UNIWERSYTET ZIELONOGÓRSKI

WYDZIAŁ NAUK ŚCISŁYCH I PRZYRODNICZYCH

kierunek Inżynieria danych

specjalność Projektowanie i obsługa systemów analitycznych

Dawid Drapiński

Modelowanie 3D na podstawie zdjęć

Photo-based 3D modelling

Promotor pracy
dr Jacek Bojarski

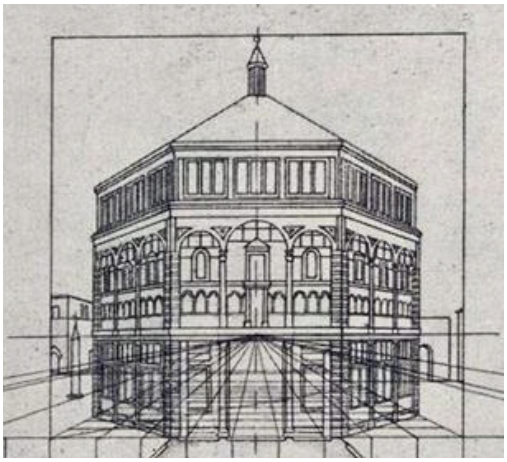
Zielona Góra, 2025

Spis treści

Wstęp

Podstawową metodą odzwierciedlania trójwymiarowego świata na dwuwymiarowej płaszczyźnie w sztuce i architekturze jest perspektywa. Początki jej świadomego wykorzystania jako narzędzia przedstawiania rzeczywistości sięgają starożytności, gdzie Grecy i Rzymianie eksperymentowali z technikami perspektywicznymi. Przykładem tego są freski w willi Publiusza Fanniusza Synistora w Boscoreale, w którym zastosowano nawet dwa typy projekcji jednocześnie, aby osiągnąć efekt głębi i przestrzeni. Egipscy artyści traktowali perspektywę w sposób symboliczny, skupiając się na przedstawieniu doświadczeń religijnych zamiast realistycznej przestrzeni. Dominowała perspektywa hieratyczna, gdzie wielkość i pozycja postaci na obrazie odzwierciedlały ich duchowe znaczenie, a nie rzeczywiste relacje przestrzenne. Pojęcie perspektywy zyskało jednak swoje prawdziwe znaczenie w epoce renesansu, kiedy artyści i architekci we Włoszech rozpoczęli naukowe badania nad geometrią i optyką, które pozwoliły na bardziej precyzyjne przedstawienie przestrzeni. Przełomowym były eksperymenty florenckiego rzeźbiarza i architekta Filippo Brunelleschiego, którego uważa się za twórcę zasad perspektywy linearnej. Włoch przy pomocy luster szkicuje perfekcyjne odwzorowanie Baptysterium San Giovanni we Florencji. Na rysunku tym umieszcza on linie, które prowadzą do jednego punktu w linii horyzontu i pozwalają mu na dokładne przedstawienie trójwymiarowego budynku. W centrum umieszcza dodatkowo mały otwór i podnosi rysunek na wysokość swojej twarzy i obraca go tyłem. W drugiej ręce trzyma natomiast lustro, które gdy znajduje się blisko niego, odbija jego obraz, oddalone odsłania prawdziwe Baptysterium. Brunelleschi zobaczył, że jego dzieło idealnie odzwierciedla budynek. Kilka lat później w 1435 roku, artysta i architekt Leon Battista Alberti ujął idee Brunelleschiego w swojej pracy *De Pictura*, które staje się fundamentem sztuki zachodniej

Baptysterium San Giovanni, Florencja, Włochy. *Zdjęcie Bradley N. Weber*



Budynek Baptysterium San Giovanni w technice perspektywy liniowej Filippo Brunelleschiego, *Źródło Instytut Historii Sztuki we Florencji, Instytut Maxa Plancka. © 2006, SCALA, Florencja / ART RESOURCE, N.Y.*

W niniejszej pracy dyplomowej zajmiemy się zrozumieniem i praktycznym wykorzystaniem przekształceń wykonywanych przez silniki graficzne, umożliwiających tworzenie złożonych scen trójwymiarowych w grach komputerowych oraz wirtualnych spacerach, takich jak Google Street View. Wprowadzimy teoretyczne podstawy geometrii z uwzględnieniem przestrzeni wektorowych, transformacji liniowych i rzutowania perspektywicznego. Następnie przeanalizujemy kolejność i zależności między układami współrzędnych w potoku graficznym, koncentrując się na operacjach takich jak translacja, rotacja i skalowanie. Dodatkowo przeprowadzimy przykłady obliczeniowe, dla lepszego zrozumienia omawianych zagadnień. Omówimy wpływ parametrów kamery na finalny wygląd sceny, obejmujący efekty wynikające z rzutowania perspektywicznego. W części praktycznej zaprezentujemy proces wykorzystania zdjęć panoramicznych do stworzenia wirtualnego spaceru, w tym konwersję zdjęć i budowę interaktywnego modelu otoczenia. Ostatecznym celem pracy będzie stworzenie aplikacji webowej pozwalającej na wirtualny spacer w Instytucie Matematyki Uniwersytetu Zielonogórskiego.

1 Geometria

Geometria jest jednym z fundamentów matematyki, której znaczenie wykracza daleko poza czysto teoretyczne rozważania. Od starożytnej Grecji, kiedy to Euklides opracował swoje „Elementy", po współczesne zastosowania w grafice komputerowej, inżynierii czy analizie danych, geometria stanowi narzędzie pozwalające opisywać i modelować otaczającą nas rzeczywistość. W rozdziale tym przyjrzymy się klasycznym pojęciom jak i zagadnieniom związanymi z rzutowaniem, które pomogą nam opisywać procesy zachodzące w następnych rozdziałach pracy.

Przestrzeń

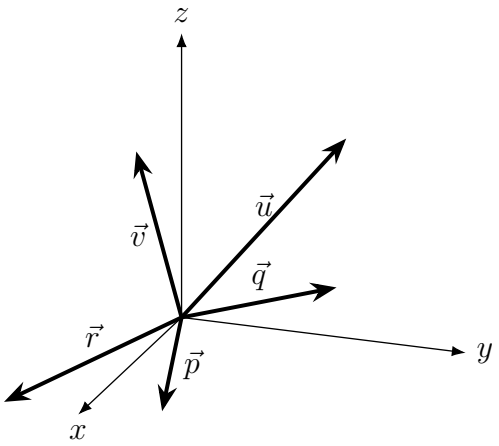
Przestrzenią w której będziemy się poruszać będzie przestrzeń liniowa. Definiujemy ją jako niepusty zbiór $\mathbf{V}$ w którym

1. dla dowolnych elementów $\overrightarrow{u}, \overrightarrow{v} \in \mathbf{V}$ określona jest ich suma $\overrightarrow{u} + \overrightarrow{v} \in \mathbf{V}$,
2. dla każdej liczby $\alpha \in \mathbb{R}$ i dla każdego elementu $\overrightarrow{u} \in \mathbf{V}$ określony jest ich iloczyn $\alpha\overrightarrow{u} \in \mathbf{V}$.

Elementy takiej przestrzeni nazywamy wektorami a rzeczywistą przestrzeń liniową nazywamy przestrzenią wektorową. Dalej w pracy będziemy poruszać się w przestrzeni $\mathbb{R}^3$, definiowaną jako zbiór dowolnych uporządkowanych n liczb rzeczywistych

$$\mathbb{R}^3 = \{\overrightarrow{x} = (x_1, x_2, x_3)^T : x_k \in \mathbb{R} \text{ dla } 1 \leqslant k \leqslant 3\}.$$

Na poniższym rysunku przedstawiono poglądowy wykres przestrzeni wektorowej $\mathbb{R}^3$, ilustrujący przykładowe wektory w tej przestrzeni



Rysunek 1.1: Wykres poglądowy przestrzeni wektorowej $\mathbf{R}^3$. *Opracowanie własne na podstawie* [13].

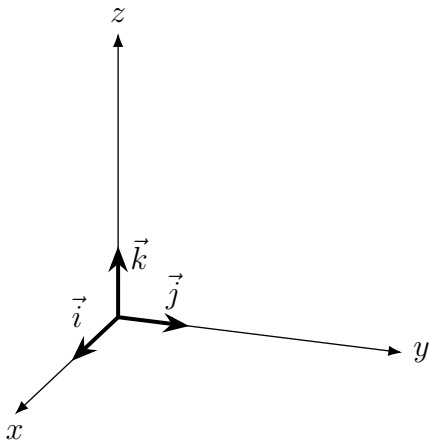
Baza przestrzeni

Układ trzech wektorów $\overrightarrow{e_1}$, $\overrightarrow{e_2}$, $\overrightarrow{e_3}$ tworzy bazę przestrzeni wektorowej $\mathbb{R}^3$, jeśli są one liniowo niezależne. O liniowej niezależności mówimy wtedy, gdy jedynym rozwiązaniem równania

$$\alpha_1 \overrightarrow{e_1} + \alpha_2 \overrightarrow{e_2} + \alpha_3 \overrightarrow{e_3} = \overrightarrow{0} \tag{1.1}$$

jest $\alpha_1 = \alpha_2 = \alpha_3 = 0$.
Wyjątkowym układem wektorów spełniającym powyższy warunek jest baza standardowa. Taką bazę dla przestrzeni $\mathbb{R}^3$ tworzy trójka wektorów o długości 1, nazywanych wersorami, których układ widoczny jest poniżej

$$\overrightarrow{i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \overrightarrow{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \overrightarrow{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$



Rysunek 1.2: Wykres poglądowy standardowej bazy przestrzeni wektorowej $\mathbb{R}^3$. *Opracowanie własne na podstawie* [13].

Iloczyn skalarny

Iloczynem skalarnym w przestrzeni wektorowej $\mathbb{R}^3$ nazywamy odwzorowanie, które każdej parze wektorów $\overrightarrow{u}, \overrightarrow{v} \in \mathbb{R}^3$ przyporządkowuje liczbę rzeczywistą oznaczaną jako $(\overrightarrow{u}, \overrightarrow{v})$ i określaną wzorem

$$(\overrightarrow{u}, \overrightarrow{v}) = u^T v = u_x v_x + u_y v_y + u_z v_z, \tag{1.2}$$

gdzie $\overrightarrow{u} = (u_x, u_y, u_z)^T$ oraz $\overrightarrow{v} = (v_x, v_y, v_z)^T$.
Wykorzystując iloczyn skalarny, możemy wprowadzić pojęcie normy wektora, czyli jego długości

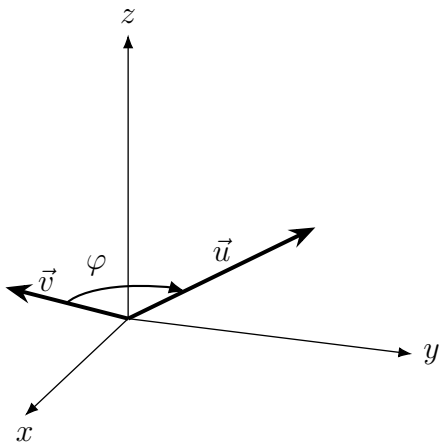
$$|\overrightarrow{u}| = \sqrt{(\overrightarrow{u}, \overrightarrow{u})}. \tag{1.3}$$

Iloczyn skalarny można również przedstawić w postaci geometrycznej

$$(\overrightarrow{u}, \overrightarrow{v}) = |\overrightarrow{u}| \cdot |\overrightarrow{v}| \cdot \cos\varphi, \tag{1.4}$$

gdzie φ to kąt między wektorami, przyjmujący wartości z przedziału $[0, \pi]$ przy założeniu, że $\overrightarrow{u} \neq \overrightarrow{0}$ oraz $\overrightarrow{v} \neq \overrightarrow{0}$. W przypadku, gdy jeden z wektorów jest zerowy, kąt między nimi nie jest zdefiniowany.

Warto zauważyć, że iloczyn skalarny przyjmuje wartość zero wtedy i tylko wtedy, gdy $\cos\varphi = 0$, co zachodzi, gdy $\varphi = \frac{\pi}{2}$. Oznacza to, że wektory $\overrightarrow{u}$ i $\overrightarrow{v}$ są prostopadłe, czyli ortogonalne. Przykładowy układ wektorów $\overrightarrow{v}$, $\overrightarrow{u}$ i kąta φ pomiędzy nimi można przedstawić następująco



Rysunek 1.3: Wykres poglądowy iloczynu skalarnego dwóch wektorów $\overrightarrow{v}$, $\overrightarrow{u}$ wraz z zaznaczonym kątem φ. *Opracowanie własne na podstawie* [13].

Punkt

Punkt to podstawowy bezwymiarowy obiekt w geometrii, który poza położeniem nie ma innych cech. Jest swego rodzaju „atomem" geometrii, z którego zbudowane są inne obiekty, tj. odcinki, proste, figury, bryły, płaszczyzny. W przestrzeni Euklidesa $\mathbb{R}^3$ położenie punktu określane jest przez trójkę liczb rzeczywistych $(x, y, z)^T$, które nazywamy jego współrzędnymi:

- x: położenie wzdłuż osi X odpowiadającej wersorowi $\overrightarrow{i}$,
- y: położenie wzdłuż osi Y odpowiadającej wersorowi $\overrightarrow{j}$,
- z: położenie wzdłuż osi Z odpowiadającej wersorowi $\overrightarrow{k}$.

Zauważmy, że punkt $P = (x, y, z)^T$ można traktować jako końcowy punkt wektora $\overrightarrow{v} = (x, y, z)^T$ zaczepionego w początku układu współrzędnych $O = (0, 0, 0)^T$, a przypomnijmy

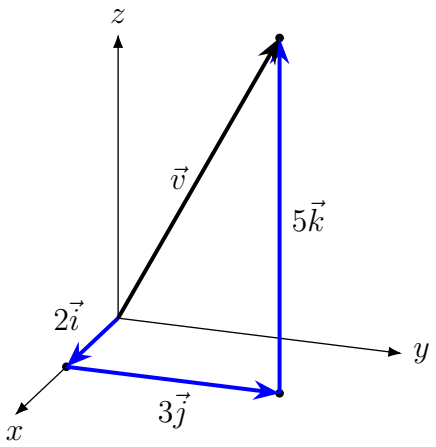
$$\overrightarrow{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = x\,\overrightarrow{i} + y\,\overrightarrow{j} + z\,\overrightarrow{k}.$$

Przykład 1.1. Punkt jako wektor

Punkt

$$P = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$$

znajduje się 2 jednostki wzdłuż osi X, 3 jednostki wzdłuż osi Y i 5 jednostek wzdłuż osi Z. Przedstawienie wektora $\overrightarrow{v}$ przy pomocy wersorów bazy standardowej będzie wyglądać następująco



Zatem inaczej, wektor $\overrightarrow{v}$ opisuje przemieszczenie od początku O do punktu P. Taki wektor $\overrightarrow{v}$ często określa się jako wektor zaczepiony. Każdy punkt $P = (x, y, z)^T$ wyznacza dokładnie jeden wektor zaczepiony $\overrightarrow{OP} = (x, y, z)^T$.

Prosta

Prosta L w przestrzeni trójwymiarowej $\mathbb{R}^3$ definiowana jest jako zbiór punktów spełniających równanie parametryczne, wyrażone za pomocą punktu $P_0 = (x_0, y_0, z_0)^T$ oraz wektora kierunkowego $\overrightarrow{v} = (v_x, v_y, v_z)^T$ zaczepionego w punkcie P_0. Równanie to przyjmuje wtedy postać

$$L: \; P(t) = P_0 + \overrightarrow{v}t, \quad t \in \mathbb{R}. \tag{1.5}$$

Równanie to w postaci rozwiniętej wygląda następująco

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} t \quad \equiv \quad \begin{cases} x = x_0 + v_x t, \\ y = y_0 + v_y t, \\ z = z_0 + v_z t. \end{cases} \tag{1.6}$$

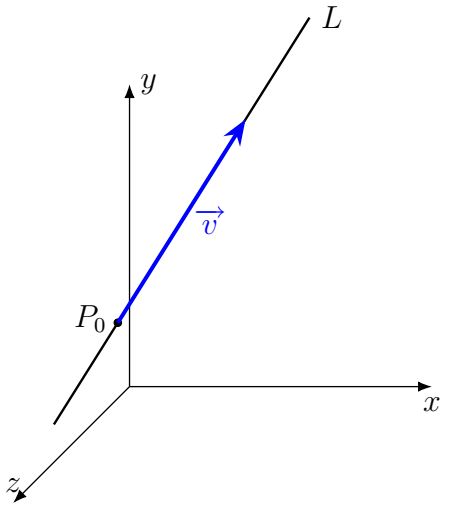
Prosta zdefiniowana w ten sposób jest nieograniczona i jednoznacznie określona przez punkt P_0 oraz wektor $\overrightarrow{v}$. Parametr t opisuje poruszanie się wzdłuż prostej w obu kierunkach, dla $t > 0$ zgodnie ze zwrotem wektora $\overrightarrow{v}$, a dla $t < 0$ przeciwnie.

Przykład 1.2.

Rozważmy prostą L przechodzącą przez punkt $P_0 = (1, 2, 3)^T$ i mającą wektor kierunkowy $\overrightarrow{v} = (4, 5, 6)^T$. Równanie parametryczne tej prostej wygląda następująco

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + t \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \quad \equiv \quad \begin{cases} x = 1 + 4t, \\ y = 2 + 5t, \\ z = 3 + 6t. \end{cases}$$

Geometrycznie będzie prezentować się następująco



Odcinek

Odcinek w przestrzeni $\mathbb{R}^3$ rozumiemy jako fragment prostej, zadany dwoma punktami należącymi do tej prostej. Jeśli dane są dwa różne punkty $P_A = (x_A, y_A, z_A)^T$ oraz $P_B = (x_B, y_B, z_B)^T$, to odcinek $\overline{P_A P_B}$ jest definiowany jako zbiór punktów, dających sie zapisać w postaci kombinacji liniowej tych punktów. W szczególności, każdy punkt P należący do odcinka $\overline{P_A P_B}$ przedstawimy jako

$$\overline{P_A P_B} : \; P(\lambda) = (1 - \lambda)P_A + \lambda P_B, \quad \lambda \in [0, 1]. \tag{1.7}$$

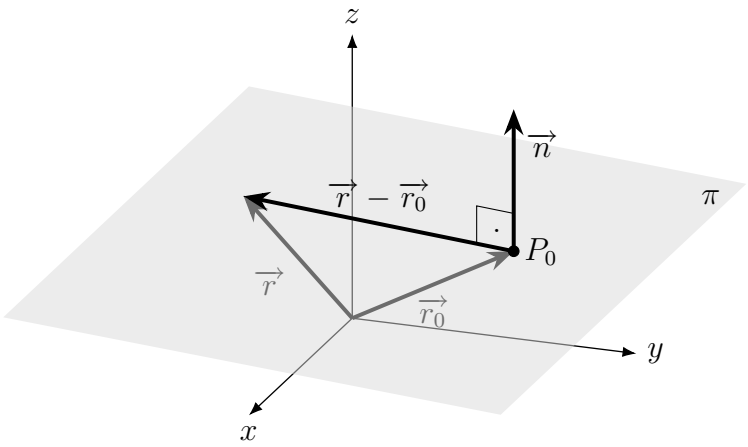
Innymi słowy, parametr λ opisuje „przemieszczanie się" od punktu P_A do P_B, tak że dla $\lambda = 0$ otrzymujemy punkt początkowy P_A, a dla $\lambda = 1$ punkt końcowy P_B.

