Oprogramowanie

1. Informacje ogólne - Jagielski

Kluczowymi elementami pracy było stworzenie oprogramowania umożliwiającego odtwarzanie nieskompresowanych sekwencji wideo oraz przeprowadzenie testów subiektywnych z użyciem autorskiego odtwarzacza.

1. Wybór narzędzi - Jagielski

Wybór narzędzi wiązał się z koniecznością przeglądu dostępnych zasobów ludzkich pod kątem umiejętności tworzenia oprogramowania w danym języku. Kolejną niezbędną kwestią była analiza istniejących rozwiązań w celu znalezienia jak najlepszego narzędzia do rozwiązania problemu.

Odtwarzacz nieskompresowanych sekwencji wideo musi charakteryzować się jak największą wydajnością. Poprzez wydajność rozumiane jest jak najlepsze zarządzanie zasobami w taki sposób, aby na dostępnym sprzęcie komputerowym uzyskać jak najlepsze parametry wyświetlania kolejnych klatek. Zdecydowano się zrezygnować z języków korzystających z maszyn wirtualnych na korzyść takich, które pozwalają na dużą swobodę w zarządzaniu pamięcią. Wybór padł na język C++ ze względu na doświadczenie autorów w pracy z nim, zarówno zawodowe jak i nabyte podczas studiów.

Wybranym systemem operacyjnym został Linux, dystrybucja Xubuntu 16.04.2 posiadająca jądro w wersji 4.4.0-72-generic. Decydującymi czynnikami były łatwość instalowania kolejnych pakietów bibliotek, dostęp do narzędzi konwertujących parametry filmów oraz niskie zużycie zasobów sprzętowych przez biernie działający system.

[https://xubuntu.org/]

Zdecydowano się użyć środowiska programistycznego CLion dostarczanym przez firmę JetBrains, korzystając z licencji studenckiej, która pozwala na użycie IDE (ang. *Integrated Development Environment*) w celach edukacyjnych.

[https://www.jetbrains.com/]

Analiza istniejących odtwarzaczy wideo pozwoliła wyselekcjonować bibliotekę, która dała możliwość darmowego użycia i jak największej możliwości modyfikacji, czyli posiadała licencję *open source*. Ze względu na popularność odtwarzacza VLC w systemach operacyjnych Linux, podjęto decyzję o wykorzystaniu biblioteki z użyciem której powyższy odtwarzacz został stworzony – libVLC.

[http://www.videolan.org/vlc/libvlc.html]

FFmpeg, czyli narzędzie pozwalające na edycje parametrów wzorcowego wideo zostało wybrane ze względu na łatwość użycia, swobodę w wyborze zmienianych aspektów filmu oraz licencję *open source.* Wszystkie użyte w pracy sekwencje wideo zostały wygenerowane przy pomocy FFmpeg z filmów wzorcowych.

[https://ffmpeg.org/]

1. Proces tworzenia - Jagielski

Organizacja pracy w grupie jest zależna od ilości jej członków, natury rozwiązywanego problemu oraz wielu innych czynników np. ograniczeń czasowych lub sposobu prezentacji postępów.

Obecnie w informatyce dąży się do wdrożenia tak zwanych metod zwinnych programowania. Pozwalają one zminimalizować straty w przypadku, gdy część wymogów klienta ulegnie zmianie. Wynika to z faktu, że kod źródłowy programu dostarczany jest iteracyjnie w jak najmniejszych częściach. Podejście to pozwala również zmaksymalizować czas faktycznej pracy każdego z pracowników, ponieważ eliminowane są sytuacje, w których pojawia się konieczność oczekiwania na wartość dostarczaną przez innych pracowników.

Oprogramowanie przedstawione w pracy tworzone było w zespole dwuosobowym. Jednym z pierwszych kroków podczas realizacji pracy magisterskiej było zdefiniowanie zasad współpracy między opiekunem, uczelnią a osobami odpowiedzialnymi za nią samą. W związku z koniecznością dotrzymania terminów odgórnie narzuconych przez uczelnie zdecydowano się wyznaczyć kilka terminów wraz z zakresem funkcjonalności, które na dany termin miały by być dostarczone. Pozwoliło to zsynchronizować wiedzę o projekcie między studentami, a prowadzącym oraz na skupienie się na wyznaczonych celach. Dopiero po akceptacji postępów przez opiekuna następowało przejście do kolejnego etapu. Na każdym ze spotkań projektowych przeprowadzano pokaz aktualnej wersji, na którym omawiano aktualny sposób działania, najnowsze zmiany w funkcjonowaniu programu oraz wyszukiwano ewentualne niedociągnięcia.

W czasie realizacji pracy magisterskiej obaj członkowie zespołu pracowali zawodowo oraz uczęszczali na zajęcia prowadzone na uczelni. W związku z tym proces tworzenia pracy magisterskiej musiał być dostosowany do ich obłożenia czasowego. Naturalnym wyborem było więc działanie w metodyce *Kanban* – skupiającej się na produkcie, bez dodatkowych, niepotrzebnych nakładów pracy oraz minimalizacji bezczynności. Stworzono prostą *tablicę kanbanową* czyli złożenie danych o postępie prac nad projektem przy podziale na poszczególne jego zadania. Zaletą takiego rozwiązania jest wiedza o przewidywanym terminie zakończenia prac oraz dowolność w wyborze czasu, w którym są one realizowane (oprócz terminów końcowych). Takie podejście pozwoliło powiązać pracę zawodową z rozwojem naukowym.

