

**Dokumentacja
Interfejsy Użytkownika i Biblioteki
Graficzne**

Przejścia między obrazami



Wydział Fizyki i Informatyki Stosowanej
Martyna Lach
Julia Mucha
Krystsina Mironenka
Mikhail Shupliakou
03.01.2026

Spis treści

1 Opis projektu	4
2 Podział pracy i analiza czasowa	5
3 Założenia wstępne przyjęte w realizacji projektu	6
3.1 Decyzja o wyborze narzędzi programistycznych	6
4 Analiza projektu	7
4.1 Specyfikacja danych wejściowych	7
4.2 Opis oczekiwanych danych wyjściowych	7
4.3 Zdefiniowanie struktur danych	8
4.4 Specyfikacja interfejsu użytkownika	8
4.5 Wyodrębnienie i zdefiniowanie zadań	10
4.6 Decyzja o wyborze narzędzi programistycznych	10
5 Opracowanie i opis niezbędnych algorytmów	10
5.1 Podstawy matematyczne animacji i sterowanie czasem	10
5.1.1 Zmienna znormalizowana (Progress)	11
5.1.2 Interpolacja Liniowa (LERP)	11
5.1.3 Zarządzanie stanem i determinizm renderowania (State Reset)	12
5.2 Algorytmy podstawowych przekształceń afinicznych 2D	13
5.2.1 Translacja (Przesunięcie liniowe)	13
5.2.2 Skalowanie względem punktu centralnego (Zoom)	13
5.3 Algorytmy przetwarzania rastrowego na CPU (Blur i Luma)	14
5.3.1 Algorytm Luma Wipe (Przejście jasnością)	14
5.3.2 Algorytm Separowanego Rozmycia Pudełkowego (Box Blur) .	15
5.4 Opracowanie własnych algorytmów 3D i rzutowania perspektywicznego	15
5.4.1 Matematyka obrotu (Transformacje w przestrzeni euklidesowej)	16
5.4.2 Rzutowanie perspektywiczne (Perspektywa jednoskrotna) .	16
5.4.3 Rozwiązywanie problemu teksturowania (Dyskretyzacja powierzchni)	17
5.5 Deformacje geometryczne i ruch orbitalny (Ring Transition)	17
5.5.1 Parametryczne równanie ruchu po okręgu	17
5.5.2 Rzutowanie i Skalowanie Perspektywiczne	18
5.5.3 Implementacja (Dual Rendering)	18
5.5.4 Rozwiązywanie widoczności (Depth Sorting)	18
5.6 Algorytm Fly Away (Efekt „Odlotu”)	19
5.7 Techniczne aspekty implementacji i struktura danych	19
5.7.1 Bezpośrednia manipulacja geometrią (Vertex Arrays)	19
5.7.2 Rendering pozaekranowy	20
5.7.3 Potok eksportu	20

6 Kodowanie	20
6.1 Funkcje pomocnicze (API Systemu Windows)	22
6.2 Główna logika renderowania	22
6.3 Pętla główna i interfejs (GUI)	22
6.4 Kluczowe zmienne i struktury danych	23
6.5 Wykres przepływu danych.	23
6.6 Graf algorytmu renderującego	24
6.7 Implementacja efektów rastrowych (Blur i Luma)	25
6.7.1 Funkcja ApplyCpuLumaWipeOptimized	26
6.7.2 Funkcja ApplyCpuBlurOptimized	27
6.7.3 Obsługa w pętli renderującej (Case 11 i 14)	28
7 Testowanie	29
7.1 Zestawy danych testowych	29
7.2 Wyniki testów wydajnościowych	29
7.3 Analiza wyników	30
7.4 Wyniki testów	31
8 Wdrożenie, raport i wnioski	34
8.1 Uruchomienie z niezależnymi danymi	34
8.2 Raport z wykonania	34
8.3 Wnioski i możliwości rozwoju	35
9 Źródła	36

1 Opis projektu

Projekt ma na celu stworzenie programu umożliwiającego generowanie animowanych przejść pomiędzy dwoma obrazami bitmapowymi o identycznym rozmiarze. Program pozwala użytkownikowi na interaktywne przeglądanie efektu przejścia przy użyciu suwaka, który reprezentuje „linię czasu” animacji. Użytkownik może wybrać liczbę klatek, z których ma się składać animacja, a po jej wygenerowaniu program umożliwia zapisanie wszystkich klatek na dysk w postaci sekwencji bitmap.

W wersji podstawowej program obsługuje następujące rodzaje przejść:

- **Proste wjazdy:** przesuwanie obrazu z lewej, prawej, góry lub dołu ekranu.
- **Boks wchodzący i wychodzący:** drugi obraz pojawia się lub znika poprzez powiększanie lub zmniejszanie centralnego punktu na ekranie.
- **Ciemnienie i rozjaśnienie:** pierwszy obraz stopniowo ciemnieje do całkowitej czerni, a drugi pojawia się poprzez rozjaśnianie od ciemności do pełnej jasności.
- **Zmiana kanału alfa:** płynne przejście poprzez zmianę wartości alfa, co daje efekt przenikania obrazów.
- **Obrót kartki:** jeden obraz znajduje się po jednej stronie „kartki”, drugi po drugiej; kartka może być obracana w pionie lub poziomie.
- **Obrót wokół krawędzi:** pierwszy obraz obraca się wokół prawej krawędzi ekranu „w głąb”, a drugi pojawia się z lewej strony.

Wersja rozszerzona umożliwia dodatkowe, bardziej zaawansowane efekty przejścia:

- **Jasność pikseli:** kolejność pojawiania się pikseli zależy od ich jasności – najpierw najjaśniejsze, na końcu najciemniejsze.
- **Obrót pudełka w przestrzeni:** dwa obrazy umieszczone na bokach wirtualnego pudełka, które obraca się w przestrzeni.
- **Rozmycie i wyostrzenie:** obraz wyjściowy jest rozmywany, następnie przechodzi w rozmyty drugi obraz poprzez zmianę kanału alfa, a na końcu następuje wyostrzenie do ostatecznego obrazu.
- **Efekt „zdmuchiwania”:** pierwszy obraz „odlewa” w przestrzeń, a następnie pojawia się drugi obraz.
- **Efekt „ring”:** pierwszy obraz oddala się po półkolu w prawo, podczas gdy drugi pojawia się z lewej i zbliża się do środka.

Program przewiduje możliwość modyfikowania wybranych efektów w wersji rozszerzonej po uzgodnieniu z prowadzącym. Dzięki temu użytkownik może eksperymentować z różnymi typami przejść i tworzyć spersonalizowane animacje.

2 Podział pracy i analiza czasowa

Poniższa tabela przedstawia podział pracy między autorów oraz szacunkowy czas poświęcony na realizację poszczególnych zadań. Każdy uczestnik przygotowuje dokumentację dla swoich części.

Autor	Zakres pracy	Czas pracy [h]
Mikhail Shupliakou	<ul style="list-style-type: none">• Konfiguracja projektu• Wymagania podstawowe• Wymagania rozszerzone: przejście jasnością• Dokumentacja	11
Martyna Lach	<ul style="list-style-type: none">• Wymagania rozszerzone: efekt pudełka 3D• Wymagania rozszerzone: efekt ring• Dokumentacja	9.5
Julia Mucha	<ul style="list-style-type: none">• Wymagania rozszerzone: efekt rozmycia (blur)• Sprawdzenie i naprawa błędów w kodzie• Dokumentacja	8
Krystsina Mironenka	<ul style="list-style-type: none">• Wymagania rozszerzone: efekt zdmuchnięcia• Współautorstwo sortowania pikseli• Licznik FPS i testowanie• Dokumentacja	8

Tabela 1: Podział pracy i zakres obowiązków autorów

3 Założenia wstępne przyjęte w realizacji projektu

W trakcie realizacji projektu przyjęto następujące założenia wstępne, będące doprecyzowaniem wymagań zawartych w zleceniu oraz dodatkowymi założeniami wprowadzonymi przez zespół:

- **Obsługa obrazów:** Program będzie działał na bitmapach o identycznym rozmiarze – każda funkcja przejścia zakłada zgodność wymiarów obu obrazów. Obrazy mogą być wczytywane w formatach: .jpg, .png, .bmp, .tga.
- **Interfejs użytkownika:** Umożliwia wybór rodzaju przejścia, liczby klatek animacji oraz zapis sekwenacji bitmap na dysk. Zawiera podgląd miniatur wczytywanych obrazów, nowoczesny ciemny motyw oraz natywne okna dialogowe systemu Windows do wyboru plików i folderu docelowego.
- **Funkcjonalność wersji podstawowej:** Obejmuje proste wjazdy i zjazdy, efekt boksu wchodzącego i wychodzącego, ciemnienie pierwszego slajdu, płynne przejście przez kanał alfa, obracanie „kartki” oraz obracanie pierwszego obrazu wokół krawędzi.
- **Funkcjonalność wersji rozszerzonej:** Może zawierać dodatkowe efekty: sortowanie pikseli według jasności, efekt 3D obracanego pudełka, rozmycie (blur) z płynnym przejściem do kolejnego obrazu, efekt „zdmuchnięcia” oraz efekt „ring”.
- **Odtwarzalność animacji:** Każde przejście powinno być odtwarzalne w dowolnym punkcie czasu przy użyciu suwaka w interfejsie, co wymaga niezależnego renderowania każdej klatki animacji.
- **Modularność projektu:** Projekt zakłada modularną budowę programu, umożliwiającą łatwe dodawanie nowych efektów w przyszłości.
- **Dokumentacja techniczna:** Dokumentacja i opis użytych algorytmów będą tworzone równolegle z implementacją, aby każdy członek zespołu dokumentował swoje moduły.
- **Automatyzacja obsługi folderów wyjściowych:** Wprowadzony zostanie mechanizm automatycznego tworzenia folderu wyjściowego w razie jego braku.

3.1 Decyzja o wyborze narzędzi programistycznych

- **Język programowania:** C++ – wybrany ze względu na wysoką wydajność oraz pełną kontrolę nad zarządzaniem pamięcią i renderowaniem grafiki.
- **Środowisko programistyczne i kompilator:** Microsoft Visual Studio (kompilator MSVC) – zapewnia wygodne narzędzia do debugowania oraz stabilne wsparcie dla standardu C++17.

- **Biblioteka graficzna:** SFML (Simple and Fast Multimedia Library) – wykorzystana do obsługi okna aplikacji, renderowania grafiki 2D, shaderów oraz zapisu obrazów do plików.
- **Interfejs użytkownika:** ImGui (Dear ImGui) – użyty do stworzenia interaktywnego interfejsu graficznego w czasie rzeczywistym, zintegrowanego z SFML.
- **Obsługa systemu plików:** `std::filesystem` – zastosowana do zarządzania ścieżkami, katalogami wyjściowymi oraz zapisu klatek animacji.

