Politechnika Śląska
Wydział Matematyki Stosowanej
Kierunek Informatyka
Studia stacjonarne I stopnia

Projekt inżynierski

FilmUpper

Kierujący projektem:
dr inż. Adam Zielonka

Autorzy:
Kamil Rutkowski
Jakub Rup

Gliwice 2018

Projekt inżynierski:

FilmUpper

kierujący projektem: dr inż. Adam Zielonka

1. **Kamil Rutkowski** – (55%)
   Struktura aplikacji, Algorytmika

2. **Jakub Rup** – (45%)
   Kodowanie i dekodowanie plików, Interfejs użytkownika, Algorytmika

Podpisy autorów projektu

1. .....
2. .....

Podpis kierującego projektem

.....

Oświadczenie kierującego projektem inżynierskim

Potwierdzam, że niniejszy projekt został przygotowany pod moim kierunkiem i kwalifikuje się do przedstawienia go w postępowaniu o nadanie tytułu zawodowego: inżynier.

Data Podpis kierującego projektem

Oświadczenie autorów

Świadomy/a odpowiedzialności karnej oświadczam, że przedkładany projekt inżynierski na temat:
FilmUpper
został napisany przez autorów samodzielnie.
Jednocześnie oświadczam, że ww. projekt:

- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2000 r. Nr 80, poz. 904, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/am w sposób niedozwolony,
- nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów wyższej uczelni lub tytułów zawodowych.
- nie zawiera fragmentów dokumentów kopiowanych z innych źródeł bez wyraźnego zaznaczenia i podania źródła.

Podpisy autorów projektu

1. Kamil Rutkowski, nr albumu:246604, ...(podpis:).....
2. Jakub Rup, nr albumu:246603, ...(podpis:).....

Gliwice, dnia .....

Spis treści

Wstęp

Od końca dziewiętnastego wieku sztuka filmowa towarzyszyła człowiekowi. Uważany za pierwszy film pokazany publicznie, stworzony przez braci Lumière film „Wyjście robotników z fabryki" trwał 46 sekund przy 16 klatkach na sekundę o stosunku klatki 1.33:1. Od tamtej pory ludzkość przeszła proces cyfryzacji, zmieniały się aspekty ekranów i kręconych filmów (standardowe w telewizorach kineskopowych 4:3, współczesne 16:9 a także szerokoekranowe 21:9). Zmianą podlegał także aspekt prędkości wyświetlanych klatek na sekundę, które dzisiaj najczęściej wynoszą 25, 30, 50 oraz 60 klatek na sekundę. Obsługiwane są także standardy pozwalające na wyświetlanie nawet 300 klatek na sekundę.

Celem naszej aplikacji jest możliwość zapewnienia prostego narzędzia umożliwiającego większej liczbie ludzi ulepszenia jakości posiadanych przez nich filmów. Poprzez implementację i dostarczenie użytkownikowi zbioru algorytmów wraz z opisami, będzie on w stanie zmienić rozdzielczość filmu ulepszając przy tym jego jakość w zależności od użytego algorytmu. Będzie mógł on także zwiększyć ilość klatek na sekundę w filmie, przez co oglądany przez niego obraz zyska na wrażeniu płynności. Żaden z tych algorytmów nie da wyglądu jaki posiadałby oryginał kręcony w docelowej jakości, jednakże różnica między filmem nieprzetworzonym a ulepszonym przez nasz program będzie zauważalna.

Aby sprostać założeniom projektu musieliśmy rozważyć różne jego aspekty zaczynając od technologii i języka programowania poprzez strukturę programu w jakiej chcemy pracować finalnie kończąc na algorytmach które chcieliśmy zaimplementować.

1. Technologia i język programowania

Przetwarzanie plików filmowych nawet w przypadku zastosowania najłatwiejszych algorytmów sposób jest zadaniem bardzo wymagającym pod względem wydajnościowym, dlatego wybór technologii ma kluczowe znaczenie, gdyż wpływa na wydajność całej aplikacji. Należy zwrócić uwagę na wiele czynników które mogą wpłynąć na działanie programu jak również na sposób jego implementacji.

1.1. Biblioteka do dekodowania oraz kodowania plików wideo

Od wyboru narzędzi do dekodowania/kodowania zależało powodzenie wykonania przez nas projektu. Szukając dostępnych rozwiązań pozwalających na wykonywanie tychże czynności zwracaliśmy szczególną uwagę na ich szybkość, niezawodność i opinie społeczności. W ostatecznym rozrachunku pod uwagę braliśmy następujące biblioteki programistyczne.

1.1.1. FFmpeg / libav

Narzędzia oraz biblioteki *FFmpeg* (czy też *libav* będącym jej odgałęzieniem) są jednymi z najbardziej popularnych otwartych oprogramowań służących do dekodowania, kodowania oraz manipulacji plikami multimedialnymi. Posiadają one zbiór powszechnie używanych kodeków audio i wideo. Biblioteki te napisane są w języku C.

1.1.2. OpenCV

OpenCV (Open Source Computer Vision) to zbiór bibliotek napisanych w języku C++, przeznaczonych głównie do rozpoznawania obrazów w czasie rzeczywistym. Biblioteka *cudacodec* zawarta w tym zbiorze pozwala na dekodowanie oraz kodowanie plików wideo. Jest ona oparta na kodekach *FFmpeg*-a, do procesu przetwarzania multimediów używa karty graficznej.

1.1.3. DirectShow

Platforma programistyczna opracowana przez firmę *Microsoft* dla systemów rodziny *Windows*, przeznaczona do manipulacji plikami multimedialnymi. Oparta jest ona na standardzie *Component Object Model (COM)*. Dzieli ona proces przetwarzania multimediów na moduły nazywane filtrami. Każdy filtr niezależnie wykonuje określone operacje na danych, przekazując je do kolejnego filtru, zapisując je lub wyświetlając na ekranie. Kodeki audio i wideo dostarczane są w formie takich filtrów.

1.1.4. Media Foundation

Następca *DirectShowa* dostępny dla systemów *Windows Vista* oraz nowszych. Działa na podobnej zasadzie co swój poprzednik, wspierając domyślnie większa liczbę formatów plików multimedialnych.

1.1.5. Wybór biblioteki do dekodowania/kodowania

Przy wyborze spośród narzędzi do dekodowania/kodowania kierowaliśmy się liczbą dostępnych pomocy naukowych na ich temat oraz prostotą ich używania. Ostatecznie przy tworzeniu projektu zastosowaliśmy rozwiązania oferowane przez biblioteki *FFmpeg* do otrzymania danych obrazu z plików wideo oraz gotowych narzędzi *FFmpeg*-a do późniejszego dodania dźwięku do otrzymanych plików multimedialnych.

1.2. Biblioteka interfejsu graficznego

1.2.1. Qt

Zbiór narzędzi i bibliotek programistycznych przeznaczonych do tworzenia graficznych interfejsów użytkownika. *Qt* jest biblioteką dostępną dla wielu systemów operacyjnych oraz języków programistycznych. Interfejs użytkownika budowany jest przy pomocy tzw. „widgetów". Biblioteki te pozwalają także na operacje takie jak:

- zarządzanie plikami,
- wyświetlanie grafiki 3D,
- zarządzanie bazami danych SQL.

1.2.2. GTK+

Zestaw bibliotek zbudowanych początkowo z myślą o programie *GIMP*. Posiada on gotowe komponenty graficzne, pozwalając na łatwe i szybkie budowanie interfejsów graficznych. Biblioteki te zostały napisane w języku C, jednakże zaprojektowana została wykorzystując obiektowe paradygmaty programowania. Zestaw ten posiada także nakładkę *gtkmm* pozwalającą na pracę z nim wykorzystując składnie języka C++.

1.2.3. FLTK

Multiplatformowa biblioteka przeznaczona do budowy graficznych interfejsów użytkownika napisana w języku C++. Zaprojektowana została ona tak aby programy ją wykorzystujące wyglądały tak samo na różnych platformach oraz aby były one małych rozmiarów.

1.2.4. Wybór biblioteki do tworzenia interfejsu graficznego

W naszym programie zdecydowaliśmy się zastosować biblioteki *Qt* gdyż dostarczane przez nią komponenty i mechanizmy pozwalają na szybkie i łatwe tworzenie interfejsów użytkownika. Zdecydowanym plusem był też fakt iż mieliśmy już doświadczenie przy pracy z tą technologią.

1.3. Język programowania

Wybór właściwego języka programowania jest dla nas zadaniem kluczowym, ze względu na to, że użycie odpowiedniego języka znacząco wpływa na prędkość działania programu oraz na dostępność narzędzi i bibliotek wspomagających pracę nad przetwarzanymi danymi. Znajomość danego języka także była dla nas jednym z kluczowych czynników przy jego wyborze. Naszym rozważaniom poddaliśmy następujące języki programowania.

1.3.1. C++

Język C++ jest jednym z najczęściej używanych języków niskopoziomowych. Jego popularność jest skutkiem bardzo długiego czasu na rynku oraz pewnej prostoty użycia. Kolejne wersje tego wciąż rozwijającego się języka dodają nowe, sprawdzone i ułatwiające tworzenie programów rozwiązania z innych języków programowania.