System kontroli wersji zastosowany w projekcie pozwolił na równoczesną pracę nad wieloma funkcjonalnościami programu, wprowadził historię zmian oraz pozwolił na bezproblemową ich synchronizację między członkami zespołu. Wykorzystano popularną platformę *GitHub*, która dostarcza gotowego rozwiązania wraz z serwerem do przechowywania danych.

[https://github.com/Kondix/UHDPlayer]

1. Analiza systemu - Jagielski

Analiza systemu została przeprowadzona przed implementacją rozwiązania. Podczas tej fazy stworzono wymagania, które będą stanowiły o tym kiedy projekt można uznać za zakończony. Ich definicja była konieczna, aby ukierunkować tok prac.

Lista wymagań projektowych:

* Odtwarzanie nieskompresowanych sekwencji wideo o wybranych parametrach
* Utrzymanie jakości filmu przez cały jego czas trwania
* Możliwość przeprowadzenia różnych scenariuszy testowych
* Możliwość wygenerowania filmów o zadanych parametrach z filmu źródłowego
* Automatyczna prezentacja filmów osobie testowanej
* Zbieranie danych o ocenie filmu osoby testowanej

Wszystkie powyższe wymagania są przedstawione z punktu widzenia nietechnicznego, co pozwala na ich zrozumienie również osobie, która nie posiada wiedzy specjalistycznej.

Testy subiektywne przeprowadzane są z użyciem nieskompresowanych sekwencji wideo, w celu wyeliminowania wpływu procesu dekodowania na otrzymany obraz, a co za tym idzie otrzymania lepszej jakości wyników samego testu. Różne implementacje tego samego sposobu dekodowania mogły by spowodować, że przeprowadzenie teoretycznie tego samego testu na dwóch różnych, niezależnych maszynach, otrzyma wyniki dla rożnych obrazów widzianych przez osoby testujące.

Odtwarzacz takich sekwencji oraz maszyna, na której jest uruchomiony powinny być skonfigurowane tak, aby film w zadanej jakości odtwarzał się płynnie, wyświetlane były wszystkie klatki oraz ich kolejność i jakość została zachowana. Bardzo ważnym elementem analizy było ustalenie zbioru jakości, które odtwarzacz będzie mógł odtworzyć. Ograniczeniem w tej kwestii pozostaje dostępny sprzęt elektroniczny, budżet na zakup nowych części i rozwiązań oraz sama implementacja odtwarzacza. O tym czy zapewniona została odpowiednia jakość filmów decydował zestaw testów wydajnościowych, o których więcej pojawi się w dalszej części pracy.

Przeprowadzenie testów powinno odbywać się z jak najmniejszym udziałem człowieka, tj. po uprzednim przygotowaniu scenariuszy kolejne uruchomienia testu powinny być zautomatyzowane, a sama osoba testująca powinna być „prowadzona za rękę” przez cała długość jego trwania. Tester musi mieć dodatkowo możliwość interakcji z systemem w celu podawana kolejnych odpowiedzi na zadane pytania.

Kluczową dla pracy magisterskiej jest kwestia porównania między sobą wybranych scenariuszy testowych. Dlatego wymagana jest również możliwość ich przeprowadzenia, a co za tym idzie skonfigurowania. Konfiguracja i struktura plików nie powinna być dostępna dla testera ze względu na przykład na ich nazwy, które podczas generacji przez skrypt zawierają sugestie, w jakiej jakości zapisany jest dany film.

Jakości poszczególnych filmów testowych powinny być definiowane przez administratora lub osobę za to odpowiedzialną, a nie być narzucone odgórnie przez skrypt, ze względu na różnice pomiędzy koncepcjami testowymi. Niektóre scenariusze mogą zakładać użycie materiałów o nieznaczne zmienionych parametrach, inne korzystać z takich, których parametry różnią się znacznie. Generacja filmów powinna odbywać się automatycznie, korzystając z pliku źródłowego oraz informacji o pożądanej jakości wynikowego filmu. Należy zaznaczyć, że skrypt generujący został stworzony tylko na potrzeby wewnętrzne projektu w celu zaspokojenia zapotrzebowania na dane wejściowe systemu. Użytkownik końcowy powinien zapewniać jakość sekwencji wideo we własnym zakresie.

1. Generacja danych wejściowych - Jagielski
2. Szczegóły implementacyjne - Jagielski

Kolejnym krokiem po szczegółowej analizie systemu było zaplanowanie architektury kodu. Naturalnym początkiem było wydzielenie części odpowiedzialnej za przetwarzanie i wyświetlanie nieskompresowanych sekwencji wideo. Po ukończeniu odtwarzacza ruszyły prace, mające na celu stworzenie warstwy widocznej przez użytkownika, wraz z funkcją przeprowadzenia wybranych scenariuszy testowych.