4 Analiza projektu

4.1 Specyfikacja danych wejściowych

- **Rodzaje danych:** obrazy bitmapowe (zdjęcia, grafiki), które mają być podane animacji przejścia.
- **Postać danych:** każdy obraz jest wczytywany jako plik graficzny i mapowany na teksturę w pamięci programu.
- **Formaty danych:** obsługiwane formaty plików graficznych to `.jpg`, `.png`, `.bmp`, `.tga`.
- **Ograniczenia:** liczba klatek animacji musi zawierać się w zakresie 10–1000, a program działa wyłącznie w środowisku Windows ze względu na użycie natywnych okien dialogowych.
- **Uwagi dodatkowe:** obrazy muszą być poprawnie wczytane (rozmiar większy niż zero pikseli). Obrazy o różnych rozmiarach są automatycznie skalowane do rozdzielczości animacji (1200×800 pikseli), więc identyczny rozmiar nie jest wymagany.

4.2 Opis oczekiwanych danych wyjściowych

- **Rodzaj danych:** sekwencja obrazów bitmapowych reprezentujących kolejne klatki animacji przejścia między dwoma obrazami wejściowymi. Każda klatka odpowiada określonemu punktowi czasu animacji i umożliwia odtworzenie efektu w dowolnym momencie.
- **Postać danych:** każda klatka jest zapisywana jako niezależny plik graficzny zawierający pełną informację o kolorze i przezroczystości pikseli (RGBA, jeśli używane są efekty przezroczystości lub maski). Pliki mogą być odczytywane przez standardowe programy graficzne lub używane jako wejście do dalszej obróbki.
- **Formaty danych:** obsługiwany format wyjściowy to `.png` (bezstratny, z obsługą kanału alfa).

- **Uwagi dodatkowe:** liczba klatek i miejsce zapisu są definiowane przez użytkownika w interfejsie. Każda klatka jest renderowana niezależnie, co umożliwia pełną kontrolę nad animacją i dalszą obróbkę (np. eksport do GIF, wideo lub innych narzędzi graficznych).

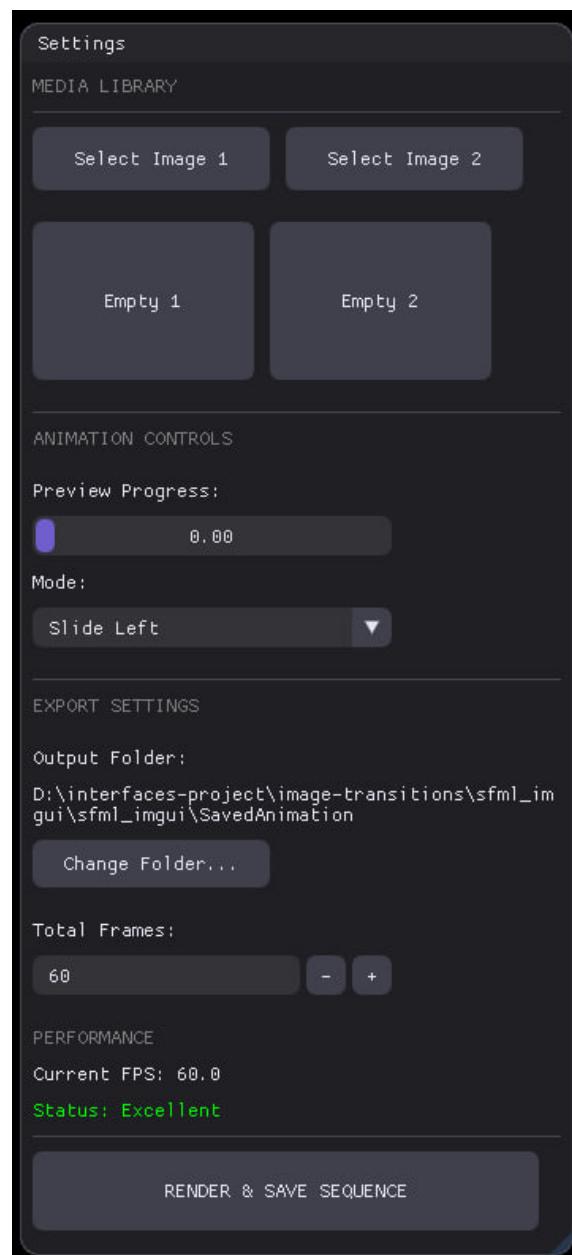
4.3 Zdefiniowanie struktur danych

W projekcie zastosowano struktury danych umożliwiające przechowywanie obrazów wejściowych, generowanie klatek animacji oraz zapis danych wyjściowych.

- **sf::Texture** – przechowuje obrazy wejściowe w pamięci GPU, wykorzystywane do renderowania podglądu oraz animacji przejść.
- **sf::Sprite** – reprezentuje obraz na ekranie i przechowuje informacje o jego położeniu, skali, obrocie oraz przezroczystości.
- **sf::RenderTexture** – wirtualna powierzchnia renderująca w pamięci GPU, służąca do generowania kolejnych klatek animacji przed zapisem na dysk.
- **sf::Image** – struktura przechowująca dane pikselowe w pamięci CPU, wykorzystywana do zapisu wygenerowanych klatek w postaci plików graficznych.
- **std::filesystem::path oraz std::string** – przechowują ścieżki do plików wejściowych oraz folderu wyjściowego.

4.4 Specyfikacja interfejsu użytkownika

- Interfejs oparty na ImGui + SFML, umożliwia interakcję myszką i klawiaturą w czasie rzeczywistym.
- **Sekcja Media Library:** wczytywanie obrazów źródłowych poprzez przyciski Select Image 1/2 (dialog Windows), podgląd miniaturow w okienkach 140x100 px.
- **Sekcja Animation Controls:** suwak Preview Progress do odtwarzania animacji w dowolnym punkcie czasu, lista rozwijana Mode do wyboru efektu przejścia, liczba klatek do eksportu.
- **Sekcja Export Settings:** wyświetlanie bieżącego folderu wyjściowego, przycisk Change Folder... (dialog Windows), przycisk Render & Save Sequence do generowania klatek, popup informacyjne w przypadku błędów.
- **Dodatkowe cechy:** ciemny motyw graficzny, nowoczesne zaokrąglenia i kontrastowe kolory przycisków, estetyczne rozmieszczenie elementów GUI.
- Interakcja użytkownika: wczytanie obrazów → wybór efektu → ustawienie suwaka → eksport sekwencji → automatyczne otwarcie folderu wynikowego.



Rysunek 1: GUI

4.5 Wyodrębnienie i zdefiniowanie zadań

Projekt został podzielony na mniejsze, jednoznacznie zdefiniowane moduły funkcjonalne, co ułatwia jego rozwój, testowanie oraz przyszłą rozbudowę. Każdy moduł odpowiada za konkretny aspekt działania aplikacji.

- **Moduł wejścia danych** – odpowiedzialny za wczytywanie obrazów źródłowych z dysku, weryfikację ich poprawności oraz przygotowanie danych do dalszego przetwarzania.
- **Moduł efektów przejścia** – zawiera implementacje poszczególnych animacji przejścia oraz logikę ich parametryzacji w zależności od postępu animacji.
- **Moduł animacji i czasu** – zarządza linią czasu animacji, obliczaniem aktualnego stanu przejścia oraz generowaniem kolejnych klatek.
- **Moduł renderowania** – odpowiada za rysowanie aktualnej klatki animacji na ekranie oraz przygotowanie obrazu do zapisu.
- **Moduł interfejsu użytkownika** – realizuje komunikację z użytkownikiem, umożliwiając wybór obrazów, parametrów animacji oraz uruchomienie eksportu.
- **Moduł eksportu danych** – odpowiada za zapis wygenerowanych klatek animacji do plików graficznych oraz zarządzanie folderem wyjściowym.

4.6 Decyzja o wyborze narzędzi programistycznych

- **Język programowania:** C++ - wymaganie projektowe.
- **Środowisko programistyczne i kompilator:** Microsoft Visual Studio (kompilator MSVC) – zapewnia wygodne narzędzia do debugowania oraz stabilne wsparcie dla standardu C++17.
- **Biblioteka graficzna:** SSFML (Simple and Fast Multimedia Library), użyta w wersji statycznie linkowanej – do obsługi okna aplikacji, renderowania grafiki 2D, shaderów oraz zapisu obrazów do plików.
- **Interfejs użytkownika:** ImGui – użyty do stworzenia interaktywnego interfejsu graficznego w czasie rzeczywistym, zintegrowanego z SFML.
- **Obsługa systemu plików:** std::filesystem – zastosowana do zarządzania ścieżkami, katalogami wyjściowymi oraz zapisu klatek animacji.

5 Opracowanie i opis niezbędnych algorytmów

5.1 Podstawy matematyczne animacji i sterowanie czasem

Podstawą funkcjonowania wszystkich zaimplementowanych efektów przejść jest matematyczne odwzorowanie upływu czasu na zmiany właściwości geometrycznych i

graficznych obiektów. Zamiast operować na numerach klatek, system wykorzystuje podejście oparte na czasie znormalizowanym oraz funkcjach interpolacyjnych.

5.1.1 Zmienna znormalizowana (Progress)

Kluczowym elementem sterującym logiką renderowania jest zmienna *progress* (oznaczana dalej jako p), która reprezentuje postęp animacji. Jest to wartość zmienno-przecinkowa z zakresu domkniętego:

$$p \in [0.0, 1.0]$$

Gdzie:

- $p = 0.0$ oznacza początek tranzycji (wyświetlany jest obraz źródłowy).
- $p = 1.0$ oznacza koniec tranzycji (wyświetlany jest w pełni obraz docelowy).

Za wybór zmiennej *progress* w naszej aplikacji odpowiada tak zwany progress bar.



Rysunek 2: Progress Bar

Dzięki normalizacji czasu, algorytmy przejść są niezależne od liczby klatek na sekundę (FPS) czy całkowitego czasu trwania animacji. W pętli renderującej wartość ta jest przekazywana do shaderów GLSL oraz funkcji transformujących wierzchołki.

5.1.2 Interpolacja Liniowa (LERP)

Większość przekształceń w projekcie (zmiana pozycji, przezroczystości, skali) opiera się na algorytmie interpolacji liniowej. Pozwala ona na znalezienie wartości pośredniej między stanem początkowym A a stanem końcowym B dla danego momentu p . Ogólny wzór zastosowany w kodzie to:

$$V(p) = A + (B - A) \cdot p$$

W projekcie algorytm ten przyjmuje różne formy w zależności od zastosowania:

- **Ruch (Translacja):** Dla efektu Slide Left, pozycja pozioma x obliczana jest jako $x(p) = -width \cdot (1.0 - p)$. Jest to szczególny przypadek LERP, gdzie obiekt przemieszcza się od $-width$ do 0.

```
case 0: // Slide Left
// Sprite 2 moves from Right (width) to Center (0)
xOffset = -width * (1.0f - progress);
s2.setPosition({ xOffset, yOffset });
break;
```

- **Przezroczystość (Alpha Blending):** Dla efektu Cross-Fade, kanał alfa drugiego obrazu jest bezpośrednio powiązany z postępem: $\alpha = 255 \cdot p$.

```
case 7: // Cross-Fade (Alpha)
{
    // Image 1 stays as background. Image 2 fades in on top.
    s2.setColor({ 255, 255, 255, (std::uint8_t)(255 * progress) });
    s2.setPosition({ 0.f, 0.f });
}
break;
```

- **Skalowanie:** W efektach typu Zoom, skala obiektu jest interpolowana od wartości 1.0 do 0.0 (lub odwrotnie).