Płaszczyzna

Płaszczyzna π w przestrzeni trójwymiarowej $\mathbb{R}^3$ definiowana jest jako zbiór punktów spełniających równanie normalne płaszczyzny, wykorzystujące punkt $P_0 = (x_0, y_0, z_0)^T$, przez który przechodzi płaszczyzna, jej wektor wodzący $\overrightarrow{r_0}$ i prostopadły do niej wektor normalny $\overrightarrow{n} = (x_n, y_n, z_n)^T$

$$\pi : \; ((\overrightarrow{r} - \overrightarrow{r_0}), \overrightarrow{n}) = 0,$$

gdzie $\overrightarrow{r} = (x, y, z)^T$ jest wektorem wodzącym punktów przestrzeni [12]. Powyższą definicję możemy zobrazować następująco



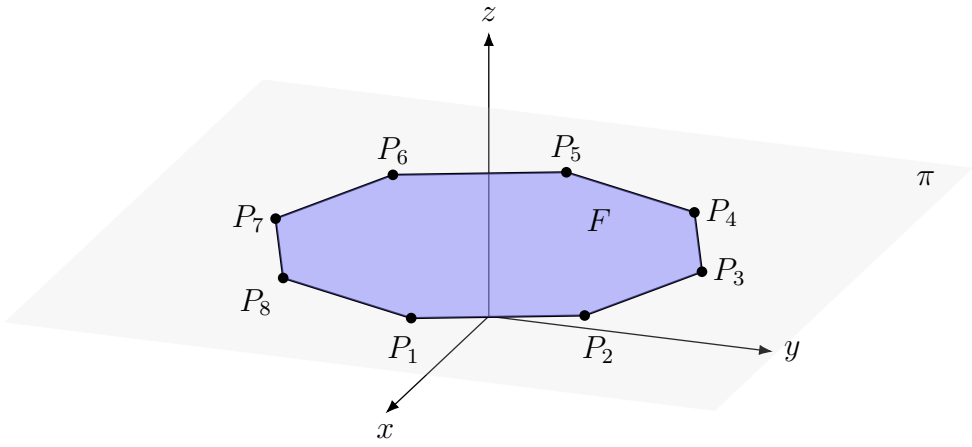
Rysunek 1.4: Wykres płaszczyzny wyznaczonej przez punkt i wektor normalny. *Opracowanie własne na podstawie* [13].

Wielokąt

Wielokątem F nazywamy pewien ograniczony obszar należący do dowolnej płaszczyzny π, składający się z co najmniej 3 niewspółliniowych punktów $P_1, P_2, ..., P_k$ których pary są połączone kolejno odcinkami, to znaczy $\overline{P_1P_2}, \overline{P_2P_3}, ..., \overline{P_{k-1}P_k}$ oraz ostatni punkt połączony jest z pierwszym $\overline{P_kP_1}$ tworząc zamkniętą linię łamaną

$$F : \overline{P_1P_2...P_k}, \quad k \in \mathbb{N}, k \geqslant 3. \tag{1.8}$$

Punkty te nazywamy wierzchołkami wielokąta, natomiast odcinki łączące kolejne wierzchołki – jego bokami [7]. Poniższy przykład ośmiokąta ukazuje zamkniętą linię łamaną złożoną z ośmiu punktów i odcinków



Rysunek 1.5: Przykład wielokąta, ośmiokąt. *Opracowanie własne.*

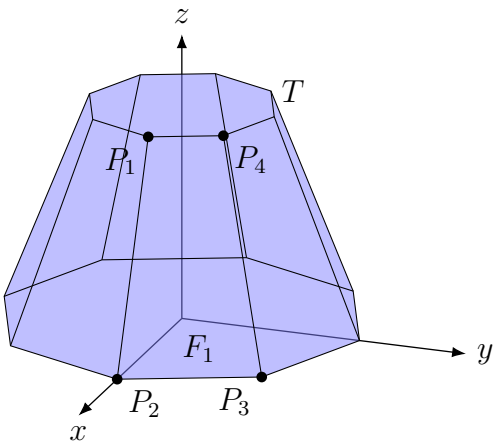
Bryła

Bryłą T nazywamy trójwymiarowy obiekt geometryczny, który składa się ze skończonej liczby wielokątów $F_1, F_2, ..., F_k$ połączonych w taki sposób, że tworzą zamknięty, spójny

i jednoznacznie wyznaczony obszar

$$T : \langle F_1, F_2, \ldots, F_k \rangle, \quad k \in N, k \geqslant 4.$$

Przykład tak zdefiniowanej bryły przedstawiono na poniższym rysunku. Graniastosłup ścięty o podstawie ośmiokąta ilustruje ideę łączenia wielokątów (w tym przypadku dwóch ośmiokątów i ośmiu trapezów) w zamkniętą, trójwymiarową strukturę

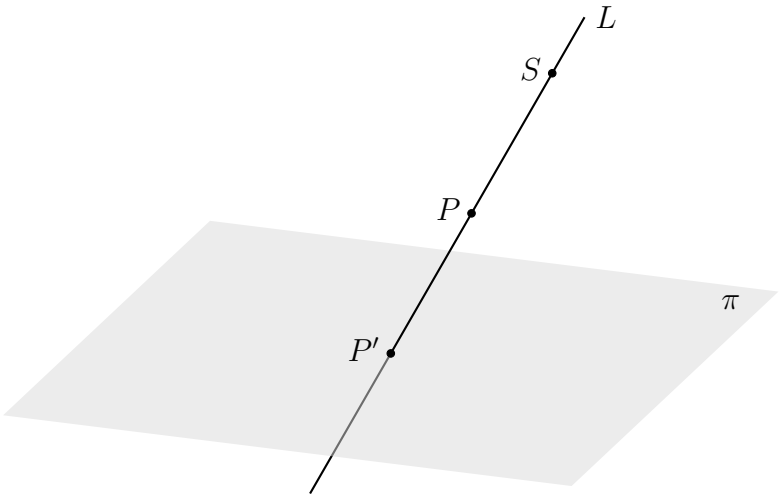


Rysunek 1.6: Przykład bryły, graniastosłup ścięty o podstawie ośmiokąta. Dla przejrzystości zaznaczono wierzchołki tylko jednej z bocznych ścian. *Opracowanie własne.*

Rzut środkowy

Dla zadanej płaszczyzny π oraz nienależącego do niej punktu S, rzutem środkowym R nazywamy odwzorowanie dowolnego punktu P na tej płaszczyźnie, przez znalezienie punktu przecięcia prostej przechodzącej przez punkty S i P oraz płaszczyzny π. Mechanizm ten zobrazowano na poniższym rysunku

$$P' = L \cap \pi \tag{1.9}$$



Rysunek 1.7: Przykład rzutu środkowego na płaszczyznę. *Opracowanie własne.*

Płaszczyznę π, na którą wykonywany jest rzut, nazywamy rzutnią, natomiast punkt S, z którego prowadzone są proste przechodzące przez rzutowane punkty, nazywamy środkiem rzutu.

Przykład 1.3.

Załóżmy następujące elementy rzutu:

- środek rzutu $S = \begin{pmatrix} 2 \\ -6 \\ 7 \end{pmatrix}$,
- rzutnia $\pi : \left(\left(\overrightarrow{r} - \begin{pmatrix} 0 \\ 5 \\ 4 \end{pmatrix} \right), \begin{pmatrix} 0 \\ 1 \\ -2 \end{pmatrix} \right)$,
- wielokąt $F : \overline{P_1P_2P_3}$, gdzie

$$P_1 = \begin{pmatrix} 4 \\ -4 \\ 3 \end{pmatrix} \quad P_2 = \begin{pmatrix} 1 \\ -3 \\ \frac{7}{2} \end{pmatrix} \quad P_3 = \begin{pmatrix} 2 \\ -\frac{3}{2} \\ 3 \end{pmatrix}.$$

Aby wyznaczyć równanie prostej L łączącej punkty P_1 i S, wykorzystujemy parametryczne równanie prostej (1.5)

$$L : P(t) = P_1 + (S - P_1)t,$$

Po podstawieniu współrzędnych punktów otrzymamy

$$P(t) = \begin{pmatrix} 4 \\ -4 \\ 3 \end{pmatrix} + \begin{pmatrix} -2 \\ -2 \\ 4 \end{pmatrix} t,$$

Zatem równanie parametryczne prostej łączącej punkty P_1 i S wygląda następująco

$$\begin{cases} x(t) = 4 - 2t, \\ y(t) = -4 - 2t, \\ z(t) = 3 + 4t. \end{cases}$$

Aby znaleźć punkt wspólny $P_1' = L \cap \pi$, musimy znaleźć zmienną t. W tym celu podstawiamy równanie parametryczne prostej L do równania płaszczyzny π

$$\left(\begin{pmatrix} 4 - 2t \\ -4 - 2t \\ 3 + 4t \end{pmatrix} - \begin{pmatrix} 0 \\ 5 \\ 4 \end{pmatrix} \right) \circ \begin{pmatrix} 0 \\ 1 \\ -2 \end{pmatrix} = -7 - 10t$$

$$\begin{gathered} -7 - 10t = 0 \\ t = -\tfrac{7}{10}. \end{gathered}$$

Przykład 1.3.

Podstawiamy t do równania parametrycznego prostej, aby znaleźć współrzędne punktu P_1'

$$\begin{cases} x(-\frac{7}{10}) = 4 - 2 \cdot \left(-\frac{7}{10}\right) = \frac{27}{5}, \\ y(-\frac{7}{10}) = -4 - 2 \cdot \left(\frac{7}{10}\right) = -\frac{13}{5}, \\ z(-\frac{7}{10}) = 3 + 4 \cdot \left(\frac{7}{10}\right) = \frac{1}{5}. \end{cases}$$

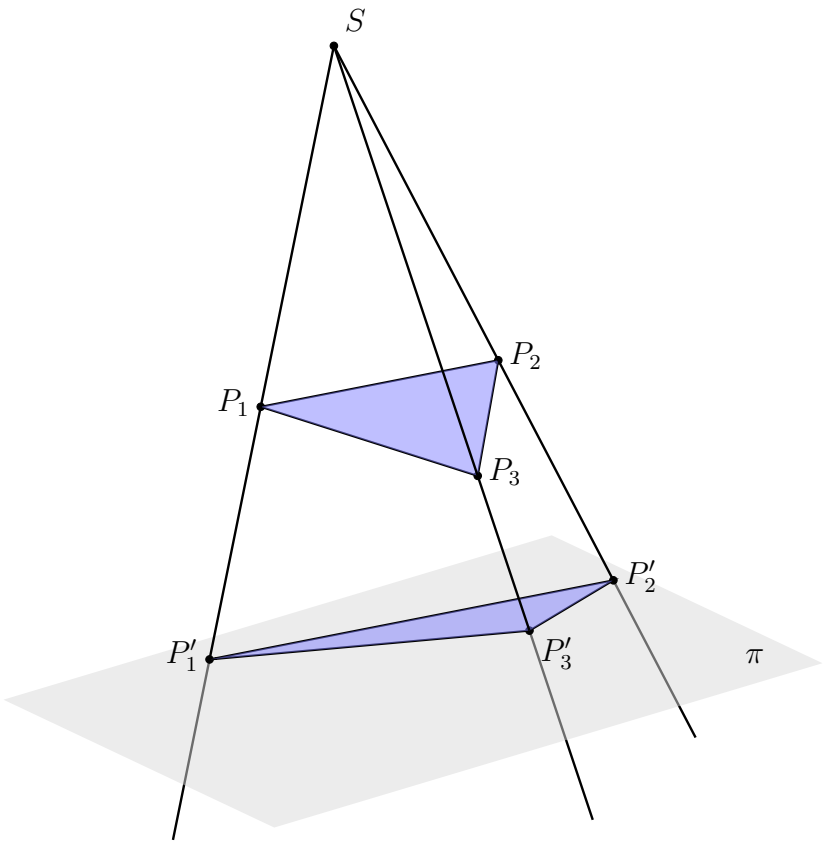
Zatem rzut środkowy punktu P_1 na płaszczyznę π przy użyciu punktu środkowego S ma współrzędne

$$P_1' = \begin{pmatrix} \frac{27}{5} \\ -\frac{13}{5} \\ \frac{1}{5} \end{pmatrix}.$$

Podobny proces należy przeprowadzić dla pozostałych wierzchołków wielokąta F, aby znaleźć ich rzuty P_2' i P_3' na płaszczyznę π. Jednak dla uproszczenia pominiemy te obliczenia i podamy gotowe współrzędne

$$P_2' = \begin{pmatrix} \frac{3}{10} \\ -\frac{9}{10} \\ \frac{21}{20} \end{pmatrix}, \quad P_3' = \begin{pmatrix} 2 \\ \frac{3}{25} \\ \frac{39}{25} \end{pmatrix}.$$

Geometrycznie rzut wygląda następująco



Odwzorowanie walcowe równoodległościowe

Odwzorowanie walcowe równoodległościowe (*equirectangular projection* lub *equidistant cylindrical projection*) stosowane jest w kartografii podczas tworzenia map świata oraz fotografii przy przetwarzaniu zdjęć 360°. Rzutuje punkty z powierzchni bryły sfero-podobnej na boczną powierzchnię walca. Następnie walec ten jest „rozwijany", co pozwala uzyskać płaski widok danej powierzchni [21, 22].
Załóżmy sferę o promieniu r oraz punkcie środkowym P_O. Oś obrotu obiektu sfero-podobnego łączy dwa przeciwległe punkty nazywane biegunami. Bieguny te, połączyć możemy też prowadząc linie po płaszczyźnie bryły i nazywać je będziemy południkami. Płaszczyzna równika to płaszczyzna przechodząca przez punkt P_O i prostopadła do osi obrotu, dzieląca horyzontalnie obiekt na dwie części. Każdy punkt P na sferze można opisać w układzie współrzędnych sferycznych za pomocą dwóch kątów

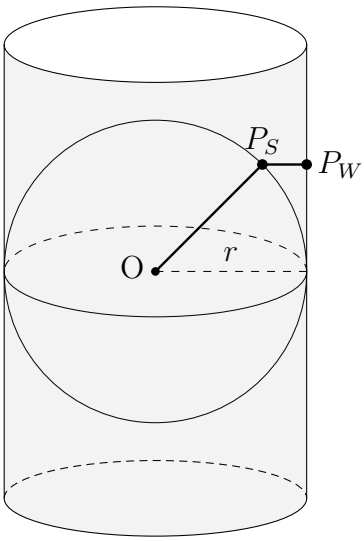
- φ - kąt między płaszczyzną równika a odcinkiem $\overline{P_O P_S}$, przyjmuje wartości $[-\frac{\pi}{2}, \frac{\pi}{2}]$, nazywany szerokością geograficzną,
- λ - kąt mierzony od dowolnego wybranego południka a południka zawierającego punkt P, przyjmuje wartości $[-\pi, \pi]$.

$$P_S = \begin{pmatrix} r\cos(\varphi)\cos(\lambda) \\ r\cos(\varphi)\sin(\lambda) \\ r\sin(\varphi) \end{pmatrix},$$

gdzie współrzędne x i y leżą w płaszczyźnie równika bryły sfero-podobnej, a współrzędna z wskazuje na wysokość punktu nad płaszczyzną. Załóżmy walec, który jest styczny do sfery wzdłuż jej przecięcia z płaszczyzną równika. Jego promień, jest równy promieniowi r sfery, a oś z jest wspólna dla obu brył. Aby odwzorować punkt P_S leżący na sferze na punkt P_W leżący na bocznej powierzchni walca, należy zachować długość geograficzną λ oraz wysokość, która odpowiada szerokości geograficznej φ

$$P_W = \begin{pmatrix} r\cos(\lambda) \\ r\sin(\lambda) \\ r\varphi \end{pmatrix}.$$

Poglądowo tą projekcję można przestawić w następujący sposób

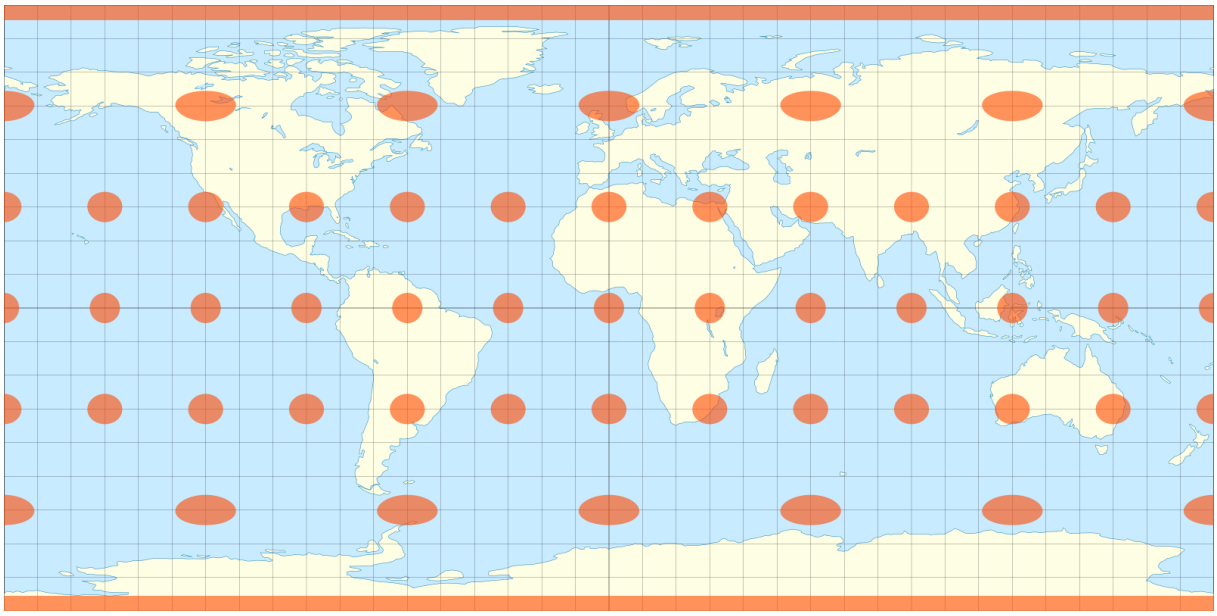


Rysunek 1.8: Przykład kuli umieszczonej wewnątrz walca, na którego płaszczyznę boczną rzucany będzię punkt. *Opracowanie własne na podstawie* [21].

Po „rozwinięciu" walca na płaszczyznę, współrzędne punktu P_C w układzie płaskim (x', y') przyjmują postać

$$x' = r\lambda, \quad y' = r\varphi.$$

W ten sposób odwzorowanie walcowe równoodległościowe przekształca punkty sfery na płaszczyznę, zachowując proporcje kątowe wzdłuż równoleżników i południków [21, 22]. Odwzorowanie to nie zachowuje powierzchni ani kształtów, co prowadzi do deformacji obiektów w miarę oddalania się od równika



Rysunek 1.9: Przykład projekcji walcowej równoodległościowej mapy Ziemi wraz z elipsami zniekształceń Tissot'a. *Źródło* [14].

2 Transformacje liniowe

Wyświetlenie zaprojektowanej przez nas sceny poprzedza wykonanie przez silnik graficzny kilku etapów. Jednym z nich jest odpowiednie ułożenie wszystkich obiektów we wspólnym układzie tak, by osoba oglądająca scenę mogła zrozumieć co się na niej znajduje. Poprawne ułożenie otrzymać można dzięki odpowiednio nałożonym elementarnym transformacją liniowym: skalowaniu, rotacji oraz translacji.