Biblioteka *libVLC* zapewnia dostęp do wielu gotowych metod obsługi materiału wideo, nie posiada jednak bezpośrednich rozwiązań do odtwarzania nieskompresowanych sekwencji wideo. W przypadku sekwencji skompresowanych jednym ze standardowych sposobów odtwarzania jest wczytywanie nie całych klatek, a tylko zmian zachodzących między kolejnymi dwoma poprzez dekompresje. Takie podejście pozwala znacząco ograniczyć rozmiar wczytywanych do pamięci danych. Mniejsza ilość danych pozwala na zwiększenie maksymalnej jakości filmu odtwarzanego na tym samym komputerze względem programu ładującego każdą klatkę od nowa.

W przypadku inkrementacyjnego ładowania zmian w jednym z ostatnich kroków procesu otrzymywana jest klatka reprezentowana w ten sam sposób, co nieskompresowana. Dzięki tej obserwacji, możliwe jest wstrzyknięcie wczytanej nieskompresowanej klatki do standardowego procesu odtwarzania filmu i jej wyświetlenie z użyciem biblioteki *libVLC*. Szczegółowy opis implementacji tego sposobu zostanie przedstawiony w dalszej części rozdziału.

* 1. Podstawy działania programu - Jagielski

W tej części opisany zostanie sposób na stworzenie najprostszego programu z użyciem biblioteki *libVLC*. Program otworzy plik wideo dostępnego na dysku lokalnym, lub zasobie sieciowym a następnie odtworzy zdefiniowaną wcześniej długość filmu, np. 10 sekund. Program nie posiada wiedzy o długości trwania dostępnego materiału, dlatego w przypadku gdy film jest krótszy od zadanej wartości program przerwie swoje działanie zwracając odpowiedni kod błędu. Później zostaną również opisane podstawowe typy dostarczane przez bibliotekę.

Pierwszym, podstawowym typem biblioteki jest *libvlc\_instance\_t*, struktura reprezentująca całą instancję *libVLC*. Obiekt tego typu inicjalizowany jest przez funkcję *libvlc\_new*, zwracającą typ wskaźnika na wyżej wymienioną strukturę lub *NULL* gdy podczas jej wykonywania wystąpił błąd. Funkcja inicjalizująca przyjmuje dwa parametry, podobne do tych z funkcji głównej programu. Pierwszy z nich *argc* oczekuje danych w formacie całkowitoliczbowym z ilością argumentów przekazanych w postaci stałego ciągu znaków *argv*. Przekazywane argumenty są bardzo ważne z punktu widzenia pracy magisterskiej. To dzięki nim możliwe było sterowanie odtwarzanym filmem w postaci nieskompresowanej, co stanowiło kluczowy element podczas jej realizacji. Szczegółowy opis przekazanych argumentów zostanie przedstawiony wraz z interfejsem do zarządzania pamięcią w dalszej części rozdziału. Na potrzeby prostego programu przyjęto, że nie będą przekazywane żadne argumenty, a więc parametry będą ustawione kolejno na 0 i *NULL*.

Kolejnym ważnym typem dostarczanym przez bibliotekę jest *libvlc\_media\_t* odpowiadającym za reprezentację odtwarzanych mediów. Przez media rozumiane są dowolne, wspierane przez bibliotekę dane w odpowiednim formacie. Mogą to być przykładowo filmy znajdujące się na dysku twardym komputera lub sekwencje wideo dostępne w Internecie. W zależności od źródła danych biblioteka dostarcza różne funkcje, pozwalające na stworzenie obiektu wyżej wymienionego typu.

Lista dostępnych funkcji inicjujących obiekt typu *libvlc\_media\_t.* Każda z poniższych funkcji jako pierwszy argument przyjmuje wskaźnik na obiekt instancji *libVLC* - *llibvlc\_instance\_t.*

* *libvlc\_media\_new\_location* – pozwala na obsługę mediów dostępnych zarówno  
   w sieci jak i na lokalnej maszynie. Jako drugi parametr przyjmuje ścieżkę do żądanego zasobu (np. odnośnik protokołu http) w postaci ciągu znaków.
* *libvlc\_media\_new\_path* – służy do obsługi mediów dostępnych w lokalnym systemie plików. Jako drugi parametr przyjmuje ścieżkę do odtwarzanego pliku w postaci ciągu znaków.
* *libvlc\_media\_new\_fd* – umożliwia obsługę mediów z otwartego wcześniej deskryptora pliku. Jako drugi parametr przyjmuje otwarty deskryptor pliku w formacie całkowitoliczbowym.

Ostatnim z omawianych typów jest *libvlc\_media\_player\_t* czyli typ reprezentujący odtwarzacz mediów. Pozwala on na odtworzenie w oknie wybranych mediów. W programie podstawowym nie będzie on wymagał żadnej wstępnej konfiguracji. Tworzony jest z użyciem funkcji *libvlc\_media\_player\_new\_from\_media*, która jako parametr przyjmuje wskaźnik na obiekt reprezentujący media, oraz zwraca wskaźnik na nowo powstały obiekt lub *NULL* w przypadku niepowodzenia. Po wywołaniu funkcji, a więc zainicjowaniu obiektu odtwarzacza mediów, same media nie są już wymagane. Usunięcia mediów można dokonać z użyciem funkcji *libvlc\_media\_release*, przyjmującej jako parametr wskaźnik na usuwane media.