Bardzo duża wydajność oraz mnogość dostępnych bibliotek związanych z dekodowaniem i enkodowaniem plików filmowych jest bardzo ważnym aspektem tego wyboru. Wybór ten wiąże się także z kilkoma negatywnymi cechami tego języka, wymóg ręcznego zarządzania pamięcią, mało przejrzysta składnia przy tworzeniu rozwiązań o dużym stopniu skomplikowania, mało czytelne komunikaty odnośnie błędów podczas kompilacji i działania programu oraz brak ułatwień które poznaliśmy w językach wysokopoziomowych są jednymi z nich.

1.3.2. C#

Wysokopoziomowe rozwinięcie języka z rodziny C. Mimo posiadania składni podobnej do C++, język ten porzuca wiele z nieprzyjemnych jego aspektów. Poprzez automatyczne zarządzanie pamięcią, usunięcie składni charakterystycznej dla wskaźników oraz dodanie wielu nowych mechanizmów, wygląd kodu oraz prędkość tworzenia programów znacząco wzrasta. Bardzo ważnym składnikiem C# jest także LINQ które umożliwia bardzo kompaktowe i przejrzyste działanie na kolekcjach co jest dużym plusem przy przetwarzaniu plików wideo, jako że są one kolekcjami pojedynczych klatek złożonych z kolekcji pikseli. Wszystkie z tych udogodnień mają jednak cenę w postaci mniejszej wydajności w stosunku do C++ oraz brak wystarczającego wsparcia dla zarządzania plikami wideo jako że język ten jest stworzony z myślą o szybkim tworzeniu aplikacji biurowych.

1.3.3. Rust

Prędkość działania porównywalna z językiem C++, duże bezpieczeństwo pod względem zarządzania pamięcią, łatwość w konwersji programu jednowątkowego na wielowątkowy, składnia języka oraz wiele udogodnień zaciągniętych z języków wysokiego poziomu. To jedne z wielu punktów które zachęcały do wyboru tego języka. Niestety, z uwagi na to, że jest to język stosunkowo młody zauważalny jest brak lub wczesna wersja bibliotek umożliwiających przyjemną pracę na plikach filmowych. Jest to także język z którym nie mamy dużego doświadczenia.

1.3.4. Python

Python mimo bycia językiem wysokopoziomowym ma duże możliwości w przetwarzaniu ogromnych ilości danych dzięki bibliotece numpy. Jest to bardzo szybka biblioteka zaimplementowana w języku niskiego poziomu. Mieliśmy także styczności

z tym językiem podczas naszych studiów. Dużym minusem jest dla nas brak silnych statycznych typów który znacząco ułatwia naukę nowych rozwiązań.

1.3.5. Wybór języka programowania

Rozważając cechy każdego z języków, nasze umiejętności w posługiwaniu się nimi oraz specyfikę programu który chcemy napisać wybraliśmy język C++. Kluczowymi cechami które przekonały nas do wyboru tego języka były: mnogość bibliotek związanych z tematem projektu lub uniwersalnie wspomagających często napotykane problemy, prędkość działania oraz duża znajomość samego języka.

2. Dekodowanie oraz kodowanie filmów

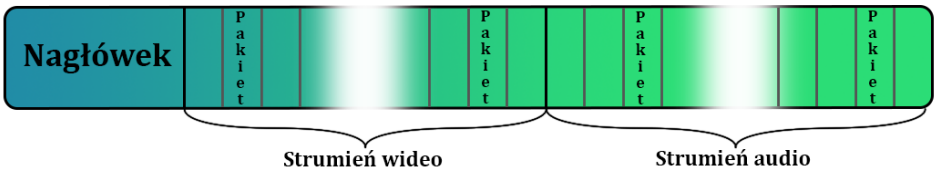
2.1. Format pliku wideo

Formatem pliku wideo nazywamy sposób reprezentacji danych wideo i audio w pamięci komputera. Na format wideo składa się kontener przechowujący strumienie między innymi audio i wideo oraz kodeki określające jak te strumienie są kodowane i kompresowane. Kontenery obsługują ściśle określone kodeki audio i wideo. Większość kontenerów jest w stanie przechowywać wyłącznie jeden strumień wideo oraz jeden strumień dźwięku, niektóre z nich mogą przechowywać informacje o napisach (przykładem kontenera pozwalającego na przechowywanie wielu strumieni takiego samego typu jest *Matroska*).

2.2. Struktura pliku wideo

W pliku wideo wyróżnić można następujące elementy:

- nagłówek zawierający dane formacie pliku, jego kontenerze, zastosowanych kodekach oraz opcjonalne metadane takie jak tytuł czy też autor,
- pakiety posiadające dane o pewnych określonych częściach strumieni które są zapisane w pliku.



Rysunek 1: Schemat struktury pliku wideo

W przypadku strumienia wideo pojedynczy pakiet przechowuje dane o jednej jego klatce (bądź też, w przypadku niektórych kodeków, części która uległa zmianie względem poprzedniej klatki). Dla strumieni audio pakiet przechowuję dane o pewnej ilości próbek dźwięku zależnej od zastosowanego podczas kodowania próbkowania oraz ilości klatek wyświetlanych na sekundę. Pakiety przechowują także informacje pozwalające na synchronizacje czasową strumieni danych potrzebne do poprawnego odtwarzania pliku.

2.3. Dekodowanie

Aby odczytać z pliku wideo pojedynczą klatkę obrazu potrzebujemy wydobyć z niego pojedynczy pakiet i zdekodować go określonym przez format tego pliku kodekiem. Zanim do tego przystąpimy musimy jednak wydzielić pakiety należące do strumienia wideo. Proces ten nazywany jest demultipleksowaniem, czyli podziałem pliku na oddzielne strumienie danych obrazu i dźwięku. Po wydzieleniu z pliku ścieżki wideo odczytujemy kolejne pakiety, które przesyłane są do dekodera. Dekoder zwraca nam przetworzoną klatkę oraz informacje o niej.

Przed przystąpieniem do przetwarzania otrzymanych danych o pikselach należy określić w jakiej przestrzeni barw zostały one zapisane w pliku. Większość formatów zapisuje piksele korzystając z modelu barw YUV, w którym składowa Y odpowiada za luminancje obrazu, a składowe UV odpowiadają za nadanie mu barwy. Dla potrzeb naszych algorytmów wymagana jest konwersja klatek do przestrzeni barw RGB. Aby tego dokonać stosuje się poniższe działanie mnożenia macierzy przez wektor:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1,13983 \\ 1 & -0,39465 & -0,58060 \\ 1 & 2,03211 & 0 \end{bmatrix} * \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

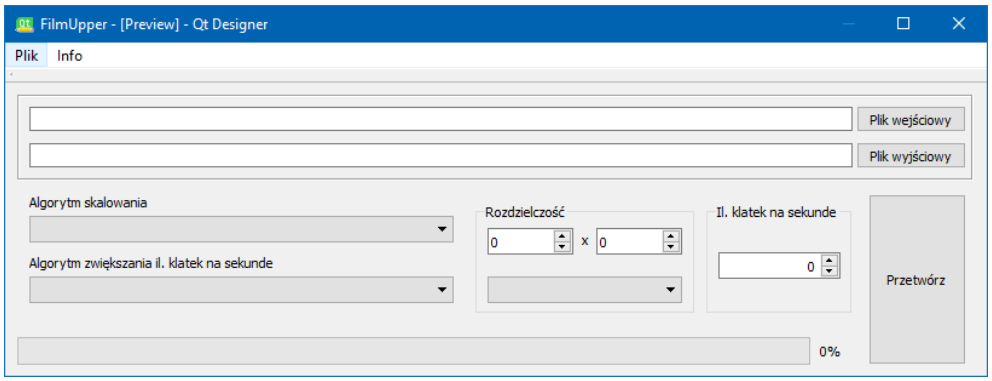
2.4. Kodowanie

Kodowanie jest procesem odwrotnym do dekodowania. Na początku tworzony jest plik do którego zapisywany jest jego nagłówek. Przetworzona przez nas klatka obrazu konwertowana jest na model przestrzeni barw obsługiwany przez format

pliku docelowego. Następnie klatka ta przesyłana jest do kodera który koduje obraz i tworzy na jego podstawie pakiet. W przypadku gdy zapisujemy do formatu który zapisuje klatki pośrednie jako różnicę między klatką kluczową a obecnie zapisywaną, koder buforuje obrazy do momentu otrzymania określonej przez format jej ilości, generując różnicę między nimi i tworząc pakiety na ich podstawie. Następnie pakiety zapisywane są do pliku przechodząc przez proces multipleksowania jeżeli zapisujemy do pliku z istniejącym już strumieniem danych. Ostatecznie plik uzupełniany jest o dodatkowe dane potrzebne do jego odtworzenia i zamykany.

3. Interfejs użytkownika

Interfejs naszej aplikacji zbudowaliśmy w oparciu o zbiór bibliotek i narzędzi programistycznych *Qt*. Interfejs prosi użytkownika o podanie pliku źródłowego oraz



Rysunek 2: Główny interfejs programu

pliku docelowego. W tym celu może on skorzystać z odpowiednich pól wpisując do nich ścieżki plików, wykonując akcję „przeciąg-upuść" na te pola lub korzystając z typowego okna wyboru pliku. Po wyborze parametrów obróbki filmu, użytkownik może przejść do procesu przetwarzania pliku klikając w odpowiedni przycisk. Podczas procesu obróbki program stara się wyświetlić aktualny jej postęp.