Skalowanie

Skalowanie jest liniowym przekształceniem geometrycznym, które zmienia rozmiar obiektu poprzez mnożenie współrzędnych punktów przez odpowiednie współczynniki skalowania wzdłuż osi układu współrzędnych. W przestrzeni wektorowej R^3 reprezentowane jest za pomocą macierzy diagonalnej M_S o wymiarach 3×3, która składa się z współczynników skalowania s_x, s_y, s_z, gdzie każdy z nich odpowiada jednej osi współrzędnych

$$M_S = \text{scale}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}. \tag{2.1}$$

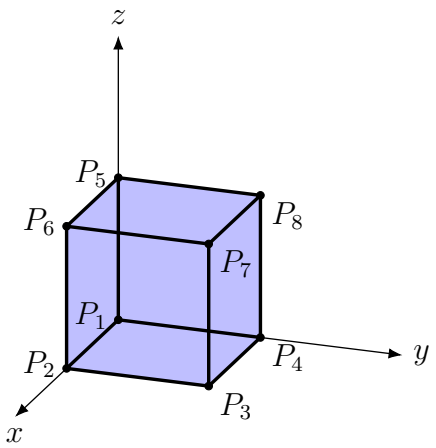
Dla dowolnego punktu $P = (x, y, z)^T$ wynik operacji skalowania wyraża się jako

$$\text{scale}(s_x, s_y, s_z) \cdot P = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \end{pmatrix}.$$

Przykład 2.1. Skalowanie sześcianu

Rozważmy sześcian o wierzchołkach

$$P_1 = (0,0,0)^T, \quad P_2 = (2,0,0)^T, \quad P_3 = (0,2,0)^T, \quad P_4 = (2,2,0)^T,$$
$$P_5 = (0,0,2)^T, \quad P_6 = (2,0,2)^T, \quad P_7 = (0,2,2)^T, \quad P_8 = (2,2,2)^T,$$



Przykład 2.1.

na którym chcemy przeprowadzić operację skalowania z współczynnikami $s_x = \frac{3}{2}$, $s_y = 2$, $s_z = \frac{1}{2}$

$$\text{scale}\left(\frac{3}{2}, 2, \frac{1}{2}\right) = \begin{bmatrix} \frac{3}{2} & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix}.$$

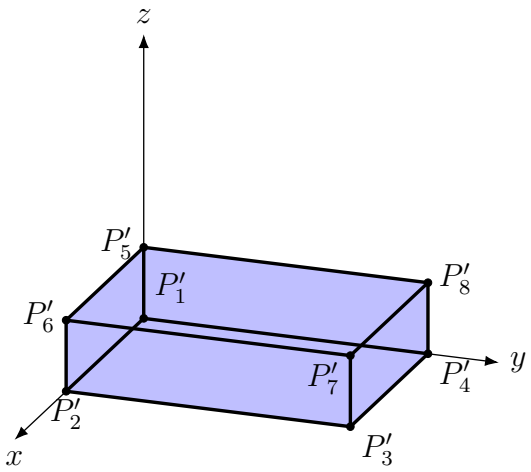
Dla przykładu weźmy punkt $P_8 = (2, 2, 2)^T$, który po przekształceniu będzie miał następujące współrzędne

$$P'_8 = \text{scale}\left(\frac{3}{2}, 2, \frac{1}{2}\right) \cdot P_8 = \begin{bmatrix} \frac{3}{2} & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix} \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 1 \end{pmatrix}.$$

Przekształcone współrzędne wszystkich wierzchołków wynoszą

$$\begin{array}{llll} P'_1 = (0,0,0)^T, & P'_2 = (3,0,0)^T, & P'_3 = (0,4,0)^T, & P'_4 = (3,4,0)^T, \\ P'_5 = (0,0,1)^T, & P'_6 = (3,0,1)^T, & P'_7 = (0,4,1)^T, & P'_8 = (3,4,1)^T. \end{array}$$

Wtedy przeskalowany sześcian wygląda następująco



Rotacja

Rotacja jest liniowym przekształceniem geometrycznym, zmieniającym orientację obiektu wokół wybranej osi. W przestrzeni R^3 rotację można przedstawić za pomocą macierzy obrotu M_{Rx}, M_{Ry} i M_{Rz}, o wymiarach 3×3 i reprezentujących obrót wokół odpowiednio osi x, y i z. Każda macierz jest jednoznacznie zdefiniowana poprzez kąt obrotu θ, który określa wielkość przekształcenia w radianach

$$M_{Rx} = \text{rotate\_x}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \tag{2.2}$$

$$M_{Ry} = \text{rotate\_y}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{2.3}$$

$$M_{Rz} = \text{rotate\_z}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.4}$$

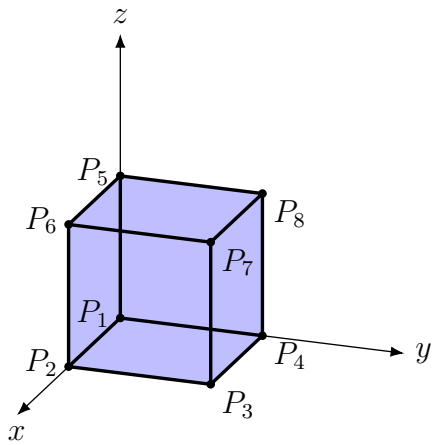
Dla dowolnego punktu $P = (x, y, z)^T$ wynik zastosowania rotacji wokół osi z wyraża się jako

$$P' = \text{rotate\_z}(\theta) \cdot P = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \end{pmatrix}.$$

W analogiczny sposób można zastosować operacje obrotu wokół osi x oraz y. Dowolną trójwymiarową rotację uzyskuje się poprzez składanie tych elementarnych przekształceń. Należy jednak pamiętać, że postać macierzy złożonej M_R zależy od kolejności ich aplikacji.

Przykład 2.2. Rotowanie sześcianu

Rozważmy sześcian z przykładu 2.1



Zadane są obroty o $\frac{\pi}{4}$ względem osi x oraz o $\frac{\pi}{6}$ względem osi y. Macierze obrotu mają więc postać

$$\text{rotate\_x}\left(\tfrac{\pi}{4}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\frac{\pi}{4}) & -\sin(\frac{\pi}{4}) \\ 0 & \sin(\frac{\pi}{4}) & \cos(\frac{\pi}{4}) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix},$$

$$\text{rotate\_y}(\tfrac{\pi}{6}) = \begin{bmatrix} \cos(\frac{\pi}{6}) & 0 & \sin(\frac{\pi}{6}) \\ 0 & 1 & 0 \\ -\sin(\frac{\pi}{6}) & 0 & \cos(\frac{\pi}{6}) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{bmatrix}.$$

Wynikowa macierz obrotu jest złożeniem obu obrotów

$$\text{rotate\_x}\left(\frac{\pi}{4}\right) \cdot \text{rotate\_y}\left(\frac{\pi}{6}\right) = \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & -\frac{\sqrt{6}}{4} \\ -\frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \end{bmatrix}.$$

Przykład 2.2.

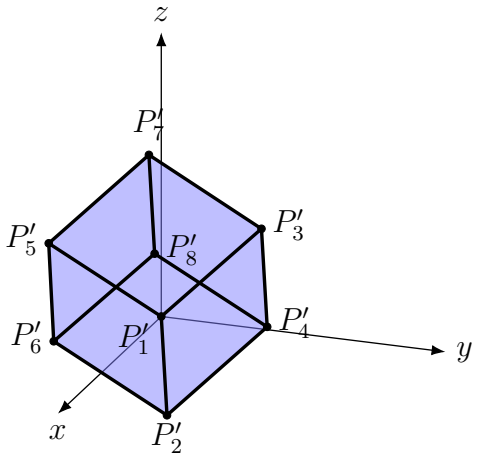
Aby uzyskać odpowiednio obrócone współrzędne wierzchołków, każdy z punktów należy pomnożyć przez tak złożoną macierz obrotów. Na przykład, punkt $P_7 = (0, 2, 2)^T$ będzie miał następujące współrzędne

$$\left[\text{rotate\_x}\left(\frac{\pi}{4}\right) \cdot \text{rotate\_y}\left(\frac{\pi}{6}\right)\right] \cdot P_7 = \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & -\frac{\sqrt{6}}{4} \\ -\frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \end{bmatrix} \begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{2\sqrt{2}-\sqrt{6}}{2} \\ \frac{2\sqrt{2}+\sqrt{6}}{2} \end{pmatrix}.$$

Przekształcone współrzędne wszystkich wierzchołków wynoszą

$$P'_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad P'_2 = \begin{pmatrix} \sqrt{3} \\ \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{pmatrix}, \quad P'_3 = \begin{pmatrix} 0 \\ \sqrt{2} \\ \sqrt{2} \end{pmatrix}, \quad P'_4 = \begin{pmatrix} \sqrt{3} \\ \frac{3\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{pmatrix},$$
$$P'_5 = \begin{pmatrix} 1 \\ -\frac{\sqrt{6}}{2} \\ \frac{\sqrt{6}}{2} \end{pmatrix}, \quad P'_6 = \begin{pmatrix} \sqrt{3}+1 \\ \frac{\sqrt{2}-\sqrt{6}}{2} \\ \frac{-\sqrt{2}+\sqrt{6}}{2} \end{pmatrix}, \quad P'_7 = \begin{pmatrix} 1 \\ \frac{2\sqrt{2}-\sqrt{6}}{2} \\ \frac{2\sqrt{2}+\sqrt{6}}{2} \end{pmatrix}, \quad P'_8 = \begin{pmatrix} \sqrt{3}+1 \\ \frac{3\sqrt{2}-\sqrt{6}}{2} \\ \frac{\sqrt{2}+\sqrt{6}}{2} \end{pmatrix}$$

Sześcian po obrocie wygląda więc następująco



Translacja

Translacja jest operacją przesunięcia obiektu w przestrzeni. W $\mathbb{R}^3$ reprezentowana jest za pomocą wektora, którego każdy element wskazuje przesunięcie względem odpowiedniej osi współrzędnych

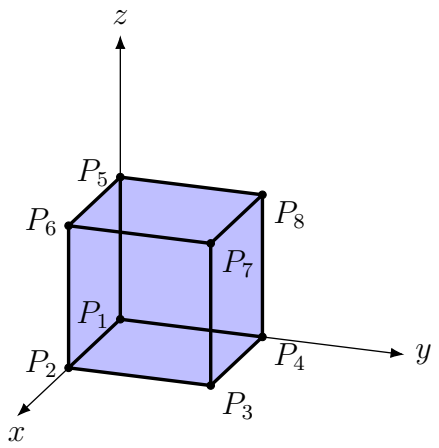
$$\overrightarrow{t} = \text{translate}(t_x, t_y, t_z) = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}. \tag{2.5}$$

Dla dowolnego punktu $P = (x, y, z)^T$, wynik operacji translacji można zapisać jako

$$P' = P + \text{translate}(t_x, t_y, t_z) = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \end{pmatrix}.$$

Przykład 2.3. Przesunięcie sześcianu

Rozważmy ponownie nasz sześcian z przykładu 2.1



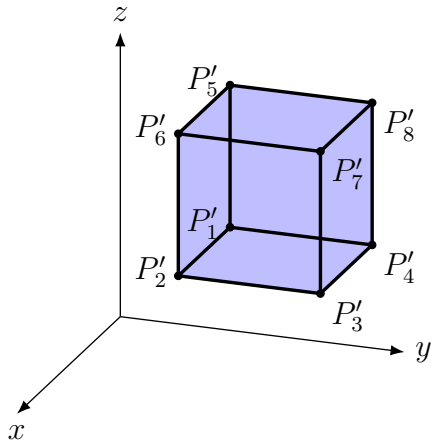
na którym chcemy wykonać zadaną translację translate$(4, 3, 3)$ Aby uzyskać współrzędne wierzchołków sześcianu, każdy z nich przesuwamy poprzez dodanie do jego współrzędnych wektora przesunięcia. Dla przykładu, punkt $P_6 = (2, 0, 2)^T$ po przesunięciu będzie miał współrzędne

$$P_6' = P_6 + \text{translate}(4, 3, 3) = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix} + \begin{pmatrix} 4 \\ 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 6 \\ 3 \\ 5 \end{pmatrix}.$$

Przekształcone współrzędne wszystkich wierzchołków wynoszą

$$P_1' = \begin{pmatrix} 4 \\ 3 \\ 3 \end{pmatrix}, \quad P_2' = \begin{pmatrix} 6 \\ 3 \\ 3 \end{pmatrix}, \quad P_3' = \begin{pmatrix} 4 \\ 5 \\ 3 \end{pmatrix}, \quad P_4' = \begin{pmatrix} 6 \\ 5 \\ 3 \end{pmatrix},$$
$$P_5' = \begin{pmatrix} 4 \\ 3 \\ 5 \end{pmatrix}, \quad P_6' = \begin{pmatrix} 6 \\ 3 \\ 5 \end{pmatrix}, \quad P_7' = \begin{pmatrix} 4 \\ 5 \\ 5 \end{pmatrix}, \quad P_8' = \begin{pmatrix} 6 \\ 5 \\ 5 \end{pmatrix}.$$

Sześcian po przesunięciu wygląda więc następująco



Współrzędne jednorodne

Opisane transformacje pozwalają zmieniać położenie obiektów geometrycznych. Należy jednak zwrócić uwagę na istotne ograniczenie związane z translacją. Skalowanie i rotacja są przekształceniami liniowymi – można je reprezentować za pomocą macierzy 3×3, które działają na współrzędne punktu poprzez mnożenie macierzowe. Translacja natomiast jest przekształceniem afinicznym, polegającym na dodaniu stałego wektora przesunięcia. W przeciwieństwie do przekształceń liniowych, translacja nie spełnia warunku zachowania początku układu współrzędnych, dlatego nie da się jej wyrazić za pomocą standardowej macierzy 3×3. Rozwiązaniem tego problemu jest zastosowanie współrzędnych jednorodnych, które wprowadzają dodatkowy wymiar do przestrzeni poprzez dodanie do współrzędnych elementu o wartości 1, co pozwala na reprezentowanie translacji jako transformacji liniowej [17]. W ten sposób każdy punkt $P = (x, y, z)^T$ przy użyciu współrzędnych jednorodnych zapisywany będzie jako

$$P = \begin{pmatrix} x \\ y \\ z \\ \omega = 1 \end{pmatrix}. \tag{2.6}$$

Operacje skalowania i rotacji będą zapisywane jako macierze 4×4, przez dodanie czwartego wiersza i kolumny, gdzie ostatni element będzie równy 1 i będą mieć postać

$$M_S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

$$M_{Rx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

$$M_{Ry} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

$$M_{Rz} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

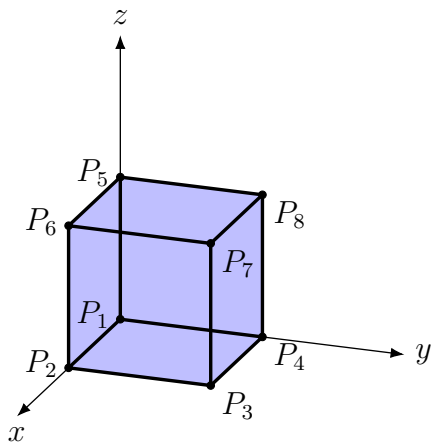
Wektor operacji translacji teraz będzie zapisywany jako macierz 4×4, gdzie ostatnia kolumna będzie przechowywać elementy wskazujące na przesunięcie obiektu

$$\text{translate}(t_x, t_y, t_z) = M_T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Mając tak zdefiniowane macierze transformacji obiektów, chcąc wykonać kilka operacji, możemy je przechować za pomocą jednej macierzy 4×4.

Przykład 2.4.

Rozważmy sześcian z przykładu 2.1



Zadane są następujące transformacje

- skalowanie scale $\left(\frac{3}{2}, 2, \frac{1}{2}\right)$,
- rotacja o kąt $\frac{\pi}{4}$ wokół osi x,
- rotacja o kąt $\frac{\pi}{6}$ wokół osi y,
- translacja o wektor translate$(4, 3, 3)$.

Wykorzystując powyżej zdefiniowane macierze operacji transformacji dla współrzędnych jednorodnych, przedstawmy je za pomocą jednej macierzy. Rozważmy kolejno operacje

$$M = \text{translate}(4,3,3) \cdot \text{rotate\_y}\left(\tfrac{\pi}{6}\right) \cdot \text{rotate\_x}\left(\tfrac{\pi}{4}\right) \cdot \text{scale}\left(\tfrac{3}{2}, 2, \tfrac{1}{2}\right) =$$
$$= \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{3}{2} & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$
$$= \begin{bmatrix} \frac{3\sqrt{3}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{8} & 4 \\ 0 & \sqrt{2} & -\frac{\sqrt{2}}{4} & 3 \\ -\frac{3}{4} & \frac{\sqrt{6}}{2} & \frac{\sqrt{6}}{8} & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Tak złożoną macierz transformacji M możemy wykorzystać by uzyskać nowe współrzędne przekształcanego sześcianu. Przykładowo, współrzędne punktu P_4 po przekształceniu będą równe

$$P_4' = MP_4 = \begin{bmatrix} \frac{3\sqrt{3}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{8} & 4 \\ 0 & \sqrt{2} & -\frac{\sqrt{2}}{4} & 3 \\ -\frac{3}{4} & \frac{\sqrt{6}}{2} & \frac{\sqrt{6}}{8} & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 2 \\ 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \sqrt{2} + \frac{3\sqrt{3}}{2} + 4 \\ 2\sqrt{2} + 3 \\ \frac{3}{2} + \sqrt{6} \\ 1 \end{pmatrix}.$$

Przykład 2.4.

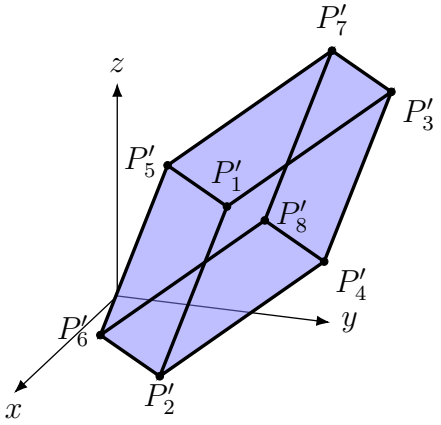
Przekształcone współrzędne wszystkich wierzchołków wynoszą

$$P_1 = \begin{pmatrix} 4 \\ 3 \\ 3 \end{pmatrix}, \quad P_2 = \begin{pmatrix} \frac{3\sqrt{3}}{2} + 4 \\ 3 \\ \frac{3}{2} \end{pmatrix}, \quad P_3 = \begin{pmatrix} \sqrt{2} + 4 \\ 2\sqrt{2} + 3 \\ \sqrt{6} + 3 \end{pmatrix},$$

$$P_4 = \begin{pmatrix} \sqrt{2} + \frac{3\sqrt{3}}{2} + 4 \\ 2\sqrt{2} + 3 \\ \frac{3}{2} + \sqrt{6} \end{pmatrix}, \quad P_5 = \begin{pmatrix} \frac{\sqrt{2}}{4} + 4 \\ 3 - \frac{\sqrt{2}}{2} \\ \frac{\sqrt{6}}{4} + 3 \end{pmatrix}, \quad P_6 = \begin{pmatrix} \frac{\sqrt{2}}{4} + \frac{3\sqrt{3}}{2} + 4 \\ 3 - \frac{\sqrt{2}}{2} \\ \frac{\sqrt{6}}{4} + \frac{3}{2} \end{pmatrix},$$

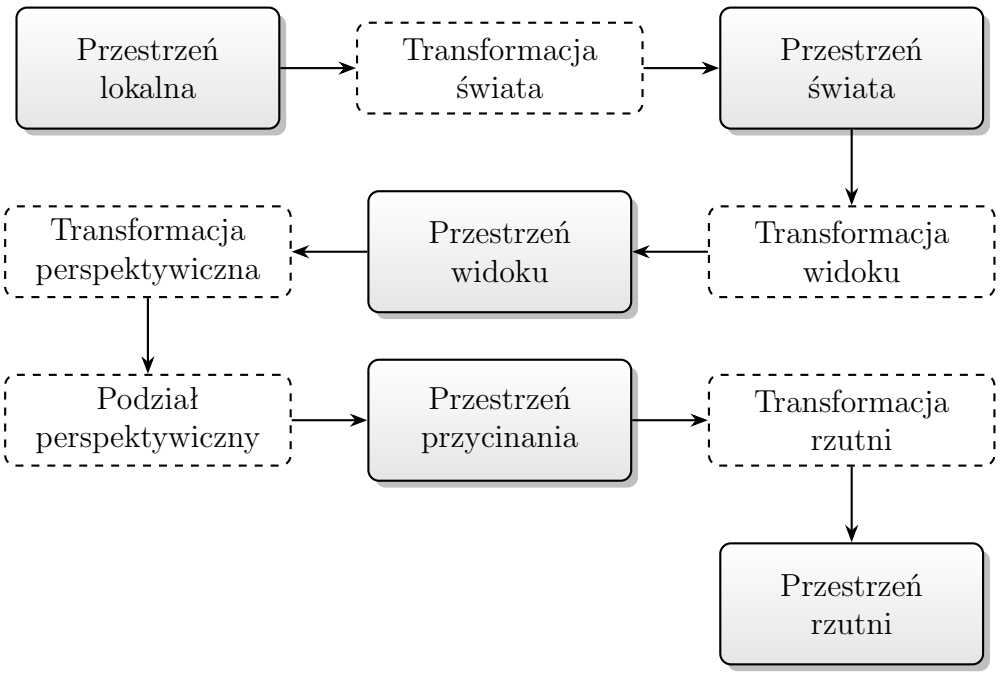
$$P_7 = \begin{pmatrix} \frac{5\sqrt{2}}{4} + 4 \\ \frac{3\sqrt{2}}{2} + 3 \\ 3 + \frac{5\sqrt{6}}{4} \end{pmatrix}, \quad P_8 = \begin{pmatrix} \frac{5\sqrt{2}}{4} + \frac{3\sqrt{3}}{2} + 4 \\ \frac{3\sqrt{2}}{2} + 3 \\ \frac{3}{2} + \frac{5\sqrt{6}}{4} \end{pmatrix}.$$

Sześcian po tych transformacjach będzie wyglądać następująco



3 Układy współrzędnych w potoku graficznym

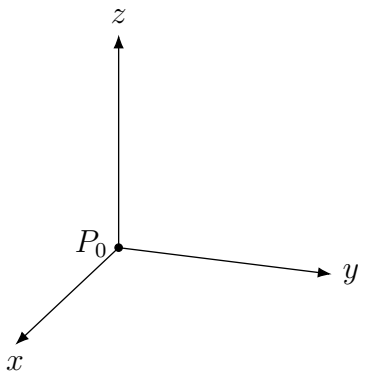
Układy współrzędnych są podstawowym narzędziem wykorzystywanym w potoku graficznym do reprezentacji i transformacji danych przestrzennych. Pozwalają one na precyzyjne określenie położenia, orientacji i relacji między obiektami w przestrzeni trójwymiarowej. W tym rozdziale omówione zostaną najważniejsze układy współrzędnych wykorzystywane w potoku graficznym, takie jak układ świata, układ kamery czy układ projekcji. Szczególną uwagę poświęcimy tworzeniu macierzy transformacji pomiędzy tymi układami, które są kluczowe dla poprawnego odwzorowania geometrii w procesie renderowania.



