AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

KATEDRA FIZYKI I MATERII SKONDENSOWANEJ

Projekt dyplomowy

Algorytm optymalizacji siatki modeli 3D
The remeshing algorithm for 3D models

Autor: Arkadiusz Trojanowski
Kierunek studiów: Informatyka Stosowana
Opiekun pracy: dr inż. Ireneusz Bugański

Kraków, 2023

Spis treści

1 Wstęp 5
1.1 Cele projektu 6
2 Teoria 6
2.1 Omówienie zagadnień teoretycznych 6
2.2 Przegląd algorytmów optymalizacji siatki 8
2.2.1 Optymalizacja siatki przy pomocy centroidalnej tesselacji Woronoja 8
2.2.2 Siatka modelu na bazie trójkątów równobocznych 9
2.2.3 Błąd średniokwadratowy jako warunek minimalizacji 10
2.2.4 Izotropowa optymalizacja siatki 11
2.3 Opis użytego języka, technologii i bibliotek 12
2.3.1 Język C++ 12
2.3.2 Visual Studio 13
2.3.3 OpenGL 13
2.3.4 Biblioteka GLEW 13
2.3.5 Biblioteka GLFW 13
2.3.6 Biblioteka GLM 13
2.3.7 Biblioteka Dear ImGui 14
3 Opis implementacji 14
3.1 Klasa `Camera` 14
3.2 Klasy elementów siatki 15
3.2.1 Klasa `Vertex` 15
3.2.2 Klasa `SimpleVertex` 15
3.2.3 Klasa `Pair` 15
3.2.4 Klasa `Face` 15
3.3 Klasa `Shader` 16
3.3.1 Pliki GLSL 16
3.4 Klasa `Material` 17
3.5 Klasa `ObjLoader` 17
3.6 Klasa `Mesh` 17
3.7 Klasa `Model` 23
3.8 Klasa `App` 23
3.9 Klasa `Gui` 25
4 Przykłady działania aplikacji 27
4.1 Poruszanie się po aplikacji 27

4.2 Przykładowe wyniki algorytmów optymalizacyjnych 32
4.2.1 Czasy przetwarzania modelu kapibary 33
4.2.2 Czasy przetwarzania modelu ptaka 34
4.2.3 Czasy przetwarzania modelu kota 36
5 Konkluzja 38
5.1 Pomysły na dalszy rozwój aplikacji 38
5.2 Wniosek końcowy 39
6 Załącznik 40
Bibliografia 41

1 Wstęp

Wobec postępującej technologii, w celu oddania realizmu, trójwymiarowe modele posiadają coraz więcej wierzchołków. Modele powstałe poprzez operacje skanowania rzeczywistych obiektów charakteryzują się siatką złożoną z milionów wierzchołków. Często ich ilość w takiej wczytanej siatce[1] (ang. *mesh*) jest zbyt duża, co objawia się wysokim poziomem wykorzystywanej pamięci komputera czy nadmiernym obciążeniem procesora podczas ich przetwarzania (patrz: rys. 1.1). Tymczasem efekt wizualny dla podobnego modelu o mniejszej liczbie ścian – na przykład w grze wideo czy animacji – byłby niemal niedostrzegalny.

Dlatego wciąż aktualnym i istotnym procesem jest optymalizacja siatki 3D. Dla zaoszczędzenia zasobów komputera, modele powinny posiadać niską liczbę wielokątów (inaczej poligonów, z ang. *polygon*) przy zachowaniu oryginalnej jakości[2] modelu. W tym celu powstało już wiele algorytmów[3], ale wciąż potrzebne są nowe, coraz wydajniejsze i dokładniejsze.

Rysunek 1.1: Fragmenty obszaru edycji w ogólnodostępnym programie Blender w trybie *rendered*, po prawej z dodatkową opcją rysowania siatki. Niewspółmiernie duża ilość wierzchołków modelu negatywnie wpływa na szybkość wprowadzania zmian czy czas renderowania[4] animacji. Źródło: [3]

[1] Siatka wielokątów to struktura powstała ze zbiorów wierzchołków, krawędzi i wielokątowych powierzchni (ang. *faces*) [1].

[2] „Jakość" modelu rozumiemy jako posiadanie przezeń na tyle dużo detali, że utrzymuje on przekonujący poziom realizmu [2].

[3] Temat rozwinięty w sekcji 2.2.

[4] Tu: *rendering* - tworzenie statycznego obrazu 2D na podstawie trójwymiarowej sceny. Ciąg takich obrazów tworzy animację 3D.

1.1 Cele projektu

Celem projektu było stworzenie aplikacji wczytującej plik w formacie OBJ (*Wavefront Object file*) i poddającej załadowany model operacjom przetwarzania za pomocą wybranego algorytmu optymalizacji siatki, generując nową siatkę zachowującą topologię oryginału. Program powinien również umożliwić dobór gęstości punktów nowej siatki. Dodatkowym zadaniem było postawienie na generowaniu modelu złożonego z czworokątów, a nie dużo powszechniejszych trójkątów. Autor miał też możliwość wyboru dowolnej technologii i języka programowania w celach napisania aplikacji.

2 Teoria

2.1 Omówienie zagadnień teoretycznych

Ze względu na poruszanie się w przestrzeni trójwymiarowej w aplikacji, konieczne jest zrozumienie operacji na wektorach i działania macierzy transformacji [4].

Układ, w którym wykonywane są obliczenia w tym projekcie, to kartezjański układ współrzędnych X, Y oraz Z. Wersory układu kartezjańskiego są do siebie wzajemnie prostopadłe.

Iloczyn skalarny dwóch wektorów to rezultat pomnożenia przez siebie ich długości oraz cosinusa kąta między nimi:

$$\boldsymbol{v}_1 \circ \boldsymbol{v}_2 = \|\boldsymbol{v}_1\| \cdot \|\boldsymbol{v}_2\| \cdot cos\,\theta, \tag{2.1}$$

gdzie $\boldsymbol{v}_1$ – wektor o współrzędnych $\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$, $\boldsymbol{v}_2$ – wektor o współrzędnych $\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$. Ponieważ cosinus kąta prostego oraz półpełnego daje wartości odpowiednio 0 i 1 (lub -1), można w ten sposób łatwo sprawdzać, czy wektory są do siebie prostopadłe, ewentualnie równoległe (jeśli mają długości jednostkowe). W niniejszym projekcie iloczyn skalarny wykorzystywany jest do obliczenia współrzędnych punktów płaszczyzny zawierającej dany poligon siatki.

Iloczyn wektorowy przyjmuje na wejściu dwa wektory $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ i produkuje wektor do nich prostopadły, więc operacja działa tylko w przestrzeni trójwymiarowej. Jego współrzędne można pozyskać ze wzoru:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 \cdot z_2 - z_1 \cdot y_2 \\ z_1 \cdot x_2 - x_1 \cdot z_2 \\ x_1 \cdot y_2 - y_1 \cdot x_2 \end{bmatrix}. \tag{2.2}$$

Powstały w ten sposób wektor służy w projekcie do przesuwania kamery na podstawie wektorów frontowego i tzw. wektora *world-up*[5] (sekcja 3.1).

W trzech wymiarach do przemieszczania, zmieniania rozmiarów i obracania wektorów wykorzystywane są macierze transformacji (przekształcenia) o rozmiarze 4 na 4, powstające ze złożenia macierzy skalowania, translacji (przemieszczenia) i rotacji (obrotu). Wektor wskazujący położenie punktu również ma rozmiar 4, gdzie ostatnia współrzędna wynosi 1, aby

[5] Wektor jednostkowy, który powinien mieć współrzędne $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$, a więc wskazywać dodatni zwrot zgodnie z pionową osią Z.

można było dokonać przemieszczenia. Do wykonania operacji transformacji należy taki wektor (pionowy) pomnożyć z lewej strony przez macierz transformacji. Najprostszą taką macierzą jest macierz jednostkowa, przez którą pomnożenie wektora zwraca ten sam wektor.

Operacja translacji o wektor $\begin{bmatrix} T_x \\ T_y \\ T_z \\ 1 \end{bmatrix}$ wykonana na wektorze $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ daje w wyniku:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{bmatrix}. \tag{2.3}$$

Operacja skalowania wykonana na tym samym wektorze:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot S_x \\ y \cdot S_y \\ z \cdot S_z \\ 1 \end{bmatrix}. \tag{2.4}$$

Operacja rotacji wygląda inaczej dla różnych osi, wokół których obracany jest punkt. Dla obrotu wektora wokół osi X:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{bmatrix}, \tag{2.5}$$

wokół osi Y:

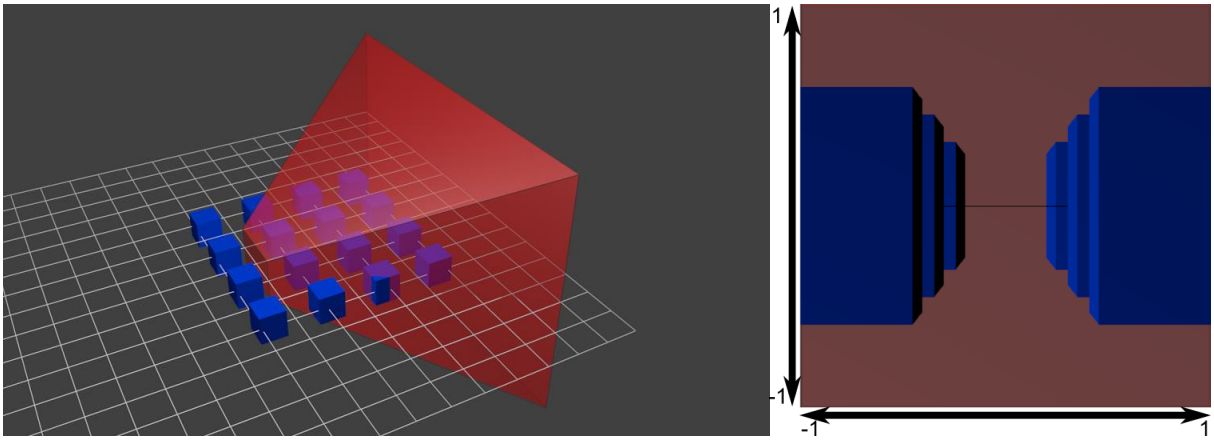
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{bmatrix} \tag{2.6}$$

oraz wokół osi Z:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{bmatrix}. \tag{2.7}$$

Kwestią kluczową przy działaniu programów wykorzystujących grafikę 3D jest fakt, iż wbrew pozorom to nie użytkownik (kamera) przemieszcza i obraca się w przestrzeni, ale to wszystkie obiekty zmieniają położenie względem niego, a dokonuje się tego poprzez zastosowanie tzw. macierzy widoku (ang. *view matrix*) [5]. W świecie rzeczywistym przemieszczanie obiektu w celach dokonania obserwacji jego powierzchni z innego punktu jest często niepraktyczne, a nawet niemożliwe, jednak w grafice komputerowej jest to trywialne proste. Najłatwiej jest wyobrazić sobie chęć przesunięcia kamery w wybranym kierunku. W tym celu należy dokonać translacji wszystkich trójwymiarowych struktur, wykorzystując macierz widoku, ustawiając w niej na odpowiednich osiach wartości przeciwne, co da pożądany efekt.

W celu wyświetlania i wprowadzenia perspektywy wymagana jest ponadto macierz projekcji (ang. *projection matrix*) [6]. Można otrzymać ją[6] poprzez nadanie na wejściu wartości FOV (*field of view*, pole widzenia) w radianach (im niższy FOV, tym bardziej kamera „skupia się" na centralnym punkcie obrazu), współczynnik proporcji szerokości do wysokości obrazu (ang. *aspect ratio*) oraz dwie wartości oznaczające odległości prostopadłych do wektora frontowego kamery płaszczyzn ograniczających zasięg rysowania od punktu centralnego. Przykład wykorzystania macierzy projekcji zamieszczono na rys. 2.1.



Rysunek 2.1: Zastosowanie macierzy projekcji. Na fragmencie po lewej stronie na czerwono zaznaczono obszar zasięgu rysowania sceny, z widocznym kątem tworzącym FOV oraz dwie prostopadłe płaszczyzny wskazujące początek i koniec rysowania. Po prawej efekt zastosowania macierzy projekcji. Źródło: [6]

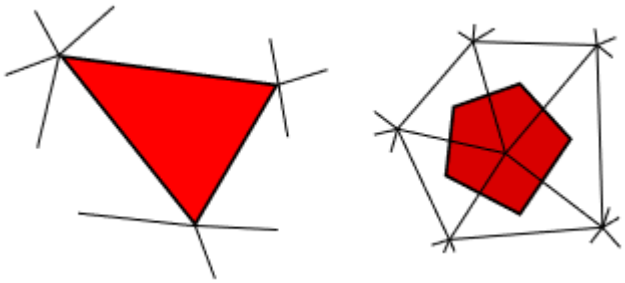
2.2 Przegląd algorytmów optymalizacji siatki

Algorytmów optymalizacji siatki 3D jest bardzo dużo i poza innymi strategiami i wewnętrznymi procedurami, rezultaty ich użycia różnią się między sobą szybkościami wykonywania i podejściem do jakości modelu wyjściowego. Mogą one skupiać się na poprawianiu jednorodności przestrzennego rozkładu wierzchołków, zunifikowaniu długości krawędzi modelu, usuwaniu wierzchołków tworzących zbyt małe lub zbyt duże kąty w poligonach, czy po prostu zmniejszania liczby elementów siatki w celu redukcji zużycia pamięci komputera [7]. Warto przyjrzeć się kilku popularniejszym z nich.

2.2.1 Optymalizacja siatki przy pomocy centroidalnej tesselacji Woronoja

Algorytm ten [8] bazuje na dzieleniu (klastrowaniu) siatki modelu wejściowego w celach wydajnego rozdysponowania zadanej ilości nowych wierzchołków. Można wybrać, czy klastry mają być trójkątami siatki, czy wielokątami dla każdego wierzchołka, z bokami przechodzącymi przez środki odpowiednich jemu krawędzi (rys. 2.2, bardziej polecana jest druga opcja ze względu na około dwukrotnie mniejszą ilość węzłów siatek, niż poligonów).

[6] Temat rozwinięty w sekcji 3.8.