Kolejnym krokiem programu jest uruchomienie odtwarzacza mediów. Realizowane jest to za pomocą funkcji *libvlc\_media\_player\_play*, która jako parametr przyjmuje wskaźnik na obiekt odtwarzacza mediów. W tym momencie następuje uruchomienie wyświetlania filmu. Program posiada podstawowe dane na temat filmu, jednak nie wie jak długo trwa. Dlatego w kolejnym kroku wołana jest funkcja *sleep*, która powoduje zatrzymanie wykonywania wątku, z którego została wywołana na określony w parametrze czas, wyrażony w sekundach. Problem pojawia się, gdy długość filmu nie przekracza wartości na którą wątek został uśpiony. Taki scenariusz powoduje nagłe zatrzymanie wykonywania programu, który kończy się zwracając odpowiedni kod błędu. Rozwiązanie zostanie przedstawione w dalszej części rozdziału, wraz z opisem bardziej zaawansowanego kodu.

Jeśli film okazał się dłuższy od zadanej wartości uśpienia, program wykonuje się dalej. Następnym etapem jest zatrzymanie odtwarzacza mediów funkcją *libvlc\_media\_player\_stop*. Podobnie jak rozpoczęcie odtwarzania, jako parametr przyjmuje wskaźnik na odtwarzacz mediów. Funkcja *libvlc\_media\_player\_release* zwalnia obiekt odtwarzacza mediów. Po jej wywołaniu obiekt nie nadaje się do ponownego użycia. Konieczna jest jego ponowna inicjalizacja.

Ostatnim krokiem jest zwolnienie obiektu instancji, poprzez wywołanie *libvlc\_release*, przyjmującej jako parametr wskaźnika na zwalnianą instancję.

  
Rysunek 1. Widok ekranu z kodem źródłowym programu podstawowego.

[https://wiki.videolan.org/LibVLC\_Tutorial/]  
[https://www.videolan.org/developers/vlc/doc/doxygen/html/]

Na rysunku 1 zamieszczono cały kod omawianego programu. W trakcie jego pisania korzystano ze środowiska *CLion*, które opiera się na *cmake*. Jest to narzędzie skryptowe o otwartej licencji pozwalające zbudować (ang. *build*) dany program w różnych systemach operacyjnych, w tym używanego w pracy *Linuxa*. Jego działanie opiera się na sekwencyjnym wykonywaniu kolejnych dyrektyw i tworzeniu plików z regułami, wykorzystywanymi później przez kompilator. Aby zapewnić dostęp do wszystkich potrzebnych źródeł i bibliotek stworzono listę poleceń przedstawioną na   
rysunku 2.

  
Rysunek 2. Widok ekranu z listą dyrektyw narzędzia *cmake*.

Polecenie *cmake\_minimum\_required* określa jaka najstarsza wersja *cmake* może być użyta do wykonania skryptu. Dyrektywa *set* pozwala na ustawienie zmiennej, której nazwa jest przekazywana jako pierwszy parametr, na wartość przekazywaną jako drugi. Może to być zarówno zmienna środowiskowa jak i lokalna. Komenda *project* ustawia nazwę projektu, w tym przypadku na „Odtwarzacz”. Od tego momentu nazwa jest kojarzona z konkretnym projektem i jej użycie spowoduje odniesienie się do niego. *Add\_executable* dodaje pliki źródłowe do projektu w którym mają zostać zbudowane. W tym przypadku dodany jest plik *main.cpp* zawierający wcześniej opisywany program. W *cmake* aby odnieść się do wartości przechowywanej przez zmienna należy użyć składni: nazwa zmiennej w nawiasach klamrowych, poprzedzonych przez znak dolara. Polecenie *LINK\_DIRECOTIES* specyfikuje ścieżkę w której konsolidator (ang. *linker*) powinien szukać bibliotek, z których powstanie plik wykonywalny. Ścieżka do biblioteki *libVLC* w przykładzie jest przekazywana przez odwołanie do wartości zmiennej *VLC\_LIB*. Ostatecznie dyrektywa *TARGET\_LINK\_LIBRARIES* specyfikuje które konkretnie biblioteki mają być użyte przez konsolidator podczas budowania danego projektu. Jako parametr przyjmuje też flagi z jakimi proces ma zostać przeprowadzony.

* 1. Wstrzykiwanie klatek do pamięci - Jagielski

Jednym z największych problemów dotyczących tej części pracy było napisanie odtwarzacza w taki sposób, aby możliwe było odtwarzanie nieskompresowanych sekwencji wideo. Z powodu braku dostępu do materiałów wyjaśniających sposób w jaki można to osiągnąć zdecydowano się szczegółowo opisać problem oraz sposoby jego rozwiązania. W trakcie rozwoju odtwarzacza pojawiło się wiele ograniczeń które również zostaną opisane w poniższym rozdziale.

Podczas prowadzenia analizy istniejących rozwiązań dla tego typu odtwarzaczy natknięto się na wątek dotyczący przesyłania obrazu z kamery za pomocą Internetu oraz protokołu UDP. Autor pytania chciał odbierać dane i przekazywać je programowi napisanemu z użyciem *libVLC* z pomocą interfejsu *imem*. Moduł interfejsu nie cieszy się dużą popularnością, nie jest również opisany w dokumentacji dostarczonej przez autorów *libVLC*. Największym źródłem informacji o nim zdobyto podczas analizy statycznej kodu źródłowego, który na szczęście jest dostępny na zasadach otwartej licencji. Korzystano również z informacji dostarczonych przez użytkownika Arkaid z wątku na forum videolan.