Rysunek 3.1: Diagram układów współrzędnych i transformacji między nimi. *Opracowanie własne na podstawie* [9].

3.1 Przestrzeń lokalna

Przestrzeń modelu (*Model space*), określana również jako przestrzeń lokalna (*Local space*) lub przestrzeń obiektu (*Object space*), jest podstawowym układem współrzędnych, w którym definiowane są obiekty trójwymiarowe. W tej przestrzeni powstaje obiekt zdefiniowany za pomocą wierzchołków, które zdefiniowane są względem lokalnego punktu odniesienia $P_0 = (0, 0, 0)^T$, co widoczne jest na rysunku poniżej



Rysunek 3.2: Układ osi współrzędnych w przestrzeni lokalnej. *Opracowanie własne.*

Aby utworzyć dowolny obiekt T, w tym miejscu opisujemy wierzchołki za pomocą macierzy oraz relacje pomiędzy nimi. W grafice komputerowej wszystkie ściany nanosimy za pomocą trójkątów, ponieważ definiowane poprzez określenie trzech wierzchołków i relacji między nimi, jednoznacznie definiują płaszczyznę.

Przykład 3.1. Definicja obiektu

Powróćmy do sześcianu z przykładu 2.1. Wierzchołki sześcianu zdefiniować można przy pomocy macierzy, gdzie każda kolumna odpowiada jednemu punktowi

$$T = \begin{matrix} P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 \end{matrix} \begin{bmatrix} 0 & 2 & 2 & 0 & 0 & 2 & 2 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

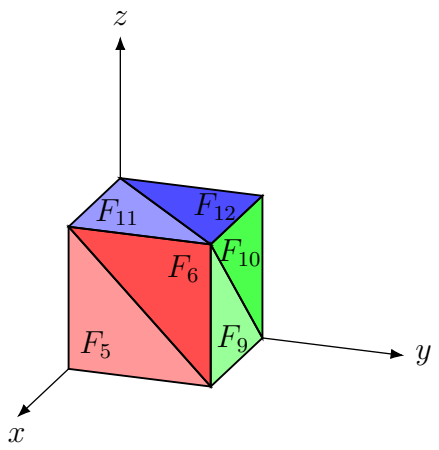
Następnie musimy zdefiniować relacje między nimi, by utworzyć ściany bryły. Zrobimy to za pomocą zagnieżdżonych macierzy, gdzie każda taka macierz składa się z oznaczeń wierzchołków (odpowiadającym tym nad kolumnami macierzy T) i pokazuje w jaki sposób połączone mają być odcinki tworzonych trójkątów

$$\begin{matrix} F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 & F_9 & F_{10} & F_{11} & F_{12} \end{matrix} \begin{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix} & \begin{bmatrix} P_3 \\ P_4 \\ P_1 \end{bmatrix} & \begin{bmatrix} P_1 \\ P_4 \\ P_5 \end{bmatrix} & \begin{bmatrix} P_5 \\ P_4 \\ P_8 \end{bmatrix} & \begin{bmatrix} P_2 \\ P_3 \\ P_6 \end{bmatrix} & \begin{bmatrix} P_6 \\ P_3 \\ P_7 \end{bmatrix} & \begin{bmatrix} P_1 \\ P_2 \\ P_6 \end{bmatrix} & \begin{bmatrix} P_6 \\ P_1 \\ P_5 \end{bmatrix} & \begin{bmatrix} P_3 \\ P_4 \\ P_7 \end{bmatrix} & \begin{bmatrix} P_7 \\ P_4 \\ P_8 \end{bmatrix} & \begin{bmatrix} P_6 \\ P_5 \\ P_7 \end{bmatrix} & \begin{bmatrix} P_7 \\ P_5 \\ P_8 \end{bmatrix} \end{bmatrix}.$$

Dolna ściana | Tylna ściana | Przednia ściana | Lewa ściana | Prawa ściana | Górna ściana

Przykład 3.1.

Tak zdefiniowany sześcian będzie wyglądał następująco

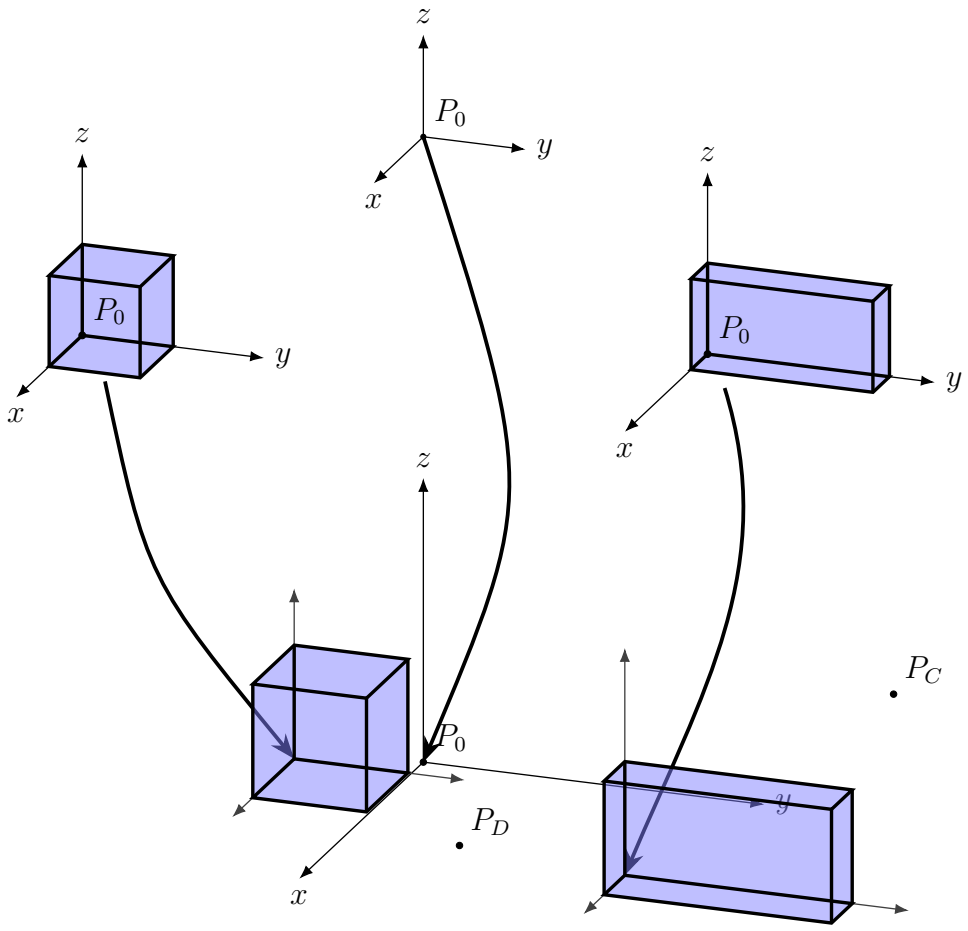


3.2 Przestrzeń świata

Przestrzeń świata (*World space*) jest kolejnym kluczowym etapem potoku graficznego, w którym budujemy globalny układ współrzędnych dla całej trójwymiarowej sceny. W tym układzie każdy obiekt, utworzony wcześniej w przestrzeni lokalnej, umieszczany jest względem ustalonego punktu odniesienia $P_0 = (0, 0, 0)^T$. Proces ten można porównać do budowy drzewa dziedziczenia, w którym każdy węzeł (obiekt) posiada swoją lokalną definicję, ale jego ostateczne położenie i orientacja zależą od położenia rodzica w tym drzewie. Matematycznie, transformację do przestrzeni świata wyraża się jako złożenie operacji lokalnych i dziedziczonych

$$T_{\text{world}} = (\text{translate}(t_x, t_y, t_z) \cdot \text{scale}(s_x, s_y, s_z) \cdot \text{rotate\_x}(\theta_1) \cdot \text{rotate\_y}(\theta_2) \cdot \text{rotate\_z}(\theta_3)) \cdot T.$$

Dzięki temu złożeniu, obiekty zachowują swoją autonomię w lokalnym układzie współrzędnych, ale jednocześnie dziedziczą transformacje od przodków w drzewie sceny. Na przykład, ruch rodzica automatycznie wpływa na wszystkie jego dzieci, zachowując przy tym ich względne położenie. Korzeniem tej hierarchii jest zawsze globalny układ współrzędnych, co zapewnia spójność wszystkich obliczeń przestrzennych. W tej przestrzeni również pojawiają się dwa kolejne obiekty. Pierwszy to obiekt kamery, odzwierciedlający miejsce z którego użytkownik obserwuje scenę, oznaczane jako punkt $P_C = (c_x, c_y, c_z)^T$. Drugi to obiekt celu, przedstawiający miejsce w które skierowana jest soczewka kamery, oznaczany jako punkt $P_D = (d_x, d_z, d_y)^T$. Na rysunku poniżej przedstawiono przykład drzewa sceny, gdzie obiekt rodzica (układ współrzędnych z punktem P_0) stanowi punkt odniesienia dla dwóch obiektów potomnych



Rysunek 3.3: Przykład przeniesienia obiektów do wspólnej przestrzeni świata. *Opracowanie własne.*

Przykład 3.2. Transformacja świata

Rozważmy macierz wierzchołków sześcianu T z przykładu 3.1. Zaprezentujmy ten sześcian w przestrzeni świata, wiedząc że powinien zostać poddany operacjom

- przesunięcia translate$(2, 1, 3)$,
- skalowania scale$(1, \frac{1}{2}, 2)$,

oraz kamerę w punkcie $P_C = (5, 7, 4)^T$ skierowaną w kierunku początku układu współrzędnych $P_D = (0, 0, 0)^T$.
Najpierw, określmy ogólną macierz operacji liniowych M

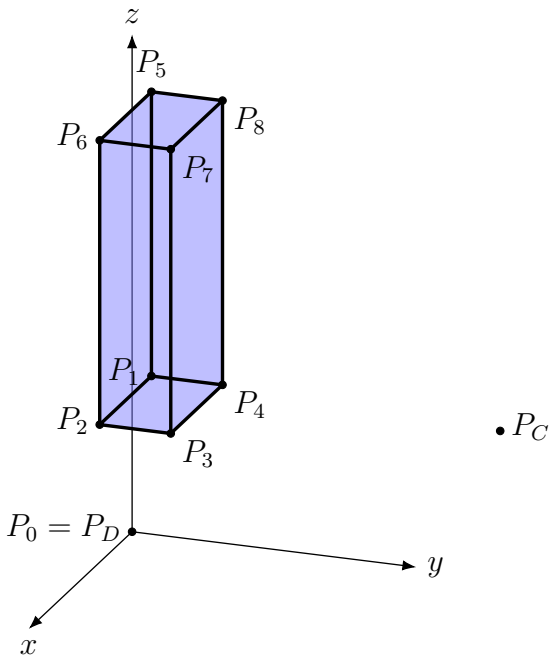
$$M = \text{translate}(2,1,3) \cdot \text{scale}(1, \frac{1}{2}, 2) = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & \frac{1}{2} & 0 & 1 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Przykład 3.2.

Następnie przemnóżmy macierz wierzchołków T przez uzyskaną macierz operacji liniowych M

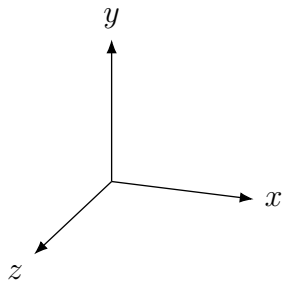
$$T_{\text{world}} = MT = \begin{bmatrix} 2 & 4 & 4 & 2 & 2 & 4 & 4 & 2 \\ 1 & 1 & 2 & 2 & 1 & 1 & 2 & 2 \\ 3 & 3 & 3 & 3 & 7 & 7 & 7 & 7 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Zatem przekształcony sześcian oraz kamera w przestrzeni świata będą wyglądać następująco



3.3 Przestrzeń widoku

Przestrzeń widoku (*View space*), określana również jako przestrzeń kamery (*Camera space*) lub przestrzeń oka (*Eye space*), jest układem współrzędnych w którym punktem odniesienia sceny jest obiekt kamery P_C skierowanej wzdłuż negatywnej osi współrzędnych z. Ponieważ OpenGL używa prawoskrętnego układu współrzędnych, by poprawić czytelność następnych wykresów, zmienimy ułożenie osi współrzędnych, co widać poniżej



Rysunek 3.4: Prawoskrętny układ współrzędnych. *Opracowanie własne.*

Przejście do przestrzeni widoku wymaga ustalenia postaci macierzy modelu kamery. Intuicja podpowiada, że kluczowe będą operacje translacji i rotacji. Aby uzyskać macierz operacji translacji, musimy wykorzystać fakt z rozdziału pierwszego - każdy punkt możemy opisać jako wektor, z punktem zaczepienia w punkcie odniesienia układu, zatem postać operacji przesunięcia składa się z współrzędnych opisujących pozycję kamery. Dodatkowo musimy zwrócić uwagę na fakt, że potrzebna jest nam macierz operacji przesunięcia z zanegowanymi współczynnikami przesunięcia, ponieważ chcemy punkt P_C przesunąć z swojej obecnej pozycji do punktu odniesienia układu. [17]

$$M_T = \text{translate}(-c_x, -c_y, -c_z) = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Następnie aby określić wielkości obrotów względem osi współrzędnych musimy utworzyć bazę, składającą się z trzech znormalizowanych wektorów, wszystkich zaczepionych w punkcie P_C. Pierwszy z nich, to wektor „naprzód" $\overrightarrow{f} = (f_x, f_y, f_z)^T$, który reprezentuje kierunek w jakim kamera jest skierowana i wyznaczamy za pomocą dwóch punktów P_C i P_D

$$\overrightarrow{f} = -\frac{P_C - P_D}{\|P_C - P_D\|}. \tag{3.1}$$

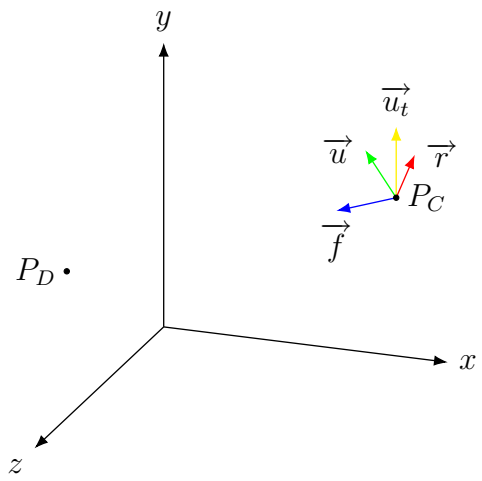
Kolejny to wektor $\overrightarrow{u} = (u_x, u_y, u_z)^T$ wskazuje „górę". Musi on znajdować się w płaszczyźnie dzielącej widok kamery na dwie równe części, do której należy już wektor $\overrightarrow{f}$ i jeśli nie jest określony, domyślnie przyjmujemy jego wartość za $(0, 1, 0)^T$ oraz oznaczymy u_t. Mając te dwa wektory, używając iloczynu wektorowego możemy obliczyć ostatni z nich, czyli wektor $\overrightarrow{r} = (r_x, r_y, r_z)^T$ wskazujący "w prawo", który jest wektorem normalnym płaszczyzny powstałej z wektorów $\overrightarrow{f}$ i $\overrightarrow{u_t}$ [2][18]

$$\overrightarrow{r} = \frac{\overrightarrow{u_t} \times \overrightarrow{f}}{\|\overrightarrow{u_t} \times \overrightarrow{f}\|}. \tag{3.2}$$

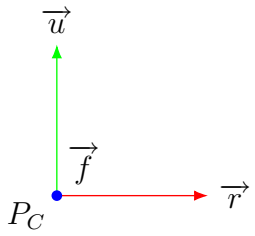
Na koniec, jeśli dotychczasowy wektor wskazujący górę był wektorem domyślnym $\overrightarrow{u_t} = (0, 1, 0)^T$, musimy ustalić wartości prawdziwego wektora $\overrightarrow{u}$. Ponieważ mamy już dwa ortogonalne wektory $\overrightarrow{f}$ i $\overrightarrow{r}$, użyjemy znów iloczynu wektorowego

$$\overrightarrow{u} = \overrightarrow{f} \times \overrightarrow{r}. \tag{3.3}$$

Poniżej przedstawiono dwa rysunki z widocznym obiektem kamery P_C oraz układem trzech wektorów $\overrightarrow{f}$, $\overrightarrow{u}$, $\overrightarrow{r}$



Rysunek 3.5: Widok układu trzech wektorów $\overrightarrow{f}, \overrightarrow{u}, \overrightarrow{r}$ w punkcie P_C w przestrzeni świata. *Opracowanie własne.*



Rysunek 3.6: Widok układu trzech wektorów $\overrightarrow{f}, \overrightarrow{u}, \overrightarrow{r}$ w punkcie P_C, odzwierciedlający widok z „soczewki" kamery.