Rysunek 2.2: Po lewej klastrowanie trójkąta (potrzebne regiony stają się nim samym); po prawej klastrowanie wierzchołka [8]

Optymalizacja CVD (*Centroidal Voronoi Diagrams*) polega na minimalizacji energii E danej analitycznym wzorem:

$$E = \sum_{i=1}^{n} \int_{C_i} \rho(\boldsymbol{x}) \|\boldsymbol{x} - \boldsymbol{z}_i\|^2 d^3\boldsymbol{x}, \tag{2.8}$$

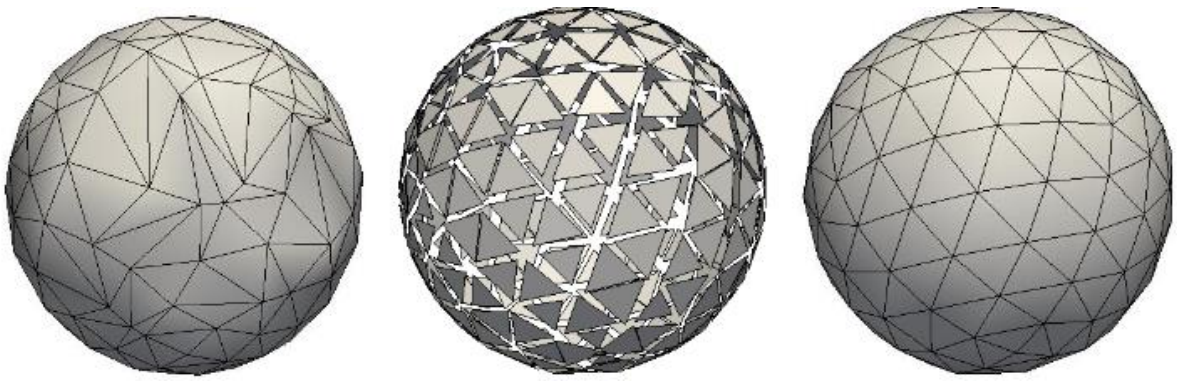
gdzie $\boldsymbol{z}_i$ to tzw. wektor środka masy klastra, który można policzyć ze wzoru:

$$\boldsymbol{z}_i = \frac{\int_{C_i} \boldsymbol{x} \cdot \rho(\boldsymbol{x}) d^3\boldsymbol{x}}{\int_{C_i} \rho(\boldsymbol{x}) d^3\boldsymbol{x}}, \tag{2.9}$$

C_i – klaster o indeksie i, $\rho(\boldsymbol{x})$ – funkcja gęstości.

2.2.2 Siatka modelu na bazie trójkątów równobocznych

Trójkątne siatki mogą zostać uznane za niskojakościowe ze względu na obecność wielu trójkątów o silnie różniących się długościach boków. Tymczasem trójkąty równoboczne pozytywnie wpływają na dokładność symulacji numerycznych np. w metodach elementów skończonych. Jednym z algorytmów pozwalających uzyskać taką siatkę jest *As-equilateral-as-possible* (AEAP) *surface remeshing* [9], którego działanie przedstawiono na rys. 2.3.



Rysunek 2.3: Upraszczanie siatki procedurą AEAP. Od lewej: oryginalna siatka; przygotowane trójkąty równoboczne na powierzchni; rezultat remeshingu [9]

Działanie algorytmu wykonuje się w kilku punktach. Pierwszym z nich jest krok lokalny, w którym dla każdego trójkąta $\boldsymbol{x}_t$ (gdzie t – nr trójkąta) znajduje się odpowiedni mu

trójkąt równoboczny o tym samym polu powierzchni. Najpierw porównuje się go do jednostokowego trójkąta równobocznego $\boldsymbol{u}_t$, by uzyskać macierz transformacji $\boldsymbol{T}_t$, a następnie pozyskiwana jest z niej macierz rotacji $\boldsymbol{R}_t$ przy pomocy SVD (*singular value decomposition*, rozkład głównych składowych [10]). Na końcu z pól powierzchni uzyskiwana jest wartość skali, co pozwala wyliczyć najbliższy $\boldsymbol{x}_t$ trójkąt równoboczny $\boldsymbol{e}_t$ oraz lokalną macierz transformacji:

$$\boldsymbol{L}_t = e_t \cdot \boldsymbol{x}_t^{\dagger}, \tag{2.10}$$

gdzie $\boldsymbol{x}_t^{\dagger}$ jest pseudo-odwróconym[7] trójkątem $\boldsymbol{x}_t$.

W kroku globalnym rozmieszczamy uzyskane trójkąty równoboczne odpowiednio na siatce, uzyskując współrzędne nowego trójkąta $\boldsymbol{x}_t'$ za pomocą wzoru:

$$\boldsymbol{x}_t' \cdot \boldsymbol{x}_t^T \cdot \left(\boldsymbol{x}_t^{\dagger}\right)^T + \boldsymbol{x}_t^T \cdot \left(\boldsymbol{x}_t^{\dagger}\right)^T \cdot (\boldsymbol{x}_t')^T = \frac{\boldsymbol{L}_t + \boldsymbol{L}_t^T}{A_t}, \tag{2.12}$$

gdzie A_t – powierzchnia trójkąta o numerze t. W ten sposób uzyskujemy strukturę odpowiadającą środkowej siatce na rys. 2.3.

Aby uzupełnić luki w modelu, należy przemieścić wierzchołki tak, by wypełnić model.

2.2.3 Błąd średniokwadratowy jako warunek minimalizacji

Surface simplification using quadric error metrics [2] bazuje na iteratywnym łączeniu par wierzchołków – krawędzi lub węzłów z ustaloną między nimi przez użytkownika maksymalną odległością będącą progiem (ang. *threshold*), jak przedstawiono na rys. 2.4.

Rysunek 2.4: Operacje zwierania krawędzi (po lewej) i par wierzchołków w bliskiej odległości (po prawej) [2]

Na początku pracy algorytmu należy obliczyć symetryczną macierz $\boldsymbol{Q}$ dla każdego wierzchołka $\boldsymbol{v}$ bazując na jego błędzie aproksymacji powierzchni (Δ), na którego podstawie oceniany jest stan geometryczny modelu:

[7] Pseudo-odwrotną macierz $A^{\dagger}$ liczymy za pomocą wzoru:

$$A^{\dagger} = (A^T \cdot A)^{-1} \cdot A^T \text{ [9]}. \tag{2.11}$$

$$\begin{aligned} \boldsymbol{Q} = \Delta(\boldsymbol{v}) = \Delta([x \quad y \quad z \quad 1]^T) = \\ = \sum_{\boldsymbol{p} \in powierzchnie(v)} (\boldsymbol{p}^T \cdot \boldsymbol{v})^2 = \\ = \boldsymbol{v}^T \cdot \left(\sum_{\boldsymbol{p} \in powierzchnie(v)} \boldsymbol{K_p}\right) \cdot \boldsymbol{v}, \end{aligned} \tag{2.13}$$

gdzie $\boldsymbol{p} = [a \quad b \quad c \quad d]^T$ reprezentuje powierzchnię zdefiniowaną przez równanie $ax + by + cz + d = 0$ (gdzie $a^2 + b^2 + c^2 = 1$), a $\boldsymbol{K_p}$ jest macierzą o wartościach:

$$\boldsymbol{K_p} = \boldsymbol{p} \cdot \boldsymbol{p}^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}. \tag{2.14}$$

Sumując takie macierze otrzymujemy pojedynczą macierz $\boldsymbol{Q}$. Wszystkie struktury w ten sposób zainicjalizowane wystarczą do udanego przetworzenia algorytmu – w przypadku zwarcia wierzchołków, nowy węzeł jako $\boldsymbol{Q}$ przyjmie sumę macierzy ze składających się na niego węzłów.

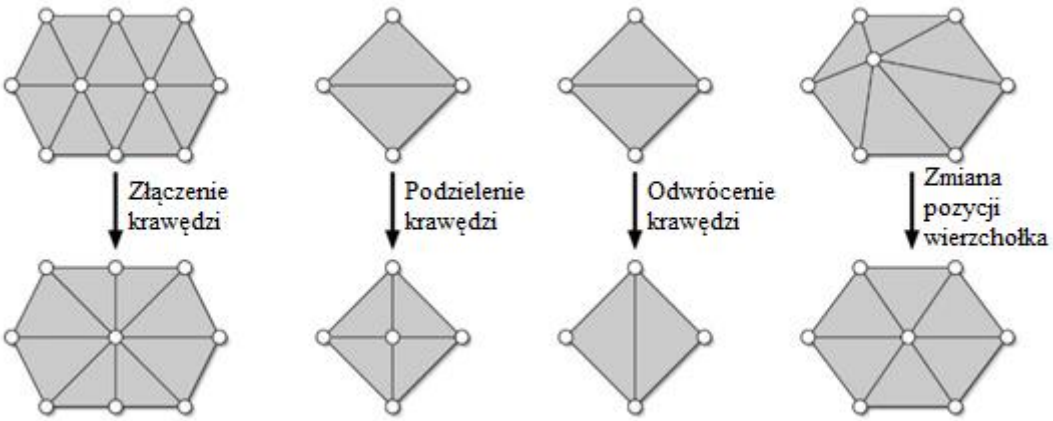
Po wydzieleniu wszystkich par $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ w siatce (krawędzi i wierzchołków będących w niewielkiej odległości) wyznaczana jest pozycja nowego wierzchołka $\boldsymbol{v}$ (na przykład z użyciem wartości średniej) i obliczany jest koszt (błąd) dla każdej pary ze wzoru:

$$cost = \boldsymbol{v}^T \cdot (\boldsymbol{Q}_1 + \boldsymbol{Q}_2) \cdot \boldsymbol{v}. \tag{2.15}$$

Następnie para o najniższym koszcie zostaje usunięta, a koszty wszystkich pozostałych, zaangażowanych w operację par – zaktualizowane. Algorytm wykonuje liczbę iteracji redukcji par zgodnej z wymaganiem użytkownika.

2.2.4 Izotropowa optymalizacja siatki

Przewagą tego algorytmu nad innymi jest łatwość jego implementacji [11]. Główną jego funkcją jest dążenie do zrównania długości wszystkich krawędzi oraz wartościowości (ilości sąsiadów) każdego wierzchołka. Izotropowa optymalizacja siatki składa się z kilku kroków, przedstawionych na rys. 2.5.



Rysunek 2.5: Operacje wykonywane podczas pracy algorytmu izotropowego. Od lewej: zwieranie krawędzi, dzielenie krawędzi, odwracanie krawędzi, przemieszczenie wierzchołka. Źródło: [11]

Na początku użytkownik ustala docelową długość krawędzi L, na której podstawie wyliczane są długości minimalna i maksymalna, wynoszące odpowiednio $\frac{4}{5}L$ oraz $\frac{4}{3}L$. Następie wykonywane są odpowiednie procedury w iteracjach, na wszystkich parach w siatce.

Łączenie krawędzi występuje w przypadku, gdy krawędź jest krótsza niż $\frac{4}{5}L$ i działa identycznie, jak w algorytmie z sekcji 2.2.3.

Dzielenie krawędzi występuje, gdy krawędź jest dłuższa niż $\frac{4}{3}L$. Tworzy nowy węzeł między dwoma trójkątami (dokładnie w połowie długości krawędzi) i dodaje nowe połączenia z wolnymi wierzchołkami tych trójkątów.

Wartość *deviation* (odchylenie), na podstawie której określa się, czy krawędź powinno się odwrócić, liczona jest ze wzoru:

$$\begin{aligned} deviation = {} & |valence(a) - target(a)| + \\ & |valence(b) - target(b)| + \\ & |valence(c) - target(c)| + \\ & |valence(d) - target(d)|, \end{aligned} \tag{2.16}$$

gdzie a, b, c, d – wierzchołki w trójkącie, $valence$ – aktualna wartościowość wierzchołka (czyli liczba jego sąsiadów), $target$ – doskonała wartościowość wierzchołka, do której dąży algorytm wynosząca 6 dla węzła wewnętrznego i 4 dla węzła na granicy modelu. Jeżeli po odwróceniu krawędzi $deviation$ jest mniejsze niż przed odwróceniem, zmiana jest zachowywana, w przeciwnym wypadku operacja jest cofana.

W celu wygładzania (ang. *smoothing*) siatki można dokonać relaksacji stycznej. Nowa pozycja wierzchołka $\boldsymbol{p}$ (oznaczona jako $\boldsymbol{p}'$) obliczana jest wykorzystując wzór:

$$\boldsymbol{p}' = \boldsymbol{q} + \boldsymbol{n} \cdot \boldsymbol{n}^T(\boldsymbol{p} - \boldsymbol{q}), \tag{2.17}$$

gdzie $\boldsymbol{n}$ – wektor normalny węzła $\boldsymbol{p}$, $\boldsymbol{q}$ – pozycja liczona ze wzoru:

$$\boldsymbol{q} = \frac{1}{|N(\boldsymbol{p})|} \cdot \sum_{\boldsymbol{p}_j \epsilon N(\boldsymbol{p})} \boldsymbol{p}_j, \tag{2.18}$$

$N(\boldsymbol{p})$ – liczba wszystkich wierzchołków.

2.3 Opis użytego języka, technologii i bibliotek

2.3.1 Język C++

Języki programowania służą do przekazywania maszynie poleceń, przy dostarczeniu zestawu pojęć, które wykorzystuje programista do wyrażania swoich myśli. Język C pozwala na bezpośrednie odwzorowanie wbudowanych operacji i typów na elementy sprzętu, co pozwala na efektywne wykorzystanie pamięci. W 1985 r. Bjarne Stroustrup opracował język C++, zwany „C z klasami”. Jest to język programowania ogólnego przeznaczenia – nie jest wyspecjalizowany w żadnym obszarze, ale jest zaprojektowany z myślą o wykorzystywaniu go w jak największej liczbie dziedzin. Rozwinął C o takie elementy jak klasy, konstruktory, destruktory, wyjątki czy szablony. C++ wprowadził lekkie i elastyczne mechanizmy abstrakcji, dzięki którym możliwe stało się definiowanie własnych typów danych mających taki sam zakres zastosowań i wydajność, co typy wbudowane. C++ jest językiem wysokiego poziomu [12].

C++ wciąż cieszy się statusem jednego z najpopularniejszych na świecie [13].

Obecnym standardem języka jest ISO C++20 [14].

2.3.2 Visual Studio

Visual Studio to opracowane przez firmę Microsoft IDE (*integrated development environment*, zintegrowane środowisko programistyczne), które poza możliwością edycji kodu pozwala na kompilowanie (domyślnie za pomocą kompilatora Microsoftu), linkowanie, debugowanie, a także wydawanie i rozwijanie aplikacji o nowe rozszerzenia [15].

Do zrealizowania niniejszego projektu wykorzystano udostępnioną powszechnie wersję Community 2017.

2.3.3 OpenGL

Open Graphics Library to API (*application programming interface*, interfejs programistyczny aplikacji) pozwalające na renderowanie przy pomocy karty graficznej. Jest to biblioteka wyłącznie niskopoziomowa – nie dostarcza żadnych funkcji obsługujących animacje, czas, wczytywanie i zapisywanie plików czy graficzny interfejs. OpenGL charakteryzuje się stabilnością, niezawodnością oraz łatwością w użyciu. Jest również systematycznie rozwijany. OpenGL wprowadza język GLSL (*OpenGl Shading Language*) służący pisania shaderów[8]. Obecna wersja biblioteki oznaczona jest numerem 4.6 [16].

Autor do zaprojektowania aplikacji wykorzystał najnowszą wersję OpenGL.

2.3.4 Biblioteka GLEW

OpenGL Extension Wrangler Library to opcjonalna biblioteka dostarczająca mechanizmy sprawdzające, które rozszerzenia OpenGL są wspierane na platformie użytkownika. Numer aktualnej wersji to 2.1 [17].