Po wykonaniu procesu przetwarzania użytkownik ma opcję porównania jakości filmu źródłowego z tym otrzymanym w wyniku działania programu. Odbywa się to poprzez wyświetlenie obu filmów obok siebie w nowym oknie programu.

Do powiadamiania użytkownika o ewentualnych błędach programu zastosowaliśmy mechanizm *„sygnałów i slotów"* bibliotek *Qt*.

```
public slots:
    void exampleSlot(int param);
signals:
    void exampleSignal(int param);
```

Pozwala on na komunikacje oraz interakcje pomiędzy komponentami tego zbioru.

Mechanizm ten działa na prostej zasadzie. W przypadku wykonania przez program pewnego działania wysyłany jest sygnał poleceniem

```
emit exampleSignal( exParam );
```

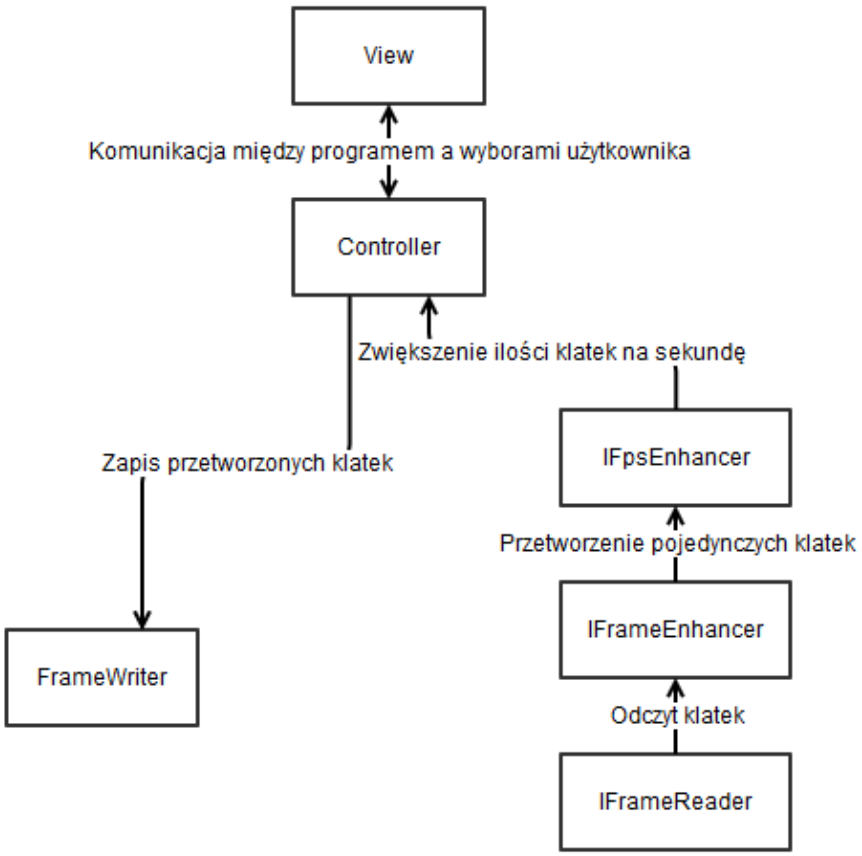
a następnie wykonywana jest funkcjonalność slotów „podpiętych" wcześniej do danego sygnału.

```
QObject::connect(const QObject *sender, const char *signal,
const QObject *receiver, const char *method,
Qt::ConnectionType type = Qt::AutoConnection)
```

Jeden sygnał potrafi być obsługiwany przez wiele slotów, jednakże jeden slot może być przypisany tylko do jednego sygnału - relacja $1doN$. W przypadku naszego programu jeżeli program miałby zwrócić wyjątek, emitowany jest sygnał *unexpected_error* w wyniku którego wyświetlany jest komunikat z treścią wyjątku.

4. Struktura programu

Głównym celem projektu jest takie zaprojektować naszą aplikację, aby możliwa była dalsza jej rozbudowa polegająca na dodawaniu nowych algorytmów. Struktura w jakiej organizowalibyśmy kod programu jest bardzo istotnym zagadnieniem z punktu widzenia przejrzystości systemu oraz ergonomii pracy z nim. Struktura naszego programu dzieli się na 2 główne części: część odpowiadającą za komunikację z użytkownikiem oraz wybór odpowiednich algorytmów, oraz część polegającą na przetwarzaniu pliku wideo przez wybrane przez użytkownika algorytmy. Tworzyliśmy architekturę naszej aplikacji zgodnie z paradygmatem odwrócenia sterowania (IoC), polegającym na przekazywaniu gotowych obiektów do klas ich potrzebujących. Znacząco ułatwiło to projektowanie całej aplikacji a zwłaszcza jej testowanie.



Rysunek 3: Główna struktura programu

4.1. Komunikacja z użytkownikiem

Do komunikacji z użytkownikiem stosujemy klasy FilmUpperView (dalej View) i FilmUpperController (dalej controller). Jest to uproszczona wersja wzorca projektowego MVC(Model, View, Controller) Początkowo chcieliśmy stworzyć infrastrukturę opartą na bardzo podobnym w założeniach wzorcu MVVM (Model, View Model, Controller) oraz zastosować rozwiązania reaktywne w kontakcie z użytkownikiem. Mimo, że głównym celem systemów reaktywnych jest asynchroniczne zarządzanie zdarzeniami poprzez stosowanie strumieni zamiast tradycyjnego podejścia do danych, to reaktywne podejście do zdarzeń bardzo upraszcza ich obsługę oraz jest bardzo przejrzyste pod względem przepływu sterowania. Niestety z uwagi na to, ze rozwiązania te stosowaliśmy tylko w aplikacjach opartych na języku C#, nie byliśmy zaznajomieni z tą technologią w językach niższego poziomu. Nasza aplikacja nie jest także na tyle rozbudowana pod względem złożoności możliwych stanów ani jej elementy nie reagują w sposób znaczny na zmiany które użytkownik wykonuje w interfejsie, więc zauważyliśmy, że lepszym rozwiązaniem będzie zastosowanie prostszego modelu. Komunikacja między użytkownikiem a programem odbywa się pomiędzy klasami View a Controller. W klasie View zarządzamy interfejsem użytkownika, wyświetlającym możliwe do wyboru algorytmy i przekazujemy jego decyzje do Controllera. W klasie Controller następuje ustalenie jakie algorytmy są dostępne do wykorzystania przez użytkownika, obsługiwany jest proces ulepszania jakości obrazu oraz zapisywania wyniku tej operacji. Obecnie algorytmy które są dostępne do wykorzystania są wpisane na stałe w kodzie programu, przez co aby dodać kolejny algorytm należy go dopisać do odpowiedniego wektora algorytmów. Jedną z możliwych i podstawowych opcji, które rozważaliśmy do rozbudowy funkcjonalności programu jest możliwość automatycznego dodawania dostępnych algorytmów poprzez walidację zawartości odpowiednich folderów. Umożliwiało by to rozbudowę funkcjonalności programu przez następnych użytkowników, a mianowicie poprzez tworzenie bibliotek dynamicznych o odpowiedniej budowie (opartej o odpowiednie klasy bazowe) i ich automatyczne wczytywanie przy starcie programu. Dało by to bardzo dużą możliwość rozbudowy bazowej funkcjonalności, jednakże w obecnej fazie, funkcjonalność ta nie jest uważana przez nas za podstawową.

4.2. Implementacja procesu dekodowania/kodowania przy pomocy bibliotek FFmpeg

4.2.1. Demultipleksowanie i dekodowanie

Przed przystąpieniem do odczytu jakiegokolwiek pliku wideo przy pomocy bibliotek *FFmpeg*-a musieliśmy powołać funkcję

```
av_register_all();
```

Funkcja ta inicjalizuje oraz rejestruje wszystkie możliwe multipleksery, demultipleksery oraz kodeki formatów plików wideo. Możliwy jest także wybór formatów które obsługiwać będą funkcje bibliotek poprzez wywołanie funkcji:

```
av_register_input_format(AVInputFormat* format);
av_register_output_format(AVOutputFormat* format);
```

gdzie jako parametr przekażemy wybrane przez nas formaty. Następnie poleceniem:

```
avformat_open_input(AVFormatContext** formatContext,
const char* filename, AVInputFormat* fmt, AVDictionary** options);
```

otwieramy plik znajdujący się pod ścieżką podaną w parametrze *filename* zapisując informacje o nim w strukturze *formatContext*. Struktura ta posiada informacje między innymi o formacie pliku, dostępnych strumieniach audio/wideo oraz sposobie ich kodowania. Wskaźniki do strumieni danych zawarte są w tablicy *streams* będącej polem struktury *formatContext*. Aby odnaleźć strumień obrazu wideo użyliśmy polecenia

```
av_find_best_stream(AVFormatContext* formatContext,
enum AVMediaType type, int wanted_stream_nb, int related_stream,
AVCodec** decoder_ret, intflags);
```

które zwraca jego indeks w tablicy *streams*. Następnie na podstawie pola *codec_id* znajdującego się w strukturze opisującej strumień wideo odnajdujemy dekoder i inicjalizujemy go funkcjami

```
avcodec_find_decoder(enum AVCodecID codec_id);
avcodec_open2(AVCodecContext* codecContext, const AVCodec* codec,
AVDictionary** options);
```