Następnie, musimy utworzyć macierz operacji obrotu M_R korzystając z współczynników obliczonych wektorów [18]

$$M_R = \begin{bmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

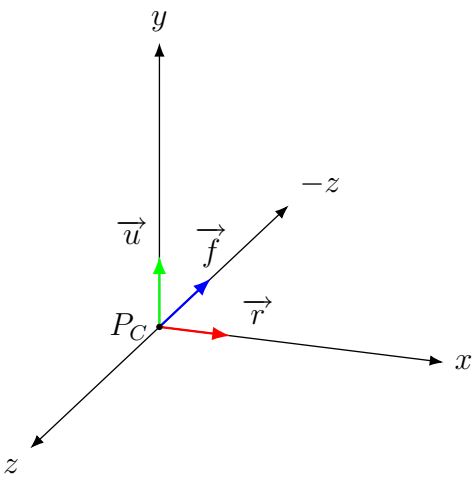
Macierz modelu kamery wygląda zatem następująco

$$P'_C = M_R M_T = \begin{bmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r_x & u_x & f_x & -(r_x c_x + u_x c_y + f_x c_z) \\ r_y & u_y & f_y & -(r_y c_x + u_y c_y + f_y c_z) \\ r_z & u_z & f_z & -(r_z c_x + u_z c_y + f_z c_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Wykorzystując tę macierz modelu do transformacji wszystkich obiektów w naszej scenie, znajdziemy się w przestrzeni widoku

$$T_{\text{cam}} = P'_C \cdot T_{\text{world}}. \tag{3.4}$$

Po przesunięciu obiektu kamery do początku układu współrzędnych, układ powinien wyglądać jak na poniższym rysunku



Rysunek 3.7: Przykład przestrzeni widoku z obiektem kamery jako punktem odniesienia. *Opracowanie własne.*

Przykład 3.3. Transformacja widoku

Kontynuując przykład 3.2 przekształćmy przestrzeń świata do przestrzeni widoku. Przypomnijmy informacje jakie aktualnie mamy

- kamera w punkcie $P_C = (5, 7, 4)^T$,
- punkt w który skierowana jest kamera $P_D = P_0 = (0, 0, 0)^T$,
- macierz przekształconego obiektu $T_{\text{world}} = \begin{bmatrix} 2 & 4 & 4 & 2 & 2 & 4 & 4 & 2 \\ 1 & 1 & 2 & 2 & 1 & 1 & 2 & 2 \\ 3 & 3 & 3 & 3 & 7 & 7 & 7 & 7 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$

Przygotujmy macierz modelu kamery, zaczynając od ustalenia macierzy transformacji kamery

$$M_T = \text{translate}(-5, -7, -4) = \begin{bmatrix} 1 & 0 & 0 & -5 \\ 0 & 1 & 0 & -7 \\ 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Teraz zdefiniujmy wektory $\overrightarrow{f}, \overrightarrow{u}, \overrightarrow{r}$, przy czym nie mamy podanego wektora $\overrightarrow{u}$, więc będziemy musieli użyć wektora tymczasowego $\overrightarrow{u_t} = (0, 1, 0)^T$

$$P_C - P_D = \begin{pmatrix} 5 \\ 7 \\ 4 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 4 \end{pmatrix},$$

Przykład 3.3.

$$\overrightarrow{f} = -\frac{\begin{pmatrix}5\\7\\4\end{pmatrix}}{\left\|\begin{pmatrix}5\\7\\4\end{pmatrix}\right\|} = -\begin{pmatrix}\frac{\sqrt{10}}{6}\\ \frac{7\sqrt{10}}{30}\\ \frac{2\sqrt{10}}{15}\end{pmatrix},$$

$$\overrightarrow{u_t} \times \overrightarrow{f} = \begin{pmatrix}0\\1\\0\end{pmatrix} \times \begin{pmatrix}-\frac{\sqrt{10}}{6}\\ -\frac{7\sqrt{10}}{30}\\ -\frac{2\sqrt{10}}{15}\end{pmatrix} = \begin{pmatrix}-\frac{2\sqrt{10}}{15}\\ 0\\ \frac{\sqrt{10}}{6}\end{pmatrix},$$

$$\overrightarrow{r} = \frac{\begin{pmatrix}-\frac{2\sqrt{10}}{15}\\ 0\\ \frac{\sqrt{10}}{6}\end{pmatrix}}{\left\|\begin{pmatrix}-\frac{2\sqrt{10}}{15}\\ 0\\ \frac{\sqrt{10}}{6}\end{pmatrix}\right\|} = \begin{pmatrix}-\frac{4\sqrt{10}}{41}\\ 0\\ \frac{5\sqrt{10}}{41}\end{pmatrix},$$

$$\overrightarrow{u} = -\begin{pmatrix}\frac{\sqrt{10}}{6}\\ \frac{7\sqrt{10}}{30}\\ \frac{2\sqrt{10}}{15}\end{pmatrix} \times \begin{pmatrix}-\frac{4\sqrt{10}}{41}\\ 0\\ \frac{5\sqrt{10}}{41}\end{pmatrix} = \begin{pmatrix}\frac{35}{123}\\ -\frac{3}{41}\\ \frac{28}{123}\end{pmatrix}.$$

Macierz M_R ma więc postać

$$M_R = \begin{bmatrix}-\frac{4\sqrt{10}}{41} & \frac{35}{123} & -\frac{\sqrt{10}}{6} & 0\\ 0 & -\frac{3}{41} & -\frac{7\sqrt{10}}{30} & 0\\ \frac{5\sqrt{10}}{41} & \frac{28}{123} & -\frac{2\sqrt{10}}{15} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

Macierz modelu kamery wygląda więc następująco

$$P'_C = \begin{bmatrix}1 & 0 & 0 & -5\\ 0 & 1 & 0 & -7\\ 0 & 0 & 1 & -4\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}-\frac{4\sqrt{10}}{41} & \frac{35}{123} & -\frac{\sqrt{10}}{6} & 0\\ 0 & -\frac{3}{41} & -\frac{7\sqrt{10}}{30} & 0\\ \frac{5\sqrt{10}}{41} & \frac{28}{123} & -\frac{2\sqrt{10}}{15} & 0\\ 0 & 0 & 0 & 1\end{bmatrix} =$$

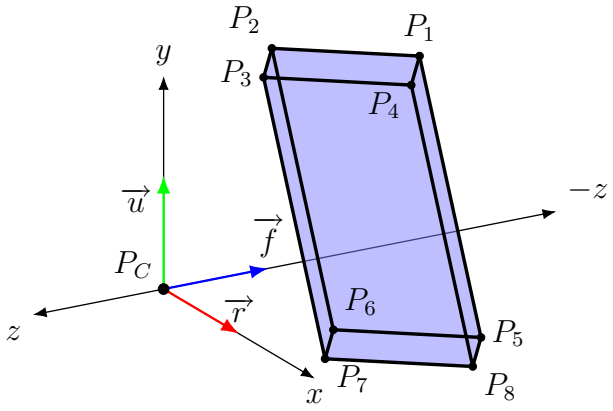
$$= \begin{bmatrix}-\frac{4\sqrt{10}}{41} & \frac{35}{123} & -\frac{\sqrt{10}}{6} & \frac{20\sqrt{10}}{41} - \frac{245}{123} + \frac{2\sqrt{10}}{3}\\ 0 & -\frac{3}{41} & -\frac{7\sqrt{10}}{30} & \frac{21}{41} + \frac{28\sqrt{10}}{30}\\ \frac{5\sqrt{10}}{41} & \frac{28}{123} & -\frac{2\sqrt{10}}{15} & -\frac{25\sqrt{10}}{41} - \frac{196}{123} + \frac{8\sqrt{10}}{15}\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

Przykład 3.3.

Zaktualizujmy więc macierz wierzchołków obiektu T do współrzędnych przestrzeni widoku

$$T_{\text{cam}} = P'_C T_{\text{world}} = \begin{bmatrix} -\frac{70}{41} + \frac{19\sqrt{10}}{41} & -\frac{70}{41} + \frac{11\sqrt{10}}{41} & -\frac{58}{41} + \frac{11\sqrt{10}}{41} & -\frac{58}{41} + \frac{19\sqrt{10}}{41} \\ \frac{18}{41} + \frac{7\sqrt{10}}{30} & \frac{18}{41} + \frac{7\sqrt{10}}{30} & \frac{15}{41} + \frac{7\sqrt{10}}{30} & \frac{15}{41} + \frac{7\sqrt{10}}{30} \\ -\frac{56}{41} - \frac{19\sqrt{10}}{82} & -\frac{56}{41} + \frac{2\sqrt{10}}{41} & -\frac{46}{41} + \frac{2\sqrt{10}}{41} & -\frac{46}{41} - \frac{19\sqrt{10}}{82} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -\frac{70}{41} - \frac{17\sqrt{10}}{41} & -\frac{70}{41} - \frac{33\sqrt{10}}{41} & -\frac{58}{41} - \frac{33\sqrt{10}}{41} & -\frac{58}{41} - \frac{17\sqrt{10}}{41} \\ \frac{18}{41} - \frac{7\sqrt{10}}{10} & \frac{18}{41} - \frac{7\sqrt{10}}{10} & \frac{15}{41} - \frac{7\sqrt{10}}{10} & \frac{15}{41} - \frac{7\sqrt{10}}{10} \\ -\frac{56}{41} - \frac{31\sqrt{10}}{41} & -\frac{46}{41} - \frac{31\sqrt{10}}{41} & -\frac{46}{41} - \frac{25\sqrt{10}}{41} & -\frac{56}{41} - \frac{25\sqrt{10}}{41} \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

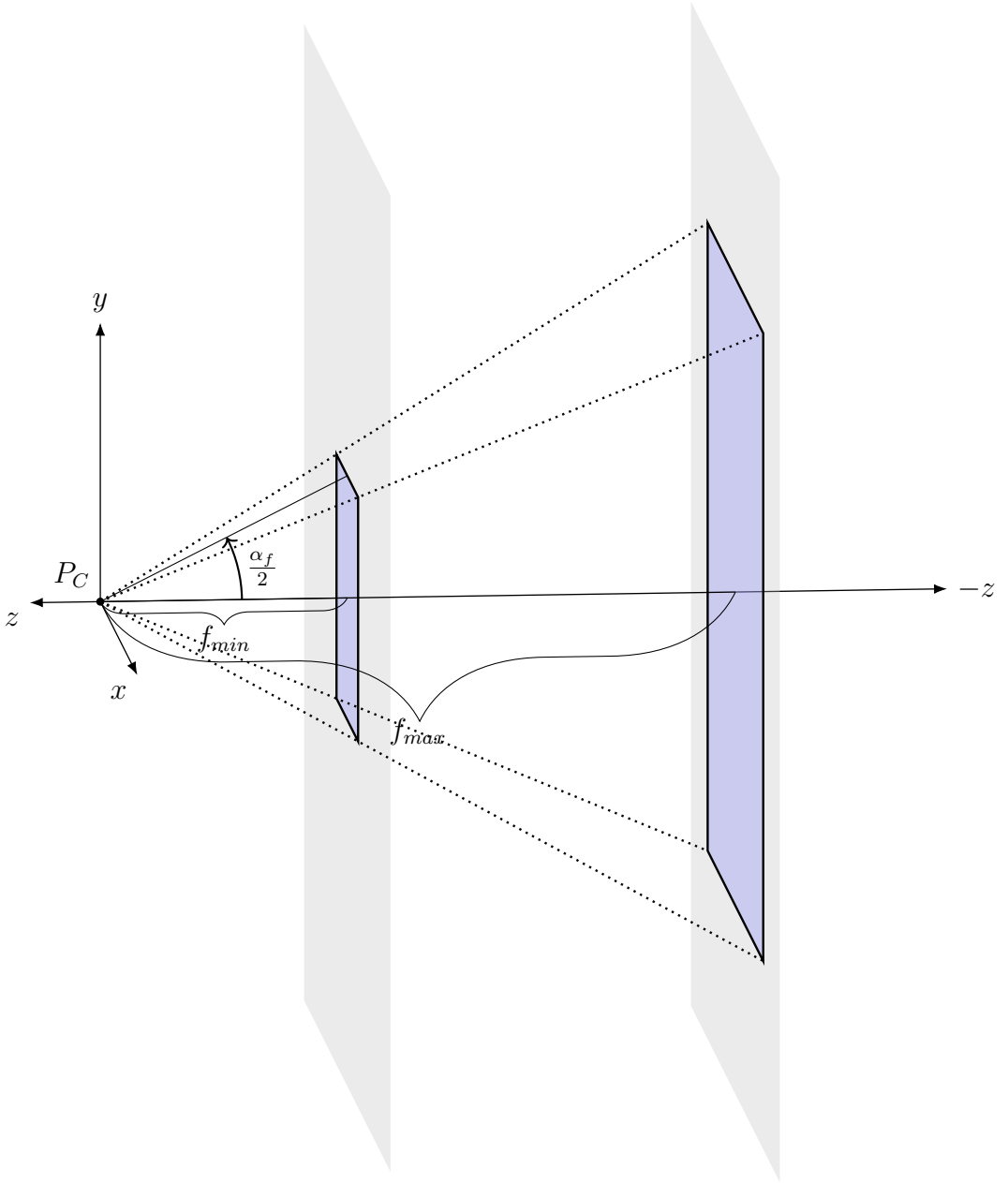


3.4 Przestrzeń przycinania

Przestrzeń przycinania (Clip space) jest miejscem w którym wszystkie obiekty są oceniane pod kątem ich widoczności na ekranie. Dzieje się to przy użyciu bryły widoku, która w zależności od użytego rodzaju projekcji, będzie przyjmować różne kształty. W naszej pracy będziemy używać projekcji perspektywicznej, dlatego będzie to frustrum ostrosłupa czworokątnego, gdzie frustrum wskazuje część bryły, znajdującą się pomiędzy dwoma równoległymi płaszczyznami przecinającymi bryłę oraz zakładamy, że frustrum jest symetryczne. Obszar ten wyznaczamy przy użyciu czterech cech obiektu kamery

- α_f - pionowe pole widzenia, określane w stopniach,
- p - proporcja szerokości i wysokości ekranu na którym będzie wyświetlany obraz,
- f_{min} - odległość kamery na osi $-z$ do najbliższej płaszczyzny ograniczającej ostrosłup,
- f_{max} - odległość kamery na osi $-z$ do najodleglejszej płaszczyzny ograniczającej ostrosłup.

Na poniższym rysunku przedstawiono przykład frustrum perspektywicznego zdefiniowanego przez parametry kamery: kąt widzenia α_f, proporcje ekranu p oraz płaszczyzny przycinania $f_{\min}$ i $f_{\max}$.



Rysunek 3.8: Przykład frustrum ostrosłupa czworokątnego. *Opracowanie własne.*

Jednak żeby wyświetlić obraz na ekranie, musimy odnaleźć odwzorowanie $P_p = (x_p, y_p, z_p)^T$ wierzchołków obiektów, czyli punktów z przestrzeni kamery które oznaczymy jako $P_e = (x_e, y_e, z_e)^T$, na najbliższą kamerze płaszczyznę o wzorze

$$z = -f_{\min}.$$

Aby to zrobić musimy poprowadzić prostą przez środek rzutu, czyli obiekt kamery P_C oraz rzutowany punkt $P_e = (x_e, y_e, z_e)^T$, a następnie obliczyć współrzędne punktu $P_p = (x_p, y_p, z_p)$

$$L : P_p = P_C + P_e \cdot t, \tag{3.5}$$

$$\begin{cases} x_p = x_e \cdot t, \\ y_p = y_e \cdot t, \\ z_p = z_e \cdot t. \end{cases}$$

Wiedząc, że rzutnia jest określona jako $z = -f_{\min}$, możemy podstawić tą wartość do z_p i obliczyć t

$$t = \frac{z_p}{z_e} = \frac{-f_{\min}}{z_e},$$

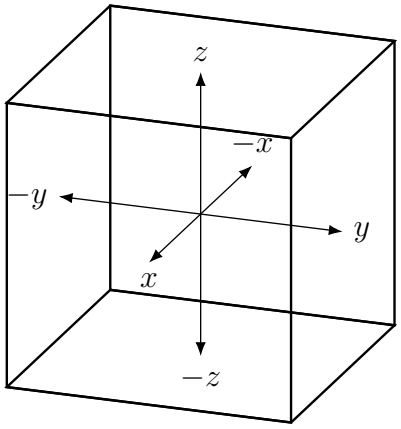
w ten sposób pierwsze dwa rzędy macierzy projekcji M_P zapiszemy jako

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ \omega_p \end{pmatrix} = \begin{bmatrix} f_{\min} & 0 & 0 & 0 \\ 0 & f_{\min} & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{pmatrix} -\frac{x_e}{z_e} \\ -\frac{y_e}{z_e} \\ z_e \\ 1 \end{pmatrix}. \tag{3.6}$$

Mimo że znamy już współrzędne punktu na płaszczyźnie, przed oceną jego widoczności konieczne jest dokonanie pewnych modyfikacji. Aby wyeliminować wierzchołki, które nie będą wyświetlane, musimy przejść do układu znormalizowanych współrzędnych urządzenia (NDC). Ten układ przekształca frustrum w sześcian o boku 2, obejmujący wszystkie punkty spełniające warunek

$$-1 \leqslant x_n, y_n, z_n \leqslant 1.$$

Na poniższym rysunku przedstawiono frustrum przekształcone do układu NDC – regularnego sześcianu o zakresie współrzędnych od -1 do 1 wzdłuż każdej osi



Rysunek 3.9: Przykład frustrum w znormalizowanych współrzędnych urządzenia. *Opracowanie własne.*

W nowo uzyskanej przestrzeni płaszczyzna $z = -1$ odpowiada ścianie frustrum położonej najbliżej kamery, natomiast płaszczyzna $z = 1$ ścianie znajdującej się najdalej. Samo przekształcenie odbywa się w trakcie podziału perspektywicznego (perspective divide), który polega na podzieleniu wszystkich współrzędnych przez współrzędną jednorodną. Dodatkowo zauważmy, że zarówno x_p oraz y_p są odwrotnie proporcjonalne do $-z_e$, zatem wiedząc o podziale perspektywicznym, operację uzyskania współrzędnych punktu P_p możemy zapisać jako [2]

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ \omega_p \end{pmatrix} = \begin{bmatrix} f_{\min} & 0 & 0 & 0 \\ 0 & f_{\min} & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix}. \tag{3.7}$$

Wyobraźmy sobie sytuację, w której dwa punkty w przestrzeni widoku znajdą się na tej samej prostej poprowadzonej ze środka rzutu. Wtedy oba te punkty znajdą się w tym samym miejscu na rzutni. Aby więc silnik graficzny mógł ocenić, który punkt powinien zostać wyświetlony, musimy w punkcie P_p zawrzeć informację o odległości danego punktu P_e od kamery na osi $-z$. Ponieważ wiemy, że wartość z_p nie jest powiązana z x_e i y_e, musimy wykorzystać z_e oraz ω_p

$$z_p = \begin{bmatrix} 0 & 0 & A & B \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ \omega_e \end{pmatrix} = Az_e + B\omega_e = Az_e + B.$$

Następnie wykonajmy podział perspektywiczny

$$\frac{z_p}{\omega_p} = \frac{Az_e + B}{-z_e}.$$

Ponieważ płaszczyzny frustrum wyznaczone odległościami $f_{\min}$ i $f_{\max}$ podczas przejścia do układu znormalizowanych współrzędnych urządzenia zostają zrzutowane do wartości

$$\begin{aligned} z_e = -f_{\min} \to -1, \\ z_e = -f_{max} \to 1, \end{aligned}$$

możemy ułożyć układ równań i wyznaczyć wartości A i B

$$\begin{cases} \frac{-Af_{\min}+B}{f_{\min}} = -1, \\ \frac{-Af_{\max}+B}{f_{\max}} = 1, \end{cases}$$

$$\begin{aligned} A = \frac{-f_{\min}-f_{\max}}{f_{\max}-f_{\min}}, \\ B = \frac{-2f_{\max}f_{\min}}{f_{\max}-f_{\min}} \end{aligned}$$

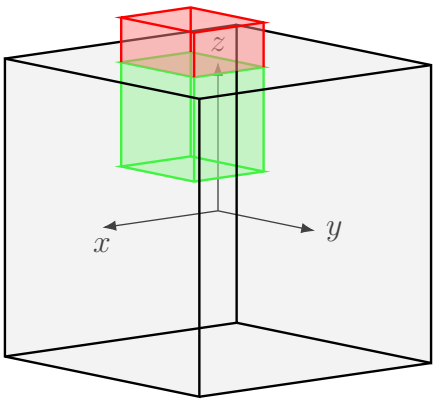
Finalnie macierz projekcji perspektywicznej będzie miała postać

$$\begin{bmatrix} f_{\min} & 0 & 0 & 0 \\ 0 & f_{\min} & 0 & 0 \\ 0 & 0 & \frac{-f_{\min}-f_{\max}}{f_{\max}-f_{\min}} & \frac{-2f_{\max}f_{\min}}{f_{\max}-f_{\min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}. \tag{3.8}$$

Na sam koniec, po przemnożeniu współrzędnych punktu przez macierz projekcji perspektywicznej, wykonujemy wspominany już podział perspektywiczny

$$x_n = \frac{x_p}{\omega_p}, \quad y_n = \frac{y_p}{\omega_p}, \quad z_n = \frac{z_p}{\omega_p}.$$

Dzięki tej operacji otrzymujemy ostateczne współrzędne w układzie znormalizowanych współrzędnych urządzenia. Jeśli którakolwiek z otrzymanych współrzędnych $(x_n, y_n, z_n)^T$ wychodzi poza przedział $[-1, 1]$, dany punkt lub fragment obiektu zostaje odcięty (*clipped*) jako niewidoczny z perspektywy kamery. W ten sposób zachowujemy wyłącznie te elementy sceny, które rzeczywiście mieszczą się w wyświetlanym obszarze oraz podlegają dalszemu etapowi renderowania. Na poniższym rysunku przedstawiono odcięcie części obiektu, która wychodzi poza sześcian NDC



Rysunek 3.10: Przykład przycinania obiektu. *Opracowanie własne.*

Przykład 3.4. Transformacja przycinania

Załóżmy, że w naszym układzie projekcji perspektywicznej przyjmujemy

$$f_{\min} = 2, \quad f_{\max} = 12.$$

Wówczas macierz M_P ma postać

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & -\frac{14}{10} & -\frac{48}{10} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

Mając macierz wierzchołków obiektu w przestrzeni widoku T_{cam}, definiujemy współrzędne przycinania przez

$$T_{\text{clip}} = M_P T_{\text{cam}} = \begin{bmatrix} -\frac{140}{41} + \frac{38\sqrt{10}}{41} & -\frac{140}{41} + \frac{22\sqrt{10}}{41} & -\frac{116}{41} + \frac{22\sqrt{10}}{41} & -\frac{116}{41} + \frac{38\sqrt{10}}{41} \\ \frac{36}{41} + \frac{7\sqrt{10}}{15} & \frac{30}{41} + \frac{7\sqrt{10}}{15} & \frac{30}{41} + \frac{7\sqrt{10}}{15} & \frac{36}{41} + \frac{7\sqrt{10}}{15} \\ -\frac{592}{205} - \frac{14\sqrt{10}}{205} & -\frac{662}{205} - \frac{14\sqrt{10}}{205} & -\frac{662}{205} + \frac{133\sqrt{10}}{410} & -\frac{592}{205} + \frac{133\sqrt{10}}{410} \\ \frac{56}{41} - \frac{2\sqrt{10}}{41} & \frac{46}{41} - \frac{2\sqrt{10}}{41} & \frac{19\sqrt{10}}{82} + \frac{46}{41} & \frac{19\sqrt{10}}{82} + \frac{56}{41} \end{bmatrix}$$