5.1.3 Zarządzanie stanem i determinizm renderowania (State Reset)

Istotnym aspektem algorytmicznym w bibliotece **SFML** jest stanowość obiektów klasy `sf::Sprite`. Obiekt ten zachowuje swoje właściwości (obrót, kolor, pozycję) pomiędzy kolejnymi wywołaniami pętli rysującej. Aby zapewnić poprawność obliczeń matematycznych dla każdej klatki niezależnie, zaimplementowano mechanizm resetowania stanu na początku funkcji **RenderTransitionFrame**. Przed wykonaniem jakichkolwiek obliczeń dla bieżącej wartości p , algorytm przywraca macierz transformacji do wartości neutralnych:

- Pozycja: $(0, 0)$ – lewy górny róg.
- Obrót: 0° .
- Kolor: Biały z pełną nieprzezroczystością (`sf::Color::White`).
- Skala: Dopasowana do rozmiaru okna renderowania.

Dzięki temu podejściu funkcja renderująca staje się funkcją czystą (deterministyczną) – dla tego samego wejściowego p zawsze generuje identyczną klatkę wyjściową, co jest kluczowe przy eksportowaniu animacji do plików wideo, gdzie klatki mogą być generowane w innej kolejności lub tempie niż w podglądzie na żywo.

5.2 Algorytmy podstawowych przekształceń aficznego 2D

Druga grupa zaimplementowanych algorytmów opiera się na standardowych przekształceniach aficznego w przestrzeni dwuwymiarowej. Wykorzystują one wbudowane w silnik graficzny metody manipulacji macierzą modelu (Model Matrix), która określa pozycję, rotację i skalę obiektu w świecie gry. W przeciwieństwie do operacji na pikselach (realizowanych przez shadery), operacje te wykonywane są na wierzchołkach czworokątów (sprite'ów).

5.2.1 Translacja (Przesunięcie liniowe)

Efekty z grupy Slide (Left, Right, Top, Bottom) realizują algorytm translacji, czyli przesunięcia wszystkich punktów obiektu o stały wektor v . Wektor ten jest funkcją czasu p . Dla przejścia typu Slide (wjazd drugiego obrazu na pierwszy), pozycja (x, y) drugiego sprite'a jest obliczana w każdej klatce według wzoru:

$$\begin{cases} x(p) = x_{\text{start}} + (x_{\text{end}} - x_{\text{start}}) \cdot p \\ y(p) = y_{\text{start}} + (y_{\text{end}} - y_{\text{start}}) \cdot p \end{cases}$$

5.2.2 Skalowanie względem punktu centralnego (Zoom)

Algorytmy Box In oraz Box Out wymagają zmiany rozmiaru obiektu w czasie. Kluczowym elementem tego algorytmu jest zmiana punktu odniesienia transformacji (Origin). Domyślnie transformacje w grafice 2D odbywają się względem lewego górnego rogu $(0, 0)$. Aby uzyskać efekt "Zoom In" (powiększanie od środka), algorytm wykonuje następujące kroki:

1. **Ustawienie Origin:** Punkt odniesienia zostaje przesunięty na środek tekstuury:

$$\text{Origin} = \left(\frac{\text{Width}_{\text{tex}}}{2}, \frac{\text{Height}_{\text{tex}}}{2} \right)$$

2. **Ustawienie Pozycji:** Obiekt jest pozycjonowany na środku ekranu renderowania.

3. **Obliczenie Skali:** Współczynnik skali S jest interpolowany liniowo:

- **Dla Box In:** $S(p) = S_{\text{max}} \cdot p$ (obraz rośnie od 0 do pełnego rozmiaru).
- **Dla Box Out:** $S(p) = S_{\text{max}} \cdot (1.0 - p)$ (obraz maleje do 0).

```
case 4: // Box In (Zoom In)
{
    // Sprite 2 grows from the center of the screen
    sf::Vector2u sz2 = t2.getSize();
    // Set Origin to center of the image
    s2.setOrigin({ (float)sz2.x / 2.f, (float)sz2.y / 2.f });
    // Set Position to center of the screen
    s2.setPosition({ width / 2.f, height / 2.f });
}
```

```

// Calculate scale factor based on progress (0.0 to 1.0)
float tX = width / sz2.x;
float tY = height / sz2.y;
s2.setScale({ tX * progress, tY * progress });
}
break;

```

Dzięki temu operacja mnożenia macierzy skali odbywa się względem centrum, a nie narożnika, co tworzy pożądany efekt głębi.

5.3 Algorytmy przetwarzania rastrowego na CPU (Blur i Luma)

Ze względu na specyficzne wymagania projektowe (brak możliwości wykorzystania shaderów GLSL), algorytmy przetwarzania obrazu (rozmycie oraz przejście oparte na jasności) zostały zaimplementowane bezpośrednio na jednostce centralnej (CPU). Kluczowym wyzwaniem było zapewnienie płynności 60 FPS przy operacjach na obrazach o wysokiej rozdzielczości (1200x800). W tym celu, oprócz separacji splotu, wprowadzono zaawansowane techniki zarządzania pamięcią oraz zastosowano technikę redukcji rozdzielczości przetwarzanego obrazu (downsampling) dla algorytmów o wysokiej złożoności obliczeniowej, co pozwoliło na drastyczne zmniejszenie liczby operacji na sekundę bez widocznej utraty jakości efektu.

5.3.1 Algorytm Luma Wipe (Przejście jasnością)

Algorytm ten decyduje o widoczności pikseli drugiego obrazu na podstawie ich jasności (luminancji). Proces składa się z dwóch etapów:

- Obliczenie mapy luminancji:** Dla każdego piksela obrazu wejściowego obliczana jest wartość jasności L zgodnie ze standardem ITU-R BT.601:

$$L = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Gdzie R, G, B to składowe koloru. W celu optymalizacji obliczenia wykonywane są na liczbach całkowitych.

- Progowanie (Thresholding):** Wartość postępu animacji $p \in [0, 1]$ jest mapowana na próg odcięcia $T \in [255, 0]$.

$$\text{Piksel}_{\text{wynik}} = \begin{cases} \text{Piksel}_B & \text{jeśli } L \geq T \\ \text{Piksel}_A & \text{w przeciwnym razie} \end{cases}$$

Dzięki temu najpierw pojawiają się najjaśniejsze elementy obrazu docelowego (np. niebo, światła), a na końcu elementy najciemniejsze.

Optymalizacja wydajności Luma Wipe:

- **Buforowanie mapy luminancji (Caching):** Wartości jasności L są obliczane tylko raz i przechowywane w statycznym wektorze, co eliminuje powtarzalne obliczenia zmiennoprzecinkowe.
- **Optymalizacja dostępu do pamięci:** Zastosowano wysoko wydajną pętlę jednowątkową, która lepiej wykorzystuje mechanizm Hardware Prefetcher procesora przy liniowym dostępie do danych, unikając narzutu związanego z synchronizacją wątków.

5.3.2 Algorytm Separowanego Rozmycia Pudełkowego (Box Blur)

Standardowe rozmycie o promieniu R wymagałoby dla każdego piksela pobrania średniej z $(2R + 1)^2$ sąsiadów, co daje złożoność obliczeniową $O(W \cdot H \cdot R^2)$. Dla dużych promieni jest to zbyt wolne dla CPU.

Zastosowano optymalizację polegającą na separacji splotu. Rozmycie dwuwymiarowe rozdzielono na dwa niezależne przebiegi jednowymiarowe:

1. **Przebieg poziomy:** Dla każdego piksela obliczana jest średnia kolorów z sąsiadów w poziomie. Wynik zapisywany jest w buforze tymczasowym.
2. **Przebieg pionowy:** Na wynikach przebiegu poziomego obliczana jest średnia z sąsiadów w pionie.

Zastosowanie Downsamplingu w procesie rozmycia: Standardowy splot nadal obciążał CPU, dlatego wdrożono technikę piramidy obrazu:

1. **Redukcja danych:** Obraz jest 4-krotnie zmniejszany przed nałożeniem filtra, co redukuje liczbę pikseli do przetworzenia o 93.75
2. **Przetwarzanie:** Splot wykonywany jest na zredukowanym buforze, obniżając złożoność do

$$O\left(\frac{W}{4} \cdot \frac{H}{4} \cdot \frac{R}{4}\right)$$

3. **Skalowanie:** Wynikowa tekstura jest rozciągana do 1200x800 przy użyciu sprzętowego mapowania (SFML Sprite Scaling). Ponieważ Blur usuwa detale, utrata jakości jest niezauważalna, a wzrost FPS jest kilkukrotny.

Dzięki temu złożoność spadła do liniowej. Dodatkowo, każdy wiersz (w przebiegu 1) i każda kolumna (w przebiegu 2) są niezależne, co pozwoliło na ich idealne zrównoleglenie.

5.4 Opracowanie własnych algorytmów 3D i rzutowania perspektywicznego

Ze względu na dwuwymiarową naturę biblioteki SFML, realizacja efektu obracającego się sześcianu (Cube Rotate) wymagała opracowania autorskiego potoku ren-

rowania (rendering pipeline) po stronie procesora (CPU). Algorytm ten ręcznie przetwarza wierzchołki w przestrzeni trójwymiarowej, rzutuje je na płaszczyznę ekranu i rozwiązuje problemy widoczności oraz teksturowania.

5.4.1 Matematyka obrotu (Transformacje w przestrzeni euklidesowej)

Pierwszym krokiem algorytmu jest umieszczenie płaskich obrazów w wirtualnej przestrzeni 3D. Obrazy traktowane są jako ściany sześcianu. Aby zrealizować obrót wokół osi pionowej Y, zastosowano macierz rotacji. Dla każdego punktu $P(x, y, z)$ jego nowa pozycja $P'(x', y', z')$ po obrocie o kąt θ obliczana jest według układu równań:

$$\begin{cases} x' = x \cdot \cos(\theta) + z \cdot \sin(\theta) \\ y' = y \\ z' = -x \cdot \sin(\theta) + z \cdot \cos(\theta) \end{cases}$$

W kodzie kąt θ jest bezpośrednio zależny od zmiennej progress (zakres 0° do 90°), co powoduje obrót sześcianu o jedną ścianę.

5.4.2 Rzutowanie perspektywiczne (Perspektywa jednoskrotna)

Aby wyświetlić trójwymiarowy obiekt na płaskim ekranie monitora, konieczne jest zastosowanie rzutowania (projekcji). Zaimplementowano model kamery otworkowej, gdzie współczynnik skali obiektu maleje wraz z oddalaniem się od obserwatora (osi Z).

Współczynnik rzutowania S wyznaczany jest wzorem:

$$S = \frac{FOV}{FOV + z'}$$

Gdzie:

- FOV (Field of View) – stała określająca "ogniskową" wirtualnego obiektywu (w projekcie przyjęto 800.0f).
- z' – głębokość punktu po obrocie.

Współrzędne ekranowe 2D (u, v) to ostatecznie:

$$u = Center_X + x' \cdot S, \quad v = Center_Y + y' \cdot S$$

Dzięki temu ściana sześcianu, która "oddala się" od widza, staje się wizualnie mniejsza, tworząc iluzję głębi.

5.4.3 Rozwiążanie problemu teksturowania (Dyskretyzacja powierzchni)

Standardowe renderowanie czworokątów (Quad) w środowiskach 2D wykorzystuje interpolację aficzną tekstur. Przy silnych skrótach perspektywicznych powoduje to błędy wizualne ("łamanie się" tekstury po przekątnej), ponieważ tekstura jest interpolowana liniowo w przestrzeni ekranu, a nie w przestrzeni 3D.