[https://forum.videolan.org/viewtopic.php?t=93842]

Bazując na programie podstawowym opisanym w poprzednim rozdziale stworzono kod z wykorzystaniem interfejsu *imem*. Jest to moduł pozwalający na dostęp do pamięci komputera i użycia danych przechowywanych bezpośrednio w niej. Sterowanie modułem odbywa się dzięki zestawowi komend, które zostaną omówione później.

  
Rysunek 3. Widok ekranu z konstruktorem parametrycznym klasy *Controler*.

Pierwszą zmianą jest ustawienie źródła mediów na omawiany interfejs. Deklaracja źródła mediów odbywa się poprzez omówioną w poprzednim rozdziale funkcję *libvlc\_media\_new\_location*. Jako drugi parametr przekazany został ciąg znaków „imem://”. Przykład implementacji znajduje się na rysunku 3. Użycie obiektu reprezentującego media, w którym użyto modułu *imem*, w programie podstawowym zakończy się jednak niepowodzeniem. Wymagane jest wprowadzenie kilku kolejnych zmian implementacyjnych aby zmienić ten stan rzeczy.

Podstawą działania modułu *imem* są funkcje nazwane przywołaniami (ang. *callback function).* Odpowiadają one za obsługę każdej klatki filmu. Wyodrębnione zostały dwie funkcje tego typu. Pierwsza odpowiedzialna jest za alokację pamięci oraz ustawienie szeregu zmiennych odpowiedzialnych za konkretne parametry wyświetlanej klatki. Druga odpowiada za ewentualne zwolnienie zaalokowanej pamięci. Oczywiście obie funkcje można edytować do swoich własnych potrzeb.

  
Rysunek 4. Widok ekranu z kodem źródłowym interfejsu *imem*. Definicje typów funkcji przywołań.

Na rysunku 4 przedstawiono definicje typów funkcji przywołań. Jest to jedno z ograniczeń, które należy uwzględnić podczas modyfikacji którejkolwiek z funkcji. Przekazane parametry funkcji muszą pokrywać się z tymi, które znajdują się w pliku imem.c dostarczanym wraz z biblioteką *libVLC*. Omówione zostaną tylko parametry przekazywane w oryginalnej, niezmodyfikowanej wersji biblioteki.

  
Rysunek 5. Widok ekranu z kodem odpowiedzialnym za ustawienie adresów funkcji przywołań.

Jak zostało już wspomniane interfejs *imem* konfigurowany jest przez zestaw komend. Komendy te przekazywane są do programu z użyciem omówionej już funkcji *libvlc\_new.* Widoczna na rysunku 5 metoda *AddOption* obiektu *optionsHandler* odpowiada za agregację kolejnych argumentów, które ostatecznie przekazywane są do funkcji *libvlc\_new.* Od tego momentu użycie obiektu instancji *llibvlc\_instance\_t* będzie wiązać się z użyciem stworzonej konfiguracji.

Argumenty „--imem-get” oraz „--imem-release” wymagają ustawienia na adresy odpowiadających im funkcji. Odbywa się to dzięki funkcji *sprintf*, która potrafi wpisać do wskazanej zmiennej ciąg znaków. Konwersja nazwy funkcji na jej adres odbywa się dzięki składni: litera ‘p’ poprzedzona znakiem kratki.

Lista parametrów alokującej funkcji przywołań:

* data – pierwszy z parametrów, odpowiadający za dane w formacie zdefiniowanym przez użytkownika. Adres przekazywany jest za pomocą argumentu „--imem-data”. Adres może zawierać dowolny typ obiektu. Dzieje się tak za sprawą typu *void\** czyli wskaźnika do obiektu o nieznanym typie. Jedynie jawne rzutowanie wskaźnika na inny typ jest bezpieczne, ponieważ w innym przypadku kompilator nie wie na jaki typ naprawdę wskazuje. Przykład przedstawiony jest na rysunku 6, gdzie *data* konwertowana jest na typ *FramesHandler\**, czyli wskaźnik na obiekt zdefiniowany przez użytkownika.

[Stroustrup]

* cookie – ciasteczko (ang. *cookie)*, zdefiniowany przez użytkownika ciąg znaków. Może służyć na przykład do wyświetlania napisów w danej klatce. Adres przekazywany jest za pomocą argumentu „--imem-cookie”. W przykładzie przedstawionym na rysunku 6 ciasteczko jest pomijane ze względu na brak konieczności jego użycia w programie.
* dts - znacznik czasowy definiujący moment zdekodowania klatki (ang. *decode timestamp)*. Wyrażony w mikrosekundach obliczanych według wzoru:  
    