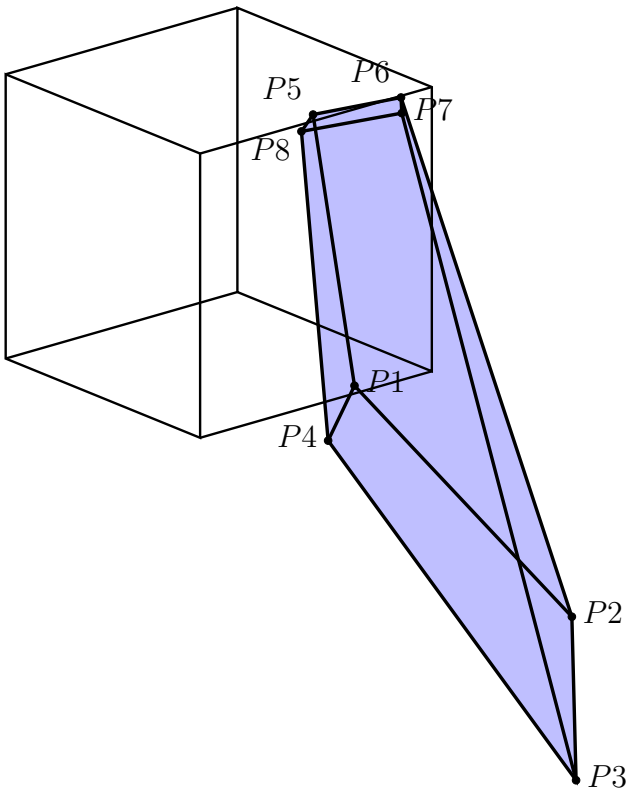
$$\begin{bmatrix} -\frac{140}{41} - \frac{34\sqrt{10}}{41} & -\frac{66\sqrt{10}}{41} - \frac{140}{41} & -\frac{66\sqrt{10}}{41} - \frac{116}{41} & -\frac{116}{41} - \frac{34\sqrt{10}}{41} \\ \frac{36}{41} - \frac{7\sqrt{10}}{5} & \frac{36}{41} - \frac{7\sqrt{10}}{5} & \frac{30}{41} - \frac{7\sqrt{10}}{5} & \frac{30}{41} - \frac{7\sqrt{10}}{5} \\ -\frac{592}{205} + \frac{217\sqrt{10}}{205} & -\frac{662}{205} + \frac{217\sqrt{10}}{205} & -\frac{662}{205} + \frac{35\sqrt{10}}{41} & -\frac{592}{205} + \frac{35\sqrt{10}}{41} \\ \frac{56}{41} + \frac{31\sqrt{10}}{41} & \frac{46}{41} + \frac{31\sqrt{10}}{41} & \frac{46}{41} + \frac{25\sqrt{10}}{41} & \frac{56}{41} + \frac{25\sqrt{10}}{41} \end{bmatrix}.$$

Przykład 3.4.

Następnie musimy wykonać podział perspektywiczny

$$T_{\text{NDC}} = \frac{1}{\omega} T_{\text{clip}} = \begin{bmatrix} \frac{-\frac{140}{41}+\frac{38\sqrt{10}}{41}}{\frac{56}{41}-\frac{2\sqrt{10}}{41}} & \frac{-\frac{140}{41}+\frac{22\sqrt{10}}{41}}{\frac{46}{41}-\frac{2\sqrt{10}}{41}} & \frac{-\frac{116}{41}+\frac{22\sqrt{10}}{41}}{\frac{19\sqrt{10}}{82}+\frac{46}{41}} & \frac{-\frac{116}{41}+\frac{38\sqrt{10}}{41}}{\frac{19\sqrt{10}}{82}+\frac{56}{41}} \\ \frac{\frac{36}{41}+\frac{7\sqrt{10}}{15}}{\frac{56}{41}-\frac{2\sqrt{10}}{41}} & \frac{\frac{30}{41}+\frac{7\sqrt{10}}{15}}{\frac{46}{41}-\frac{2\sqrt{10}}{41}} & \frac{\frac{30}{41}+\frac{7\sqrt{10}}{15}}{\frac{19\sqrt{10}}{82}+\frac{46}{41}} & \frac{\frac{36}{41}+\frac{7\sqrt{10}}{15}}{\frac{19\sqrt{10}}{82}+\frac{56}{41}} \\ \frac{-\frac{592}{205}-\frac{14\sqrt{10}}{205}}{\frac{56}{41}-\frac{2\sqrt{10}}{41}} & \frac{-\frac{662}{205}-\frac{14\sqrt{10}}{205}}{\frac{46}{41}-\frac{2\sqrt{10}}{41}} & \frac{-\frac{662}{205}+\frac{133\sqrt{10}}{410}}{\frac{19\sqrt{10}}{82}+\frac{46}{41}} & \frac{-\frac{592}{205}+\frac{133\sqrt{10}}{410}}{\frac{19\sqrt{10}}{82}+\frac{56}{41}} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

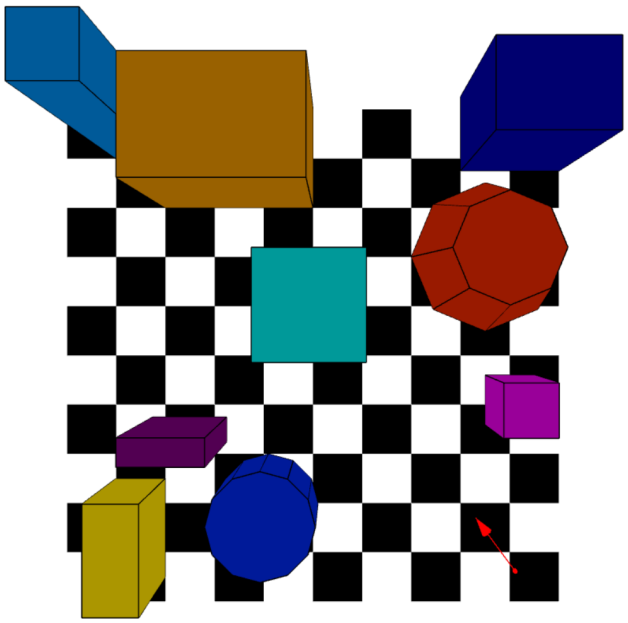
$$\begin{bmatrix} \frac{-\frac{140}{41}-\frac{34\sqrt{10}}{41}}{\frac{56}{41}+\frac{31\sqrt{10}}{41}} & \frac{-\frac{66\sqrt{10}}{41}-\frac{140}{41}}{\frac{46}{41}+\frac{31\sqrt{10}}{41}} & \frac{-\frac{66\sqrt{10}}{41}-\frac{116}{41}}{\frac{46}{41}+\frac{25\sqrt{10}}{41}} & \frac{-\frac{116}{41}-\frac{34\sqrt{10}}{41}}{\frac{56}{41}+\frac{25\sqrt{10}}{41}} \\ \frac{\frac{36}{41}-\frac{7\sqrt{10}}{5}}{\frac{56}{41}+\frac{31\sqrt{10}}{41}} & \frac{\frac{36}{41}-\frac{7\sqrt{10}}{5}}{\frac{46}{41}+\frac{31\sqrt{10}}{41}} & \frac{\frac{30}{41}-\frac{7\sqrt{10}}{5}}{\frac{46}{41}+\frac{25\sqrt{10}}{41}} & \frac{\frac{30}{41}-\frac{7\sqrt{10}}{5}}{\frac{56}{41}+\frac{25\sqrt{10}}{41}} \\ \frac{-\frac{592}{205}+\frac{217\sqrt{10}}{205}}{\frac{56}{41}+\frac{31\sqrt{10}}{41}} & \frac{-\frac{662}{205}+\frac{217\sqrt{10}}{205}}{\frac{46}{41}+\frac{31\sqrt{10}}{41}} & \frac{-\frac{662}{205}+\frac{35\sqrt{10}}{41}}{\frac{46}{41}+\frac{25\sqrt{10}}{41}} & \frac{-\frac{592}{205}+\frac{35\sqrt{10}}{41}}{\frac{56}{41}+\frac{25\sqrt{10}}{41}} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$



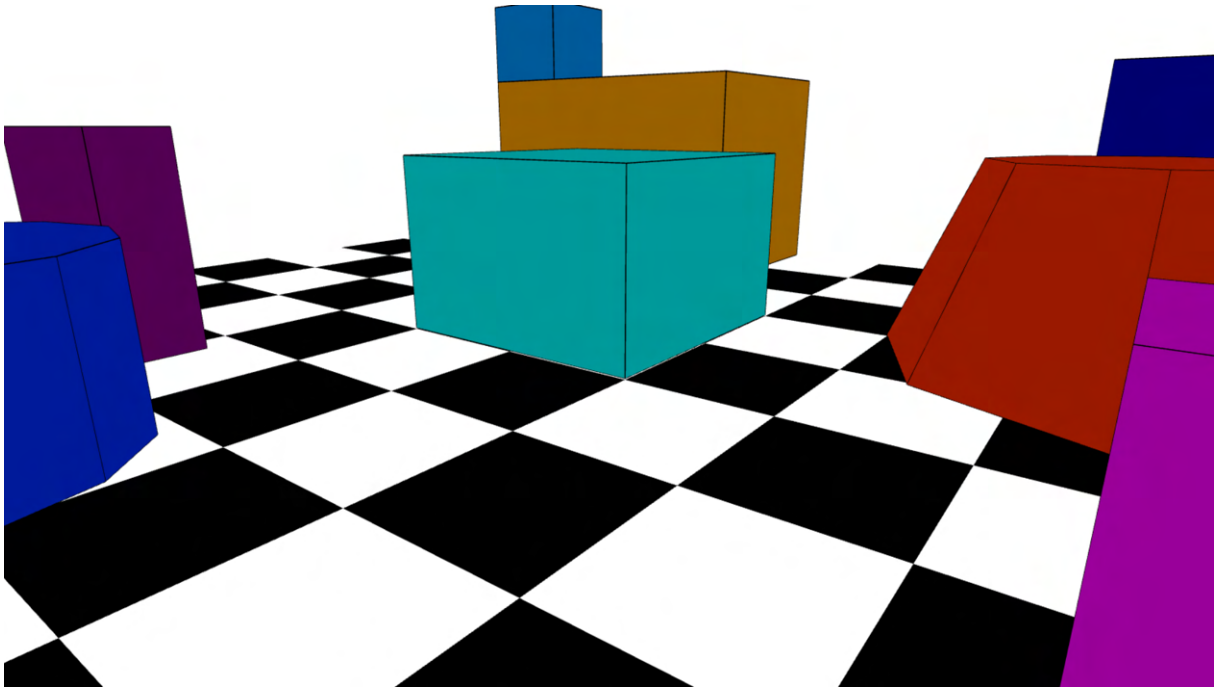
3.4.1 Wpływ parametrów kamery na wyświetlanie sceny

Zmiana parametrów kamery perspektywicznej ma istotny wpływ na finalny wygląd renderowanej sceny. W zależności od przyjętych wartości, obiekty mogą wydawać się bliższe lub dalsze, a cały kadr może być bardziej „zwarty” lub „rozszerzony”. Ustawienie niewłaściwej granicy obcinania może z kolei powodować niepożądane ucinanie elementów na

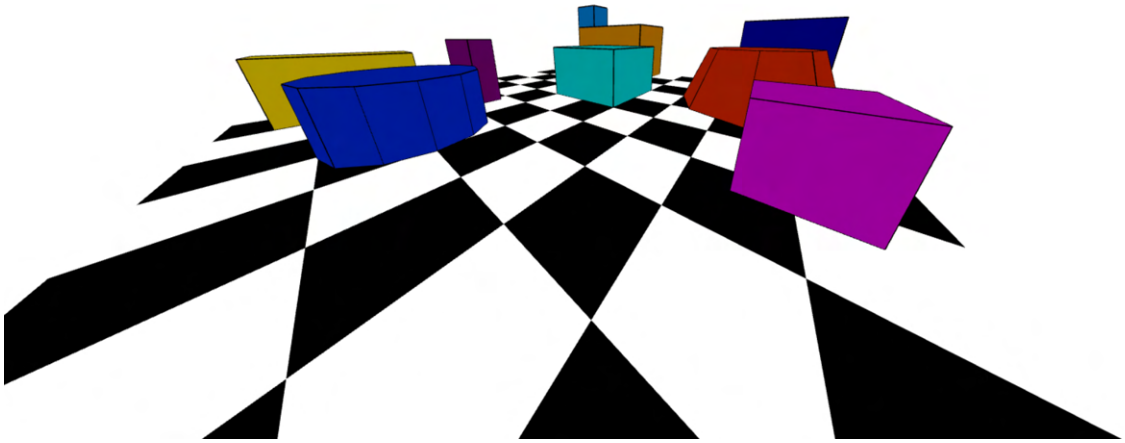
pierwszym planie bądź nadmierne „rozciągnięcie” obiektów w tle.
Poniżej przedstawiono przykładowe rezultaty uzyskane za pomocą biblioteki `Three.js` [5], w których zaprezentowano różne konfiguracje głównych parametrów projekcji α_f, p, f_{min}, f_{max}. Wszystkie przykładowe ustawienia parametrów i kod źródłowy sceny zamieszczono w Dodatku A.



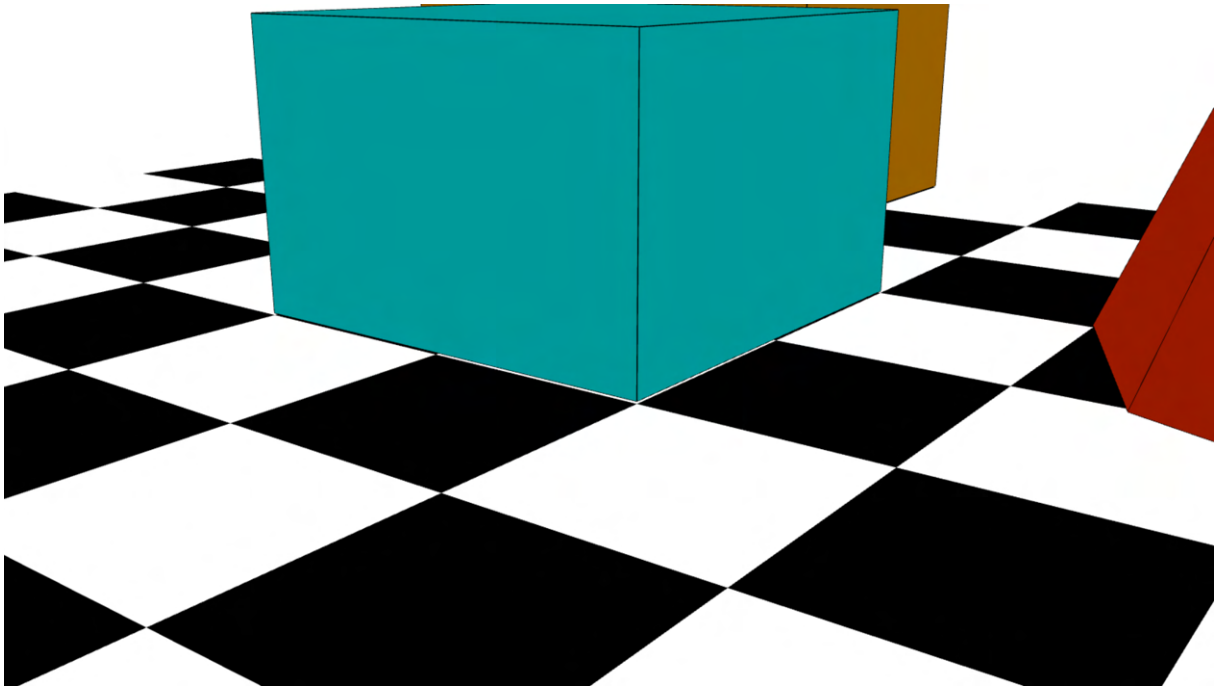
Rysunek 3.11: Widok sceny od góry, z zaznaczoną pozycją kamery i kierunkiem w którym jest skierowana jako czerwony punkt i wektor. *Opracowanie własne.*



Rysunek 3.12: Widok sceny z domyślnymi ustawieniami kamery perspektywicznej, $\alpha_f = 50°$, $p = \frac{1536}{864}$, $f_{\text{min}} = 0.1$, $f_{\text{max}} = 2000$. *Opracowanie własne.*



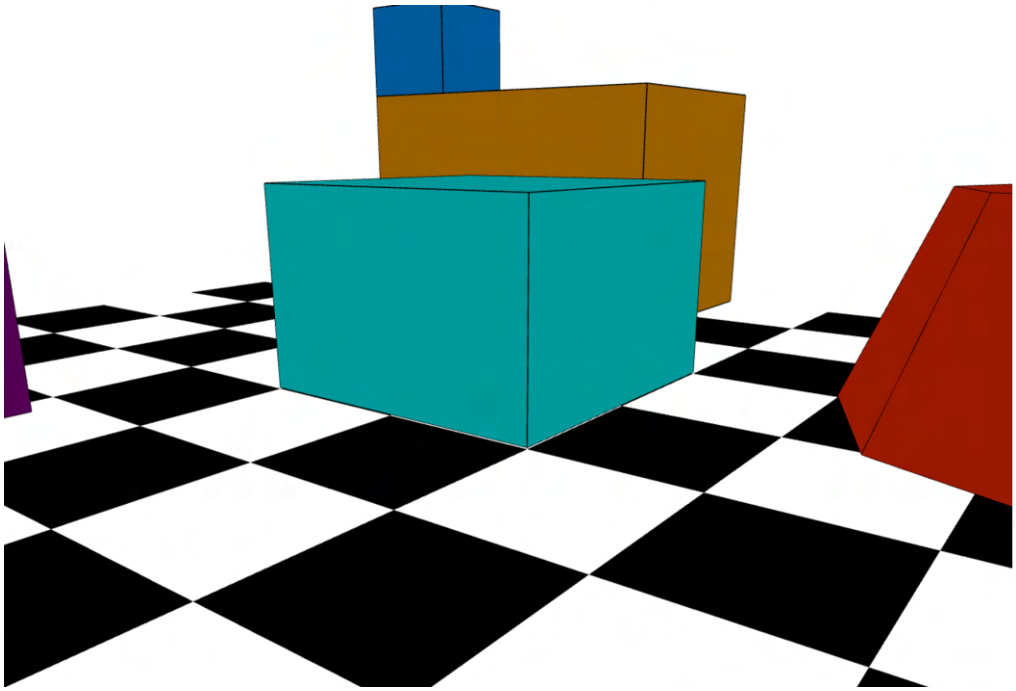
Rysunek 3.13: Widok sceny z szerokim kątem widzenia,
$\alpha_f = 120°$, $p = \frac{1536}{864}$, $f_{\min} = 0.1$, $f_{\max} = 2000$. *Opracowanie własne.*