Aby wyeliminować ten artefakt bez użycia shaderów perspektywicznych, opracowano algorytm dyskretyzacji powierzchni (Subdivision):

1. Każda ściana sześciangu (obraz) nie jest rysowana jako jeden prostokąt, lecz jest dzielona na 96 pionowych pasów (STRIPS).
2. Dla każdego paska osobno obliczane są współrzędne 3D i rzutowanie.
3. Wąskie paski są rysowane przy użyciu prymitywu sf::TriangleStrip.

Dzięki tak dużej gęstości podziału, błąd interpolacji liniowej staje się niezauważalny dla ludzkiego oka, a obraz zachowuje poprawną perspektywę geometryczną.

5.5 Deformacje geometryczne i ruch orbitalny (Ring Transition)

Symuluje algorytm ruch obrazów po obwodzie wirtualnego pierścienia (cylindra) w przestrzeni trójwymiarowej. W przeciwieństwie do liniowych przesunięć, algorytm ten wykorzystuje współrzędne biegunowe do obliczenia pozycji w przestrzeni 3D, a następnie rzutuje je na płaszczyznę 2D.

5.5.1 Parametryczne równanie ruchu po okręgu

Podstawą algorytmu jest odwzorowanie liniowego postępu animacji ($p \in [0, 1]$) na kąt obrotu θ w radianach. Pełna tranzycja odpowiada obrotowi o 90° ($\frac{\pi}{2}$ radiana).

Pozycja każdego z obrazów w przestrzeni (x, z) jest obliczana na podstawie parametrycznych równań okręgu. Zmienna z reprezentuje głębokość (oddalenie od kamery), co jest kluczowe dla efektu perspektywy.

$$\begin{cases} \theta = p \cdot \frac{\pi}{2} \\ x = R \cdot (1 - \cos(\theta)) \cdot k_{side} \\ z = R \cdot \sin(\theta) \end{cases} \quad (1)$$

Gdzie:

- R – promień orbity (w kodzie zmienna `radius`, np. `1000.0f`).
- k_{side} – kierunek ruchu (+1 dla wchodzącego, -1 dla wychodzącego obrazu).

5.5.2 Rzutowanie i Skalowanie Perspektywiczne

Po wyznaczeniu pozycji 3D, algorytm dokonuje rzutowania na ekran. W przeciwnieństwie do rzutu izometrycznego, zastosowano tu rzut perspektywiczny, w którym obiekty dalsze (większe z) wydają się mniejsze. Współczynnik skali S obliczany jest ze wzoru:

$$S = \frac{D}{D + z} \quad (2)$$

Gdzie D (zmienna `depth`) to odległość obserwatora od płaszczyzny rzutowania. Uzyskana skala S służy do modyfikacji szerokości i wysokości renderowanego czworokąta (quad), co tworzy iluzję ruchu w głąb sceny ("fisheye effect").

5.5.3 Implementacja (Dual Rendering)

Efekt wymaga jednoczesnego renderowania dwóch obiektów poruszających się w przeciwnych fazach.

- 1. Obraz 1 (Wychodzący):** Jego kąt zmienia się od 0 do $\frac{\pi}{2}$. Przesuwa się on ze środka ekranu na bok, "odpływając" w głąb (rosnące z).
- 2. Obraz 2 (Wchodzący):** Jego kąt zmienia się od $\frac{\pi}{2}$ do 0. Wynurza się on z głębi, zmierzając do centrum.

Poniższy fragment kodu w języku C++ (wykorzystujący składnię Lambda z C++17) ilustruje implementację funkcji obliczającej pozycję na pierścieniu:

```
auto ringPos = [&](float angle, float sideSign)
{
    // Obliczenie współrzędnych cylindrycznych
    float x = sideSign * (radius - std::cos(angle) * radius);
    float z = std::sin(angle) * radius;

    // Obliczenie skali perspektywicznej
    float s = depth / (depth + z);

    // Zwrócenie krotki: pozycja X, głębokość Z, skala S
    return std::tuple<float, float, float>(x, z, s);
};
```

5.5.4 Rozwiązywanie widoczności (Depth Sorting)

Ponieważ oba obrazy poruszają się w przestrzeni 3D, konieczne jest ustalenie kolejności rysowania, aby uniknąć błędów graficznych (np. obraz w tle rysowany na wierzchu). Zastosowano dynamiczny *Algorytm Malarza*. W każdej klatce program porównuje współrzędne z obu obrazów i renderuje je w kolejności od najdalszego do najbliższego:

$$\text{Order} = \begin{cases} \text{Draw}(Img_1) \rightarrow \text{Draw}(Img_2) & \text{jeśli } z_1 > z_2 \\ \text{Draw}(Img_2) \rightarrow \text{Draw}(Img_1) & \text{w przeciwnym razie} \end{cases} \quad (3)$$

Zapewnia to płynne i poprawne wizualnie mijanie się obrazów w połowie transzycji.

5.6 Algorytm Fly Away (Efekt „Odlotu”)

Efekt Fly Away realizuje dynamiczne przejście oparte na jednoczesnej manipulacji geometrią oraz przezroczystością obrazów. Transformacja ta symuluje oddalanie się pierwszego obiektu w głąb przestrzeni oraz „nadlatywanie” drugiego z oddali. Proces ten został zaimplementowany w dwóch fazach, z punktem zwrotnym przy wartości postępu $p = 0.5$.

Kluczowe aspekty techniczne algorytmu:

1. **Transformacja skali i rotacji:** Obrazy są skalowane względem ich środka geometrycznego (ustawienie punktu zakotwiczenia za pomocą `setOrigin`). W pierwszej fazie skala obrazu wejściowego maleje od 1.0 do 0, podczas gdy rotacja zmienia się liniowo od 0° do 180° . W drugiej fazie obraz docelowy rośnie od skali 0 do 1.0, obracając się z pozycji -180° do stanu spoczynku
2. **Zarządzanie przezroczystością (Alpha Blending):** Aby uzyskać płynność przejścia, zastosowano modyfikację składowej Alpha koloru. W pierwszej połowie animacji pierwszy obraz zanika do czarnego tła, a w drugiej połowie drugi obraz stopniowo odzyskuje pełną widoczność.
3. **Normalizacja postępu lokalnego:** W kodzie zastosowano zmienną pomocniczą lp (local progress), która mapuje globalny zakres postępu $[0.0, 1.0]$ na dwa podzakresy $[0.0, 1.0]$ dla każdej z faz z osobna:
 - Dla $p \leq 0.5$: $lp = p \cdot 2$
 - Dla $p > 0.5$: $lp = (p - 0.5) \cdot 2$

Zalety wydajnościowe: Mimo złożonego efektu wizualnego, algorytm ten jest niezwykle wydajny na procesorze CPU, ponieważ operuje na macierzach transformacji, nie wymagając bezpośredniej modyfikacji każdego piksela w pętli, co pozwala na utrzymanie stałych 60 FPS nawet przy słabszych jednostkach obliczeniowych.

5.7 Techniczne aspekty implementacji i struktura danych

5.7.1 Bezpośrednia manipulacja geometrią (Vertex Arrays)

Standardowa klasa `sf::Sprite` w bibliotece SFML reprezentuje prosty, teksturowany prostokąt. Jest to wystarczające dla przekształceń afanicznych 2D, ale uniemożliwia realizację efektów deformacji przestrzennych (takich jak obracający się sześcian czy pierścień).

Aby ominąć to ograniczenie, zastosowano klasę `sf::VertexArray`, która pozwala na bezpośrednie definiowanie surowych danych wierzchołków przesyłanych do karty graficznej.

- **Struktura Wierzchołka:** Każdy wierzchołek (`sf::Vertex`) przechowuje trzy kluczowe informacje:
 1. Pozycję w przestrzeni ekranu (x, y).
 2. Współrzędne tekstury (u, v) mapujące piksel z obrazu źródłowego.
 3. Kolor (używany do cieniowania ścian sześciennu).
- **Prymitywy graficzne:** W efekcie *Cube Rotation* użyto typu `sf::TriangleStrip`. Pozwala to na zdefiniowanie ciągu trójkątów, gdzie każdy kolejny wierzchołek tworzy nowy trójkąt z dwoma poprzednimi. Zredukowało to liczbę przesyłanych danych o ok. 50% w porównaniu do oddzielnych trójkątów, co jest kluczowe przy podziale ściany na 96 pasów (łącznie ponad 200 wierzchołków na klatkę).

5.7.2 Rendering pozaekranowy

Funkcjonalność eksportu animacji do sekwencji plików PNG wymagała odseparowania procesu wyświetlania obrazu na ekranie od procesu generowania klatek do zapisu. W tym celu wykorzystano mechanizm **FBO (Framebuffer Object)**, dostępny w SFML poprzez klasę `sf::RenderTexture`.

5.7.3 Potok eksportu

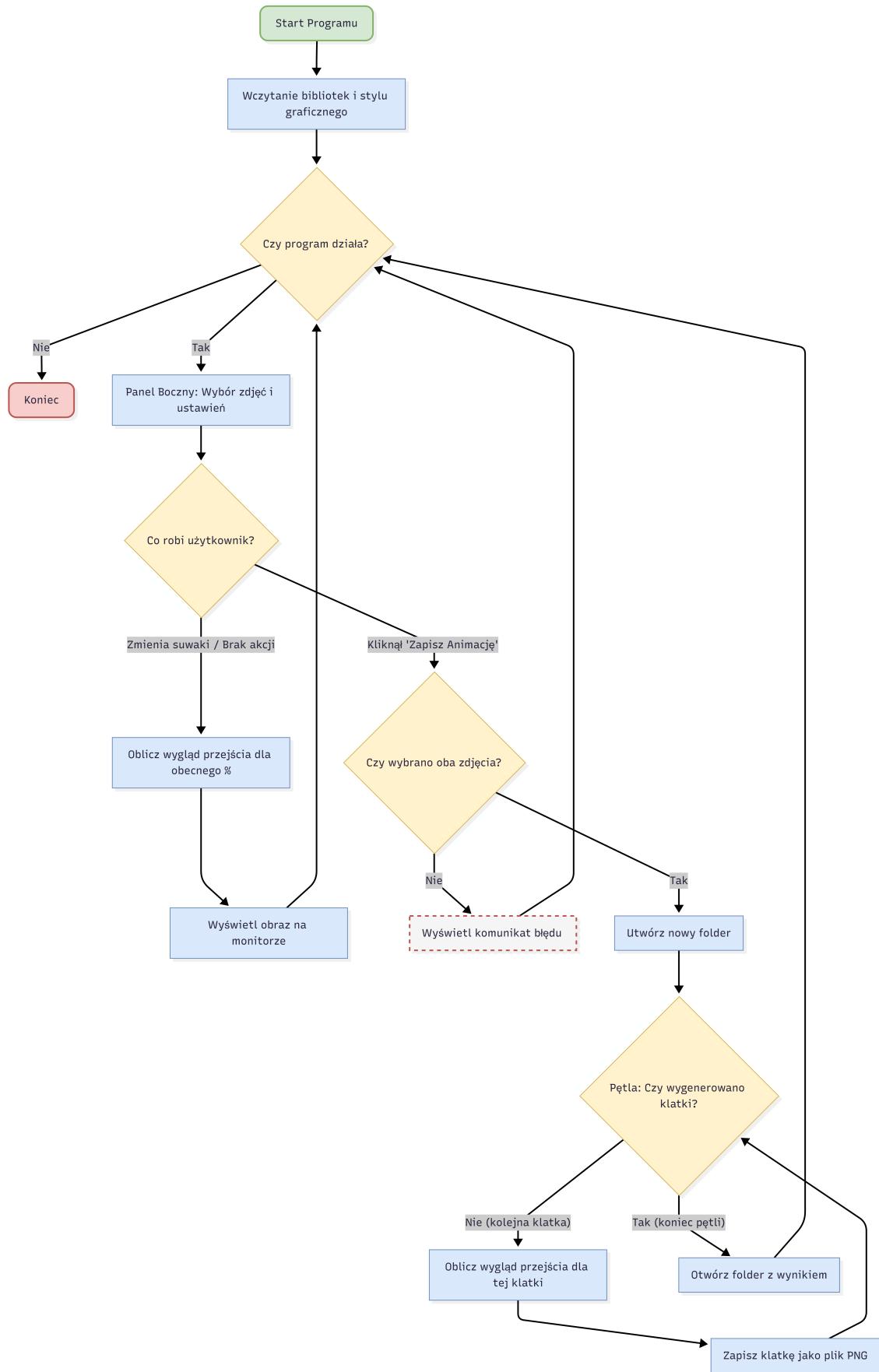
Proces zapisu sekwencji klatek przebiega w izolowanym środowisku, co eliminuje migotanie okna aplikacji i pozwala na wygenerowanie animacji o idealnym klatkażu, niezależnie od wydajności komputera w czasie rzeczywistym.