Biblioteki *FFmpeg*-a dają nam dwa sposoby dekodowania i kodowania plików. W przypadku dekodowania skorzystaliśmy z metody która pozwala nam na bezpośrednie odczytanie kolejnej w strumieniu zakodowanej klatki obrazu. Dokonujemy tego poniższą pętlą while:

```
while (ret = av_read_frame(formatCTX, packet) >= 0)
{
    if (packet->stream_index == videoStream)
    {
        avcodec_decode_video2(codecCTX, frame, &frameFinished,
                              packet);
        if (frameFinished)
            break;
    }
}
```

Pętla ta odczytuje kolejne pakiety z pliku wideo sprawdzając czy należą one do strumienia danych obrazu i dekodując je aż otrzymamy pełną klatkę. Przed zwróceniem danych o obrazie do naszych algorytmów musimy przekształcić ją tak aby była ona w modelu barw RGB. Biblioteki *FFmpeg*-a dostarczają nam funkcję

```
context = sws_getContext(int sourceWidth, int sourceHeight,
enum AVPixelFormat sourceFormat, int destinationWidth,
int destinationHeight, enum AVPixelFormat destinationFormat,
int flags, SwsFilter* srcFilter, SwsFilter* dstFilter,
const double* param);


sws_scale(struct SwsContext* context, uint8_t* const sourceData[],
const int sourceStride[], int srcSliceY, int srcSliceH,
uint8_t* const destinationData[], const int destinationStride[]);
```

która pozwala na konwersje między modelami barw.

4.2.2. Kodowanie i multipleksowanie

Tak jak w przypadku dekodowania pliku wideo, przed przystąpieniem do kodowania i zapisu danych musimy zainicjalizować strukturę *formatContext*, wypełnić ją wymaganymi danymi oraz zainicjalizować i skonfigurować koder, którego chcemy używać. Dokonujemy tego metodą

```
avcodec_find_encoder(enum AVCodecID id);
```

i wypełniając otrzymaną w jej wyniku strukturę pożądanymi przez nas parametrami zapisu (miedzy innymi ilością klatek na sekundę, rozdzielczością obrazu czy przepływność). Następnie dodajemy do struktury *formatContext* nowy strumień wideo poleceniem

```
avformat_new_stream(AVFormatContext* s, const AVCodec* c)
```

Przed przystąpieniem do kodowania klatek musimy zapisać do pliku docelowego nagłówek formaty funkcją

```
avformat_write_header(AVFormatContext* s, AVDictionary** options)
```

Po skonwertowaniu danych klatki do modelu barw wymaganego przez wybrany format pliku kodowanie odbywa się następującym algorytmem:

```
if (ret = avcodec_send_frame(codecCTX, frameOut) < 0)
    throw std::runtime_error("Couldn't send frame to encoder");


ret = avcodec_receive_packet(codecCTX, packet);
if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
    return;
else if (ret < 0)
    throw std::runtime_error("Couldn't recieve packet from encoder");
if (av_interleaved_write_frame(formatCTX, packet) < 0)
    throw std::runtime_error("Couldn't save packet to file");
```

która wysyła klatki do kodera do momentu aż jego wewnętrzny bufor zostanie wypełniony i otrzymamy pakiet który zapisać możemy do pliku.

Po zapisaniu wszystkich klatek pozostaje nam jeszcze wyczyszczenie bufora kodeka poprzez wysłanie do niego wskaźnika *NULL* i ewentualne zapisanie powstałego pakietu.

Do otrzymanego pliku wideo dźwięk dodawany jest przy użyciu gotowego narzędzia należącego do pakietu *FFmpeg*. Odbywa się to poprzez skopiowanie strumienia danych audio z pliku źródłowego do pliku docelowego.

4.3. Organizacja algorytmów i sposobu przekształcania plików wideo

W naszym projekcie z założenia stosujemy dwa typy algorytmów. Są to algorytmy:

- wpływające na jakość pojedynczej klatki wideo,
- wpływające na ilość klatek wideo na sekundę.

Algorytmy te korzystają z klas które przechowują ważne informacje odnośnie przetwarzanego wideo. Klasami tymi są klasy VideoFrame oraz FilmQualityInfo.

Klasa VideoFrame przechowuje informację na temat pojedynczej klatki wideo, FilmQualityInfo przechowuje natomiast informację odnośnie parametrów pliku wideo.

Przy ich użyciu oraz przy użyciu klas bazowych IFrameReader i IFrameWriter (i stworzonych na ich podstawie implementacjach) przetwarzamy plik wideo w następujący sposób:

1. klasa IFrameReader czyta kolejne klatki z pliku wejściowego
2. FrameEnhancerBase przetwarza przeczytane klatki, zwiększając ich rozdzielczość zgodnie z zastosowanym algorytmem
3. FpsEnhancerBase przetwarza ulepszone przez FrameEnhancerBase klatki, tworząc nowe klatki zgodnie z wybranym algorytmem
4. IFrameWriter zapisuje otrzymane przez FpsEnhancerBase klatki

4.3.1. IFrameReader - odczytywanie informacji z pliku wideo

Ta klasa bazowa ma za zadanie odczytywać informację odnośnie klatek wideo oraz dźwięku. Utworzyliśmy ją jako klasę bazową ze względu na dwa aspekty:

1. możliwość stworzenia klasy testowej o ustalonych właściwościach - daje to bardzo dużą swobodę w prowadzeniu testów, bez względu na stan pracy nad właściwą implementacją

2. umożliwia stosunkowo bezbolesną zmianę biblioteki dekodującej pliki wideo bez zmiany sposobu korzystania z niej, daje nam to też swobodę na przyszłość, jeśli chcielibyśmy porównać działanie 2 różnych bibliotek umożliwiających odczyt plików wideo.

Klasa ta dostarcza następującą funkcjonalność:

```
class IFrameReader
{
public:
    virtual VideoFrame* ReadNextFrame() {};
    virtual FilmQualityInfo* GetVideoFormatInfo() {};
    virtual bool AreFramesLeft() { return true; };
};
```

Klasy dziedziczące po niej implementują następującą funkcjonalność:

```
* virtual VideoFrame* ReadNextFrame() {};
```

- metoda ta pozwala na odczytanie następnej klatki razem z odpowiadającymi jej próbkami dźwiękowymi,

```
* FilmQualityInfo* GetVideoFormatInfo() {};
```

- metoda ta przekazuje informację odnośnie formatu danych przechowywanych w pliku wideo,

```
* bool AreFramesLeft() { return true; };
```

- metoda ta informuje, czy pozostały jeszcze klatki do odczytu.

4.3.2. FrameEnhancerBase i IFrameEnhancerHeader - ulepszanie pojedynczej klatki

Zadaniem klasy bazowej FrameEnhancerBase jest przetwarzanie otrzymanej klatki w taki sposób, aby przy użyciu wybranego algorytmu zwiększyć rozdzielczość klatki wideo. Klasa bazowa IFrameEnhancerHeader jest odpowiedzialna za podstawowe informacje na temat danego FrameEnhancerBase oraz za utworzenie obiektu tej klasy. Daje nam to możliwość posiadania wszystkich wymaganych do wyświetlenia informacji bez tworzenia samych obiektów (tworzymy je dopiero w momencie w którym są potrzebne).

Możliwości FrameEnhancerBase to:

```
class FrameEnhancerBase
{
protected:
IFrameReader* _inputFrameStream;
FilmQualityInfo* _targetQualityInfo;
FilmQualityInfo* _sourceQualityInfo;



public:
FrameEnhancerBase(IFrameReader* inputFrameReader, FilmQualityInfo* targetQualityInfo) {
_inputFrameStream = inputFrameReader;
_targetQualityInfo = targetQualityInfo;
_sourceQualityInfo = _inputFrameStream->GetVideoFormatInfo();
}

virtual VideoFrame* ReadNextEnhancedFrame() { return nullptr; };
FilmQualityInfo* GetSourceQuality() { return _sourceQualityInfo; };
virtual bool AreFramesLeft() { return _inputFrameStream->AreFramesLeft(); };
};
```

Klasa ta dostarcza następującą funkcjonalność:

```
* IFrameReader* _inputFrameStream;
```

- _inputFrameStream zawiera obiekt implementacji IFrameReader dzięki któremu klasa ta może czytać klatki do przetworzenia,

```
* FilmQualityInfo* _targetQualityInfo;
```

- _targetQualityInfo jest obiektem przechowującym informację na temat docelowego formatu wideo. Dzięki temu obiektowi jesteśmy w stanie ustalić docelowy rozmiar klatki wideo,

```
* FilmQualityInfo* _sourceQualityInfo;
```

- _sourceQualityInfo jest obiektem przechowującym informację na temat źródłowego formatu wideo. Mimo tego, że mamy ciągły dostęp do tej informacji przy pomocy _inputFrameStream, to przechowywanie tej informacji w tej postaci jest o wiele wygodniejsze,

```
* FrameEnhancerBase(IFrameReader* inputFrameReader,
FilmQualityInfo* targetQualityInfo) {
        _inputFrameStream = inputFrameReader;
        _targetQualityInfo = targetQualityInfo;
        _sourceQualityInfo = _inputFrameStream->GetVideoFormatInfo();
}
```

- konstruktor klasy bazowej przypisuje wartości zmiennych oraz uzyskuje dane na temat formatu pliku źródłowego,

```
* virtual VideoFrame* ReadNextEnhancedFrame() { return nullptr; };
```

- zwraca przetworzoną klatkę wideo wraz z dźwiękiem,

```
* FilmQualityInfo* GetSourceQuality() { return _sourceQualityInfo; };
```

- zwraca informacje na temat jakości pliku źródłowego,

```
* virtual bool AreFramesLeft() { return _inputFrameStream->AreFramesLeft(); };
```

- zwraca informacje na temat tego, czy pozostały jeszcze jakieś klatki do odczytu.