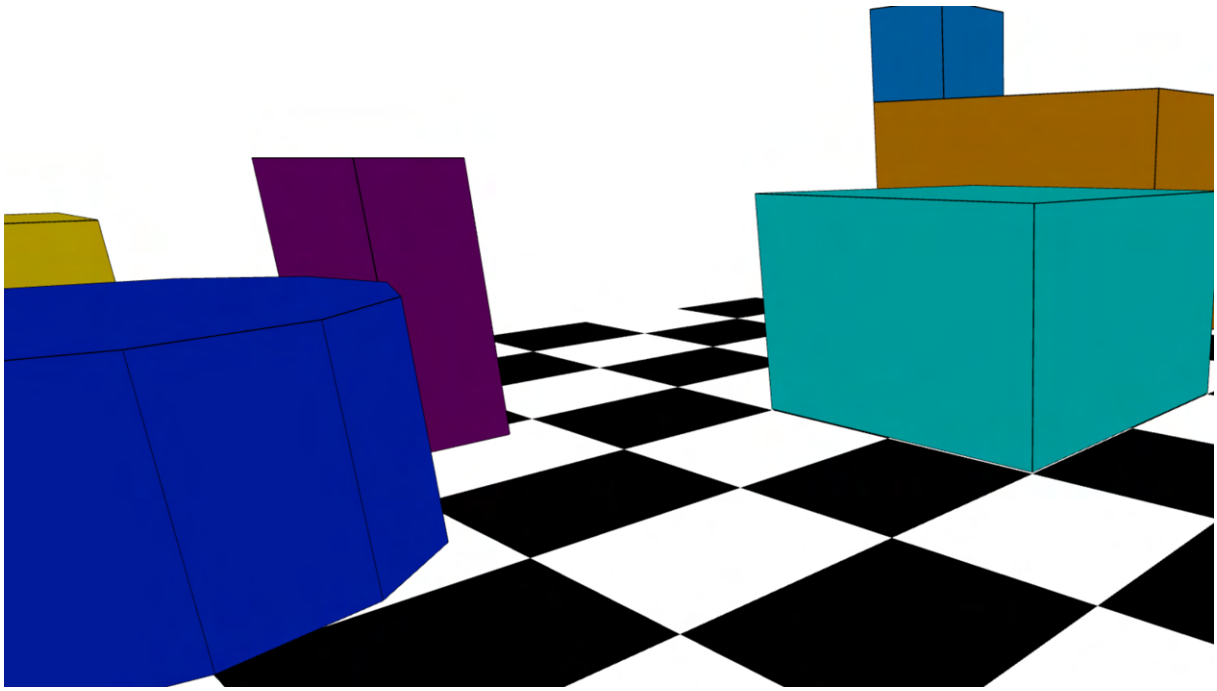
Rysunek 3.14: Widok sceny z wąskim kątem widzenia,
$\alpha_f = 30°$, $p = \frac{1536}{864}$, $f_{\min} = 0.1$, $f_{\max} = 2000$. *Opracowanie własne.*

Poniższe przykłady ilustrują wpływ zmiany proporcji obrazu parametru p na sposób wyświetlania sceny. Warto jednak zaznaczyć, że mimo modyfikowania wartości p sama rozdzielczość ekranu, na którym finalnie prezentowane są obrazy, pozostaje bez zmian. Oznacza to, że efekty „rozciągnięcia" lub „zwężenia" widoku mogą być częściowo wynikiem dostosowywania projektu kamery do innego formatu, niż rzeczywiste wymiary bieżącego okna przeglądarki czy monitora. Tego typu różnice w proporcjach mogą zakłócać wraże-

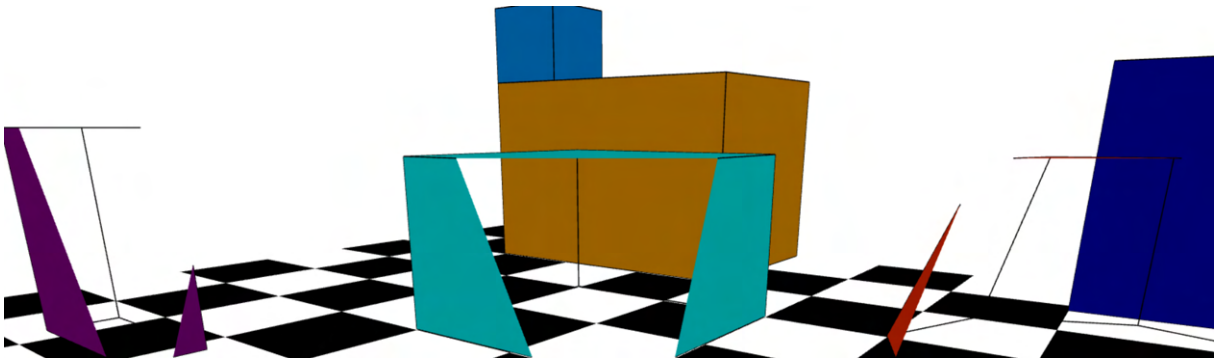
nie przestrzeni – docelowe urządzenie wyświetlało obraz w zadanej proporcji dokładnie odpowiadającej wartości p, obraz sprawiałby wrażenie naturalnej perspektywy.



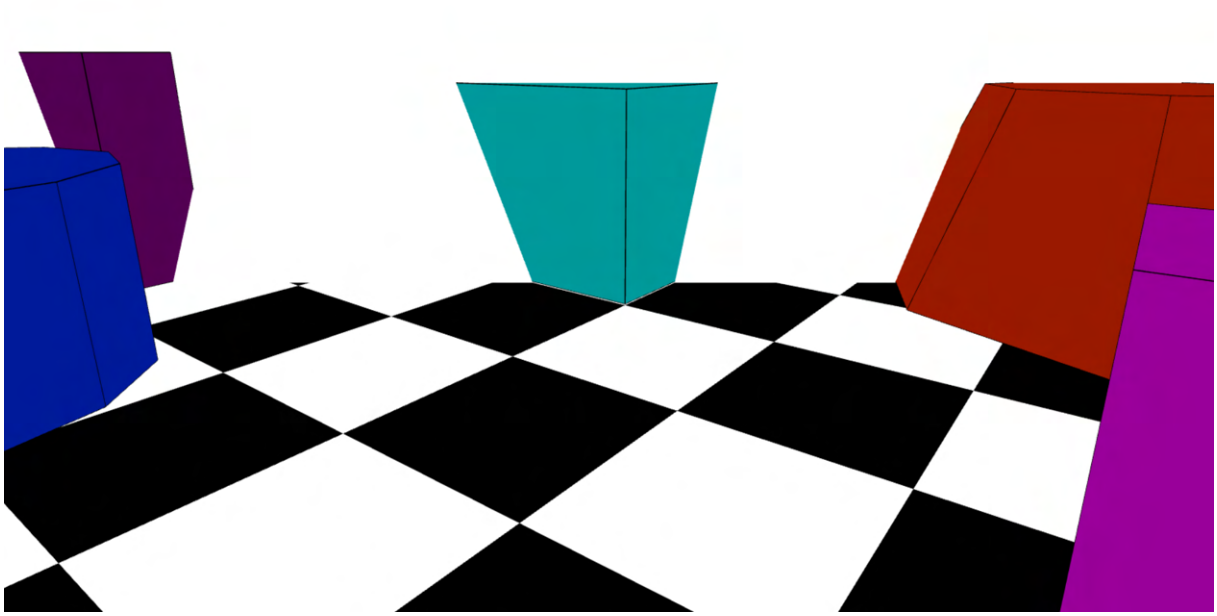
Rysunek 3.15: Widok sceny z zmniejszoną proporcją szerokości i wysokości ekranu, $\alpha_f = 50°$, $p = \frac{1280}{1024}$, $f_{\min} = 0.1$, $f_{\max} = 5$. *Opracowanie własne.*



Rysunek 3.16: Widok sceny z zwiększoną proporcją szerokości i wysokości ekranu, $\alpha_f = 50°$, $p = \frac{2560}{1080}$, $f_{\min} = 0.1$, $f_{\max} = 5$. *Opracowanie własne.*



Rysunek 3.17: Widok sceny z zwiększoną odległością do najbliższej płaszczyzny odcinania, $\alpha_f = 50°$, $p = \frac{1536}{864}$, $f_{\min} = 5$, $f_{\max} = 2000$. *Opracowanie własne.*

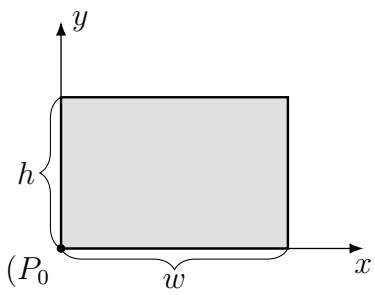


Rysunek 3.18: Widok sceny z zmniejszoną odległością do najodleglejszej płaszczyzny odcinania, $\alpha_f = 50°$, $p = \frac{1536}{864}$, $f_{\min} = 0.1$, $f_{\max} = 5$. *Opracowanie własne.*

W przedstawionych przykładach zauważyć można, że nie tylko sam kąt widzenia, ale również dobór odległości płaszczyzn obcinania od kamery decyduje o tym, jak dużo detali zostanie wyświetlonych.

3.5 Przestrzeń rzutni

Przestrzeń rzutni (*Viewport space*) lub przestrzeń ekranu (*Screen space*) jest ostatnim etapem przekształceń których musimy dokonać, aby wyświetlić obraz na ekranie. W tym miejscu musimy ustalić pozycję obiektów biorąc pod uwagę realne rozmiary obszaru wyświetlania, który definiowany jest przez wysokość h oraz szerokość w. Warto pamiętać, że specyfikacja API WebGL za początek układu współrzędnych przyjmuje punkt $P_0 = (0, 0)$, znajdujący się w lewym dolnym rogu rzutni. Na poniższym rysunku przedstawiono przykładową przestrzeń rzutni



Rysunek 3.19: Przestrzeń rzutni. *Opracowanie własne na podstawie* [4].

Macierz przekształcenia wierzchołków M_V jest połączeniem transformacji skalowania oraz translacji punktów

$$M_V = \begin{bmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Przykład 3.5. Transformacja rzutni

Kontynuując przykład 3.4, przeprowadzimy ostatni etap przekształceń. Jako rozmiary rzutni, przyjmiemy typową rozdzielczość ekranu FullHD, czyli

$$w = 1920, \quad h = 1080,$$
$$M_V = \begin{bmatrix} 960 & 0 & 0 & 960 \\ 0 & 540 & 0 & 540 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

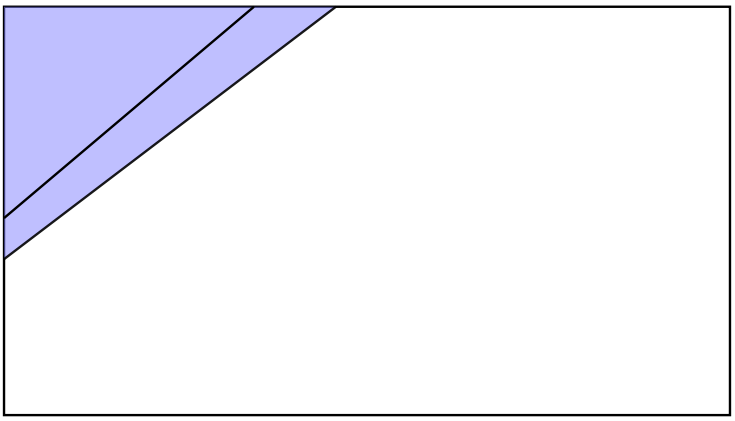
Przykład 3.5.

Zatem końcowa macierz wierzchołków będzie wyglądać następująco

$$T_{\mathrm{VP}} = M_V T_{\mathrm{NDC}} = \begin{bmatrix} \frac{960\left(-\frac{140}{41}+\frac{38\sqrt{10}}{41}\right)}{\frac{56}{41}-\frac{2\sqrt{10}}{41}}+960 & \frac{960\left(-\frac{140}{41}+\frac{22\sqrt{10}}{41}\right)}{\frac{46}{41}-\frac{2\sqrt{10}}{41}}+960 \\ 540+\frac{540\left(\frac{36}{41}+\frac{7\sqrt{10}}{15}\right)}{\frac{56}{41}-\frac{2\sqrt{10}}{41}} & 540+\frac{540\left(\frac{30}{41}+\frac{7\sqrt{10}}{15}\right)}{\frac{46}{41}-\frac{2\sqrt{10}}{41}} \\ \frac{-\frac{592}{205}-\frac{14\sqrt{10}}{205}}{\frac{56}{41}-\frac{2\sqrt{10}}{41}} & \frac{-\frac{662}{205}-\frac{14\sqrt{10}}{205}}{\frac{46}{41}-\frac{2\sqrt{10}}{41}} \\ 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{960\left(-\frac{116}{41}+\frac{22\sqrt{10}}{41}\right)}{\frac{19\sqrt{10}}{82}+\frac{46}{41}}+960 & \frac{960\left(-\frac{116}{41}+\frac{38\sqrt{10}}{41}\right)}{\frac{19\sqrt{10}}{82}+\frac{56}{41}}+960 & \frac{960\left(-\frac{140}{41}-\frac{34\sqrt{10}}{41}\right)}{\frac{56}{41}+\frac{31\sqrt{10}}{41}}+960 \\ 540+\frac{540\left(\frac{30}{41}+\frac{7\sqrt{10}}{15}\right)}{\frac{19\sqrt{10}}{82}+\frac{46}{41}} & 540+\frac{540\left(\frac{36}{41}+\frac{7\sqrt{10}}{15}\right)}{\frac{19\sqrt{10}}{82}+\frac{56}{41}} & \frac{540\left(\frac{36}{41}-\frac{7\sqrt{10}}{5}\right)}{\frac{56}{41}+\frac{31\sqrt{10}}{41}}+540 \\ \frac{-\frac{662}{205}+\frac{133\sqrt{10}}{410}}{\frac{19\sqrt{10}}{82}+\frac{46}{41}} & \frac{-\frac{592}{205}+\frac{133\sqrt{10}}{410}}{\frac{19\sqrt{10}}{82}+\frac{56}{41}} & \frac{-\frac{592}{205}+\frac{217\sqrt{10}}{205}}{\frac{56}{41}+\frac{31\sqrt{10}}{41}} \\ 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{960\left(-\frac{66\sqrt{10}}{41}-\frac{140}{41}\right)}{\frac{46}{41}+\frac{31\sqrt{10}}{41}}+960 & \frac{960\left(-\frac{66\sqrt{10}}{41}-\frac{116}{41}\right)}{\frac{46}{41}+\frac{25\sqrt{10}}{41}}+960 & \frac{960\left(-\frac{116}{41}-\frac{34\sqrt{10}}{41}\right)}{\frac{56}{41}+\frac{25\sqrt{10}}{41}}+960 \\ \frac{540\left(\frac{36}{41}-\frac{7\sqrt{10}}{5}\right)}{\frac{46}{41}+\frac{31\sqrt{10}}{41}}+540 & \frac{540\left(\frac{30}{41}-\frac{7\sqrt{10}}{5}\right)}{\frac{46}{41}+\frac{25\sqrt{10}}{41}}+540 & \frac{540\left(\frac{30}{41}-\frac{7\sqrt{10}}{5}\right)}{\frac{56}{41}+\frac{25\sqrt{10}}{41}}+540 \\ \frac{-\frac{662}{205}+\frac{217\sqrt{10}}{205}}{\frac{46}{41}+\frac{31\sqrt{10}}{41}} & \frac{-\frac{662}{205}+\frac{35\sqrt{10}}{41}}{\frac{46}{41}+\frac{25\sqrt{10}}{41}} & \frac{-\frac{592}{205}+\frac{35\sqrt{10}}{41}}{\frac{56}{41}+\frac{25\sqrt{10}}{41}} \\ 1 & 1 & 1 \end{bmatrix}.$$



4 Implementacja aplikacji webowej do wirtualnego spaceru

W niniejszym rozdziale przedstawiono proces tworzenia aplikacji webowej umożliwiającej wirtualny spacer po korytarzach Instytutu Matematyki Uniwersytetu Zielonogórskiego. Aplikacja została zbudowana z wykorzystaniem technologii `Three.js` [5] oraz uruchamiana w środowisku `Bun` [23], co zapewnia efektywne renderowanie trójwymiarowych scen i sprawną obsługę zapytań serwerowych.
Podstawą wizualną spaceru są zdjęcia sferyczne 360°, wykonane za pomocą odpowiedniego aparatu i przekształcone na mapy sześcienne (*cubemaps*). W dalszej części rozdziału omówiono zarówno przygotowanie materiałów źródłowych oraz proces ich konwersji, jak i techniczne aspekty tworzenia aplikacji — od ogólnej architektury, po funkcjonalności związane z nawigacją w środowisku wirtualnym.
Zaprojektowana aplikacja zapewnia użytkownikom wrażenie rzeczywistego przebywania w fizycznej przestrzeni. W szczególności obejmuje ona:

- swobodne obracanie widokiem w pełnym zakresie 360°,
- przechodzenie między wyznaczonymi punktami w budynku,
- realistyczne odwzorowanie perspektywy.

4.1 Wymagania techniczne

Aparat 360 stopni

Do wykonania szerokich zdjęć panoramicznych niezbędnych do poprawnego uzyskania sześciu zdjęć odzwierciedlających przestrzeń, potrzebny będzie aparat, który umożliwia robienie zdjęć w formacie $2:1$. Jest to bardzo istotny wymóg, ponieważ inne rozdzielczości mogą później spowodować nieprawidłowe stworzenie ścian sześciokąta, uniemożliwiając odpowiednie złożenie *cubemapy*. W pracy wykorzystano aparat Medion MD88190 Panorama360, a zdjęcia wykonano montując go na stojaku w celu uzyskania stałej wysokości, widocznym poniżej

Rysunek 4.1: Aparat Medion MD88190 Panorama360 na stojaku. *Materiał własny.*

Środowisko programistyczne

Do implementacji właściwej części aplikacji webowej, potrzebne będzie dowolne środowisko w którym będziemy mogli pisać kod oraz dostęp do terminalu z uprawnieniami administratora. Do pisania kodu wystarczą najprostsze narzędzia takie jak systemowy Notatnik [19] czy Notepad++ [10] lub bardziej złożone aplikacje, takie jak Visual Studio Code [6] czy WebStorm [11].
Jako język programowania wybrany został `JavaScript` [3], ze względu na jego natywną kompatybilność z przeglądarkami internetowymi, która wspomoże w nadzorowaniu interakcji użytkowników z aplikacją, takich jak poruszanie myszką czy wykrywanie miejsca kliknięć.
Jako środowisko uruchomieniowe aplikacji (*runtime*), wybrano środowisko `Bun`, które jest nowoczesną alternatywą tradycyjnego `Node.js` [20]. `Bun` wyróżnia się wysoką wydajnością, szybszym uruchamianiem aplikacji czy wbudowanym menadżerem pakietów. Aby je zainstalować, należy wejść na stronę `https://bun.sh/` i podążać za instrukcjami na stronie, zgodnymi z naszym systemem operacyjnym (*OS*). W pracy używany system operacyjny to `Ubuntu 22.04.5 LTS` [15], a wersja `Bun` to `1.1.18`. Należy wspomnieć, że

można użyć dowolnego środowiska uruchomieniowego, menadżera pakietów czy systemu operacyjnego.
Kiedy już upewnimy się, że poprawnie zainstalowaliśmy środowisko uruchomieniowe i otrzymamy odpowiedź

bash

```
> bun --version
1.1.18
```

Możemy przejść dalej, utworzyć folder dla naszej aplikacji, uruchomić w nim terminal i zainicjować projekt. `Bun` będzie proponował nam domyślne ustawienia, my jednak wpiszemy je sami

bash

```
> bun init
bun init helps you get started with a minimal project and tries to guess sensible
defaults. Press ^C anytime to quit

package name (projects): cubemaps
entry point (index.ts): index.js

Done! A package.json file was saved in the current directory.
 + index.js
 + .gitignore
 + jsconfig.json (for editor auto-complete)
 + README.md

To get started, run:
  bun run index.js
```

Powyżej możemy zobaczyć, że projekt został zainicjowany oraz nadaliśmy mu nazwę `cubemaps` i ustaliliśmy punkt wejścia, czyli plik w który stanowi centralne miejsce inicjalizacji aplikacji i jej logiki, w którym znajdują się wszystkie istotne konfiguracje aplikacji czy serwera, który nazwaliśmy `index` oraz przypisaliśmy mu rozszerzenie `.js`, czyli `JavaScript` [3]. W odpowiedzi na to, `Bun`przygotował środowisko, dodając do folderu kilka plików, którymi teraz nie będziemy się zajmować. Widoczna jest również komenda, która pozwoli sprawdzić czy wszystko zostało utworzone poprawnie

bash

```
> bun run index.js
Hello via Bun!
```

Kolejnym krokiem, będzie dodanie narzędzia budującego aplikację `Vite` [24] która da nam możliwość sprawnego rozwoju aplikacji dzięki *HMR*, czyli odświeżania widoku w przeglądarce po wykryciu zmian w plikach źródłowych oraz pozwoli na uruchomienie aplikacji i późniejsze jej zbudowanie. Dodajmy też bibliotekę `Three.js`, czyli interfejs pozwalający nam tworzyć trójwymiarowe sceny. Zainstalujmy `Vite` jako zależność deweloperską projektu, a `Three.js` jako zależność produkcyjną. Dodatkowo możemy usunąć zależność `@types/bun`, ponieważ jest to dedykowany pakiet typów dla języka `TypeScript`[1], którego nie będziemy wykorzystywać w tej pracy

bash

```
> bun add --dev vite
> bun add three
> bun remove @types/bun
```

Aby upewnić się, że narzędzie zostało zainstalowane, możemy otworzyć plik `package.json`, który przechowuje konfigurację naszego projektu. Możemy z niego ręcznie usunąć sekcję `peerDependencies` oraz dodać sekcję `scripts` z komendą `dev` uruchamiającą serwer `Vite` - {`"dev": "vite"`}. Aktualnie, plik powinien wyglądać następująco

package.json

```
{
  "name": "cubemaps",
  "type": "module",
  "devDependencies": {
    "vite": "^6.0.7"
  },
  "dependencies": {
    "three": "^0.172.0"
  },
  "scripts": {
    "dev": "vite"
  }
}
```

Następnie musimy dodać plik `index.html`, z następującą zawartością

index.html

```
<!DOCTYPE html>
<html lang="pl">
<head>
    <meta charset="UTF-8">
    <title>cubemaps</title>
    <style>
        body { margin: 0; padding: 0; overflow-x: hidden; }
        #app { width: 100vw; height: 100vh; margin: 0 auto; }
    </style>
</head>
<body>
    <div id="app"></div>
    <script src="/index.js" type="module"></script>
</body>
</html>
```

oraz uzupełnić plik `index.js` następującym kodem

index.js

```
import * as THREE from "three";

const scene = new THREE.Scene();

const wrapper = document.getElementById("app");
const width = wrapper.clientWidth;
const height = wrapper.clientHeight;

const camera = new THREE.PerspectiveCamera(75, width / height, 0.1,1000);
camera.position.z = 5;

const renderer = new THREE.WebGLRenderer();
renderer.setSize(width, height);
renderer.setAnimationLoop(animate);

wrapper.appendChild(renderer.domElement);

const geometry = new THREE.BoxGeometry(1, 1, 1);
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);

function animate() {
  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;
  renderer.render(scene, camera);
}
```

Ostatnim krokiem, który pozwoli nam upewnić się, że wszystko zostało zainstalowane poprawnie, jest wpisanie w terminalu następującej komendy

bash

```
> bun run dev
```

Po jej wykonaniu powinniśmy zobaczyć w terminalu adres i port, np. `localhost:5173`. Oznacza to, że serwer deweloperski Vite działa prawidłowo i pod podanym adresem dostępna będzie zawartość pliku `index.html` wraz z podłączonym skryptem JavaScript, który wyświetla animację obracającego się zielonego sześcianu na czarnym tle.