  W przykładzie przedstawionym na rysunku 6 ustawiany wraz ze zmienna p*ts* na tą samą wartość, jednak może się ona różnić w zależności od użytego sposobu dekodowania. Wyświetlanie nieskompresowanych sekwencji wideo nie wymaga procesu dekompresji, co pozwala na równość dwóch współczynników.
* pts - znacznik czasowy definiujący moment wyświetlenia klatki (ang. *presentation timestamp)*. Wyrażony w mikrosekundach obliczanych według wzoru:
* flags - parametr typu *unsigned\** czyli wskaźnik na obiekt całkowitoliczbowy bez znaku. W przykładzie przedstawionym na rysunku 6 nie został użyty.
* bufferSize - parametr typu *size\_t\** czyli wskaźnik na obiekt typu całkowitoliczbowego bez znaku, który może przechowywać rozmiar każdego obiektu, wyrażony w bajtach. W tym przypadku przekazywany jest rozmiar następnego parametru, czyli bufora danych. Przekazany błędnie spowoduje obcięcie wartościowych dla danej klatki danych lub wczytanie zbyt dużej ich ilości wraz ze zbędnymi, trudnymi do określenia bitami. W każdym z wyżej wymienionych przypadków klatka zostanie błędnie wyświetlona lub cały program zakończy działanie zwracając odpowiedni kod błędu.
* buffer - najważniejszy parametr odpowiedzialny za dostarczenie do interfejsu *imem* danych o klatce. Wskaźnik ustawiany jest na podany adres, a następnie brane jest tyle bajtów danych, ile wynosi zmienna bufferSize. W tym przypadku ważne jest, aby pamięć była zaalokowana w jednym bloku, to znaczy wszystkie dane klatki znajdowały się na sąsiadujących adresach w pamięci. Jeśli wywołana została funkcja *new*, aby uniknąć sytuacji gdzie zaalokowana pamięć jest nie zostaje nigdy zwolniona, w zwalniającej funkcji przywołań należy wywołać funkcję *delete*. W przykładzie przedstawionym na rysunku 6 nie alokowana jest żadna pamięć, dlatego nie ma potrzeby, aby potem ją zwalniać.

  
Rysunek 6. Widok ekranu z kodem alokującej funkcji przywołań.

Na rysunku 6 przedstawiono implementację alokującej funkcji przywołań, którą wykonano dla potrzeb pracy magisterskiej. Funkcja *MyImemGetCallback* zaczyna się i kończy operacjami na obiekcie typu *std::mutex*. Obiekty tej klasy są używane do reprezentacji wyłącznego dostępu do jakiegoś zasobu. W tym przypadku służą do ochrony przed zjawiskiem wyścigu oraz do synchronizacji dostępu do danych, w programach wielowątkowych. Wątki reprezentują obiekty typu *std::thread*. Jako parametr konstruktor klasy *std::thread* przyjmuje nazwę funkcji którą ma wykonać. W odtwarzaczu stworzonym w ramach pracy magisterskiej wątki używane są zarówno przez wewnętrzne funkcje biblioteki *libVLC* jak i podczas ładowania kolejnych klatek filmu z dysku do pamięci ram. Wczytywanie odbywa się w konfigurowalnej przez użytkownika liczbie wątków. Aby zapewnić ich synchronizację należy skorzystać z wymienionych wcześniej obiektów klasy *std::mutex.* W tym przypadku wątki współdzielą jeden obiekt tej klasy, kolejno rezerwując sobie wyłączny dostęp do zasobów dzięki użyciu metody *lock*, a następnie zwalniając go korzystając z metody *unlock.*

Rozmiar bufora określany jest na podstawie stałych, definiowanych na początku działania algorytmu. Są to kolejno wysokość i szerokość odtwarzanej klatki wyrażona w pikselach oraz informacja ile bitów reprezentuje każdy piksel. Na potrzeby algorytmu otrzymana liczba bitów przeliczana jest na bajty, poprzez proste dzielenie przez osiem. Wskaźnik *bufferSize* ustawiany jest na adres pamięci, z określoną wartością rozmiaru klatki. Jednym z założeń odtwarzacza jest, że wszystkie klatki w danym filmie mają ten sam rozmiar. Jeśli jakakolwiek klatka została już wczytana do kolejki, jej adres zostaje przypisany do zmiennej *buffer.* Następnie ustawiane są omówione już zmienne *pts* i *dts*.

  
Rysunek 7. Widok ekranu z kodem zwalniającej funkcji przywołań.

Zwalniająca funkcja przywołań przyjmuje ograniczoną ilość parametrów względem funkcji alokującej. W większości nie będą one omawiane, ponieważ ich przeznaczenie jest takie samo w obu typach funkcji przywołań. Podkreślić należy znaczenie parametru *data*. Jeśli funkcja alokująca zaalokowała jakiś blok pamięci, musi zostać on zwolniony w wyżej wymienionej funkcji.

Pierwszym zadaniem funkcji jest zarezerwowanie dostępu do zasobów w omówionym już mechanizmie opierającym się na funkcjonalnościach obiektu klasy *std::mutex*. Kolejnym krokiem jest jawne rzutowanie na typ *FramesHandler\** opisane wcześniej w tym rozdziale. Po nim następuje logika odpowiedzialna za usunięcie klatki filmu, która została już wyświetlona. Element usuwany jest z kolejki aby zwolnić miejsce w pamięci na nowo wczytane klatki. Następnie odbywa się sprawdzenie, czy ostatnia wyświetlona klatka nie była ostatnią z sekwencji filmowej. Jeśli tak, ustawiana jest flaga *DisplayHandler::m\_bDone*, odpowiedzialna za zakończenie odtwarzania filmu w sposób bezpieczny dla działania całego odtwarzacza.