2.3.5 Biblioteka GLFW

Graphics Library Framework to napisane w języku C API wykorzystywane np. przy OpenGL, umożliwiające proste tworzenie okien, powierzchni, wczytywanie plików i obsługę zdarzeń (np. wciśnięcie wybranego przez użytkownika klawisza). Biblioteka umożliwia obsługę więcej niż dwóch okien i monitorów, czy zbiera informacje o wciśnięciu przycisków z klawiatury, myszki i kontrolera do gier wideo. Najnowsza wersja GLFW oznaczona jest numerem 3.3 [18].

2.3.6 Biblioteka GLM

OpenGL Mathematics to biblioteka matematyczna dla C++ dostarczająca klasy i funkcje z tymi samymi konwencjami nazw, co w języku GLSL, ułatwiająca pisanie programów wykorzystujących shadery. GLM implementuje również funkcje matematyczne ułatwiające między innymi transformacje macierzy. Obecna wersja: 0.9.9.9 [19].

[8] Shader – program służący do opisywania własności obiektów. Shadery wykonywane są bezpośrednio na procesorze karty graficznej [22].

2.3.7 Biblioteka Dear ImGui

Jako uzupełnienie aplikacji wykorzystana została biblioteka ImGui. Jest to rozszerzenie C++ służące do tworzenia GUI (*graphics user interface*, graficzny interfejs użytkownika). Sprzyja tworzeniu interfejsów w prosty i wydajny sposób, dlatego nadają się one bardziej do prostych programów lub debugowania, a nie tych obsługiwanych przez użytkownika końcowego (klienta) – ImGui nie posiada wielu funkcji powszechnie obecnych w bibliotekach interfejsów graficznych wyższego poziomu. Numer aktualnej wersji to 1.89 [20].

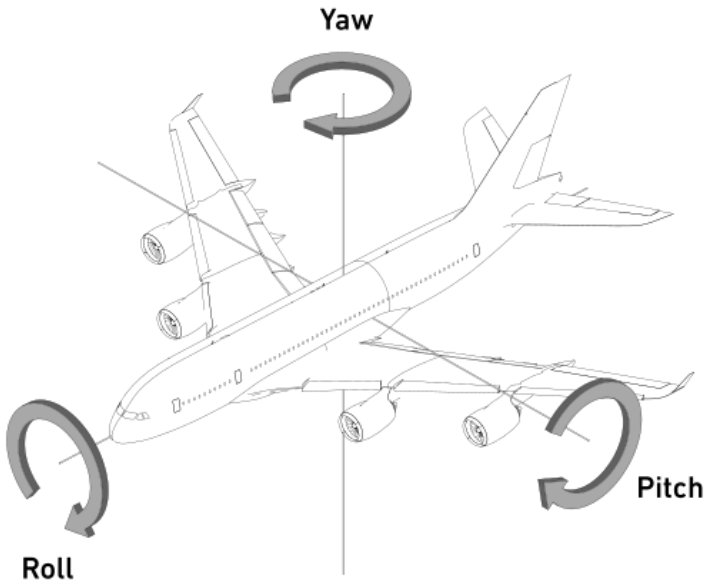
3 Opis implementacji

Aplikacja projektowa została stworzona w całości za pomocą Visual Studio IDE (sekcja 2.3.2) w architekturze 32-bitowej. Potrzebne biblioteki są linkowane dynamicznie. Algorytmy zostały napisane samodzielnie na podstawie treści teoretycznych zawartych w sekcji 2.2.

3.1 Klasa `Camera`

Klasa jest zaprzyjaźniona z klasą `Gui` (sekcja 3.9) ze względu na potrzebę umożliwienia edycji jej parametrów z poziomu interfejsu.

Prywatna metoda `updateCameraVectors` aktualizuje wektory frontowy, boczny i pionowy kamery na podstawie wartości pól `_pitch`, `_yaw` i `_roll` (rys. 3.1) uzyskanych przez obrót kamerą za pomocą myszy.



Rysunek 3.1: Terminy *pitch* (pułap), *yaw* (zmiana kursu) i *roll* („beczka”) związane są z nawigacją samolotów i używane są również w grafice 3D. Wartość *pitch* to obrót wokół osi Y, *yaw* – Z, a *roll* – X. Źródło: [21]

Funkcja wykorzystuje metody biblioteki GLM do obliczenia wartości obrotu w radianach, uzyskania wartości iloczynu wektorowego (sekcja 2.1) oraz normalizacji wektorów kamery.

Metoda `getViewMatrix` aktualizuje wektory kamery i macierz widoku za pomocą funkcji z biblioteki GLM o nazwie `lookAt`, podając na wejściu pozycję kamery, sumę wartości wektora pozycji kamery i frontowego, oraz wektor wskazujący górę „obiektywu”. Następnie uzyskane wartości macierzy są zwracane.

Funkcja `move` służy do poruszania kamerą (choć tak naprawdę zmieniana jest pozycja wszystkich obiektów wokół niej, co jest objaśnione w sekcji 2.1) po naciśnięciu odpowiednich przycisków. Przyjmuje jako wartości wejściowe zmienną *delta time*, oznaczająca czas upłynięty od ostatniego wywołania (zależna od ustawień odświeżania ekranu; dla 60 Hz jest to odpowiednio $\frac{1}{60}$ sekundy) oraz numer oznaczający kierunek, w którym ma się przesunąć.

Metoda `rotate` obraca kamerą po wykonaniu ruchu kursorem myszy przez użytkownika. Wzrost wartości `_pitch` i `_yaw` uzależniony jest od czułości myszy ustawionej w aplikacji przez użytkownika. „Dziób” kamery można obniżyć lub podwyższyć o maksymalnie 90°. Wartość `_roll` nie jest na ten moment zmieniana w aplikacji.

Metoda `reset` zmienia parametry pozycji oraz rotacji kamery do początkowych wartości wynoszących 0. Została zaimplementowania na wypadek „zgubienia” załadowanego modelu poprzez zbyt duże oddalenie kamery.

3.2 Klasy elementów siatki

3.2.1 Klasa `Vertex`

`Vertex` jest strukturą przechowującą wektory pozycji, koloru (wartości RGB – czerwony, zielony i niebieski), koordynaty tekstur oraz normalne wierzchołka. Węzły `Vertex` są wczytywane bezpośrednio z pliku OBJ i rysowane jako oryginalny model.

3.2.2 Klasa `SimpleVertex`

Jest to klasa uproszczonego wierzchołka, który pozbawiony jest niemal wszystkich wektorów klasy `Vertex` oprócz jego lokalizacji w przestrzeni. Została ona napisana na potrzeby optymalizacji siatki. `SimpleVertex` zawiera dodatkowe pola uzupełniane przez program: indeks w celu identyfikacji, macierz kwadratową $\boldsymbol{Q}$ potrzebną w algorytmie z sekcji 2.2.3, zbiór indeksów sąsiadów wierzchołka oraz zbiór indeksów wielokątów zawierających dany wierzchołek.

3.2.3 Klasa `Pair`

Struktura krawędzi w siatce. Posiada pole na unikalny identyfikator, tablicę o rozmiarze 2 przechowującą indeksy zawartych wierzchołków oraz koszt obliczany w trakcie przetwarzania siatki (sekcja 2.2.3).

Obiekty klasy `Pair` mają systematycznie sortowane tablice ID wierzchołków w celu ustawienia mniejszego indeksu węzła w pierwszej kolejności, by nie doprowadzić do „duplikacji” w zbiorze par (krawędź z wierzchołkami przykładowo $[0 \quad 1]$ oraz $[1 \quad 0]$ jest tą samą krawędzią).

3.2.4 Klasa `Face`

Klasa poligonu posiada swój indeks oraz tablicę trzech ID węzłów do niej należących. Na podstawie obiektów struktury `Face` tworzona oraz rysowana jest uproszczona siatka.

3.3 Klasa `Shader`

Shader to program opisujący obiekty 3D, uruchamiany na procesorze karty graficznej [22].

`loadShaderSource` przyjmuje nazwę pliku w formacie GLSL i wczytuje jego zawartość, zapisując ją do łańcucha znaków. Funkcja w przypadku wykrycia niewłaściwej wersji OpenGL w pliku, zmienia ją na poprawny numer.

`loadShader` wykorzystuje funkcję `loadShaderSource` i po podaniu na wejściu typu etapu shadera oraz nazwy pliku GLSL (sekcja 3.3.1) tworzony i kompilowany jest nowy obiekt z nadanym identyfikatorem wykorzystywanym do późniejszego linkowania.

Metoda `linkProgram` tworzy nowy program, któremu nadawane jest ID, a następnie podpinane są do niego stworzone etapy shaderów.

Konstruktor klasy wykorzystuje jej metody do wczytania plików i utworzenia właściwego programu.

Klasa posiada również szereg funkcji przesyłających obliczone w programie wartości skalarne, wektorów i macierzy do utworzonego shadera o wybranym identyfikatorze w celu nadania wartości tzw. uniformom[9].

3.3.1 Pliki GLSL

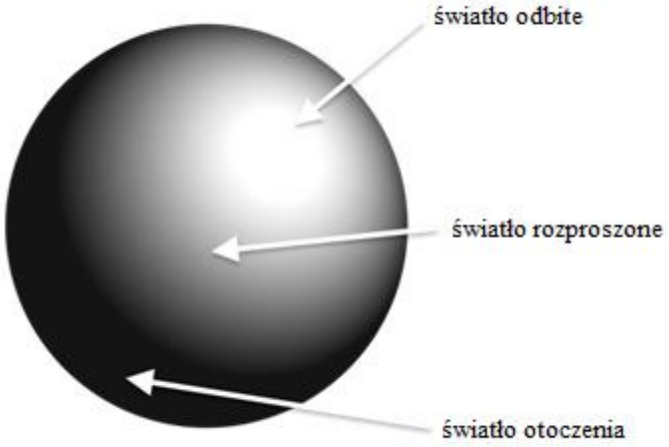
Shader w aplikacji ma dwa typy: *vertex shader* (shader wierzchołków) oraz *fragment shader* (shader fragmentów) [23], a ich kody źródłowe znajdują się w odpowiednich plikach.

Vertex shader przyjmuje na wejściu parametry wierzchołka, a także uniformy w postaci macierzy modelu, macierzy widoku i macierzy projekcji, które następnie są składane w odpowiedniej kolejności, by narysować wierzchołki obiektu w odpowiednich pozycjach względem kamery:

$$pozycja = macierz\ projekcji \cdot macierz\ widoku \cdot macierz\ modelu \cdot [pozycja\ wierzchołka \quad 1].$$

Fragment shader wykorzystywany jest do uzupełnienia obrazu efektami świetlnymi oraz nałożenia ewentualnych tekstur (w niniejszym projekcie te nie są wykorzystywane) na podstawie właściwości materiału (sekcja 3.4). Ten shader używany jest podczas wyświetlania oryginalnego modelu wczytanego z pliku OBJ, zawierającego atrybuty wektorów normalnych siatki. Zawarte są w nim metody wprowadzające *ambient light* (światło otoczenia), *diffuse light* (światło rozproszone) oraz *specular light* (światło odbite) (rys. 3.2).

[9] Uniform – zmienna globalna shadera – parametr, który użytkownik przesyła do programu w celu poprawnego wyświetlenia obiektów.



Rysunek 3.2: grafika przedstawiająca kombinację trzech rodzajów oświetlenia w OpenGL: światło otoczenia ma zwykle niewielką wartość i przychodzi ze wszystkich kierunków; światło rozproszone odbija się od powierzchni modeli i rozchodzi na różne strony; światło odbite reflektuje bezpośrednio w stronę kamery – jest charakterystyczne dla powierzchni naśladujących metalowe czy szklane. Źródło: [24]

3.4 Klasa `Material`

Klasa materiału posiada trzy pola zawierające informacje o intensywności swiateł *ambient*, *diffuse* i *specular*. Zawiera również metodę `sendToShader` wywołującą funkcje klasy `Shader`, które przypisują wartości uniformom z *fragment shader*.

3.5 Klasa `ObjLoader`

Object loader to klasa powstała w celu załadowania obiektu 3D z pliku OBJ i zebrania odpowiednich pozycji i parametrów wierzchołków w celu późniejszego przesłania ich do obiektów klasy siatki. Jest zaprzyjaźniona z klasą `Gui` celem przesyłania komunikatów do etykiet umieszczonych na interfejsie użytkownika.

Struktura pliku OBJ dzieli się na pozycje wierzchołków (poprzedzone prefiksem `v`), parametry wektorów tekstur (`vt`), parametry wektorów normalnych (`vn`) oraz wielokąty łączące odpowiednie wierzchołki ze sobą, oddzielone znakami „/” (`f`). W celu stworzenia struktury siatki autor zaimplementował samodzielnie napisaną metodę `loadObj` przechodzącą przez wszystkie linijki pliku i dodającą odpowiednie wartości do wektora `_vertices`, `_simpleVertices` i `_faces` (przechowujących obiekty klas odpowiednio `Vertex`, `SimpleVertex` oraz `Face`), a także dwóch nowych wektorów – `_indices` i `_simpleIndices` – przechowujących indeksy wierzchołków `Vertex` i `SimpleVertex` w odpowiedniej kolejności rysowania wielokątów.

3.6 Klasa `Mesh`

`Mesh` zawiera wektory przechowujące wierzchołki, wielokąty i indeksy wierzchołków w kolejności rysowania, które pozyskiwane są z odpowiedniego obiektu `ObjLoader`. Dodatkowo posiada wektor `_pairs`, który zawiera obiekty klasy `Pair` z odpowiednimi, niepowtarzającymi się wierzchołkami. Istotnym polem klasy jest `_percentage`, które

wskazuje na procentową wartość liczebności wierzchołków końcowych, w porównaniu do wierzchołków początkowych, po przejściu przez algorytm optymalizacyjny.

Konstruktor klasy `Mesh` pozyskuje wierzchołki i kolejność ich rysowania z `ObjLoader` i sprawdza, czy siatka powinna zostać poddana optymalizacji. Jeśli tak, wywoływana jest metoda `simplifyMesh` implementująca algorytm upraszczania powierzchni z błędem kwadratowym (sekcja 2.2.3). W przeciwnym wypadku od razu uruchamiana jest funkcja generująca obiekty OpenGL na podstawie wektorów wierzchołków i ich indeksów: VAO (*Vertex Array Object*), VBO (*Vertex Buffer Object*) i EBO (*Element Buffer Object*), służące do poprawnego renderowania siatki.

Metoda `render` przyjmuje argument `polygonMode`, oznaczający, czy trójkąty mają być wypełnione lub rysowane w postaci konturów, a następnie wykorzystuje funkcję OpenGL `glDrawElements` rysującą na podstawie wektora wierzchołków wielokąty w dostarczonej przez `_indices` lub `_simpleIndices` kolejności.

Do pozyskiwania odpowiednich obiektów klas wierzchołków, poligonów i par używane są specjalne metody – `getVertex`, `getFace`, `getPair` – przeszukujące wektory, na wejściu których podaje się identyfikator obiektu. Funkcje `findVertexPosition`, `findFacePosition` i `findPairPosition` zwracają indeksy w wektorach w celu ich usunięcia.