1. **Render:** Scena jest rysowana na ukrytej teksturze (`renderTex.display()`).
2. **Pobranie (Readback):** Tekstura jest kopiwana z pamięci wideo (GPU) do pamięci operacyjnej (CPU) do obiektu `sf::Image`.
3. **Zapis:** Obraz jest kompresowany i zapisywany na dysku twardym.

Użycie `sf::RenderTexture` gwarantuje również, że inne okna systemu operacyjnego przysłaniające aplikację nie wpłyną na wygląd zapisywanych klatek.

6 Kodowanie

Program został zrealizowany z wykorzystaniem biblioteki graficznej **SFML** (obsługa okna, grafiki 2D, tekstur) oraz biblioteki **ImGui** (interfejs użytkownika). Architektura aplikacji opiera się na głównej pętli zdarzeń (Event Loop), oddzielając logikę renderowania efektów od warstwy interfejsu i obsługi systemu operacyjnego.



Rysunek 3: Schemat blokowy dla programu

6.1 Funkcje pomocnicze (API Systemu Windows)

Ze względu na konieczność interakcji z systemem plików (wybór obrazów, wybór folderu zapisu), program wykorzystuje natywne funkcje Windows API (<Windows.h>).

- `std::string OpenFileDialog(HWND ownerHandle)`

Otwiera standardowe systemowe okno wyboru pliku. Konfiguruje strukturę `OPENFILENAMEA`, ustawia filtry plików (tylko obrazy: jpg, png, bmp) i zwraca ścieżkę absolutną do wybranego pliku.

- `std::string SelectFolderDialog(HWND ownerHandle)`

Otwiera systemowe okno wyboru katalogu. Wykorzystuje funkcję `SHBrowseForFolderA` do pobrania ścieżki folderu, w którym mają zostać zapisane wyrenderowane klatki animacji.

6.2 Główna logika renderowania

Sercem programu jest funkcja odpowiedzialna za obliczenia matematyczne i rysowanie klatek. Została zaprojektowana w sposób polimorficzny, aby obsługiwać zarówno ekran, jak i pliki.

Funkcja: `void RenderTransitionFrame(sf::RenderTarget& target, int type, ...)`

1. **Argumenty:** Funkcja przyjmuje referencję do abstrakcyjnego celu renderowania (`target`), indeks efektu, postęp animacji oraz referencje do tekstur i sprite'ów.
2. **Reset stanu (State Reset):** Przed wykonaniem transformacji przywraca domyślne wartości pozycji, skali, rotacji i koloru sprite'ów. Jest to kluczowe, ponieważ obiekty SFML zachowują swój stan między klatkami.
3. **Instrukcja Switch:** Wybiera odpowiedni algorytm matematyczny na podstawie zmiennej `type`.
 - Dla efektów 3D (Cube, Ring) definiuje lokalne funkcje Lambda do rzutowania i buduje geometrię `sf::VertexArray`.
4. **Rysowanie:** Wywołuje metodę `target.draw()` w odpowiedniej kolejności (algorytm malarza), aby poprawnie nałożyć warstwy.

6.3 Pętla główna i interfejs (GUI)

Funkcja `main` zarządza cyklem życia aplikacji:

1. **Inicjalizacja:** Tworzone jest okno `sf::RenderWindow` o wymiarach 1200×800 pikseli.

2. **Panel GUI (ImGui):** Wyświetla przyciski ładowania plików, suwaki sterujące postępem (progress) oraz listę wyboru efektów. Stylizacja interfejsu została zdefiniowana w funkcji `SetupModernStyle()`.
3. **Logika Eksportu:** Po naciśnięciu przycisku "RENDER & SAVE", program tworzy wirtualne płótno `sf::RenderTexture`, generuje klatki w pętli, pobiera je z pamięci GPU i zapisuje na dysku jako pliki PNG.

6.4 Kluczowe zmienne i struktury danych

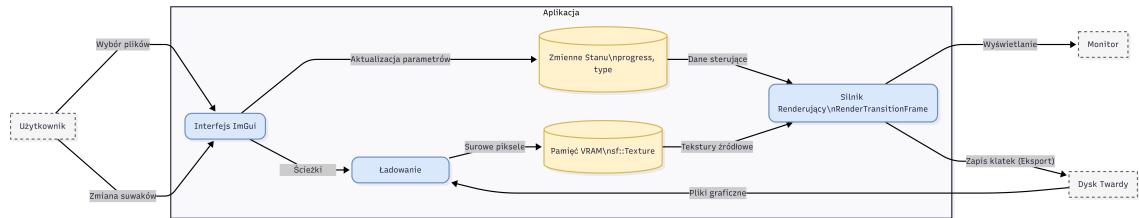
Poniższa tabela przedstawia najważniejsze zmienne globalne i lokalne sterujące działaniem programu.

Nazwa zmiennej	Typ danych	Opis funkcji
<code>window</code>	<code>sf::RenderWindow</code>	Główne okno aplikacji, kontekst OpenGL.
<code>texture1, texture2</code>	<code>sf::Texture</code>	Kontenery przechowujące surowe dane pikseli (VRAM).
<code>sprite1, sprite2</code>	<code>sf::Sprite</code>	Obiekty geometryczne służące do wyświetlania tekstur, posiadające atrybuty transformacji.
<code>progress</code>	<code>float</code>	Zmienna sterująca postępem animacji. Zakres [0.0, 1.0].
<code>transitionType</code>	<code>int</code>	Indeks aktualnie wybranego algorytmu przejścia.
<code>renderTex</code>	<code>sf::RenderTexture</code>	Obiekt FBO używany do renderowania pozaekranowego podczas eksportu.
<code>va</code>	<code>sf::VertexArray</code>	Tablica wierzchołków używana w efektach 3D do definiowania niestandardowej geometrii.

Tabela 2: Zestawienie kluczowych zmiennych w programie.

6.5 Wykres przepływu danych.

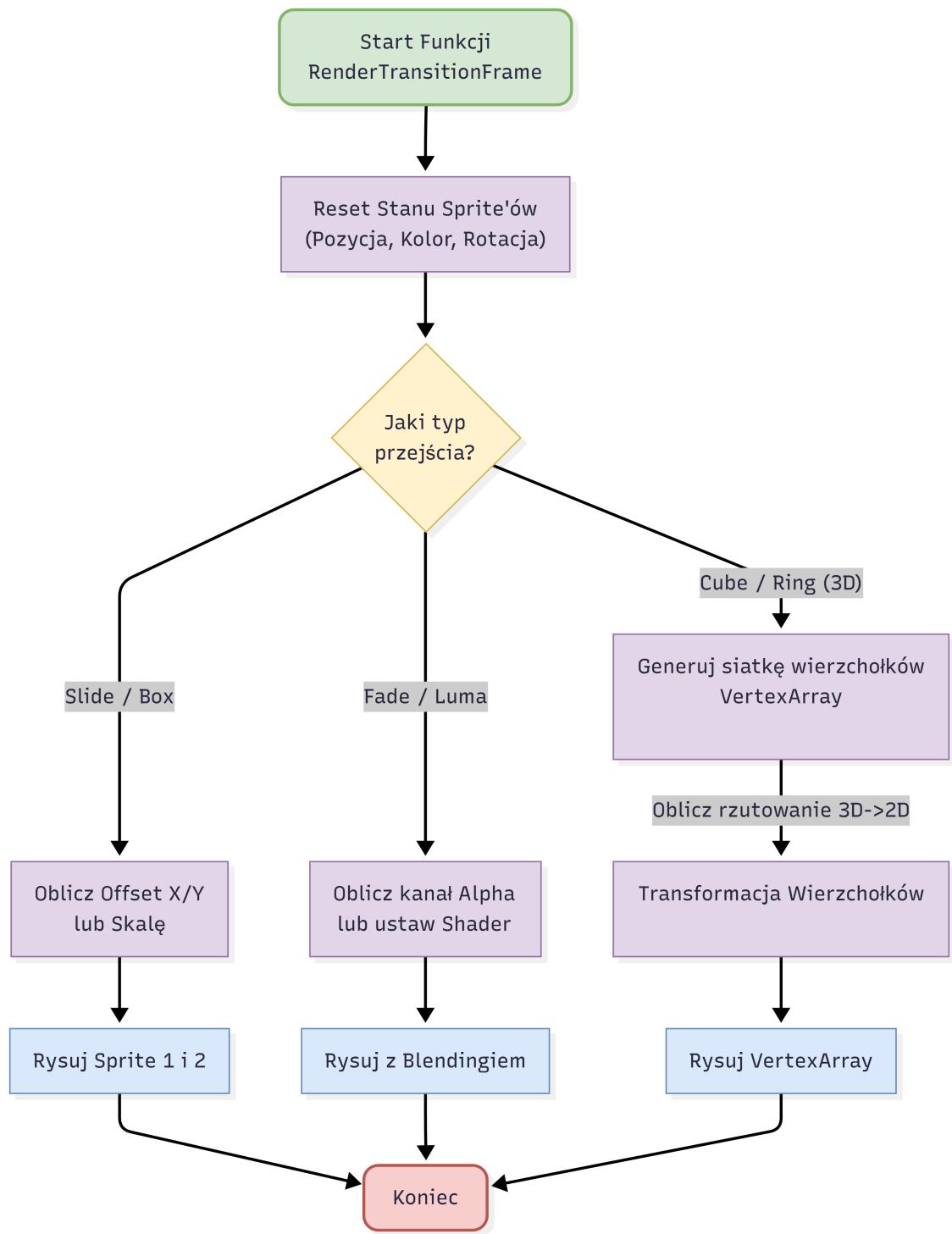
Poniższy diagram przedstawia przepływ danych w systemie (Data Flow Diagram - Level 1). Ilustruje on drogę, jaką przebywają obrazy od momentu załadowania z dysku, poprzez przetworzenie w pamięci karty graficznej (VRAM), aż do wyświetlenia na ekranie lub zapisu do pliku wynikowego.