* 1. Opis klas odtwarzacza - Jagielski

W poniższym rozdziale opisane zostaną klasy użyte w implementacji odtwarzacza nieskompresowanych sekwencji wideo. Przedstawiony zostanie diagram klas UML (angielski akronim rozwijany: *Unified Modeling Language* - czyli zunifikowany język modelowania) opisujący zależności między poszczególnymi elementami implementacji.

Rysunek 8. Uproszczony diagram klas odtwarzacza UML.

Na rysunku 8 przedstawiono hierarchię klas autorskiego rozwiązania dla odtwarzacza. Na diagramie pokazany jest uproszczony diagram klas. Ze względu na czytelność przedstawiono tylko nazwy klas. Szczegóły dotyczące idei stworzenia danych klas oraz opis ich funkcjonalności zostanie omówione w dalszej części rozdziału. Sercem całego systemu jest klasa *Controler*. Cztery różne typy obiektów obsługujących kolejno wątki, surowe dane, wyświetlanie oraz dane klatki zapewniają pokrycie wszystkich koniecznych funkcjonalności. W tym rozdziale omówione zostaną tylko klasy odtwarzacza. Elementy architektury graficznego interfejsu użytkownika oraz moduł testów zostaną omówione później.

  
Rysunek 9. Diagram UML klasy *Controler*.

*Controler* czyli najważniejsza klasa w odtwarzaczu z punktu widzenia użytkownika. Inicjuje ona w swoich konstruktorach obiekty mediów i instancji dostarczanych przez bibliotekę *libVLC* i przechowuje wskaźniki na otrzymane obiekty w swoich polach (ang. *fields).* Posiada również wskaźnik na obiekt typu *DisplayHandler*, inicjowany w konstruktorach. Jedyną metodą klasy *Controler* jest *Run*, która jako parametry przyjmuje dwie klasy pomocnicze, obsługujące klatki oraz surowe dane wczytywane z pliku. Uruchamia ona proces odtwarzania filmu. Ideą stworzenia tej klasy było uzyskanie narzędzia pozwalającego na ukrycie implementacji przed użytkownikiem oraz prosty sposób uruchomienia programu. Takie podejście pozwoliło na bardziej efektywny podział prac w zespole programistycznym oraz na zrównoleglenie rozwoju poszczególnych części funkcjonalności. Zaznaczyć trzeba, że jest to klasa kontrolująca wyświetlanie, a więc wczytywanie klatek do pamięci musi zostać wykonane przez inny element programu.

  
Rysunek 10. Diagram UML klasy *ThreadsHandler*.

*ThreadsHandler* to klasa zarządzające wątkami. Jedynym jej polem jest wektor wątków, do którego dodawane są wszystkie nowo stworzone, a usuwane te już nie aktywne. Na rysunku 10 przedstawiono diagram UML tej klasy. Pierwszą z metod tej klasy jest funkcja *AddThread* dodająca nowy wątek do wektora. Powinna być wywoływana zawsze, gdy program utworzy nowy wątek, który powinien być obsługiwany przez obiekt klasy *ThreadsHandler.* Metoda *CreateNFramesGetterThreads* tworzy ilość wątków konfigurowalną przez jeden z jej parametrów. Wszystkie stworzone w metodzie wątki korzystają z innej metody - *GetFrameToBuffer*. Podlega ona zasadom przydziału zasobów za pomocą omówionych już elementów *std::mutex.* Takie połączenie zapewnia określoną, konfigurowalną liczbę klatek które znajdują się w kolejce odtwarzacza. Jeśli jakakolwiek z nich zostanie wyświetlona jeden z wątków roboczych wczyta nową z pliku, a następnie doda ją do kolejki.

Istnieje również opcja wczytania całego filmu do pamięci RAM, korzystając z metody *PreloadCache.* Podejście to ma swoje wady i zalety. Główną wadą jest konieczność posiadanie dużej ilości wolnej pamięci RAM. Nieskompresowane dane sekwencji filmowej zajmują bardzo dużo miejsca. Jedną z zalet jest możliwość ukrycia ograniczeń sprzętowych takich jak wolny odczyt z dysku. Odtwarzacz wczytuje całą sekwencję, jednocześnie jej nie wyświetlając. Uruchomienie filmu rozpoczyna się dopiero wtedy, gdy wszystkie dane zostaną przeniesione do RAM-u. Eliminuje to ewentualne zacięcia filmu podczas jego wyświetlania, zapewnia, że wszystkie klatki zostaną wczytane oraz eliminuje zapobiega zjawiskom wyścigu. Największym argumentem za wczytywaniem filmu podczas jego odtwarzania jest możliwość odtwarzania w taki sposób sekwencji zajmujących więcej miejsca niż wynosi pojemność RAM-u. Z drugiej strony ograniczona przepustowość wczytywania z dysku nie pozwala na uruchamianie filmów wysokiej jakości w opisywany sposób.

Prawdopodobnie najlepszym rozwiązaniem byłoby połączenie obu rozwiązań w jedno. Część filmu mogłaby być ładowana jeszcze przed jego uruchomieniem, a następnie w trakcie wyświetlania wczytywane były by klatki z jego dalszej części. Należałoby stworzyć również algorytm precyzujący największą długość sekwencji przy zadanej jakości. W innym przypadku mogłyby pojawić się problemy opisane w ostatnim akapicie. Mimo wszystko dla wysokich jakości nieskompresowanych wideo najmniej wydajnym ogniwem jest sprzęt elektroniczny. Wraz z jego ulepszeniem możliwe jest osiągnięcie lepszych odtwarzanych jakości.