`simplifyMesh` wymaga podania na wejściu wartości `percentage`. Na początku dla każdego wierzchołka obliczana jest macierz $\boldsymbol{Q}$ metodą `computeInitialQuad` (listing 3.1).

```
e1 = getVertex(f._vertices[1])._position - getVertex(f._vertices[0])._position;
e2 = getVertex(f._vertices[2])._position - getVertex(f._vertices[0])._position;
normal = glm::cross(e1, e2);
plane = { normal, -glm::dot(_simpleVertices[findVertexPosition(v._id)]._position,
normal) };
quad += glm::outerProduct(plane, plane);
```

Listing 3.1: Propozycja obliczania wartości macierzy $\boldsymbol{Q}$ – zawartość pętli po indeksach trójkątów zawierających dany wierzchołek z funkcji `computeInitialQuad`. Do pozyskania współrzędnych płaszczyzny $\boldsymbol{p} = [a \quad b \quad c \quad d]^T$ wykorzystywane są funkcje biblioteki GLM obliczające iloczyn wektorowy (`glm::cross`), iloczyn skalarny (`glm::dot`) oraz macierz $\boldsymbol{K_p}$ ze wzoru 2.14 (`glm::outerProduct`)

Z siatki wydzielane są wszystkie pary przez użycie wbudowanej w C++ klasy kontrolera `std::set` i przekazywane do wektora `_pair`, a ich koszty obliczane są metodą `computeInitialCost` (listing 3.2).

```
for (auto& p : _pairs)
{
	glm::vec3 pTemp = glm::vec3(.0f);
	glm::mat4 qTemp = glm::mat4(.0f);

	pTemp = getVertex(p._vertices[0])._position +
getVertex(p._vertices[1])._position;
	qTemp = getVertex(p._vertices[0])._quad + getVertex(p._vertices[1])._quad;

	pTemp /= 2.f;
	glm::vec4 cost = glm::vec4(pTemp.x, pTemp.y, pTemp.z, 1.f);
	cost = cost * qTemp;

	p._cost = glm::dot(cost, cost);
}
```

Listing 3.2: Zawartość metody `computeInitialCost` implementująca wzór 2.15. Nowy wierzchołek tworzony jest pośrodku krawędzi

Po wykonaniu funkcji rozpoczyna się właściwy proces algorytmu. Maksymalna ilość iteracji obliczana jest na podstawie wartości procentowej podanej przez użytkownika. Wewnątrz pętli początkowo wybierana jest para o najniższym koszcie, a jej indeksy przekazywane są zwierającej krawędź metodzie `collapse` (listing 3.3).

```
glm::vec3 newPosition = (getVertex(newId)._position + getVertex(oldId)._position)
/ 2.f;
glm::mat4 newQuad = getVertex(newId)._quad + getVertex(oldId)._quad;

std::set<GLuint> newNeighbors;
for (auto& n : getVertex(newId)._neighbors)
	newNeighbors.insert(n);
for (auto& n : getVertex(oldId)._neighbors)
	newNeighbors.insert(n);

newNeighbors.erase(oldId);
newNeighbors.erase(newId);

getVertex(newId)._position = newPosition;
getVertex(newId)._neighbors = newNeighbors;
getVertex(newId)._quad = newQuad;

//	update neighbors
for (auto& n : getVertex(newId)._neighbors)
{
	getVertex(n)._neighbors.erase(oldId);
	getVertex(n)._neighbors.insert(newId);
}

//update faces
//	select faces to be removed
std::vector<GLuint> faces;
for (auto& f : getVertex(newId)._faces)
	if (contains(f, oldId))
		faces.push_back(f);

//	insert v2's faces ids to v1
for (auto& f : getVertex(oldId)._faces)
```

```
	getVertex(newId)._faces.insert(f);

//remove ereased faces from new vertex
for (auto f : faces)
	getVertex(newId)._faces.erase(f);

//	remove oldId from all faces
for (auto& f : getVertex(newId)._faces)
	for (size_t i = 0; i < 3; ++i)
		if (getFace(f)._vertices[i] == oldId)
			getFace(f)._vertices[i] = newId;

//remove deleted faces ids from all neighbors
for (auto& n : getVertex(newId)._neighbors)
	for (auto f : faces)
		getVertex(n)._faces.erase(f);

//update pairs
//	delete pairs
for (size_t i = 0; i < _pairs.size(); ++i)
	if (_pairs[i]._vertices[0] == newId || _pairs[i]._vertices[1] == newId ||
_pairs[i]._vertices[0] == oldId || _pairs[i]._vertices[1] == oldId)
	{
		_pairs.erase(_pairs.begin() + i);
		--i;
	}
//	add new pairs
for (auto n : getVertex(newId)._neighbors)
	_pairs.push_back(Pair(newId, n));

for (auto f : faces)
	_faces.erase(_faces.begin() + findFacePosition(f));

//	delete vertex
_simpleVertices.erase(_simpleVertices.begin() + findVertexPosition(oldId));
```

Listing 3.3: Implementacja operacji zwierania krawędzi (rys. 2.4) w metodzie `collapse`. Funkcja aktualizuje sąsiadów nowego wierzchołka i wielokąty, a następnie usuwa parę i poligony, których wejściowe wierzchołki tworzyły krawędzie

Na końcu iteracji obliczany jest kosz wszystkich pozostałych, zaangażowanych w operację zwierania par, za pomocą funkcji `computeCost` – bardzo podobnej do `computeInitialCost`. Po wyjściu z oryginalnej metody `simplifyMesh` wektor `_simpleIndices` budowany jest na nowo na podstawie wynikowych poligonów, a siatka poddana inicjalizacji.

Metoda `incrementalRemeshing` implementuje algorytm z sekcji 2.2.4. Jest to próba dodatkowej optymalizacji siatki, której celem jest doprowadzenie trójkątów do jak najbliższych równobocznym. Docelowa długość krawędzi L nie jest ustalona przez użytkownika, lecz obliczana jako wartość średniej arytmetycznej wszystkich długości krawędzi w siatce przetworzonej przez algorytm w metodzie `simplifyMesh`. Po wyliczeniu wartości minimalnej ($L_{min} = \frac{4}{5}L$) i maksymalnej ($L_{max} = \frac{4}{3}L$) inicjowana jest pętla przechodząca przez wszystkie pary i wywołująca metodę `collapse` (listing 3.3) dla wszystkich, których długość jest niższa, niż L_{min}. Kolejnym krokiem jest ponowne przejście przez wszystkie pary i zastosowanie dla wszystkich krawędzi o długości wyższej, niż L_{max} nowej metody `split` (listing 3.4).

```
SimpleVertex e1 = getVertex(pair._vertices[0]);
SimpleVertex e2 = getVertex(pair._vertices[1]);

//get faces
std::vector<GLuint> faces;
for (auto& f : e1._faces)
{
	if (contains(f, e2._id))
		faces.push_back(f);
}
//std::cout << faces.size() << "\t";

//other vertices
std::vector<GLuint> v;
for (int i = 0; i < faces.size(); ++i)
	for (int j = 0; j < 3; ++j)
		if (getFace(faces[i])._vertices[j] != e1._id &&
getFace(faces[i])._vertices[j] != e2._id)
			v.push_back(getFace(faces[i])._vertices[j]);

//std::cout << v.size() << "\t";

if (v.size() == faces.size() && v.size() == 2)
{
	//add new vertex
	glm::vec3 newPosition = (e1._position + e2._position) / 2.f;
	GLuint newId = _simpleVertices[_simpleVertices.size() - 1]._id + 1;

	_simpleVertices.push_back(SimpleVertex(newId, newPosition));

	//remove old faces from e2
	for (auto f : faces)
		e2._faces.erase(f);

	//change starting faces' vertrices
	for (int i = 0; i < faces.size(); ++i)
	{
		getFace(faces[i])._vertices[0] = e1._id;
		getFace(faces[i])._vertices[1] = v[i];
		getFace(faces[i])._vertices[2] = newId;
		getFace(faces[i]).sortVertices();
	}

	//add neighbors to new Vertex
	getVertex(newId)._neighbors.insert(e1._id);
	getVertex(newId)._neighbors.insert(e2._id);
	for (auto v_ : v)
		getVertex(newId)._neighbors.insert(v_);

	//remove neighbors from pair
	e1._neighbors.erase(e2._id);
	e2._neighbors.erase(e1._id);

	//add new vertex to other vertices' neighbors
	e1._neighbors.insert(newId);
	e2._neighbors.insert(newId);
	for (auto v_ : v)
		getVertex(v_)._neighbors.insert(newId);
```

```
	GLuint newFaceId;
	//add faces
	for (auto v_ : v)
	{
		newFaceId = _faces[_faces.size() - 1]._id + 1;
		_faces.push_back(Face(newFaceId, { {e2._id, v_, newId} }));
	}

	//add new faces to vertices
	for (int i = 0; i < v.size(); ++i)
	{
		e2._faces.insert(_faces[_faces.size() - 1]._id - i);
		getVertex(v[i])._faces.insert(_faces[_faces.size() - 1]._id - i);
		getVertex(newId)._faces.insert(_faces[_faces.size() - 1]._id - i);
	}
	for (auto f : faces)
		getVertex(newId)._faces.insert(f);

	//update pairs
	//	add new pairs
	for (auto v_ : v)
	{
		_pairs.push_back(Pair(newId, v_));
		_pairs[_pairs.size() - 1]._id = _pairs[_pairs.size() - 2]._id + 1;
	}
	_pairs.push_back(Pair(newId, e2._id));
	_pairs[_pairs.size() - 1]._id = _pairs[_pairs.size() - 2]._id + 1;
	//	change the starging pair
	pair.set(e1._id, newId);
}
else
	_pairs.erase(_pairs.begin() + findPairPosition(pair._id));
```

Listing 3.4: Implementacja operacji dzielenia siatki z rys. 2.5. Funkcja dodaje nowy wierzchołek, aktualizuje sąsiadów i poligony, oraz dodaje nową krawędź do zbioru

Ostatnim krokiem jest zmiana pozycji wszystkich wierzchołków za pomocą metody `vertexRelocation` (listing 3.5).

```
glm::vec3 avg(0.f);

for (auto n : vertex._neighbors)
{
	avg += getVertex(n)._position;
}

avg /= vertex._neighbors.size();
vertex._position = avg;
```

Listing 3.5: Nowym położeniem wierzchołka jest wartość średnia wszystkich jego sąsiadów

3.7 Klasa `Model`

W skład modelu teoretycznie może wchodzić wiele siatek, więc klasa `Model` posiada pole na wektor obiektów klasy `Mesh`, jednak w tym projekcie każdy model ma jedną siatkę.

Zadaniem konstruktora jest stworzenie obiektu `ObjLoader` i wczytanie pliku, a następnie przekazanie go do nowego obiektu `Mesh`, który zostanie dodany do wektora siatek.

`Model` posiada również funkcję `render`, która przesyła parametry jego materiału do shadera, a następnie rysuje po kolei wszystkie jego siatki.

3.8 Klasa `App`

Głównym rdzeniem aplikacji jest obiekt klasy `App`, który odpowiedzialny jest za wyświetlanie okna, obsługę kamery i rysowanie.

Metoda `initWindow` tworzy nowe okno GLFW – nadaje mu tytuł pojawiający się na górnej etykiecie programu oraz ustala rozdzielczość.

`initOpenGLOptions` to metoda inicjująca OpenGL. Włącza bufor głębokości[10], mieszanie kolorów i antyaliasing[11] linii siatek. Umożliwia rysowanie trójkątów od dwóch stron poprzez wyłączenie tzw. *face culling* (według wiedzy autora nie istnieje polski odpowiednik tego terminu).

Metoda `initMatrices` oblicza wartości macierzy widoku i macierzy projekcji (listing 3.6).

```
_ViewMatrix = glm::lookAt(_camPosition, glm::vec3(.0f), glm::vec3(.0f, 1.f, .0f));
	//init view matrix

_ProjectionMatrix = glm::perspective
(
	glm::radians(_fov),
	static_cast<float>(_framebufferWidth) / _framebufferHeight,
	_nearPlane,
	_farPlane
);	//init projection matrix
```

Listing 3.6: Wnętrze metody `initMatrices`. Macierze widoku i projekcji obliczane są przez wykorzystanie funkcji GLM: `glm::lookAt` oraz `glm::perspective`

`initShader` tworzy nowy obiekt klasy `Shader` i przesyła do niej numer używanej wersji OpenGL, a także ścieżki do plików GLSL. `initUniforms` wysyła do shadera macierz widoku, macierz projekcji oraz pozycję światła. Macierze są aktualizowane i przesyłane ponownie, w miarę działania programu, w funkcji `updateUniforms`.

W celu poruszania się w przestrzeni wykorzystywane są funkcje: `updateDt` i `updateInput` (oraz wchodzące w nią `updateMouseInput` i `updateKeyboardInput`). Zadaniem `updateDt` jest obliczenie czasu upłyniętego między kolejnymi pobraniami (*delta time*) funkcją `glfwGetTime` z biblioteki GLFW, zwracającą wartość `double` w sekundach.

[10] Bufor głębokości przechowuje informacje o odległości każdego fragmentu od kamery, dzięki czemu bliższe obiekty rysowane są na bardziej oddalonych [25].

[11] Antyaliasing – technika w grafice komputerowej eliminująca „poszarpane” krawędzie obiektów [26].

Dzięki uzyskanej wartości zapisanej do zmiennej `_dt` można wywołać `updateMouseInput`, która służy do obracania kamerą za pomocą myszy. Funkcja działa w dwóch dynamicznie zmieniających się trybach w zależności od tego, czy kamera jest w trakcie zmieniania pozycji (listing 3.7).

```
if
(glfwGetMouseButton(_window, GLFW_MOUSE_BUTTON_1) == GLFW_PRESS &&
	(
		!glfwGetKey(_window, GLFW_KEY_W) == GLFW_PRESS &&
		!glfwGetKey(_window, GLFW_KEY_S) == GLFW_PRESS &&
		!glfwGetKey(_window, GLFW_KEY_A) == GLFW_PRESS &&
		!glfwGetKey(_window, GLFW_KEY_D) == GLFW_PRESS
	)
)
{
	glfwSetInputMode(_window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
	glfwGetCursorPos(_window, &_mouseX, &_mouseY);
	if (_firstMouse)
	{
		_lastMouseX = _mouseX;
		_lastMouseY = _mouseY;
		_firstMouse = false;
	}

	//calculate offset
	_mouseOffsetX = -_mouseX + _lastMouseX;
	_mouseOffsetY = -_lastMouseY + _mouseY;

	//set last X and Y
	_lastMouseX = _mouseX;
	_lastMouseY = _mouseY;
}
else if
(!glfwGetMouseButton(_window, GLFW_MOUSE_BUTTON_1) == GLFW_PRESS &&
	(
		glfwGetKey(_window, GLFW_KEY_W) == GLFW_PRESS ||
		glfwGetKey(_window, GLFW_KEY_S) == GLFW_PRESS ||
		glfwGetKey(_window, GLFW_KEY_A) == GLFW_PRESS ||
		glfwGetKey(_window, GLFW_KEY_D) == GLFW_PRESS
	)
)
{
	glfwSetInputMode(_window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
	glfwGetCursorPos(_window, &_mouseX, &_mouseY);

	if (_firstMouse)
	{
		_lastMouseX = _mouseX;
		_lastMouseY = _mouseY;
		_firstMouse = false;
	}

	//calculate offset
	_mouseOffsetX = _mouseX - _lastMouseX;
	_mouseOffsetY = _lastMouseY - _mouseY;

	//set last X and Y
```

```
        _lastMouseX = _mouseX;
        _lastMouseY = _mouseY;
}
else
{
        glfwSetInputMode(_window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
        _firstMouse = true;
        _mouseOffsetX = 0.0;
        _mouseOffsetY = 0.0;
}
```

Listing 3.7: Implementacja procedury obracania kamerą[12] w metodzie `updateMouseInput`.