Rysunek 4: Enter Caption

6.6 Graf algorytmu renderującego

Kluczowym algorytmem programu jest funkcja `RenderTransitionFrame`. Odpowiada ona za matematyczne przekształcenie dwóch tekstur wejściowych w jedną klatkę wyjściową. Poniższy graf prezentuje logikę decyzyjną wewnątrz tej funkcji, w tym obsługę resetowania stanu sprite'ów oraz wybór odpowiedniego przekształcenia geometrycznego (2D/3D) lub shaderowego.



Rysunek 5: Enter Caption

6.7 Implementacja efektów rastrowych (Blur i Luma)

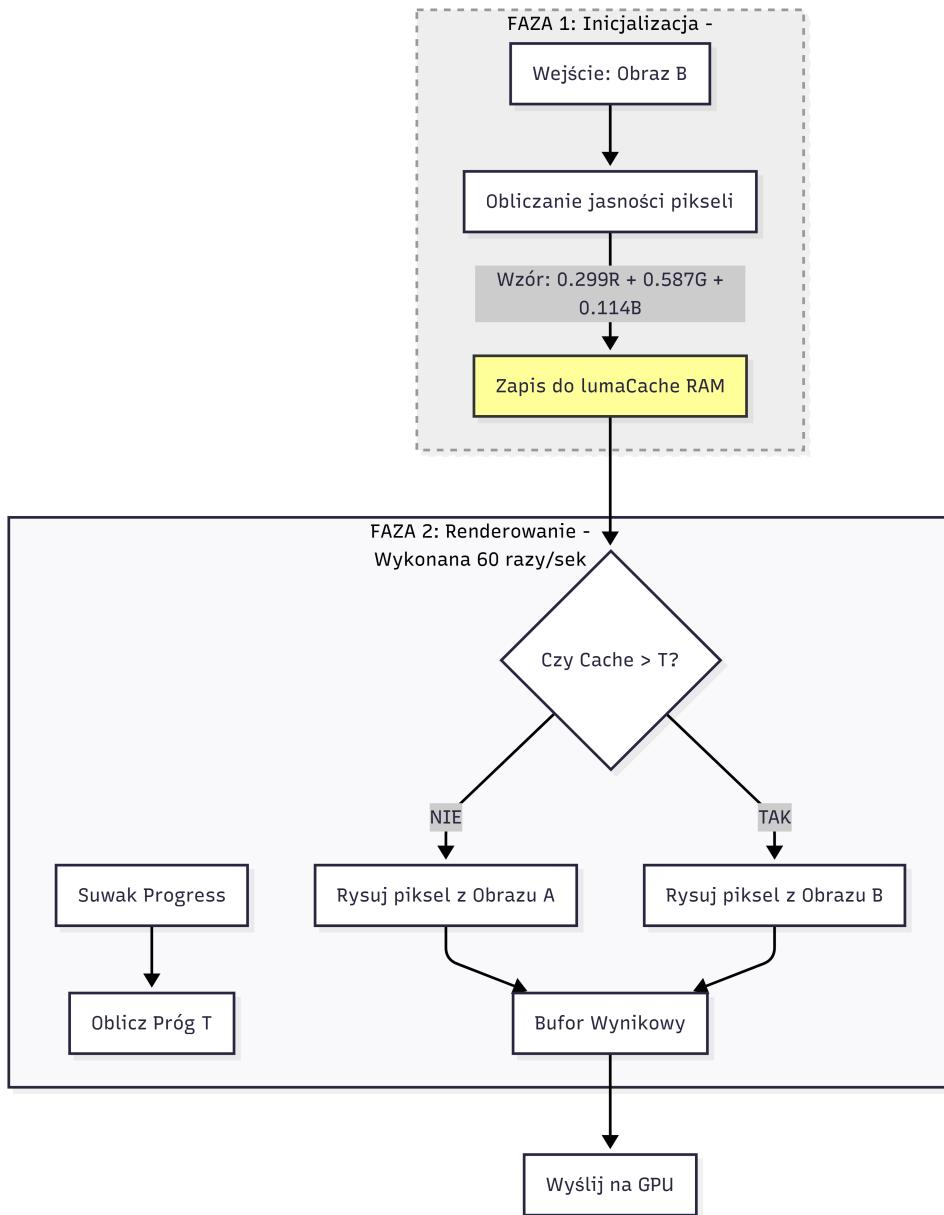
Implementacja efektów *Blur Fade* oraz *Luma Wipe* znajduje się w dedykowanych funkcjach zoptymalizowanych pod kątem wykorzystania wielu rdzeni procesora. Wy-

korzystano bibliotekę `<future>` oraz `std::async` do podziału pracy.

6.7.1 Funkcja `ApplyCpuLumaWipeOptimized`

Funkcja ta realizuje przejście jasnością. Aby uniknąć kosztownego obliczania wzoru na luminancję w każdej klatce (60 razy na sekundę), zastosowano mechanizm **Cache**:

- Zmienna globalna `lumaCache` przechowuje wstępnie obliczone wartości jasności dla drugiego obrazu.
- Obliczenia wykonywane są tylko raz (przy załadowaniu obrazu), a w trakcie animacji następuje jedynie szybkie porównanie wartości z cache z progiem wynikającym z suwaka `progress`.
- Obraz jest dzielony na poziome pasy (`rowsPerThread`), a każdy wątek przetwarza swój fragment niezależnie, zapisując wynik do wspólnego bufora wyjściowego.



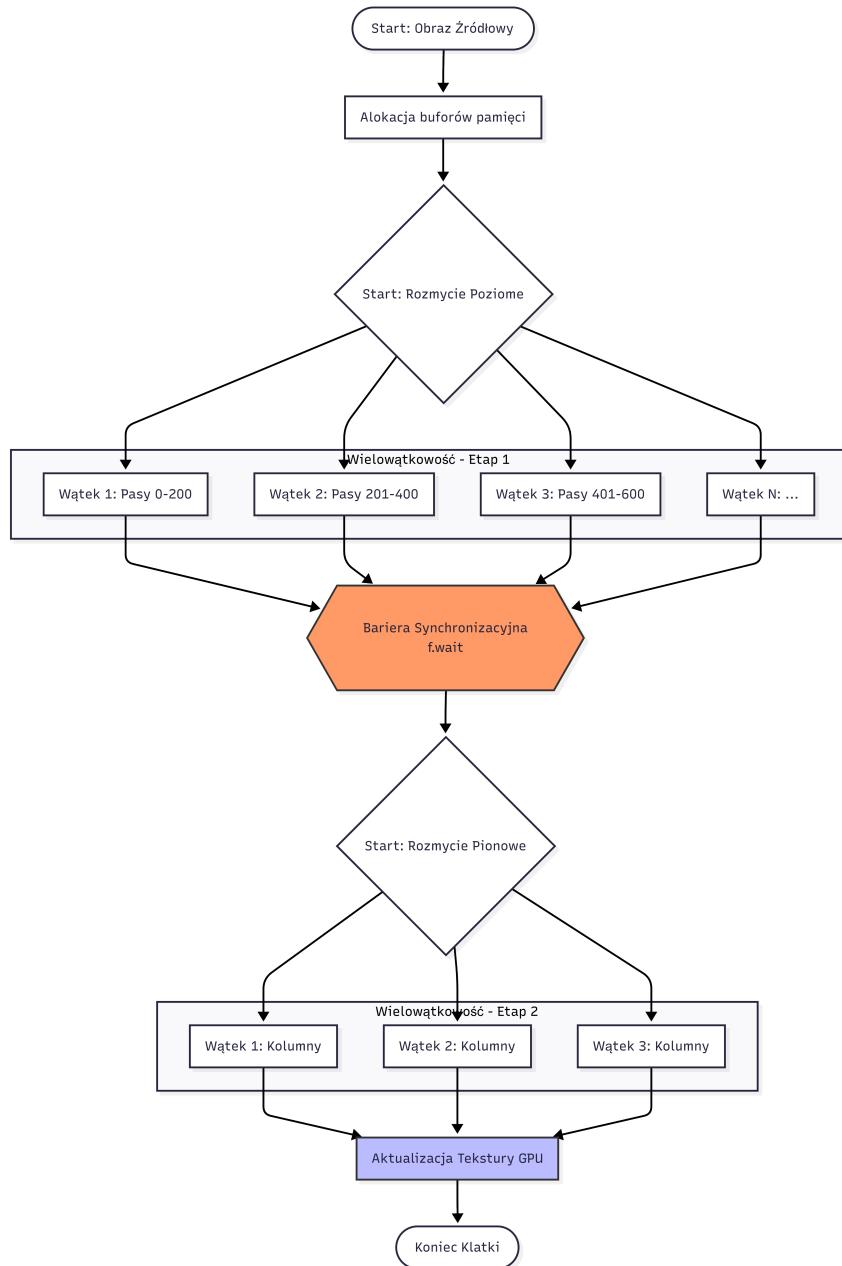
Rysunek 6: Schemat przepływu danych w algorytmie Luma Wipe (Cache + Render)

6.7.2 Funkcja ApplyCpuBlurOptimized

Funkcja realizuje rozmycie obrazu w czasie rzeczywistym. Przyjmuje parametry: obraz źródłowy, teksturę docelową oraz promień rozmycia.

- Wykorzystuje dwa statyczne bufore pamięci (`pass1`, `pass2`) aby uniknąć ciągłej alokacji pamięci RAM.
- Zastosowano barierę synchronizacyjną między przebiegiem poziomym a pionowym (`f.wait()`), aby upewnić się, że wszystkie wątki zakończyły pierwszy etap obliczeń przed rozpoczęciem drugiego.

- Wynik końcowy jest przesyłany do karty graficznej metodą `dstTex.update()`, co jest jedynym momentem komunikacji z GPU w tym procesie.



Rysunek 7: Schemat zrównoleglenia algorytmu Box Blur z barierą synchronizacyjną

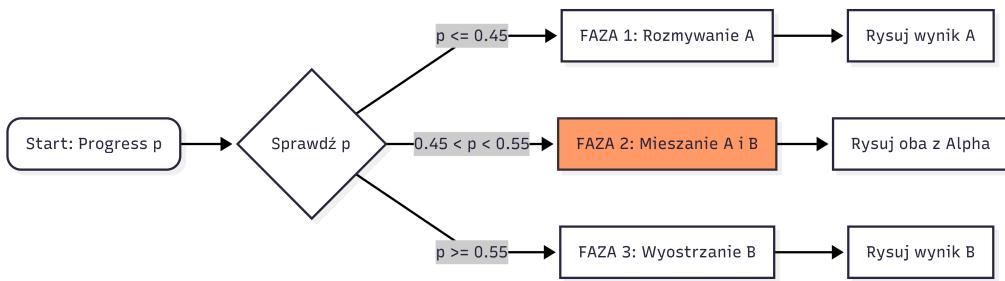
6.7.3 Obsługa w pętli renderującej (Case 11 i 14)

W funkcji `RenderTransitionFrame` zaimplementowano logikę sterowania tymi efektami:

- **Luma Wipe (Case 14):** Wywołuje funkcję CPU, a następnie skaluje wynikową teksturę do rozmiaru okna, co rozwiązuje problem dopasowania obrazów

o różnych rozdzielczościach.