Metody *StopPlayBack* oraz *StopPlayBackThread* mają za zadanie zamknąć okna odtwarzacza po skończeniu wyświetlania filmu.

  
Rysunek 11. Diagram UML klasy *FramesHandler.*

*FramesHandler* to klasa stworzona z myślą o łatwym zarządzaniu zbiorem klatek wczytanych do pamięci podręcznej. Posiada kolejkę, przechowującą wszystkie klatki w odtwarzaczu. Zapewnia również zestaw operacji na kolejce, pozwalających wydobyć z niej odpowiednią klatkę, zbiór klatek lub określić ich ilość. Zdecydowano się użyć kolejki typu *std::deque* czyli takiej, która gwarantuje dostęp do pierwszego i ostatniego elementu. Należy zwrócić uwagę na metody odpowiedzialne za usuwanie klatek z kolejki, czyli *ClearFirstFrame* oraz *ClearFrames*. Są one ważnym elementem algorytmu, pozwalają zwalniać zasoby, aby nowe klatki filmu mogły zostać wczytane. Obiekty tego typu pojawiają się w bardzo wielu miejscach kodu, dlatego dalsze zwiększanie funkcjonalności zawartych w tej klasie powinna być dobrze przemyślana. Wprowadzenie nadmiarowych, niepotrzebnych w klasie *FramesHandler* pól czy metod może prowadzić do nadmiernego skomplikowania kodu.

  
Rysunek 12. Diagram aktywności UML przedstawiający proces wczytywania filmu do odtwarzacza.

Aby lepiej zrozumieć algorytm ładowania klatek do odtwarzacza przygotowano diagram aktywności UML. Algorytm przewiduje trzy możliwe zakończenia. Pierwszym z nich jest przerwanie działania programu w wyniku jakiejś nieprawidłowości. Przykładowo może to być brak miejsca w pamięci RAM lub nieistniejący plik z danymi filmu.

Pozostałe dwa zależą od wybranego trybu ładowania klatek filmu. Przedstawiony na rysunku 12 diagram opisuje aktywności do momentu, w którym rozpoczyna się odtwarzanie filmu. Dlatego wynikami jego działania może być załadowanie całego filmu, lub tylko jego części. W przypadku gdy ładowany jest cały, podczas jego odtwarzania nie następuje już żaden odczyt filmu z dysku. Odwrotnie dzieje się gdy wybrana zostanie alternatywa. W momencie wczytania zadanej ilości klatek do kolejki następuje uruchomienie odtwarzania filmu. Zdefiniowana ilość wątków roboczych odpowiada za utrzymanie odpowiedniej ilości klatek w kolejce.

  
Rysunek 13. Diagram UML klasy *RawDataHandler.*

*RawDataHandler* to klasa, której zadaniem jest zarządzanie strumieniem danych z otwartego pliku. Jej diagram UML został przedstawiony na rysunku 13. Posiada informacje o wielkości klatki, dlatego jest w stanie odczytywać odpowiednie porcje danych i przekazywać je dalej. Posiada również wskaźnik na obiekt typu *FileOpener.* Odpowiedzialny jest on za sprawdzenie, czy plik istnieje, a następnie jego otwarcie. Przyjmuje tylko jeden parametr - ścieżkę do żądanego pliku. Głównym użytym narzędziem w omawianych dwóch klasach jest *std::fstream.* Jest to klasa dostępna w ramach biblioteki standardowej pozwalająca na czytanie oraz pisanie z/do pliku.

  
Rysunek 14. Diagram UML klasy *DisplayHandler*.

Ostatnią omawianą klasą jest *DisplayHandler.* Jej diagram UML przedstawiony jest na rysunku 14. Podobieństwo do omówionej już klasy *Controler* jest nie przypadkowe. To właśnie klasę *Controler* przysłania. *DisplayHandler* oprócz znanych już pól posiada wskaźnik na obiekt typu *libvlc\_event\_manager\_t*. Jest to klasa zarządzająca zdarzeniami, które zostaną omówione w następnym rozdziale. Głowną metodą jest *Play,* odpowiedzialna za uruchomienie odtwarzania filmu.

  
Rysunek 15. Diagram aktywności UML przedstawiający proces odtwarzania filmu.

Diagram na rysunku 15 prezentuje proces odtwarzania filmu. Niezależnie od podjętych decyzji bazuje na operacjach wykonywanych wewnątrz omówionych już funkcji wywołań. Widoczny jest podział na dwie główne ścieżki. Decyzja zależy od tego, czy wybrany jest tryb wczytywania całego filmu do pamięci, czy jego ładowania podczas odtwarzania filmu. Jeśli coś pójdzie nie tak podczas wykonywania programu, zakończy się on z odpowiednim kodem błędu. Na diagramie przedstawiono taką sytuację, gdy brakuje pamięci RAM, aby wczytać do niego nowe dane. Pojawia się ona tylko w jednej ścieżce, ponieważ podczas wczytywania całego filmu do pamięci taka sytuacja eliminowana jest już wcześniej (Rys 12.).

* 1. System zdarzeń