`updateKeyboardInput` sprawdza, czy wciśnięto odpowiedni przycisk na klawiaturze i na tej podstawie wywołuje metodę `move` klasy `Camera`, przesyłając parametr `_dt` i odpowiedni indeks oznaczający kierunek. Wszystkie metody aktualizujące wywoływane są w jednej funkcji updateInput (listing 3.8).

```
updateMouseInput();
updateKeyboardInput();
_camera.rotate(_dt, _mouseOffsetX, _mouseOffsetY);

//move light with camera
*_lights[0] = _camera.getPosition();
```

Listing 3.8: Zawartość metody `updateInput`. Funkcja wywołuje dwie metody obsługujące mysz i klawiaturę, obraca kamerą na podstawie odczytanej zmiany pozycji kursora, a także zmienia pozycję światła, by podążało za kamerą

Metoda `render` wywołuje `updateUniforms`, a następnie rysuje model wybrany przez użytkownika przez przyciski na panelu interfejsu.

3.9 Klasa `Gui`

Klasa graficznego interfejsu użytkownika to nakładka na obiekt `App`, do którego wskaźnik jest jednym z pól klasy.

Metoda `clear` czyści bufor głębokości oraz bufor koloru, wypełniając ekran ciemnoszarą barwą, przygotowując klatkę do ponownego narysowania obiektów.

Funkcja `render` odpowiedzialna jest za narysowanie bocznego panelu z interfejsem graficznym (listing 3.9).

[12] Temat rozwinięty w sekcji 4.1.

```
ImGui::Begin("front panel", 0, 1 | 2 | 4);
ImGui::SetWindowSize({ 350, 700 });
ImGui::SetWindowPos({ 10, 10 });
ImGui::Text("Use WSAD to move the camera.\nDrag the mouse cursor to rotate the
camera.");
if (ImGui::Button("reset the camera"))
{
	_app->_camera.reset();
}

ImGui::Text("camera speed");
ImGui::InputFloat("##camSpeed", &_app->_camera._movementSpeed);
ImGui::SameLine();
if (ImGui::Button("reset"))
{
	_app->_camera._movementSpeed = 10.f;
}

ImGui::Text("\nfile path");
ImGui::InputText("##filePath", _filePath, 100);
ImGui::Text("simplify modifier (%%)");
ImGui::SliderFloat("##percentage", &_percentage, 0.f, 100.f);
if (ImGui::Button("load & calculate"))
{
	_log = ">loading";
	_toInit = true;
}
ImGui::SameLine();
ImGui::Text((_log).c_str());
ImGui::Text("\n");
ImGui::Checkbox("filled polygons", &_filledPolygons);

ImGui::Text("\n");
if (ImGui::RadioButton("original model", &_meshMode, 0)) { _filledPolygons = true;
}
if (ImGui::RadioButton("original mesh", &_meshMode, 1)) { _filledPolygons = false;
}
if (ImGui::RadioButton("simplified mesh", &_meshMode, 2)) { _filledPolygons =
false; }
if (ImGui::RadioButton("quasi-regular mesh", &_meshMode, 3)) { _filledPolygons =
false; }

if (_app->_models.size() == 4 && _app->_models[0]->_meshes[0]->_vertices.size() >
0)
{
	ImGui::Text("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
	ImGui::Text(static_cast<std::string>("original mesh vertices count: " +
std::to_string(_app->_models[1]->_meshes[0]->_simpleVertices.size())).c_str());
	ImGui::Text(static_cast<std::string>("\nsimplified mesh vertices count: " +
std::to_string(_app->_models[2]->_meshes[0]->_simpleVertices.size())).c_str());
	ImGui::Text(static_cast<std::string>("processing time: " +
std::to_string(_app->_models[2]->_meshes[0]->_simplifyTime) + " s").c_str());
	ImGui::Text(static_cast<std::string>("\nquasi-regular mesh vertices count: "
+ std::to_string(_app->_models[3]->_meshes[0]->_simpleVertices.size())).c_str());
	ImGui::Text(static_cast<std::string>("processing time: " +
std::to_string(_app->_models[3]->_meshes[0]->_aeapTime) + " s").c_str());
}

ImGui::End();
```

```
ImGui::Render();
ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
```

Listing 3.9: Zawartość funkcji `render` tworzącej panel o zadanych wymiarach i pozycji, oraz uzupełniającej go etykietami, przyciskami wywołującymi odpowiednie metody i wpływającymi na zmienne, edytowalnymi polami tekstu do zmieniania prędkości kamery i otwierania plików OBJ, a także suwakiem do ustawiania wyjściowej liczby wierzchołków[13]

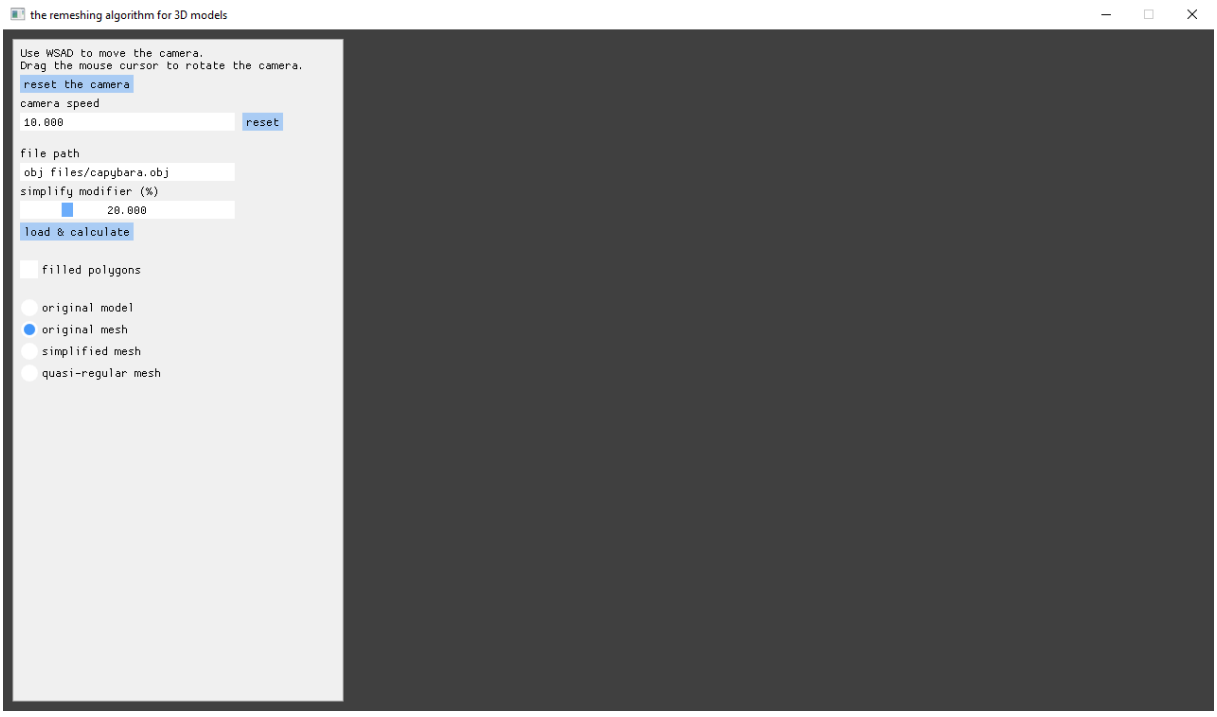
Metoda `loop` zawiera główną pętlę aplikacji odpowiedzialną za jej odświeżanie i rysowanie kolejnych klatek. Wpierw wywołuje ona metody `update` oraz `render` obiektu `App`, a w przypadku wciśnięcia przycisku „*load & calculate*" inicjuje modele na nowo metodą `initModels`. Następnie wywoływana jest metoda klasy `Gui` – `render`. Na końcu pętli uruchamiana jest funkcja `clear`.

Konstruktor klasy `Gui` oraz jej funkcja `loop` wywoływane są w głównej funkcji `main` aplikacji.

4 Przykłady działania aplikacji

4.1 Poruszanie się po aplikacji

Po uruchomieniu programu pojawia się ekran o stałej rozdzielczości 1280 na 720 pikseli, jak na rys. 4.1.



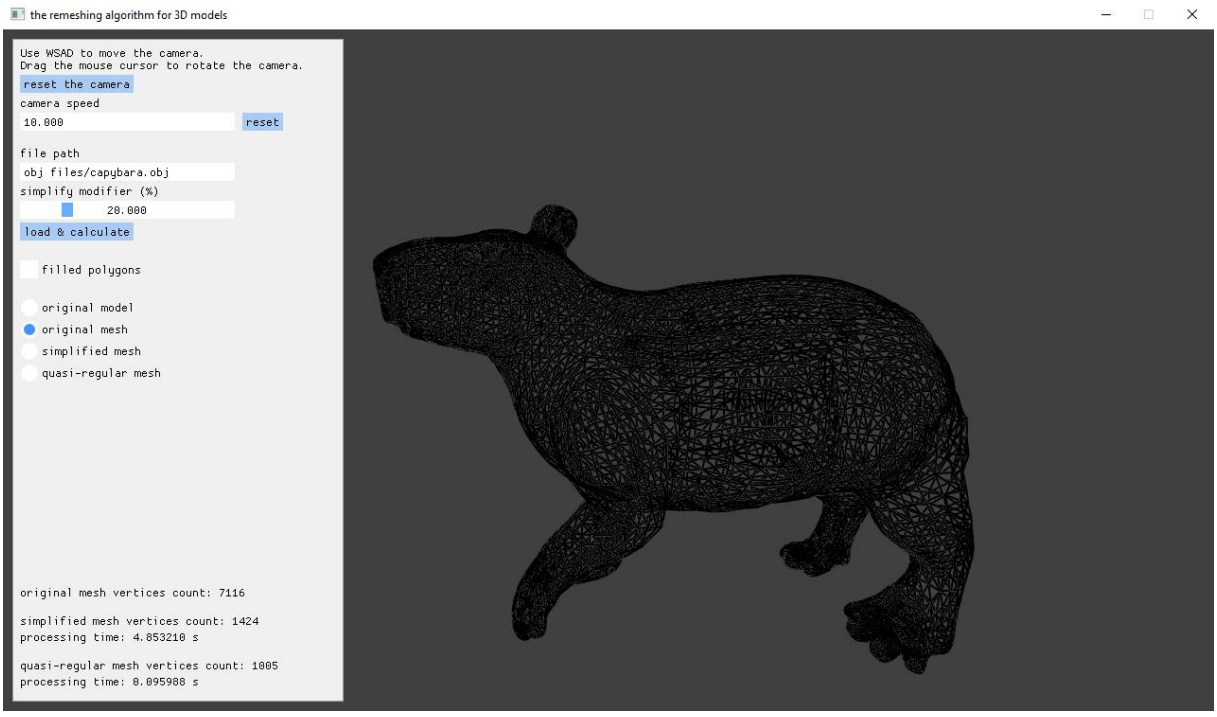
Rysunek 4.1: Okno aplikacji projektowej. Po lewej stronie widoczny panel z graficznym interfejsem użytkownika. Po prawej miejsce na rysowane modele

[13] Wyjściowe GUI przedstawione jest w sekcji 4.1.

Interfejs zawiera:

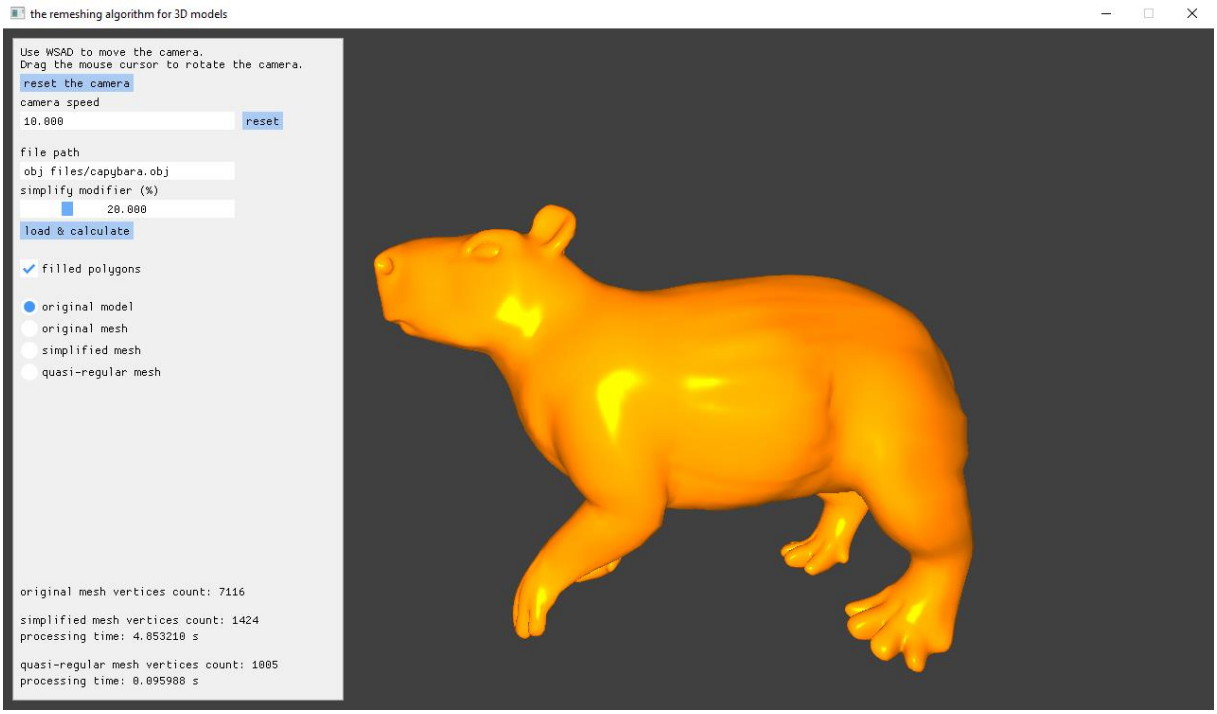
- etykiety z instrukcjami do poruszania kamerą,
- przycisk przywrócenia pozycji i rotacji kamery do wartości początkowych,
- pole edycji wartości szybkości kamery (przydatne w przypadkach wczytanych modeli znacznie różniących się rozmiarami),
- przycisk przywrócenia domyślnej wartości szybkości (10),
- pole edycji ze ścieżką do pliku OBJ (przy starcie programu wpisana jest już ścieżka do modelu kapibary [27] dostarczonego wraz z wydaniem aplikacji),
- suwak do ustawienia stosunku ilości wierzchołków wyjściowych modelu do ilości wierzchołków wejściowych w wartości procentowej,
- przycisk do wczytania i przetworzenia modelu,
- cztery przyciski opcji (*radio buttons*) do ustawienia aktualnie wyświetlanej siatki,
- etykiety z informacjami o ilości wierzchołków wejściowych, wyjściowych oraz czasach trwania algorytmów optymalizacyjnych (wyświetlane po zakończeniu wczytywania i obliczania, przykład na rys. 4.2).