- **Blur Fade (Case 11):** Implementuje trójfazowe przejście:
 1. Rozmywanie pierwszego obrazu (promień rośnie).
 2. Mieszanie (Alpha Blending) dwóch maksymalnie rozmytych obrazów.
 3. Wyostrzanie drugiego obrazu (promień maleje).



Rysunek 8: Diagram sterowania fazami w efekcie Blur Fade

7 Testowanie

Celem procedury testowej było sprawdzenie stabilności aplikacji oraz analiza płynności animacji (klatek na sekundę — FPS) w zależności od rozdzielczości przetwarzanych obrazów. Wszystkie testy zostały przeprowadzone **metodą manualną**, polegającą na bezpośredniej interakcji z interfejsem graficznym użytkownika, ręcznym wybieraniu różnych zestawów plików graficznych z biblioteki mediów oraz bieżącej obserwacji parametrów wydajnościowych wyświetlanych w panelu ustawień.

7.1 Zestawy danych testowych

Do testów przygotowano trzy zestawy obrazów o różnych stopniach złożoności:

- Zestaw A (Low): 640x480 px — standardowa niska rozdzielczość.
- Zestaw B (Medium): 1920x1080 px — natywna rozdzielczość projektu.
- Zestaw C (High): 6000x4000 px — bardzo duża rozdzielczość (test obciążeniowy).

7.2 Wyniki testów wydajnościowych

Poniższa tabela przedstawia średnią liczbę klatek na sekundę zarejestrowaną podczas podglądu animacji w trybie Release:

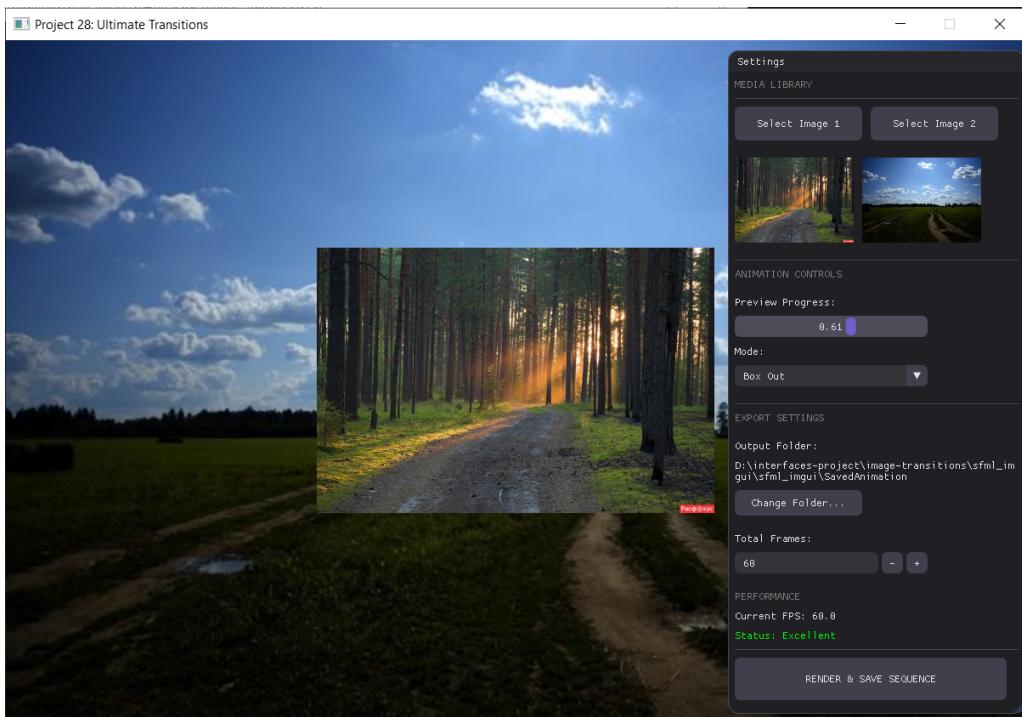
Grupa	Metody przejścia	Zestaw A	Zestaw B	Zestaw C
Transformacje Geometryczne	Slide (Left, Right, Top, Bottom), Box (In, Out)	60 FPS	59-60 FPS	58-60 FPS
Mieszanie Kolorów (Blending)	Fade to Black, Cross-Fade	60 FPS	59-60 FPS	58-60 FPS
Złożone Przekształcenia 2D/3D	3D Cube Rotation, Ring, Page Turn (H/V), Shutter Open, Fly Away	60 FPS	59-60 FPS	58-60 FPS
Przetwarzanie Rastrowe	Blur Fade, Luma Wipe	60 FPS	58-60 FPS	15-30 FPS

Tabela 3: Wyniki wydajnościowe dla różnych grup przejść.

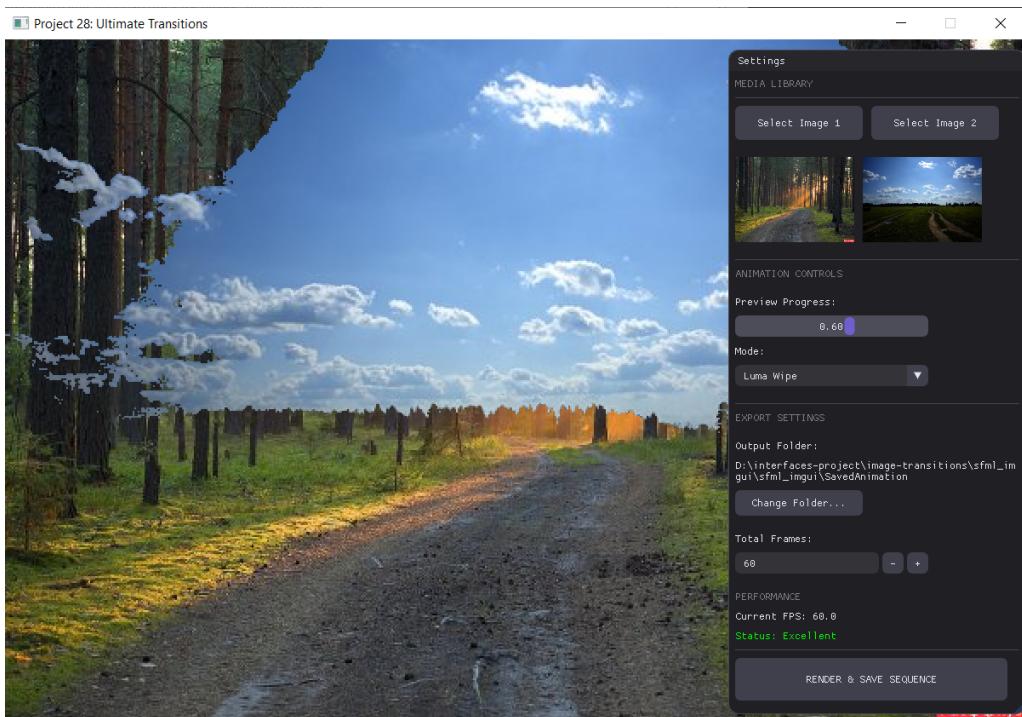
7.3 Analiza wyników

- Funkcje geometryczne (Slide, Box, Fly Away):** Wykazują stałą, idealną płynność 60 FPS niezależnie od rozdzielczości. Jest to wynikiem wykorzystania sprzętowej akceleracji transformacji wierzchołków przez bibliotekę SFML, co niemal całkowicie odciąża procesor główny (CPU).
- Funkcje rastrowe (Luma Wipe, Blur Fade):** Są najbardziej wymagająco operacjami, ponieważ wymagają bezpośredniego dostępu do każdego piksela w pamięci RAM i przesłania zmodyfikowanych danych do karty graficznej w każdej klatce.
- Skuteczność optymalizacji:** Dzięki zastosowaniu technik **Downsamplingu** oraz **Luma Caching**, aplikacja utrzymuje stabilne 60 FPS (Status: Excellent) dla natywnej rozdzielczości projektu (Zestaw B), co było głównym celem optymalizacji.