Możliwości klasy IFrameEnhancerHeader są następujące:

```
class IFrameEnhancerHeader
{
public:
IFrameEnhancerHeader(std::string name, std::string desription)
{
Name = name;
Description = desription;
}
std::string Name;
std::string Description;
virtual FrameEnhancerBase* Enhancer(IFrameReader* inputFrameReader,
 FilmQualityInfo* targetQualityInfo) {
     return new FrameEnhancerBase(inputFrameReader, targetQualityInfo); };
};
```

Dostarcza ona następującą funkcjonalność:

```
* std::string Name;
```

- jest nazwą algorytmu przetwarzającego plik wideo,

```
* std::string Description;
```

- jest opisem algorytmu dla użytkownika,

```
* virtual FrameEnhancerBase* Enhancer(IFrameReader* inputFrameReader,
 FilmQualityInfo* targetQualityInfo) {
     return new FrameEnhancerBase(inputFrameReader, targetQualityInfo); };
};
```

- zwraca nowo utworzony obiekt klasy ulepszającej jakość klatki wideo.

4.3.3. FpsEnhancerBase i IFpsEnhancerHeader - tworzenie nowych klatek pośrednich

Zadaniem klasy bazowej FpsEnhancerBase jest przetworzenie otrzymanych klatek w taki sposób aby zwiększyć ilość klatek wynikowych poprzez tworzenie klatek

pośrednich na podstawie otrzymanych od IFrameEnhancerBase, ulepszonych klatek. IFpsEnhancerHeader przechowuje informacje na temat odpowiadającego mu FpsEnhancerBase oraz umożliwia utworzenie obiektu tej klasy.

Możliwości FpsEnhancerBase to:

```
class FpsEnhancerBase {
protected:
    FrameEnhancerBase* _frameEnhancer;
    FilmQualityInfo* _targetQuality;
    FilmQualityInfo* _sourceQuality;
public:
    FpsEnhancerBase(FrameEnhancerBase* frameEnhancer,
     FilmQualityInfo* targetQuality) {
        _frameEnhancer = frameEnhancer;
        _targetQuality = targetQuality;
        _sourceQuality = _frameEnhancer->GetSourceQuality();
    }

    virtual VideoFrame* ReadNextFrame() { return nullptr; };

    virtual bool AreFramesLeft() { return _frameEnhancer->AreFramesLeft(); };
};
```

```
* FrameEnhancerBase* _frameEnhancer;
```

- przechowuje obiekt klasy przetwarzającej klatki źródłowe,

```
* FilmQualityInfo* _targetQuality;
```

- przechowuje format docelowy filmu,

```
* FilmQualityInfo* _sourceQuality;
```

- przechowuje format źródłowy filmu,

```
* virtual VideoFrame* ReadNextFrame() { return nullptr; };
```

- zwraca następną wynikową klatkę wideo

```
* virtual bool AreFramesLeft() { return _frameEnhancer->AreFramesLeft(); };
```

- informuje, czy pozostały jeszcze klatki do odczytania.

Możliwości klasy IFpsEnhancerHeader są następujące:

```
class IFpsEnhancerHeader {
public:
    IFpsEnhancerHeader(std::string name, std::string description)
    {
     Name = name;
     Description = description;
    }
    std::string Name;
    std::string Description;
    virtual FpsEnhancerBase* GetFpsEnhancer(FrameEnhancerBase* frameEnhancer,
 FilmQualityInfo* qualityInfo) {
      return new FpsEnhancerBase(frameEnhancer, qualityInfo);
      };
};
```

Dostarcza ona następującą funkcjonalność:

```
* std::string Name;
```

- jest nazwą algorytmu przetwarzającego otrzymane klatki,

```
* std::string Description;
```

- jest opisem algorytmu dla użytkownika,

```
* virtual FpsEnhancerBase* GetFpsEnhancer(FrameEnhancerBase* frameEnhancer,
 FilmQualityInfo* targetQualityInfo) {
     return new FpsEnhancerBase(frameEnhancer, qualityInfo);
};
```

- zwraca nowo utworzony obiekt klasy tworzącej pośrednie klatki wideo.

5. Algorytmy

Wykorzystywane przez nas algorytmy dzielą się na dwie poprzednio wyszczególnione kategorie – przetwarzające pojedynczą klatkę obrazu (FrameEnhancerBase) oraz tworzące nowe klatki pośrednie (FpsEnhancerBase). Aby przedstawić efekty naszych algorytmów wykorzystaliśmy fragment filmu animowanego „Big Buck Bunny".

5.1. Ulepszanie pojedynczej klatki

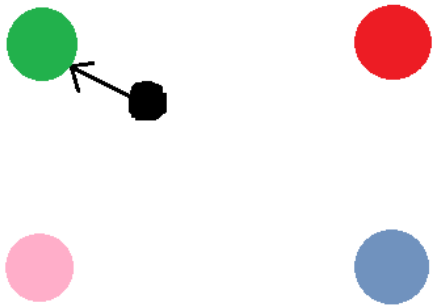
Algorytmy te ulepszają jakość klatki wideo na podstawie odpowiednich algorytmów skalowania. Wykorzystane przez nas algorytmy wykorzystują informacje zawarte tylko na klatce źródłowej (nie wykorzystujemy poprzednich lub następnych klatek obrazu). Różnią się one zastosowaniem, złożonością obliczeniową oraz otrzymanym obrazem wynikowym.

Rysunek 4: Fragment klatki źródłowej z filmu

Rysunek 4 przedstawia fragment filmu źródłowego nagranego w rozdzielczości 1280x720. Wykorzystaliśmy nasze algorytmy do przeskalowania filmu do rozdzielczości 3840x2160 i porównaliśmy wybrany fragment.

5.1.1. Algorytm najbliższego sąsiada

Algorytm ten jest najszybszym z wykorzystanych przez nas algorytmów, jednakże sytuacje w których jego wykorzystanie da bardzo dobry efekt nie są liczne. Algorytm ten polega na wyszukaniu najbliższego piksela źródłowego odpowiadającemu aktualnie ustalanemu pikselowi wynikowemu i ustawieniu jego koloru w pikselu wynikowym. Algorytm ten nie posiada żadnego wygładzania krawędzi, przez co obraz wynikowy będzie wyglądał jak złożony z większych pikseli źródłowych.



Rysunek 5: Wyszukiwanie wartości nowego piksela metodą najbliższego sąsiada

Na rysunku 5 przedstawiona jest sytuacja w której kolorem zielonym, czerwonym, różowym i niebieskim przedstawione są kolory pikseli z pewnego fragmentu obrazu źródłowego. Czarnym kołem oznaczony został nowy piksel umieszczony w odpowiadającym mu miejscu w źródłowej przestrzeni obrazu. Metoda najbliższego sąsiada dla tego piksela wynikowego wybierze kolor zielony (wskazany strzałką) z uwagi na to, że w przestrzeni źródłowej jest on najbliższy położeniu nowemu pikselowi.

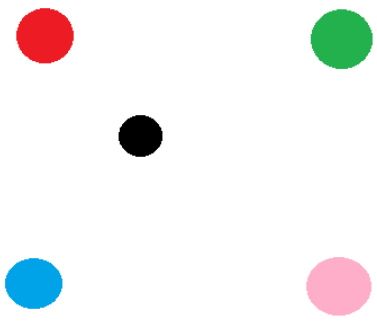
Algorytm ten można stosować z powodzeniem w skalowaniu filmów o charakterystyce tak zwanego ”pixel art”. Jest to stylistyka wzorująca się na wyglądzie gier komputerowych z czasów, gdy dominowały platformy ośmiobitowe. Styl ten charakteryzuje się widocznymi pojedynczymi pikselami, przez co wygładzanie przejść między pikselami wynikowymi w innych algorytmach jest efektem niepożądanym.

Jest to natomiast idealna sytuacja na zastosowanie algorytmu najbliższego sąsiada, jako że ten akcentuje każdy piksel powiększając go w obrazie wynikowym.