Po wciśnięciu przycisku *load & calculate* (wczytaj i oblicz) bez zmiany ustawień początkowych, jeżeli model znajduje się w odpowiednim katalogu, rysowana zostaje siatka (rys. 4.2).



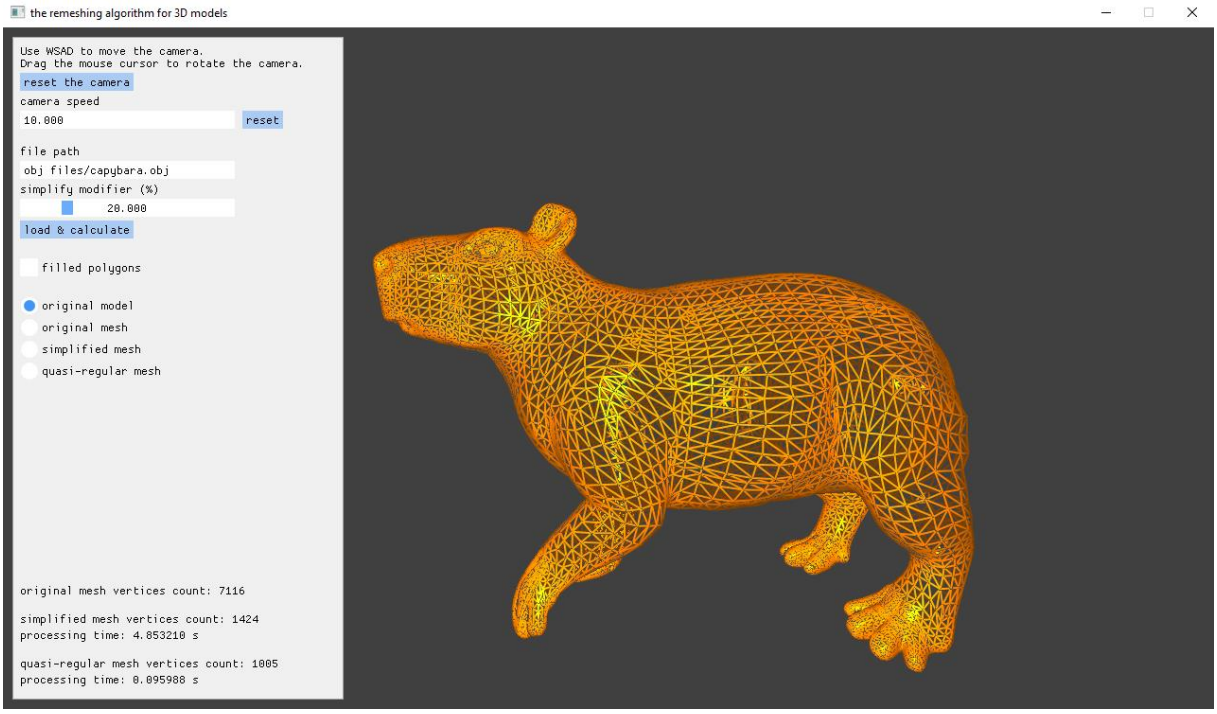
Rysunek 4.2: Wczytana siatka oryginalnego modelu.

Przełączenie się na model oryginalny, wykorzystujący wektory normalne do odbijania światła, daje rezultat jak na rys. 4.3.



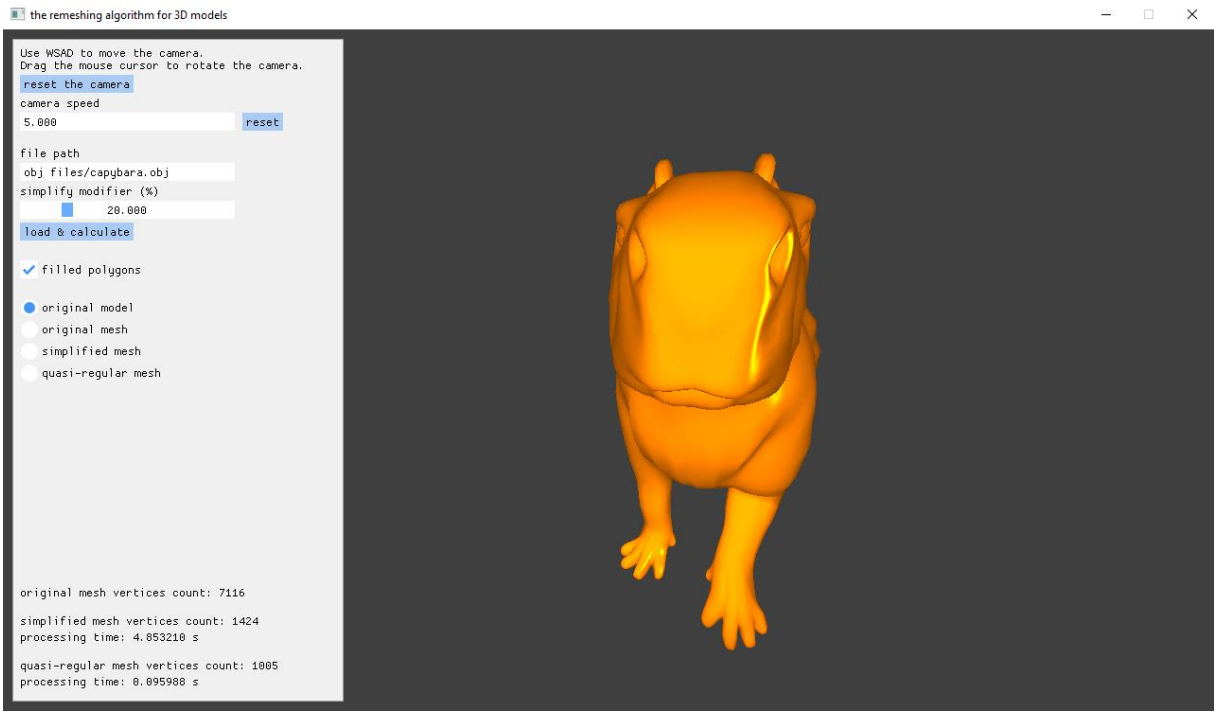
Rysunek 4.3: Oryginalny model

Opcja *filled polygons* (wypełnione poligony) jest automatycznie zaznaczana podczas przełączania się na oryginalny model i odznaczana, gdy użytkownik wybiera dowolną z trzech siatek. Można jednak zarządzać nią manualnie (rys. 4.4).



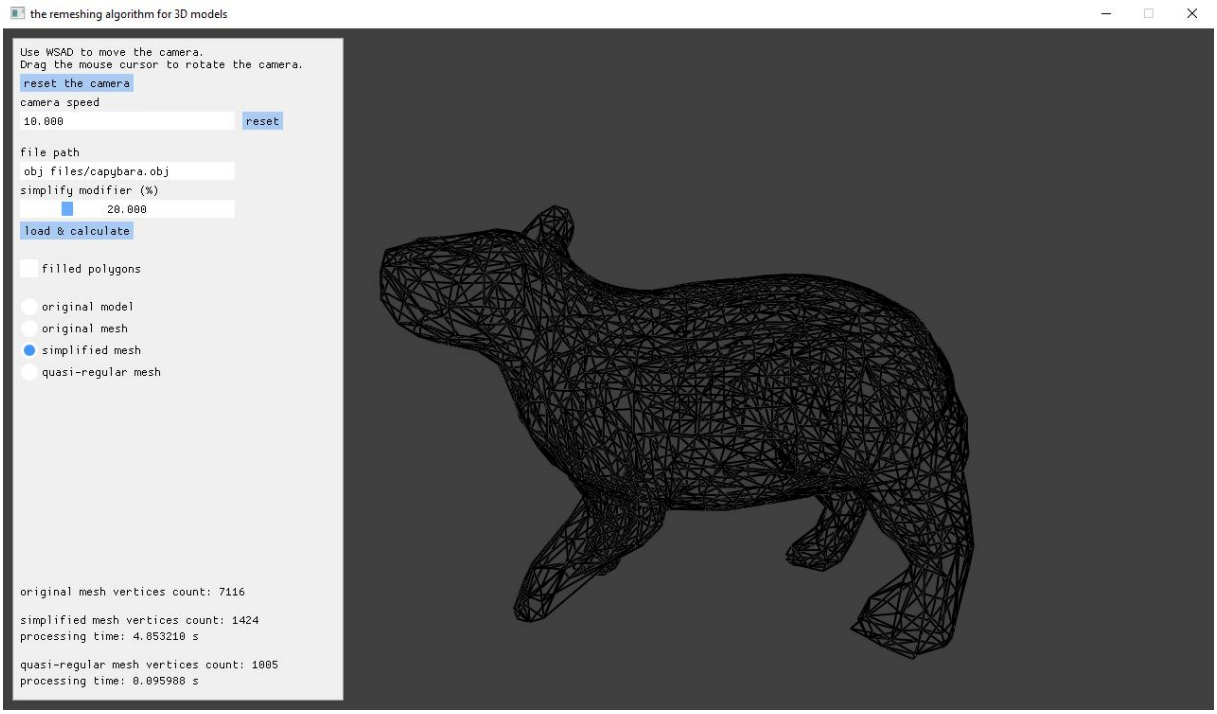
Rysunek 4.4: Oryginalny model ze ścianami w trybie konturów

Poruszanie kamerą odbywa się za pomocą przycisków na klawiaturze do zmieniania pozycji („W” – naprzód, „S” – wstecz, „A” – w lewo, „D” – w prawo) oraz myszy do obracania. Gdy kamera stoi w miejscu, kamerę obraca się wciskając lewy przycisk w dowolne miejsce w oknie aplikacji i przytrzymując go, przesuwając kursor w wybranym kierunku, a następnie puszczając. W trakcie zmieniania pozycji kamery do obracania nie trzeba wciskać przycisku myszy, co przyspiesza i ułatwia proces. Po zmianie pozycji można obserwować model z innego punktu (rys. 4.5).



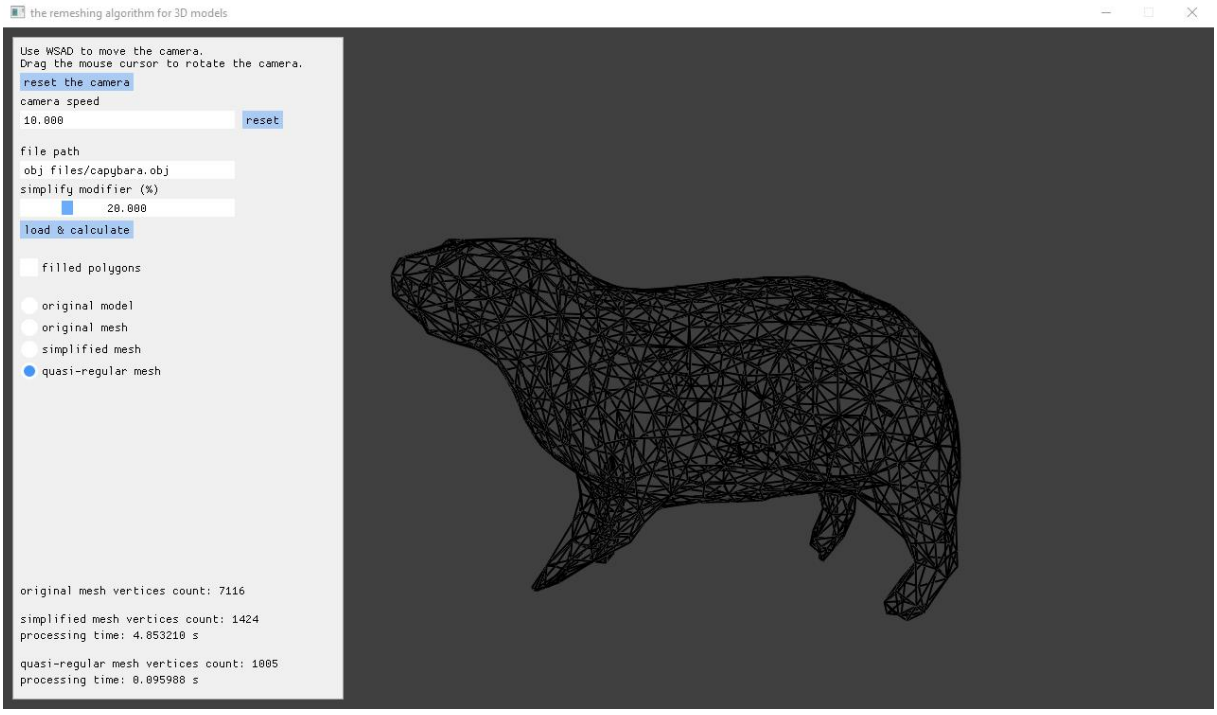
Rysunek 4.5: Widok na modelu po zmianie umiejscowienia kamery. Zmniejszono również szybkość poruszania się w celu dokładniejszego ustalenia pozycji

Siatka oryginalna (rys. 4.2) po przetworzeniu metodą `simplifyMesh` (sekcja 3.6) widoczna jest po wybraniu opcji *simplified mesh* (uproszczona siatka), jak na rys. 4.6.



Rysunek 4.6: Siatka po przetworzeniu przez algorytm błędu średniokwadratowego (sekcja 2.2.3)

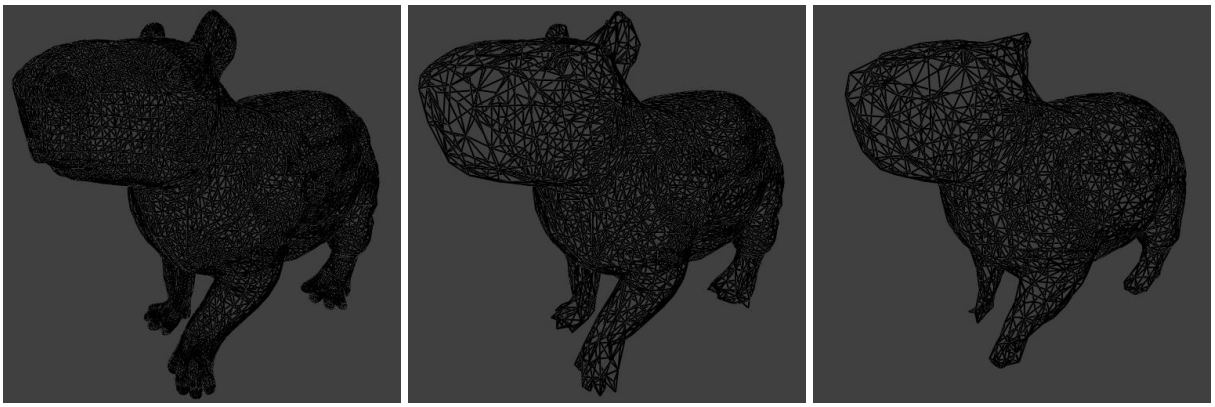
Siatka z rys. 4.6 przetworzona metodą `incrementalRemeshing` (sekcja 3.6) jest rysowania po zaznaczeniu opcji *quasi-regular mesh* (siatka częściowo regularna), jak na rys. 4.7.