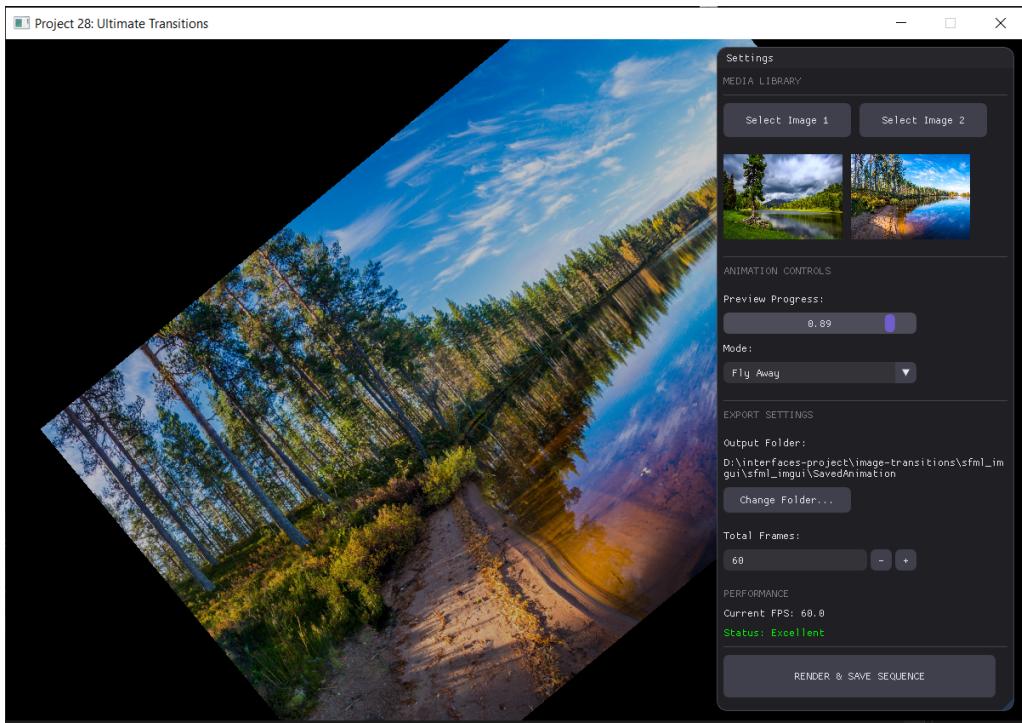
7.4 Wyniki testów



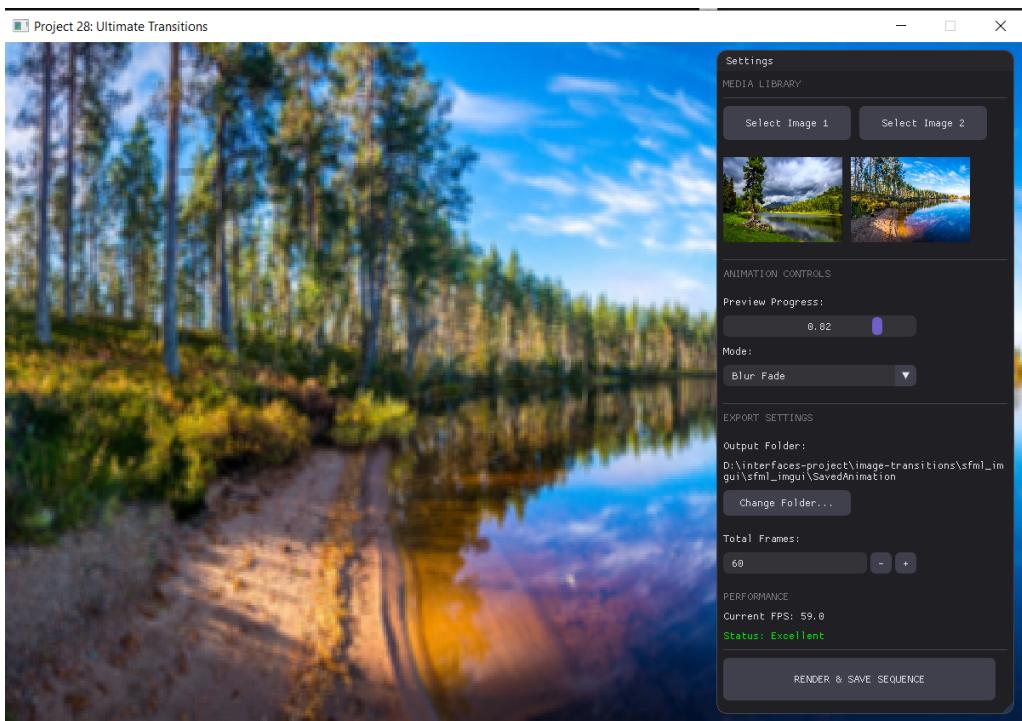
Rysunek 9: Wynik wydajności niskiej rozdzielczości z efektem Box Out



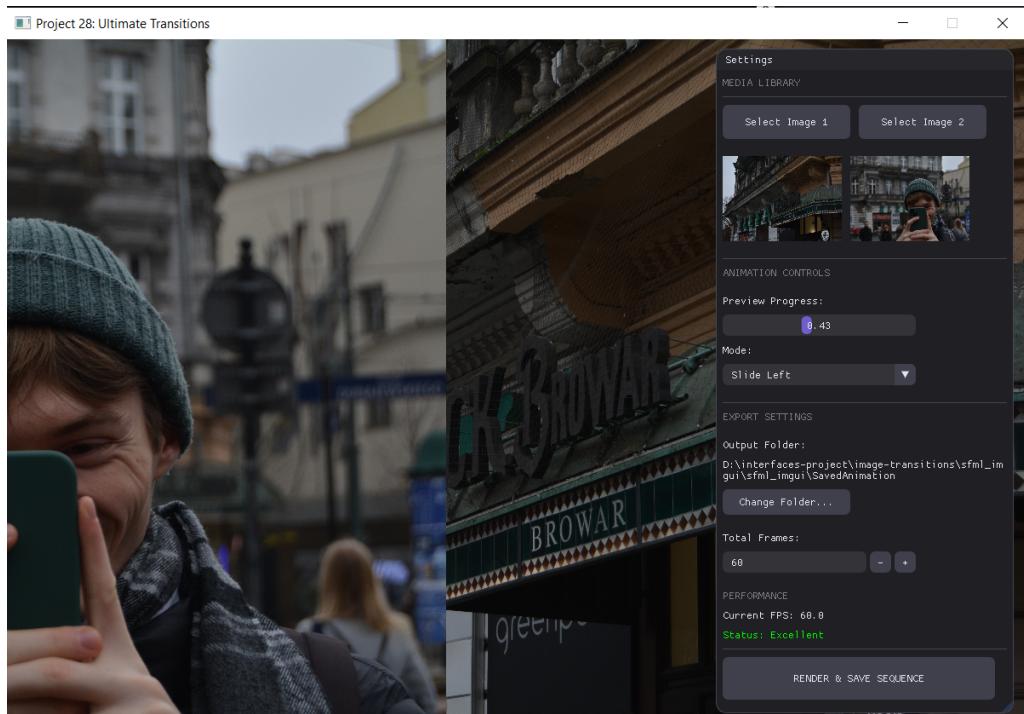
Rysunek 10: Wynik wydajności niskiej rozdzielczości z efektem Lume Wipe



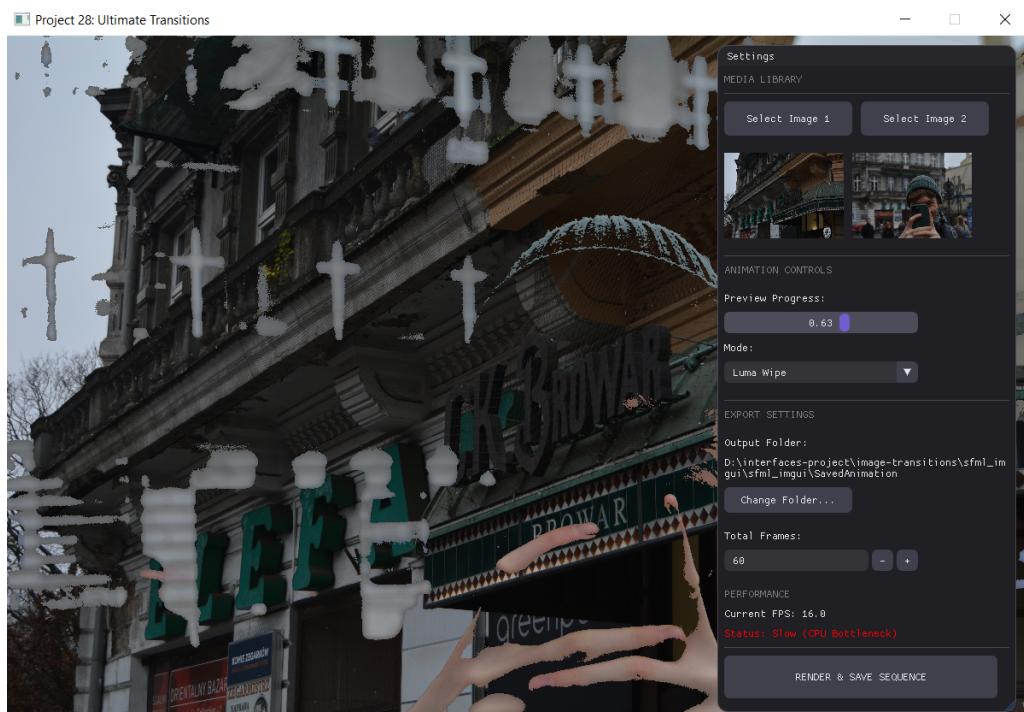
Rysunek 11: Wynik wydajności średniej rozdzielczości z efektem Fly Away



Rysunek 12: Wynik wydajności średniej rozdzielczości z efektem Blur Fade



Rysunek 13: Wynik wydajności dużej rozdzielczości z efektem Slide Left



Rysunek 14: Wynik wydajności dużej rozdzielczości z efektem Lume Wipe

8 Wdrożenie, raport i wnioski

8.1 Uruchomienie z niezależnymi danymi

W celu weryfikacji stabilności programu przeprowadzono testy z użyciem danych niezależnych (zewnętrznych). Wykorzystano mechanizm `OpenFileDialog` do wczytania losowych grafik pobranych z Internetu oraz zdjęć z aparatu cyfrowego, których programista nie uwzględnił na etapie pisania kodu.

Testy przebiegły pomyślnie pod kątem logicznym – program poprawnie interpretował różne formaty plików (.jpg, .png, .bmp) i nie ulegał awarii. Zaobserwowano jednak istotną zależność wydajnościową:

- **Obrazy małe i średnie (do Full HD):** Wszystkie animacje działały płynnie (60 FPS).
- **Obrazy o wysokiej rozdzielczości (4K i większe):** Podczas testowania efektu „Luma Wipe” zauważono wyraźne zacinanie się animacji (spadek płynności).

Stwierdzono, że płynność tego konkretnego efektu jest ściśle skorelowana z rozdzielczością obrazów wejściowych – im większa bitmapa, tym większe opóźnienia w generowaniu klatek podglądu.

8.2 Raport z wykonania

Projekt został zrealizowany zgodnie z harmonogramem, dostarczając funkcjonalną aplikację okienkową. Poniżej przedstawiono analizę sukcesów oraz napotkanych ograniczeń technicznych.

Co się udało:

- Zaimplementowano bezbłędny system wczytywania plików zewnętrznych przy użyciu natywnego API Windows.
- Większość efektów (Slide, Zoom, Cube Rotate, Ring) działa idealnie płynnie niezależnie od danych wejściowych, dzięki oparciu ich o geometrię (przekształcenia wierzchołków) a nie o edycję pikseli.
- System eksportu plików działa poprawnie – nawet jeśli podgląd Luma Wipe się zacinał, wyrenderowane pliki wyjściowe na dysku są płynne, ponieważ rendering offline nie jest ograniczony czasem rzeczywistym.

Co wymaga poprawy (analiza problemu Luma Wipe):

- Efekt „Luma Wipe” opiera się na analizie jasności każdego piksela z osobna. W obecnej implementacji operacja ta wykonywana jest na procesorze głównym (CPU).

- Usunięcie narzutu wielowątkowości: Analiza wykazała, że narzut związanego z tworzeniem i synchronizacją wątków (std::async) 60 razy na sekundę był większy niż zysk z paralelizacji. Zastąpiono go zoptymalizowaną pętlą jednowątkową.

8.3 Wnioski i możliwości rozwoju

Realizacja projektu pozwoliła zidentyfikować kluczowe różnice między przetwarzaniem grafiki na CPU a GPU. Aby wyeliminować problem zacinania się efektów przy dużych rozdzielczościach, w przyszłych wersjach oprogramowania należy wprowadzić istotne zmiany architektoniczne.

Planowane usprawnienia na przyszłość:

1. **Implementacja Shaderów (GLSL):** To najważniejsza zmiana. Przeniesienie algorytmu Luma Wipe oraz Blur z C++ (CPU) do języka shaderów (GPU) całkowicie wyeliminowałoby problem zacinania. Karta graficzna jest przystosowana do równoległego przetwarzania milionów pikseli i zrobiłaby to ułamek sekundy, niezależnie od rozdzielczości obrazu.
2. **Bezpośredni zapis wideo:** Zintegrowanie biblioteki FFmpeg pozwoliłoby użytkownikowi otrzymać gotowy plik .mp4 zamiast sekwencji obrazów, co zwiększyłoby wygodę użytkowania.
3. **Edytor krzywych czasu:** Dodanie nieliniowego sterowania czasem (np. easing functions) sprawiłoby, że proste animacje wjazdów wyglądałyby bardziej profesjonalnie i dynamicznie.

Podsumowując, program spełnia swoje zadanie jako generator klatek, a zidentyfikowane problemy wydajnościowe dotyczą jedynie podglądu w czasie rzeczywistym na bardzo dużych plikach i są możliwe do rozwiązania poprzez zmianę technologii renderowania na shadery.

9 Źródła

1. LERP - Wikipedia
2. Przekształcenie afiniczne - Wikipedia
3. Painter's algorithm - Wikipedia