Rysunek 6: Wybrany fragment przetworzony algorytmem najbliższego sąsiada

5.1.2. Algorytm interpolacji dwuliniowej

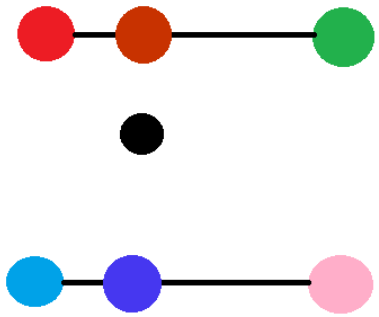
Algorytm interpolacji dwuliniowej polega na obliczeniu wynikowej wartości koloru na podstawie 4 pikseli otaczających.



Rysunek 7: Nowy piksel w przestrzeni obrazu źródłowego

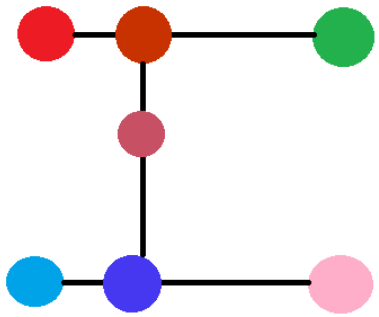
Aby obliczyć wartość koloru nowo utworzonego piksela algorytm interpolacji dwuliniowej oblicza 2 nowe piksele pośrednie. Są one wynikiem zmieszania dwóch

odpowiednich pikseli (krok ten można przeprowadzić parując ze sobą piksele horyzontalnie lub wertykalnie). Mieszanie kolorów tych pikseli traktujemy jako funkcję liniową której wartości zmieniają się od jednego koloru do drugiego. Wybieramy wartość funkcji w miejscu przecięcia tej funkcji z prostą prostopadłą przechodzącą przez punkt wynikowy. W naszej implementacji wybraliśmy mieszanie horyzontalne odpowiednich kolorów.



Rysunek 8: Obliczenie wartości kolorów pośrednich

Dzięki temu, otrzymane przez nas nowe piksele pośrednie znajdują się w jednej linii z pikselem wynikowym. Możemy teraz obliczyć wartość koloru piksela wynikowego używając wartości kolorów pośrednich do utworzenia nowej funkcji liniowej i wybrania jej wartości w miejscu przecięcia się funkcji z pikselem wynikowym.



Rysunek 9: Obliczenie wartości koloru piksela wynikowego na podstawie kolorów pośrednich

Algorytm ten jest nie jest skomplikowany obliczeniowo, ale jest zauważalnie wolniejszy od algorytmu najbliższego sąsiada. Dzięki swoim właściwością, bardzo dobrze wygładza przejścia między pikselami, przez co ma szeroką gamę zastosowań.

Rysunek 10: Wybrany fragment przetworzony algorytmem interpolacji dwuliniowej

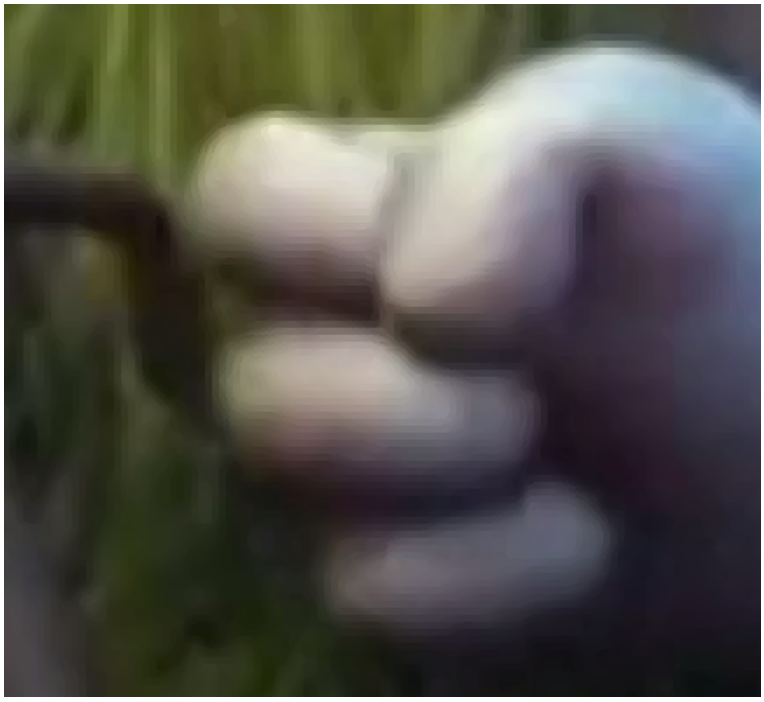
5.1.3. Algorytm interpolacji bikubicznej

Algorytm ten jest najbardziej skomplikowanym obliczeniowo algorytmem jaki zaimplementowaliśmy. Korzysta on z krzywych obliczanych na podstawie macierzy pikseli rozmiaru 4 na 4. Podobnie jak w interpolacji obliczane są wartości kolorów pikseli pośrednich obliczanych dla kolejnych wierszy macierzy dzięki którym obliczany jest splajn zgodny ze wzorem Catmulla-Roma:

$$sp(p_{x-1}, p_x, p_{x+1}, p_{x+2}) = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & -\frac{5}{2} & 2 & -\frac{1}{2} \\ -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \end{bmatrix} * \begin{bmatrix} p_{x-1} \\ p_x \\ p_{x+1} \\ p_{x+2} \end{bmatrix}$$

gdzie u jest stosunkiem umiejscowienia nowego punktu na splajnie, a punkty p_x są kolejnymi punktami z macierzy tworzącymi splain.

Obliczone na podstawie powyższego wzoru punkty wykorzystujemy do obliczenia ostatecznej krzywej która w miejscu przecięcia się z punktem końcowym daje nam ostateczną wartość koloru.



Rysunek 11: Wybrany fragment przetworzony algorytmem interpolacji bi-kubicznej

5.1.4. Algorytm splotu

Algorytm splotu oblicza wartość piksela obrazu docelowego jako sumę wartości piksela znajdującego się na tej samej pozycji oraz wartości pikseli sąsiadujących z nim przemnożonych przez wartości macierzy MxM zwanej maską lub kernelem. Działanie to można zapisać w następujący sposób:

$$p_o[x,y] = \sum_{a=-R}^{R} \sum_{b=-R}^{R} p_i[x+a, y+b] * kernel[a,b]$$

, gdzie:

p_o - wartość piksela obrazu wyjściowego,

p_i - wartość piksela obrazu wejściowego,

$kernel$ - macierz maski,

R - promień maski.

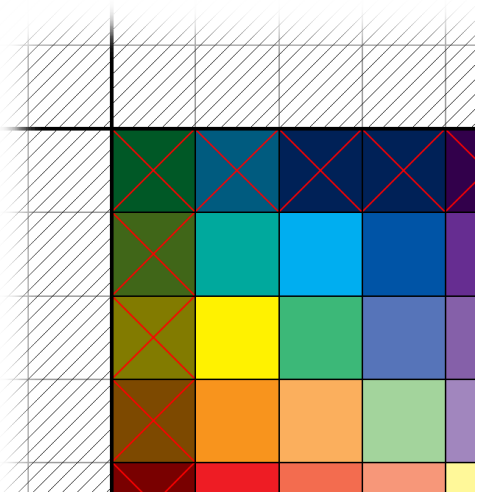
W naszym programie algorytm ten stosowany jest do wyostrzenia obrazu. Sto-

sujemy do tego celu maskę:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

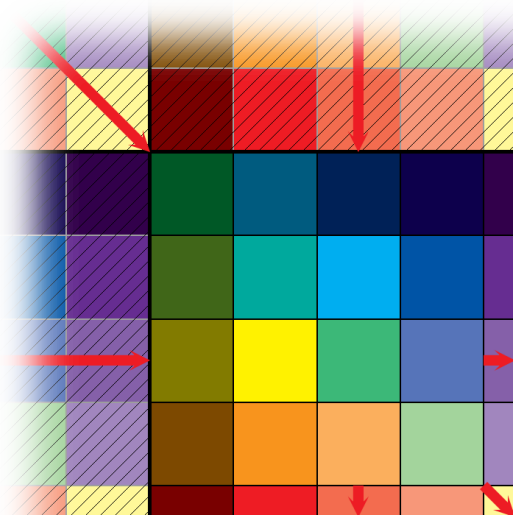
Algorytm splotu może wymagać wartości pikseli znajdujących się poza granicami obrazu źródłowego. W tym wypadku stosuję się jedno z poniższych rozwiązań

1. Obcinanie - jeżeli algorytm wymagałby do obliczenia piksela wartości znajdujących się poza krawędziami obrazu, piksel ten jest pomijany,



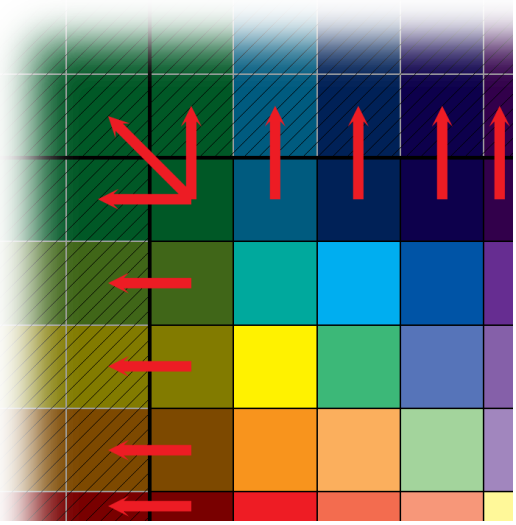
Rysunek 12: Obsługa pikseli brzegowych - obcinanie

2. Zawijanie - jeżeli wymagany jest piksel znajdujący się poza krawędziami obrazu, wartości te brane są z krawędzi przeciwnej,



Rysunek 13: Obsługa pikseli brzegowych - zawijanie

3. Rozszerzanie - krawędzie obrazu wejściowego rozszerzane są poprzez kopiowanie wartości pikseli brzegowych tyle razy ile wymaga tego algorytm.