Rysunek 4.7: Siatka po przetworzeniu przez metodę implementującą izotropową optymalizację siatki (sekcja 2.2.4)

4.2 Przykładowe wyniki algorytmów optymalizacyjnych

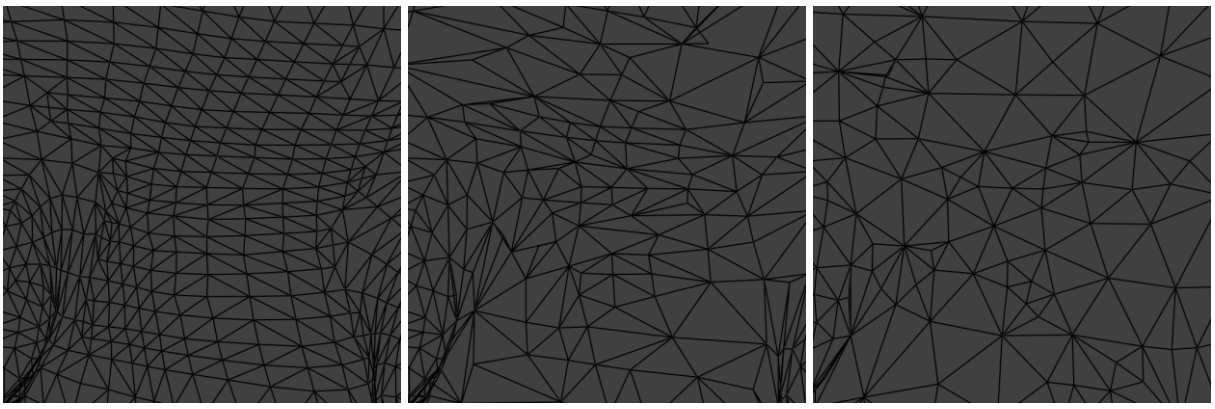
Struktura siatek wynikowych przedstawia się inaczej po przetworzeniu modelu przez różne algorytmy (przykład na rysunkach 4.8 i 4.9).



a) b) c)

Rysunek 4.8: Struktura siatki modelu kapibary: a) oryginalnej; b) po przetworzeniu przez algorytm błędu średniokwadratowego z ustawieniem na ok. 30% wierzchołków wyjściowych; c) po przetworzeniu siatki z rysunku „b)" przez algorytm optymalizacji izotropowej

Łatwo zauważyć, że na rysunku 4.8 b) szczegóły takie jak uszy czy liczba palców w łapach zwierzęcia przy 70% mniejszej ilości wierzchołków wciąż istnieją. Siatka na rysunku 4.8 c) kosztem częściowego zunifikowania poligonów traci te detale.



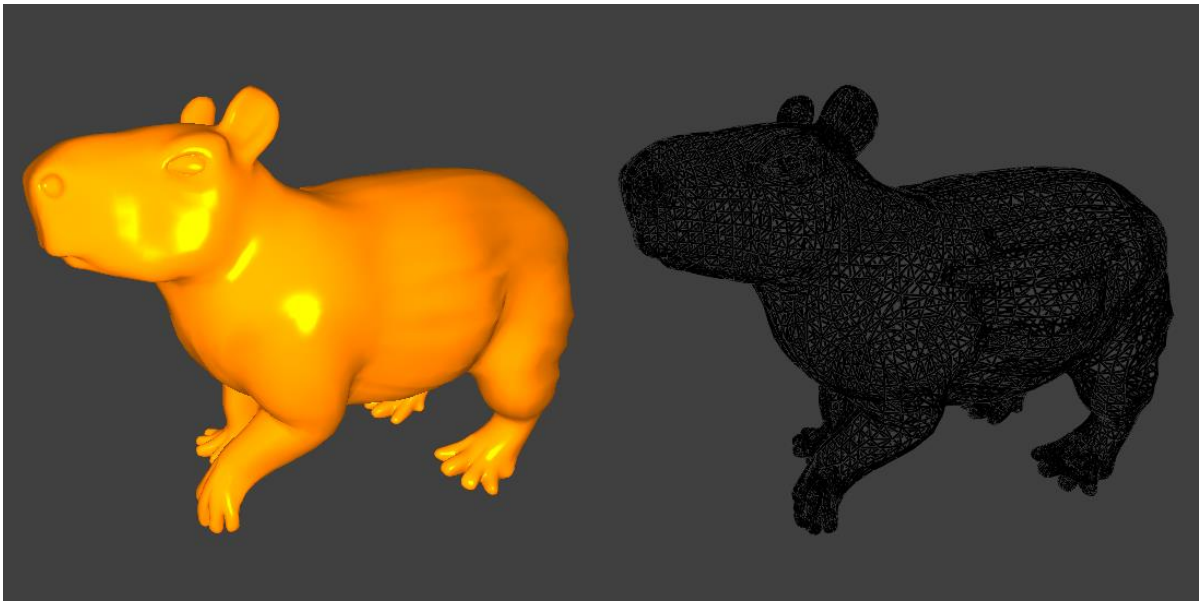
a) b) c)

Rysunek 4.9: Struktura siatki modelu kapibary w przybliżeniu: a) oryginalnej; b) po przetworzeniu przez algorytm błędu średniokwadratowego z domyślnym ustawieniem 20% wierzchołków wyjściowych; c) po przetworzeniu siatki z rysunku „b)" przez algorytm optymalizacji izotropowej

Na rysunku 4.9 b) widać, że ilość wierzchołków została względem siatki z 4.9 a) obniżona, a siatka 4.9 c) zmniejsza liczbę „zaostrzonych" trójkątów.

4.2.1 Czasy przetwarzania modelu kapibary

Model kapibary z pliku „capybara.obj” [27] ma 7116 wierzchołków i wygląda jak na rys. 4.10.



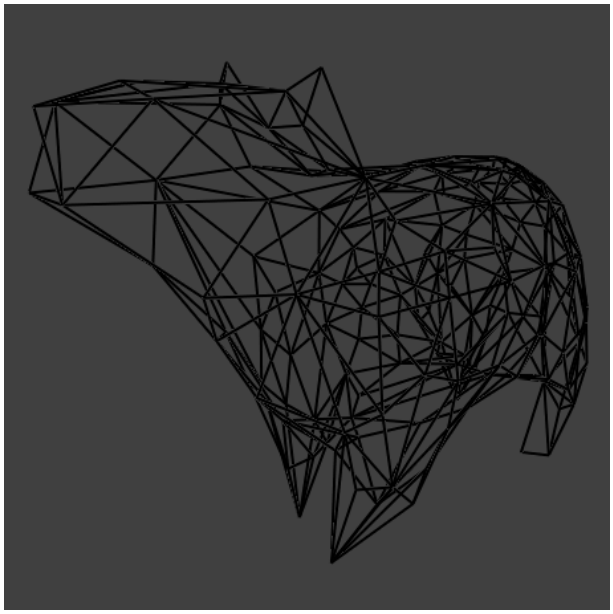
Rysunek 4.10: Model kapibary i jego oryginalna siatka

Czasy przetwarzania siatki przez funkcje błędu średniokwadratowego i izotropowej optymalizacji przedstawione są w tabeli 4.1.

Tabela 4.1: Liczebności wierzchołków siatki kapibary po zastosowaniu algorytmów i czasy ich przetwarzania

Algorytm błędu średniokwadratowego (sekcja 2.2.3)			Izotropowa optymalizacja siatki (sekcja 2.2.4)	
Procentowa liczebność wierzchołków wyjściowych	Liczebność wierzchołków wyjściowych (algorytm błędu średniokwadratowego)	Czas przetwarzania (s)	Liczebność wierzchołków wyjściowych (izotropowa optymalizacja)	Czas przetwarzania (s)
~100%	7116		7116	5,27813
~75%	5396	3,669507	4954	2,301695
~50%	3609	4,406017	2876	0,891226
~25%	1788	4,777913	1312	0,159825
~2,5%	169	4,913322	133	0,002779

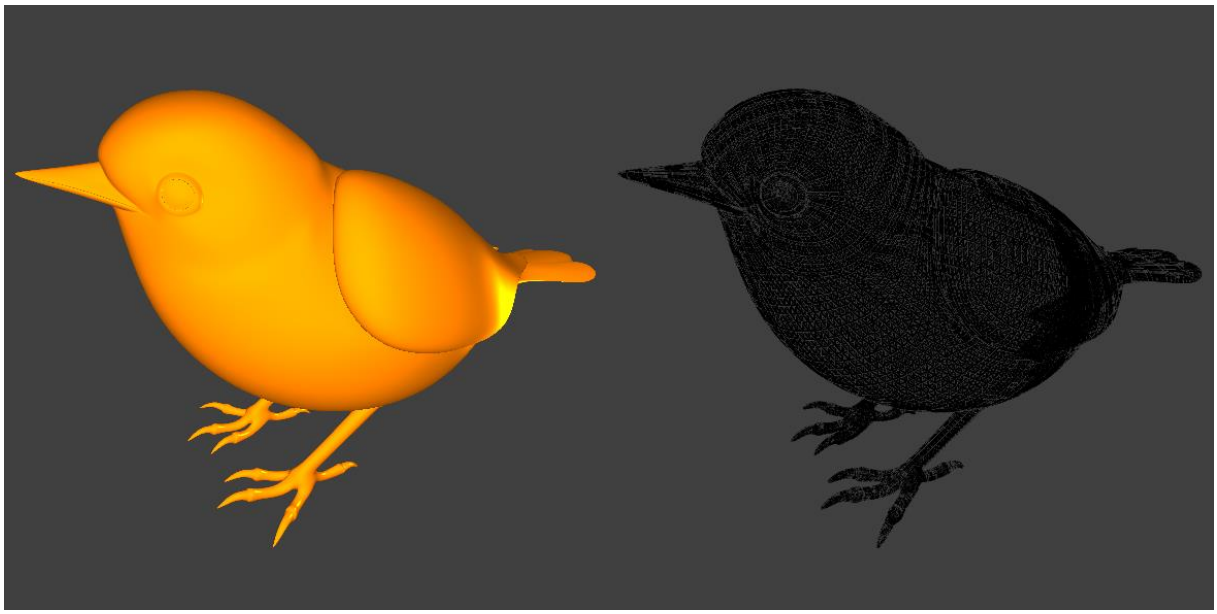
Siatka modelu z ekstremalnie małą liczbą wierzchołków wyjściowych uzyskana przy pomocy samej funkcji błędu średniokwadratowego przedstawia się na rysunku 4.11.



Rysunek 4.11: Siatka modelu z bardzo małą (~2,5%) liczbą wierzchołków wyjściowych. Topologia modelu wciąż jest zachowana

4.2.2 Czasy przetwarzania modelu ptaka

Model ptaka z pliku „bird.obj" [28] charakteryzuje się bardzo dużą liczbą wierzchołków (62338). Przedstawiony jest na rys. 4.12.



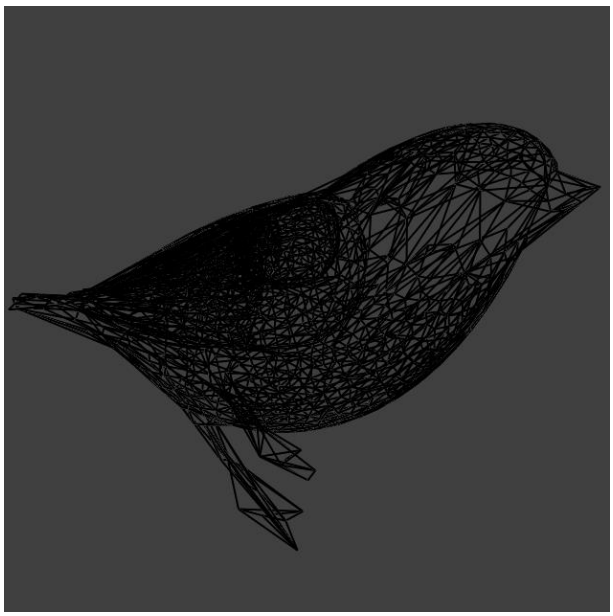
Rysunek 4.12: Model ptaka i jego oryginalna siatka

Czasy przetwarzania siatki przez funkcje błędu średniokwadratowego i izotropowej optymalizacji przedstawione są w tabeli 4.2.

Tabela 4.2: Liczebności wierzchołków siatki ptaka po zastosowaniu algorytmów i czasy ich przetwarzania

Algorytm błędu średniokwadratowego (sekcja 2.2.3)			Izotropowa optymalizacja siatki (sekcja 2.2.4)	
Procentowa liczebność wierzchołków wyjściowych	Liczebność wierzchołków wyjściowych (algorytm błędu średniokwadratowego)	Czas przetwarzania (s)	Liczebność wierzchołków wyjściowych (izotropowa optymalizacja)	Czas przetwarzania (s)
~100%	62338		56777	584,6203
~75%	46680	410,3892	43420	301,5436
~50%	31317	466,0916	28216	105,852
~25%	15659	564,1559	11203	20,04199
~2,5%	1478	642,2483	947	0,134666

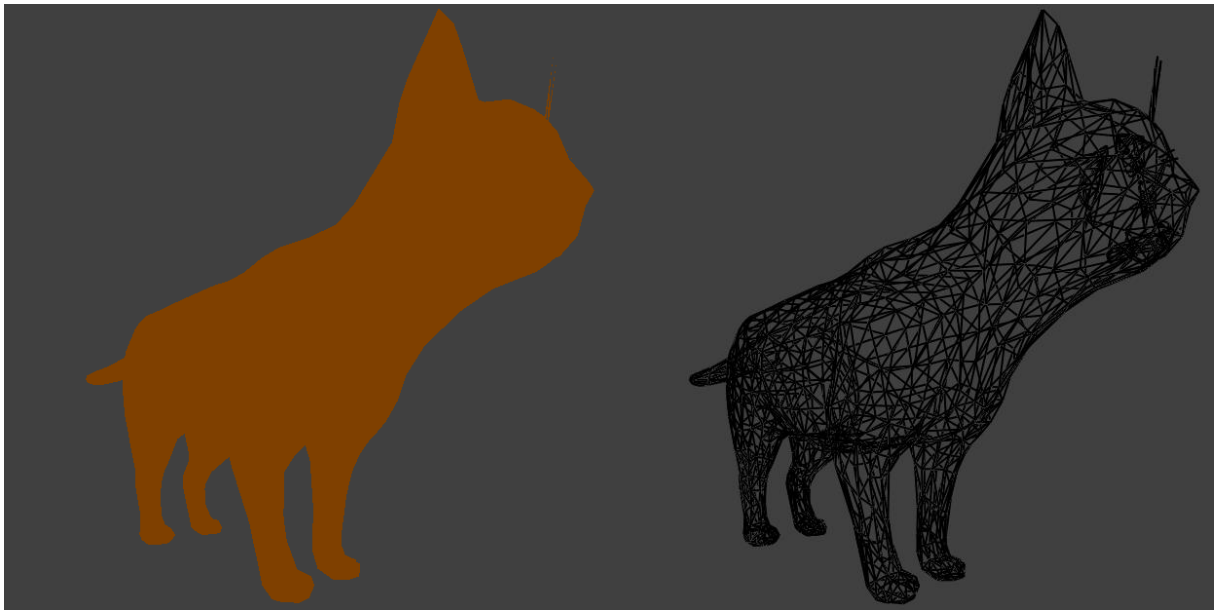
Siatka modelu z bardzo małą liczbą wierzchołków wyjściowych uzyskana przy pomocy samej funkcji błędu średniokwadratowego przedstawia się na rysunku 4.13.



Rysunek 4.13: Siatka modelu z ok. 2,5% wierzchołków wyjściowych. Topologia modelu wciąż jest zachowana

4.2.3 Czasy przetwarzania modelu kota

Model kota z pliku „cat.obj” [29] posiada małą liczbę wierzchołków (1137). Wygląda jak na rys. 4.14.



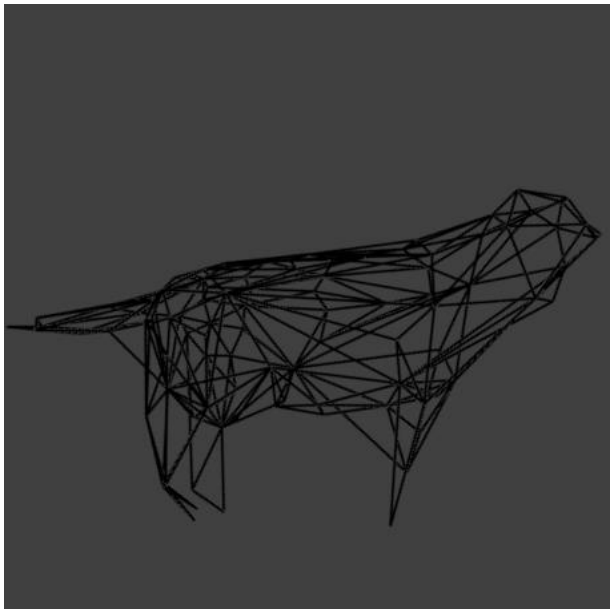
Rysunek 4.14: Model kota i jego oryginalna siatka. Nieobecność efektów świetlnych *diffuse* i *specular* spowodowana jest brakiem współrzędnych wektorów normalnych w pliku OBJ

Czasy przetwarzania siatki przez funkcje błędu średniokwadratowego i izotropowej optymalizacji przedstawione są w tabeli 4.3.

Tabela 4.3: Liczebności wierzchołków po zastosowaniu algorytmów i czasy ich przetwarzania

Algorytm błędu średniokwadratowego (sekcja 2.2.3)			Izotropowa optymalizacja siatki (sekcja 2.2.4)	
Procentowa liczebność wierzchołków wyjściowych	Liczebność wierzchołków wyjściowych (algorytm błędu średniokwadratowego)	Czas przetwarzania (s)	Liczebność wierzchołków wyjściowych (izotropowa optymalizacja)	Czas przetwarzania (s)
~100%	1137		893	0,048489
~75%	857	0,045751	604	0,037682
~50%	572	0,05869	393	0,014659
~25%	286	0,069402	188	0,006483
~2,5%	27	0,074692	26	0,000029

Siatka modelu z małą liczbą wierzchołków wyjściowych uzyskana przy pomocy samej funkcji błędu średniokwadratowego przedstawia się na rysunku 4.15.



Rysunek 4.15: Siatka modelu z bardzo małą liczbą wierzchołków wyjściowych (~10%). Poniżej tej wartości model przestaje przypominać oryginał

5 Konkluzja

Główne cele projektu zostały osiągnięte. Algorytm optymalizacji siatki został zaimplementowany i dla wybranego modelu generuje nową siatkę, zachowując jego oryginalną topologię. Program może jednak zostać wzbogacony o wiele nowych elementów, a niektóre jego błędy – naprawione.

5.1 Pomysły na dalszy rozwój aplikacji

Jednym z pierwszych elementów aplikacji, które można usprawnić, jest funkcja `loadObj` z klasy `ObjLoader` (sekcja 3.5). Metoda ta bazuje głównie na wczytywaniu kolejnych linii pliku OBJ i liczeniu znaków białych, dlatego jej działanie może się różnić w zależności od sposobu, w jaki autor zapisał swój model 3D. Przez to dla niektórych plików aplikacja może stworzyć niekompletną siatkę albo zwrócić błąd. Metodę można napisać wykorzystując inną taktykę albo skorzystać z biblioteki, np. Assimp [30].

Do usprawnienia wczytywania obiektów można zmienić sposób podawania do nich ścieżki. W tym momencie należy wpisać ją w odpowiednie pole na panelu użytkownika, jednak proces może zostać ułatwiony przez wykorzystanie np. biblioteki ImGuiFileDialog [31]. Sterowanie kamerą również mogłoby zostać poprawione, by przypominało bardziej intuicyjne poruszanie się po programie Blender, wykorzystujące kółko myszy i pozwalające na większe możliwości rotacji i przesuwania kamery.

Aby modele wyjściowe były jeszcze dokładniejsze, algorytm z metody `simplifyMesh` (sekcja 3.6) mógłby zostać wzbogacony o łączenie par niebędących krawędziami, ale znajdujących się w bliskiej odległości ustalonej przez parametr *threshold*. Dodatkowo ustalanie nowej pozycji wierzchołka tak, by zminimalizować błąd $\Delta(\boldsymbol{v})$, może odbywać się według wzoru:

$$\boldsymbol{v} = \begin{bmatrix} q_{1,1} & q_{1,2} & q_{1,3} & q_{1,4} \\ q_{1,2} & q_{22} & q_{2,3} & q_{2,4} \\ q_{1,3} & q_{2,3} & q_{3,3} & q_{3,4} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \tag{5.1}$$

gdzie q – elementy macierzy $\boldsymbol{Q}$ i pod warunkiem, że macierz jest odwracalna. W przeciwnym wypadku znajdujemy optymalną pozycje wierzchołka na krańcach lub na środku pary. Algorytm mógłby również zostać przyspieszony. W tabeli 5.1 znajdują się niektóre wyniki czasowe aplikacji porównane z tymi zawartymi w pracy opisującej algorytm błędu średniokwadratowego, o podobnej licznie wierzchołków [2].

Tabela 5.1: Porównanie pomiarów czasów optymalizacji siatki w aplikacji projektowej oraz w pracy opisującej optymalizację powierzchni z użyciem błędu średniokwadratowego

Pomiary w aplikacji projektowej			Pomiary zawarte w [2]		
Początkowa ilość wierzchołków	Końcowa ilość wierzchołków	Czas (s)	Początkowa ilość wierzchołków	Końcowa ilość wierzchołków	Czas (s)
7116	34	4,83	5804	10	0,91
1137	22	0,07	1200	10	0,42
62338	296	474,31	69451	10	15,3

Dla modelu o niewielkiej ilości wierzchołków algorytm zaimplementowany w niniejszym projekcie poradził sobie szybciej, ale w przypadku większych modeli czas obliczania jest nieporównywalnie dłuższy, więc należy zastanowić się nad jego optymalizacją.

Implementacja algorytmu izotropowego (sekcja 2.2.4) wymaga naprawienia błędu – po użyciu metody `split` w jednym z wektorów struktury może zajść nieprawidłowość, przez co kolejne wywołanie tej metody lub `collapse` doprowadza do wyłączenia programu, dlatego na ten moment algorytmu nie można umieścić w pętli. Niektóre elementy na rysunku 4.9 c) powstałe przez utworzenie nowych par metodą `split` wymagają dalszego przetworzenia.

Podczas tworzenia aplikacji podjęto próbę zaimplementowania algorytmu tworzącego siatkę na bazie trójkątów równobocznych (sekcja 2.2.2), zakończoną na wdrożeniu operacji kroku lokalnego. W przyszłości można wrócić do kwestii pisania tego algorytmu i spróbować napisać go ponownie.

Dobrym pomysłem jest umożliwienie przetwarzania siatki algorytmem średniokwadratowym w formie opcjonalnej animacji, która z pewnością wydłużyłaby czas oczekiwania, ale dałaby możliwość śledzenia efektownego usuwania kolejnych wierzchołków i łączenia par.

Pomimo wielu algorytmów optymalizacji siatki 3D wciąż potrzebne są nowe, szczególnie takie bazujące na przetwarzaniu czworokątów. Aplikacja powinna móc zarządzać również tym typem poligonów, jednak ze względu na dużo prostsze zarządzanie trójkątami w OpenGL oraz niewielką ilość dostępnych i przystępnie napisanych materiałów traktujących o tej tematyce, priorytet wzięła kwestia doprowadzenia do utworzenia jednorodnej siatki. Jednak to ograniczenie nie powinno być trudne do wprowadzenia, biorąc pod uwagę, że każdy czworokąt składa się z dwóch trójkątów.

W celu zwiększenia użyteczności aplikacji, można zaimplementować procedurę zapisu nowej siatki do pliku OBJ w celu jej późniejszego wykorzystania w innych programach.

5.2 Wniosek końcowy

Przy tworzeniu projektu zapoznano się głębiej z pojęciem grafiki 3D i operacjami na niej wykonywanymi, technologią OpenGL i jej wykorzystaniem przy pomocy odpowiednich bibliotek oraz wieloma algorytmami optymalizacji siatki 3D.

Jest to proces niezwykle istotny, gdyż moc obliczeniowa i pamięć komputerów nieustannie rośnie i pozwala na przetwarzanie trójwymiarowych modeli o coraz większej liczbie poligonów, przy czym duża część z nich wpływa na wrażenia wizualnie jedynie nieznacznie, a przy tym niepotrzebnie zajmuje zbyt dużo zasobów komputera. Zamiast tworzyć model od nowa, algorytmy optymalizacji usuwają nadmiarowe wierzchołki z minimalnym udziałem użytkownika, zachowując przy tym oryginalny kształt.

Implementacja przynajmniej jednego takiego algorytmu powiodła się, a aplikacja stanowi solidną podstawę pod dużo większy, dokładniejszy i użyteczniejszy projekt.

6 Załącznik

Hiperłącze do repozytorium na platformie GiHub: [32].

Plik wykonywalny aplikacji wraz z potrzebnymi dynamicznymi bibliotekami, skryptami GLSL i plikami OBJ znajdują się w folderze „Release”.

W celu uruchomienia projektu w Visual Studio IDE należy wypakować elementy archiwum „linking.zip” do folderu o nazwie „linking”.

Bibliografia

[1] https://www.techopedia.com/definition/31616/polygon-mesh, dostęp w dniu 02.12.2022

[2] Michael Garland, Paul S. Heckbert, *Surface Simplification Using Quadric Error Metrics*. http://www.cs.cmu.edu/~garland/Papers/quadrics.pdf, dostęp w dniu 30.11.2022

[3] Arkadiusz Trojanowski, *Animacja stworzona na potrzeby przedmiotu „Wizualizacja i grafika komputerowa”*. https://youtu.be/GqdilHNkLx8, dostęp w dniu 02.12.2022

[4] Joey de Vires, *Learn OpenGL*. https://learnopengl.com/Getting-started/Transformations, dostęp w dniu 30.11.2022

[5] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/#the-view-matrix, dostęp w dniu 01.12.2022

[6] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/#the-projection-matrix, dostęp w dniu 01.12.2022

[7] Dawar Khan, Alexander Plopski, *Surface Remeshing: A Systematic Literature Review of Methods and Research Directions*. https://par.nsf.gov/servlets/purl/10293572, dostęp w dniu 02.12.2022

[8] Sébastien Valette, Jean-Marc Chassery, *Generic remeshing of 3D triangular meshes with metric-dependent discrete Voronoi Diagrams*. https://hal.archives-ouvertes.fr/hal-00537025/document, dostęp w dniu 02.12.2022

[9] Jiang Du, Yao Jin, *As-equilateral-as-possible surface remeshing*. https://www.jstage.jst.go.jp/article/jamdsm/9/4/9_2015jamdsm0052/_pdf/-char/en, dostęp w dniu 02.12.202

[10] https://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm, dostęp w dniu 03.12.2022

[11] Yuan Yao, *Adaptive Feature-Preserving Isotropic Remeshing*. https://www.cs.ubc.ca/~rozentil/data/remesh.pdf, dostęp w dniu 02.12.2022

[12] Bjarne Stroustup, *Język C++. Kompendium Wiedzy. Wydanie IV*

[13] https://octoverse.github.com/2022/top-programming-languages, dostęp w dniu 04.12.2022

[14] https://isocpp.org/std/status, dostęp w dniu 04.12.2022

[15] https://visualstudio.microsoft.com/vs/getting-started, dostęp w dniu 04.12.2022

[16] https://www.khronos.org/opengl/wiki, dostęp w dniu 04.12.2022

[17] https://glew.sourceforge.net, dostęp w dniu 04.12.2022

[18] https://www.glfw.org, dostęp w dniu 04.12.2022

[19] https://github.com/g-truc/glm, dostęp w dniu 04.12.2022

[20] https://github.com/ocornut/imgui, dostęp w dniu 04.12.2022

[21] Jay Versluis, *What is Yaw, Pitch and Roll in 3D axis values.* https://www.versluis.com/2020/09/what-is-yaw-pitch-and-roll-in-3d-axis-values/, dostęp w dniu 06.12.2022

[22] https://thebookofshaders.com/01, dostęp w dniu 07.12.2022

[23] https://learnopengl.com/Getting-started/Shaders, dostęp w dniu 07.12.2022

[24] https://tangrams.readthedocs.io/en/main/Overviews/Lights-Overview/, dostęp w dniu 07.12.2022
[25] https://learnopengl.com/Advanced-OpenGL/Depth-testing, dostęp w dniu 10.12.2022
[26] https://www.hp.com/us-en/shop/tech-takes/what-is-anti-aliasing, dostęp w dniu 10.12.2022
[27] https://free3d.com/3d-model/capybara-v1--169437.html, dostęp w dniu 11.12.2022
[28] https://free3d.com/3d-model/bird-v1--92079.html, dostęp w dniu 13.12.2022
[29] https://free3d.com/3d-model/cat-v1--522281.html, dostęp w dniu 13.12.2022
[30] https://assimp-docs.readthedocs.io/en/v5.1.0/, dostęp w dniu 14.12.2022
[31] https://github.com/aiekick/ImGuiFileDialog, dostęp w dniu 14.12.2022
[32] https://github.com/etroark/aghust-thesis, dostęp w dniu 15.12.2022