Rysunek 14: Obsługa pikseli brzegowych - rozszerzanie

5.1.5. Normalizacja

Wartości otrzymane przez nas w wyniku operacji splotu mogą nie zawierać się w wymaganym przez nas przedziale. Wprowadza to potrzebę ich normalizacji, czego dokonać można na 2 sposoby:

1. Obcinanie wartości Jeżeli wartość przekracza wyznaczony przez nas zakres, przyjmujemy że wynikiem operacji jest wartość graniczna tego zakresu najbliższa temu wynikowi

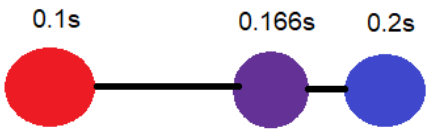
2. Normalizacja maski Aby zachować jasności pikseli obrazu możemy znormalizować maskę wykorzystywaną przez nas, poprzez podzielenie każdego z jej elementów przez sumę wszystkich elementów tejże macierzy.

5.2. Zwiększanie ilości klatek na sekundę

Algorytmy zwiększające ilość klatek obrazu na sekundę są zwykle o wiele bardziej skomplikowane od algorytmów ulepszających pojedynczą klatkę obrazu. Aby stworzyć nową klatkę obrazu z dużym oddaniem tego, jak by wyglądała w rzeczywistości wymagana by była złożona analiza wielu obrazów oraz rozwiązania korzystające z systemów opartych o sztuczną inteligencję. Stworzenie pojedynczego systemu takiego rodzaju mogłoby być tematem na osobna pracę naukową. Postanowiliśmy więc skupić się na dwóch podstawowych algorytmach które dają zadowalający efekt przy użyciu prostych technik oraz niedoskonałości ludzkiego zmysłu wzroku.

5.2.1. Algorytm interpolacji

Algorytm interpolacji polega na tym, że klatki pośrednie tworzone są na podstawie oryginalnych klatek poprzez porównywanie dwóch klatek pomiędzy którymi ma znajdować się nowo utworzona klatka. Wartość koloru w pikselach nowo utworzonej klatce obliczana jest poprzez potraktowanie pikseli klatek oryginalnych jako funkcji liniowej której parametrem jest czas. Dzięki temu każdy piksel jest obliczany na podstawie informacji kiedy (w stosunku do dwóch oryginalnych klatek) nowa klatka się pojawia.



Rysunek 15: Obliczenie wartości koloru wynikowego na podstawie czasu

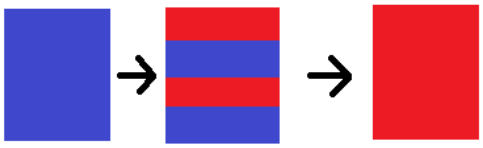
W szybko zmieniających się, dynamicznych scenach efekt ten może być widoczny jako wyraźne rozmycie, jednakże przy scenach na których nie ma gwałtownych zmian poprawia on płynność obrazu bez widzialnego negatywnego efektu.

Rysunek 16: Fragment klatki pośredniej wytworzonej algorytmem interpolacji

5.2.2. Algorytm przeplotu

Algorytm przeplotu polega na oszukiwaniu ludzkiego oka poprzez wyświetlanie tylko połowy nowych linii. Wynikiem takiego algorytmu jest film z podwojoną ilością klatek na sekundę. Powinien on być stosowany tylko dla obrazów mających stosunkowo dużą ilość klatek na sekundę z uwagi na to, że im mniejsza jest ilość klatek na sekundę tym bardziej przeplot jest widoczny. Algorytm ten stosowano już w telewizji analogowej z uwagi na niską prędkość przesyłu oraz migotanie ekranu przy wyświetlaniu obrazu w 24 lub 30(29,97) klatkach na sekundę (odpowiednio standardy PAL oraz NTSC). Naprzemienne aktualizowanie wartości wierszy nieparzystych i parzystych obrazu przy odpowiedniej szybkości wyświetlania zwiększa wrażenie płynności obrazu, jednakże im mniejsza jest źródłowa ilość klatek na sekundę tym

bardziej widoczny jest przeplot.



Rysunek 17: Tworzenie klatek pośrednich przy użyciu algorytmu przeplotu

Rysunek 18: Fragment klatki pośredniej wytworzonej algorytmem przeplotu

6. Optymalizacja i testowanie

W implementacji algorytmów ich szybkość działania jest równie ważna co ich poprawność. W trakcie implementacji algorytmów oraz ich testowania zauważyliśmy, że ich wydajność nie była zbliżona do oczekiwanej. Przeprowadziliśmy więc szereg prac optymalizacyjnych w trakcie testowania algorytmów

6.1. Procedura testowania i ich organizacja

Procedura testowania jaką założyliśmy polegała na organizacji wszystkich testów w klasie TestClassModule. Testy przez nas przeprowadzane były przeprowadzane poprzez metodę RunAllToFileTests. Uruchamiała one wszystkie metody testujące głównie algorytmy ulepszające jakość pojedynczej klatki, z uwagi na to, że test taki jest prosty do zweryfikowania w aspekcie jego poprawności poprzez porównanie pojedynczej klatki wynikowej ze źródłową. Testowaliśmy także prędkość przetwarzanej klatki.

Procedurę testowania można sprowadzić do głównej części testu skupiającej się aspekcie szybkości i następnie poprawności poprzez porównanie zapisanej klatki zgodnie z formą w jakiej chcieliśmy ją zapisać - obraz, rozpisanie pikseli na tekst oraz wyświetlenie jej przy użyciu panelu w programie.

```
int iterations = 1000;
clock_t begin = clock();
for (int xd = 0; xd < iterations; ++xd) {
    delete enhancer->ReadNextEnhancedFrame();
}
clock_t end = clock();
double elapsed_secs = (double(end - begin)/iterations) / CLOCKS_PER_SEC;
saveFile << std::to_string(elapsed_secs) + " sekund dla " +
std::to_string(iterations) +" iteracji, \n";
```

Dla 1000 iteracji pętli testujemy prędkość algorytmu przy przetwarzaniu jednej klatki obrazu. Mierzymy czas jaki zajmuje procesorowi przetworzenie 1000 klatek

i dzielimy to przez ich ilość dostając względnie dokładną estymację, jak szybko przetwarzane są klatki. Następnie zapisujemy tą wartość do pliku tekstowego.

6.2. Optymalizacja algorytmów i układu danych

Pierwszy i największy proces optymalizacji przeprowadziliśmy na klasie NNFrameEnhancer. Algorytm najbliższego sąsiada używany do skalowania klatki w tej klasie jest najszybszym zaimplementowanym przez nas algorytmem. Jednakże podczas pierwszego testu prędkość przetworzenia jednej klatki była liczona w minutach. Dzięki możliwości przechodzenia krok po kroku w programie Visual Studio oraz informacji, ile minęło od poprzedniego kroku udało nam się ustalić, że zastosowana dla uproszczenia kodu zmienna tymczasowa kopiowała pamięć klatki. Zauważyliśmy to dlatego, że operacja przypisania trwałą tam około 35 milisekund, gdy reszta pętli wykonywana była w kilka cykli procesora (ułamki milisekund). Usunięcie tej zmiennej przyśpieszyło program z około 14 sekund na jedną linię obrazu do 2 sekund na całą klatkę.

Niestety dalej było to o wiele za dużo względem tego czego oczekiwaliśmy po prędkości algorytmu (oczekiwaliśmy co najmniej kilkunastu klatek na sekundę). Następnymi podjętymi przez nas krokami było uproszczenie algorytmu poprzez usunięcie niektórych niepotrzebnych obliczeń. Uzyskaliśmy dzięki temu około 40% przyśpieszenie.

Następnie zmieniliśmy typ przechowywanych danych z QColor* na zwykłe wartości całkowite int. Po tych zmianach otrzymaliśmy wydajność na poziomie około 0.9 sekundy na klatkę obrazu.

Naszym następnym krokiem była zmiana struktury danych w której przechowywaliśmy dane odnośnie pikseli. Używana przez nas struktura

```
boost::multi_array
```

korzystała z 3 wymiarowej indeksacji. Zgodnie z tym co przeczytaliśmy w dokumentacji, klasa ta miała przechowywać dane w jednym, ciągłym bloku pamięci. Poprzez zmianę tej struktury na zwykłą tablicę int* uzyskaliśmy 30 krotne przyśpieszenie w przetwarzaniu danych (z 0.9 s do 0.03 s na klatkę).

Uzyskana przez nas prędkość przetwarzania była zgodna z naszymi oczekiwaniami. Postanowiliśmy zatem skorzystać z przetwarzania danych z użyciem wielu

wątków. Poprzez podział danych do przetworzenia na części zgodne z ilością wątków uzyskaliśmy dodatkowe przyśpieszenie. Z uwagi na koszt utworzenia wątków najlepsze wyniki uzyskaliśmy dla 2 wątków (czas przetwarzania klatki wynosił około 0.025 sekundy).

```
int threads = 2;
std::thread *tt = new std::thread[threads];

for(int t = 0; t < threads; ++t)
{
    tt[t] = std::thread(CalculateFramePararel, inputFrame, outputFrame,
    (_targetQualityInfo->Height / threads) * t,
    (_targetQualityInfo->Height / threads) * (t + 1),
    _sourceQualityInfo, _targetQualityInfo);
}

for (int t = 0; t < threads; ++t)
{
    tt[t].join();
}
```

Dodatkowo postanowiliśmy czytać następną klatkę źródłową w trakcie przetwarzania obecnej (jeśli jest to możliwe). Spowodowało to, że ostatecznie prędkość z jaką przetwarzamy klatki metodą najbliższego sąsiada wynosi około 0.02 sekundy, co daje nam możliwość przerabiania około 50 klatek na sekundę. Jest to wynik bardzo nas satysfakcjonujący.

7. Podsumowanie

7.1. Napotkane problemy podczas tworzenia projektu

7.1.1. Problemy z wybranymi narzędziami do odczytu i zapisu plików wideo

Praca z bibliotekami *FFmpeg*-a okazała się problematyczna. Wynikało to częściowo z faktu iż napisane one zostały w języku C - brak obiektowego podejścia wymagał tworzenia, inicjalizację oraz zarządzania wieloma zmiennymi i wskaźnikami. Większym problemem okazał się jednakże brak rzetelnej dokumentacji oraz innych pomocy naukowych. Duża część funkcjonalności opisana jest powierzchownie, niektóre metody nie posiadają jakichkolwiek opisów. Zbiór przykładów użycia tych bibliotek sprowadzał się do kilku plików źródłowych. W przypadku jakichkolwiek błędów czy wyników funkcji nie będących tymi jakimi oczekiwaliśmy pomocy musieliśmy szukać na forach internetowych oraz wzorować się na kodzie źródłowym innych darmowych programów korzystających z *FFmpeg*-a. Niestety ze względu an wybrany przez nas język programowania oraz używane przez nas środowisko programistyczne nie byliśmy w stanie zastosować rozwiązań konkurencyjnych.

Problemem okazało się również uzyskanie niektórych danych o pliku wideo wykorzystując biblioteki *FFmpeg*-a. Ze względu na różnorodność i ogrom różnych formatów filmów biblioteki te nie są wstanie określić ich w ogóle lub wyznaczyć je dokładnie. Wynika to z faktu, iż niektóre z formatów nie przechowują tych wartości. Przykładem informacji która nie może być wyznaczona w prosty sposób jest całkowita ilość klatek - może ona być uzyskana jedynie poprzez odczytanie całego pliku klatka po klatce lub estymowana na podstawie informacji o długości czasowej strumienia wideo i ilości klatek wyświetlanych na sekundę.

7.1.2. Problemy z technologią

Użyty przez nas język $C++$ jest językiem którego cechy były nam dobrze znane. Pomimo naszej znajomości języka musieliśmy być bardzo uważni w momencie gdy ręcznie zarządzaliśmy pamięcią. Brak wspomagania od strony języka odnośnie możliwych wycieków pamięci powodował mozolne poszukiwania miejsc, które mogły powodować problemy z ewentualnymi wyciekami pamięci.

Kolejną problematyczną częścią języka jest sposób komunikowania o występujących błędach. W nowoczesnych językach wiadomości odnośnie występujących błędów dają o nich bardzo dużą ilość informacji takich jak wartość przez którą wystąpił błąd, wartość która była oczekiwana oraz możliwe rozwiązanie problemu. Wiadomości które otrzymujemy w języku $C++$ są mało przydatne, ponieważ dostajemy tylko informację odnośnie adresu pamięci pod którym wystąpił błąd i jakiego typu był to błąd.

Problemem wynikającym z działań optymalizacyjnych był problem z odwoływaniem się do właściwych komórek pamięci w tablicy jednowymiarowej. W tablicy wielowymiarowej wybieranie właściwej komórki pamięci jest proste i intuicyjne. W przypadku tablicy jednowymiarowej prawidłowe odwoływanie się do właściwych komórek pamięci wymagało obliczania właściwego indeksu, co przy z założenia wielowymiarowych danych było problematyczne.

7.1.3. Problemy z algorytmami

Algorytmy przez nas stosowane są ogólnym rozwiązaniem matematycznym, jednakże w trakcie ich implementacji napotkaliśmy pewne problemy. Niektóre z zaimplementowanych przez nas algorytmów wymagały danych w specyficznej dziedzinie (w przypadku interpolacji bi-kubicznej wymagana była macierz źródłowa rozmiaru 4 na 4). Sprawiało to problemy w indeksach przy krawędzi, gdzie próba wyciągnięcia wartości z pamięci spowodowałaby nieprawidłowe odwołanie. Przez to musieliśmy zastosować specyficzne rozwiązania polegające na jednej z wielu możliwości: przekopiowaniu wartości źródłowych, zastosowaniu innego algorytmu lub innej, zmodyfikowanej wersji obecnego algorytmu.

7.2. Możliwości dalszego rozwoju

Mimo stworzenia aplikacji z myślą o dalszym rozwoju musieliśmy uprościć niektóre aspekty ze względu na ograniczające nas ramy czasowe. Najważniejszym zagadnieniem które rozwijalibyśmy w pierwszej kolejności byłoby stworzenie rozwiązania ładującego dostępne algorytmy na podstawie folderu z nimi, gdzie były by zamieszczone w postaci plików bibliotek dynamicznych (DLL).

Kolejnym krokiem byłoby zwiększenie dostępnych algorytmów przekształcających klatki i ich ilość na sekundę.

Przy pewnym nakładzie pracy algorytmy dekodowania i kodowania można by wyposażyć w odczyt i zapis dźwięku. Pozwoliło by to też na jego modyfikacje wykorzystując funkcje stosowane do polepszania dźwięku na przykład usuwanie szumu.

W przyszłości moglibyśmy dodać opcjonalne atrybuty dodatkowe, które mogłyby modyfikować działanie danego algorytmu, przez co jedna implementacja mogłaby korzystać z wielu możliwych opcji umożliwiających wpływających na obraz wynikowy.

Aplikację można by było zmodyfikować tak aby pozwalała na zastosowanie wielu filtrów obrazu jednocześnie.

Interfejs graficzny wzbogacić można by w obsługę wieku języków.

7.3. Wyniki działania algorytmów

Wyniki działania algorytmów jakie otrzymaliśmy były bliskie wynikom jakich się spodziewaliśmy. W przypadku stosunkowo niewielkiego zwiększania rozmiaru pojedynczej klatki i dużej źródłowej rozdzielczości przetwarzanych przez nas plików nie zauważyliśmy znaczących różnic między poszczególnymi algorytmami. Różnica była widoczna po bliższym przyjrzeniu się między algorytmem najbliższego sąsiada a resztą zaimplementowanych algorytmów. Różnica jakości klatki wynikowej między algorytmem interpolacji dwuliniowej a bi-kubicznej nie była widoczna. Według nas, algorytm interpolacji bi-kubicznej przez bardzo długi czas obliczeń oraz znikomą różnicę od znacznie szybszego algorytmu interpolacji dwuliniowej, nie jest on algorytmem godnym polecenia.

Algorytmy zmieniające ilość klatek na sekundę także zwracały wyniki bliskie oczekiwanym. W trakcie dynamicznych scen gdzie występują gwałtowne ruchy postaci wytworzone klatki pośrednie były lekko widoczne w przypadku algorytmu interpolacji. W przypadku mniej dynamicznych scen, gdzie występują delikatne ruchy postaci klatki pośrednie nie są widoczne a wrażenie większej płynności obrazu jest zauważalne. W przypadku algorytmu przeplotu efekt jest bardzo delikatnie widoczny, a wytworzone klatki pośrednie nie są wyraźnie widoczne. Oba algorytmy mają swoje zastosowania, jednakże algorytm interpolacji powinien być stosowany w mniej dynamicznych scenach, ale efekt z jego użycia będzie bardziej odczuwalny.

Literatura

[1] Tadeusiewicz R., Korohoda P., *Komputerowa analiza i przetwarzanie obrazów*, Wydawnictwo Fundacji Postępu Telekomunikacji, Kraków 1997, ISBN 83-86476-15-X

[2] Charles Poynton - Video engineering, http://poynton.ca/Poynton-video-eng.html

[3] Cristian Constantin Lalescu *Two hierarchies of spline interpolations. Practical algorithms for multivariate higher order splines.*, https://arxiv.org/pdf/0905.3564.pdf