4.2 Przygotowanie zdjęć mapy sześciennej

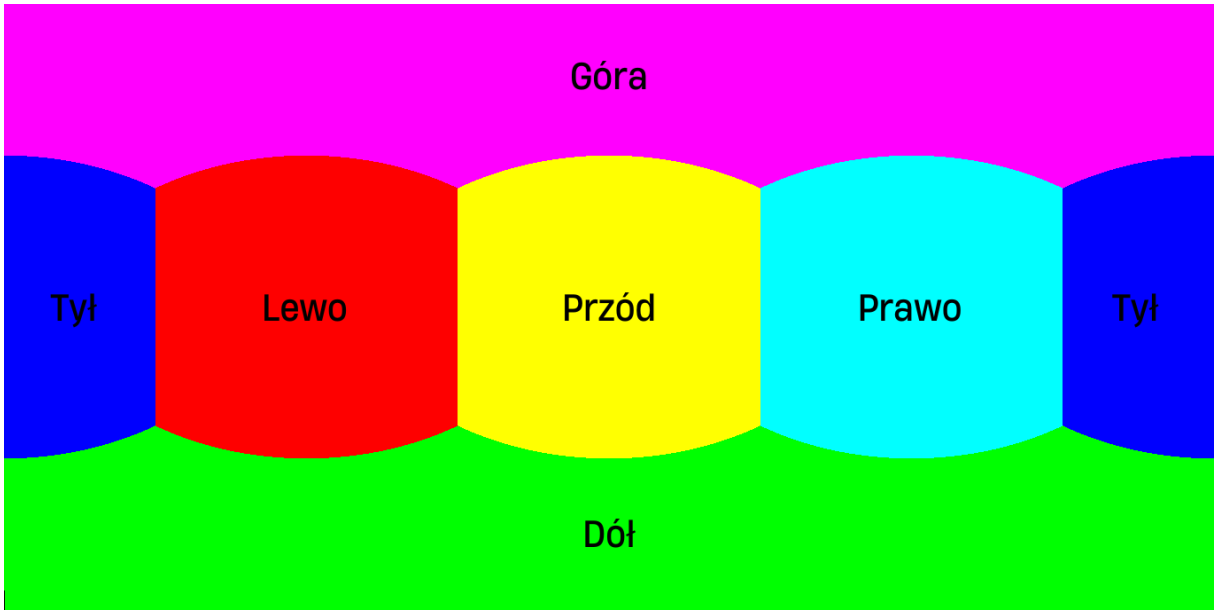
Przygotowanie mapy sześciennej z obrazu panoramicznego wymaga odpowiedniego podziału i transformacji obrazu w taki sposób, aby każdy z widoków (przód, tył, lewo, prawo, góra, dół) został poprawnie odwzorowany na ścianach sześcianu.

4.2.1 Wykonanie i podział zdjęcia panoramicznego

Zdjęcie panoramiczne wykonane aparatem 360° powinno wyglądać podobnie do poniższego

Rysunek 4.2: Przykład zdjęcia panoramicznego wykorzystującego odwzorowanie walcowe. *Materiał własny.*

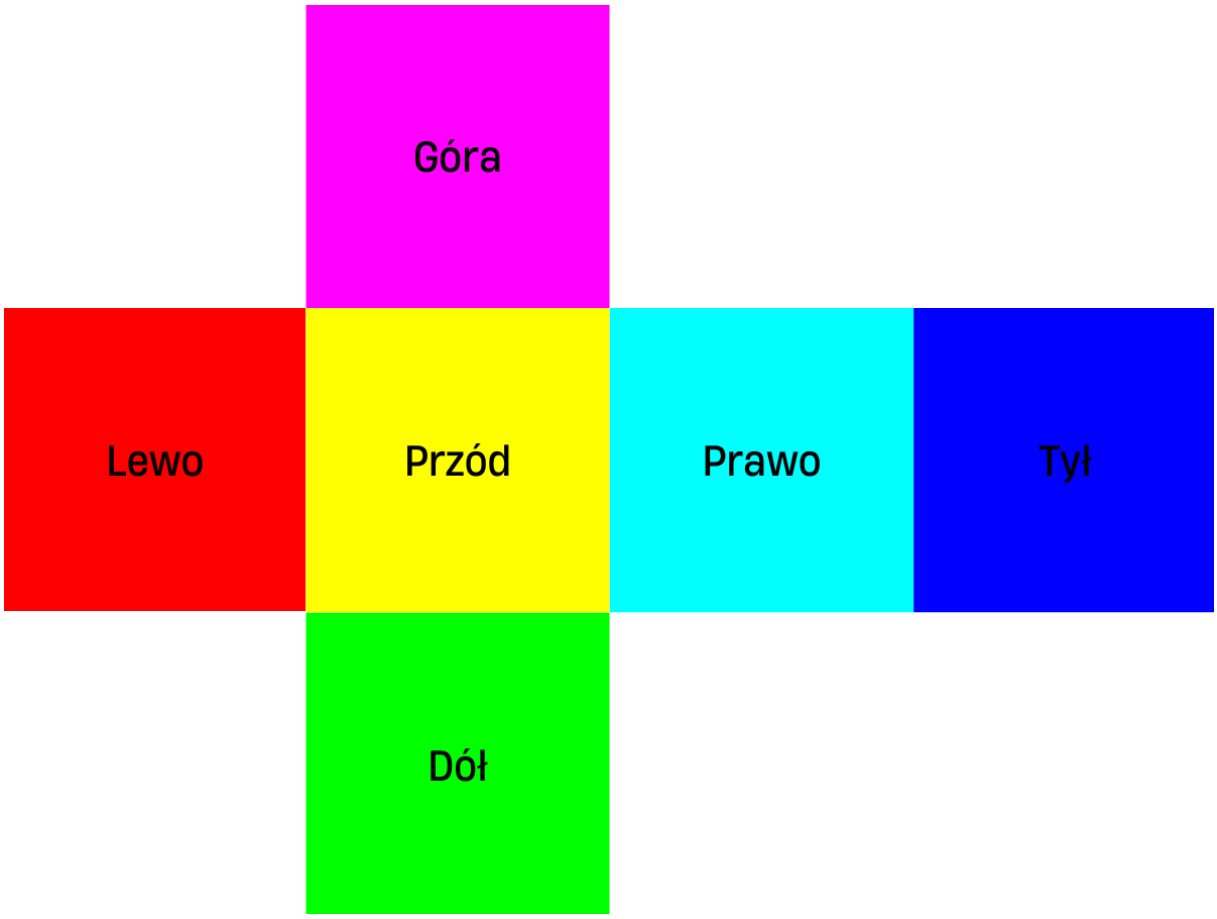
Aby przekształcić zdjęcie panoramiczne w mapę sześcienną, konieczne jest podzielenie obrazu na sześć widoków odpowiadających ścianom sześcianu: przód, tył, lewo, prawo, góra i dół



Rysunek 4.3: Lokalizacje ścian mapy sześciennej w zdjęciu panoramicznym wykorzystującym odwzorowanie walcowe. *Źródło* [16].

Proces ten wymaga zastosowania odpowiednich transformacji geometrycznych, które odwzorują fragmenty obrazu panoramicznego na płaszczyzny sześcianu. W tym celu stosuje się interpolację, aby zapewnić płynne przejścia między pikselami i uniknąć artefaktów. Jedną z najczęściej stosowanych metod interpolacji w tego typu przekształceniach jest interpolacja Lanczosa, która gwarantuje dokładniejsze przekształcenie przy dłuższym czasie wykonania obliczeń. Do podziału zdjęcia, wykorzystamy aplikację dostępną pod linkiem `https://jaxry.github.io/panorama-to-cubemap/` [8]. W sekcji *Upload*, musimy

dodać swój plik z panoramicznym zdjęciem, zmienić ustawienie obrotu kostki na 270° w sekcji *Settings*. Po chwili w sekcji *Output*, powinien pojawić się rozłożony płasko sześcian. Nawiązując do poprzedniego rysunku 4.3, podział powinien wyglądać podobnie do poniższego



Rysunek 4.4: Mapa sześcienna po wykonanej interpolacji Lanczosa na zdjęciu panoramicznym. *Źródło* [9].

Kiedy pobierzemy wynik zastosowanej interpolacji na zdjęciu, klikając w każdą z uzyskanych ścian, otrzymamy zestaw sześciu zdjęć, nazwanymi odpowiednio dla każdej osi współrzędnych oraz kierunku w którym powinny się znaleźć podczas ich złożenia. Odpowiednio więc będą to

- dla osi x, w kierunku pozytywnych wartości plik `px.png`, w kierunku negatywnych wartości plik `nx.png`,
- dla osi y, w kierunku pozytywnych wartości plik `py.png`, w kierunku negatywnych wartości plik `ny.png`,
- oś z, w kierunku pozytywnych wartości plik `pz.png`, w kierunku negatywnych wartości plik `nz.png`.

Rysunek 4.5: Grafika ściany odpowiadającej negatywnemu kierunkowi osi x, `nx.png`. *Materiał własny.*

Rysunek 4.6: Grafika ściany odpowiadającej pozytywnemu kierunkowi osi x, `px.png`. *Materiał własny.*

Rysunek 4.7: Grafika ściany odpowiadającej negatywnemu kierunkowi osi y, `ny.png`. *Materiał własny.*

Rysunek 4.8: Grafika ściany odpowiadającej pozytywnemu kierunkowi osi y, `py.png`. *Materiał własny.*

Rysunek 4.9: Grafika ściany odpowiadającej negatywnemu kierunkowi osi z, `nz.png`. *Materiał własny.*

Rysunek 4.10: Grafika ściany odpowiadającej pozytywnemu kierunkowi osi z, `pz.png`. *Materiał własny.*

4.2.2 Implementacja mapy sześciennej w Three.js

Wróćmy do części programistycznej, stwórzmy folder `public` i dodajmy do niego wszystkie uzyskane w poprzednim rozdziale zdjęcia. Struktura naszego projektu powinna wyglądać

następująco

```
cubemaps
├── node_modules
├── public
│   ├── nx.png
│   ├── ny.png
│   ├── nz.png
│   ├── px.png
│   ├── py.png
│   └── pz.png
├── .gitignore
├── bun.lockb
├── index.html
├── index.js
├── jsconfig.json
├── package.json
└── README.md
```

Następnie otwórzmy plik `index.js` i zaimplementujmy mapę sześcienną, która będzie wyświetlana na całym dostępnym oknie przeglądarki. W pierwszej kolejności wykorzystamy klasę `TextureLoader`, która umożliwia odczytywanie tekstur z wskazanych plików i późniejsze ich nałożenie na obiekty, oraz wskażemy ścieżkę do folderu w którym znajdować się będą ściany naszej mapy sześciennej.

index.js

```
const textureLoader = new THREE.TextureLoader();
textureLoader.setPath("/")
```

Jak można zauważyć, jako ścieżkę do folderu `public` ustaliliśmy `"/"`. W wielu bibliotekach (*frameworks*) webowych folder `public` jest domyślnie mapowany na katalog główny URL-a (‘/‘), co pozwala na serwowanie plików statycznych, takich jak obrazy czy style CSS, bezpośrednio z tego folderu. Dzięki temu pliki publiczne są łatwo dostępne dla użytkowników, a jednocześnie oddzielone od kodu aplikacji.
Dodajmy teraz wszystkie zdjęcia jako listę obiektów klasy `MeshBasicMaterial`, wskazując po której stronie ściany będziemy wyświetlać obraz.

index.js

```
const materials = [
  new THREE.MeshBasicMaterial({
    map: textureLoader.load("px.png"),
    side: THREE.BackSide,
  }),
  new THREE.MeshBasicMaterial({
    map: textureLoader.load("nx.png"),
    side: THREE.BackSide,
  }),
  new THREE.MeshBasicMaterial({
    map: textureLoader.load("py.png"),
    side: THREE.BackSide,
  }),
  new THREE.MeshBasicMaterial({
    map: textureLoader.load("ny.png"),
    side: THREE.BackSide,
  }),
  new THREE.MeshBasicMaterial({
    map: textureLoader.load("pz.png"),
    side: THREE.BackSide,
  }),
  new THREE.MeshBasicMaterial({
    map: textureLoader.load("nz.png"),
    side: THREE.BackSide,
  }),
];
```

Możemy też usunąć wszystkie linie kodu związane z poprzednio wyświetlaną i obracającą się zieloną kostką. W funkcji `animate` usuńmy linijki związane z rotacją kostki, zostawiając samą funkcję odpowiadającą za wyświetlenie sceny.

index.js

```
- const geometry = new THREE.BoxGeometry(1, 1, 1);
- const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
- const cube = new THREE.Mesh(geometry, material);
- scene.add(cube);
- camera.position.z = 5;

function animate() {
    - cube.rotation.x += 0.01;
    - cube.rotation.y += 0.01;
    renderer.render(scene, camera);
}
```

Stwórzmy teraz bryłę sześcianu, wykorzystując klasę `BoxGeometry`, gdzie jako wartości parametrów wymiaru, musimy trzykrotnie wpisać taką samą wartość aby uzyskać prawidłowy sześcian. Następnie stworzymy obiekt klasy `Mesh`, łącząc bryłę i listę materiałów i dodamy go do wyświetlanej sceny.

index.js

```
const geometry = new THREE.BoxGeometry(1000, 1000, 1000);
const cubemap = new THREE.Mesh(geometry, materials);
scene.add(cubemap);
```

Na sam koniec dodajmy kod odpowiedzialny za responsywność wyświetlanego obrazu. Zaimportujmy obiekt `OrbitControls` z pliku który został dodany przy instalacji pakietu `Three.js`, `three/addons/controls/OrbitControls.js`. Jest to obiekt który będzie kontrolował nasze poruszanie myszką po ekranie, dzięki czemu będziemy mogli obracać się w naszej mapie sześciennej. Dodamy również funkcję odpowiedzialną za dopasowywanie rozmiaru wyświetlanego zdjęcia, jeśli okno przeglądarki zmieni swoją szerokość lub wysokość w trakcie działania aplikacji.

index.js

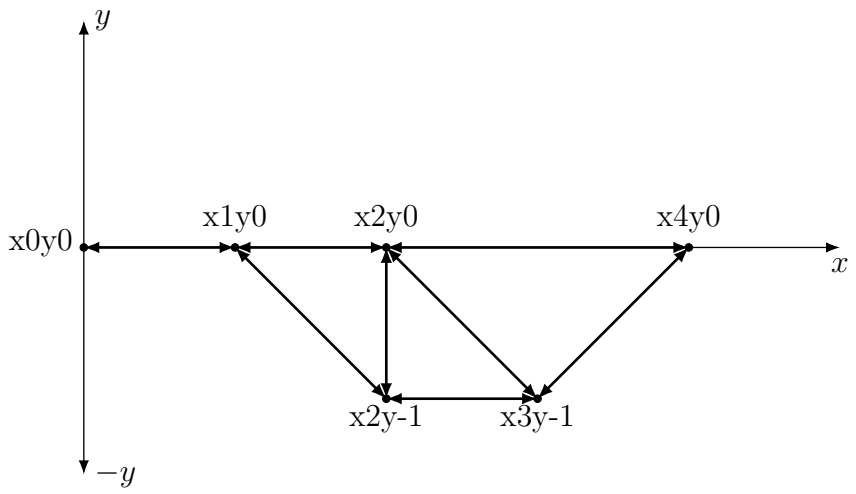
```
import { OrbitControls } from "three/addons/controls/OrbitControls.js";

const controls = new OrbitControls(camera, renderer.domElement);
controls.minDistance = 1.5;
controls.maxDistance = 6;

function onWindowResize() {
    const width = wrapper.clientWidth;
    const height = wrapper.clientHeight;

    camera.aspect = width / height;
    camera.updateProjectionMatrix();

    renderer.setSize(width, height);
}

window.addEventListener("resize", onWindowResize);

function animate() {
    controls.update();
    renderer.render(scene, camera);
}
```

W ten sposób, po uruchomieniu aplikacji komendą `bun run dev` i wpisaniu wskazanego adresu w przeglądarkę, powinniśmy zobaczyć złożoną mapę sześcienną w której możemy oglądać obraz dookoła nas.

4.2.3 Implementacja możliwości poruszania się pomiędzy zdjęciami

W ostatnim kroku implementacji aplikacji, skupimy się na możliwości poruszania się pomiędzy różnymi punktami wirtualnego spaceru, czyli pomiędzy kolejnymi mapami sześciennymi. W tym celu zaimplementujemy mechanizm nawigacji, który umożliwi płynne przełączanie się między mapami sześciennymi reprezentującymi różne lokalizacje. Omówimy sposób definiowania punktów przejścia, ich wizualizację w środowisku 3D oraz obsługę interakcji użytkownika, takich jak kliknięcia myszką.
Zacznijmy od struktury przechowywania zdjęć dla części wizualizowanego korytarza. Każdy zestaw zdjęć do mapy sześciennej będzie przechowywany w odpowiednio oznaczonym, osobnym folderze. Dla ułatwienia przechowywania informacji o miejscu w której mapie sześciennej się obecnie wykorzystamy układ współrzędnych R^2, aby ułatwić przyszłą możliwość rozszerzenia aplikacji. Każdy punkt symbolizuje mapę sześcienną, a współrzędne punktu będą pomocne przy określeniu gdzie obecnie się znajdujemy i do jakich punktów możemy przejść. Wektory natomiast pokazują pomiędzy którymi punktami użytkownik będzie mógł się poruszyć. Przykładowo, część wykonanych zdjęć w mapowanych Instytucie

Matematyki będzie wyglądać



Rysunek 4.11: Wizualizacja struktury przechowywania folderów ze zdjęciami. *Opracowanie własne.*

Na rysunku widać połączenie pomiędzy mapą sześcienną x2y0 a x4y0, czyli z pominięciem współrzędnej x = 3, mimo, że punkt x3y0 nie istnieje. Dzieje się tak, ponieważ tworzony przez nas mechanizm połączeń będzie poszukiwał punktów oddalonych od nas o maksymalnie 3 jednostki w określonym kierunku. To znaczy, że jeśli punkt oddalony o 1 nie istnieje, będziemy szukać punktu odległego o 2, a następnie 3. Jeśli nadal nie znajdziemy punktu w danym kierunku, program nie pokaże możliwości przejścia dalej. Zaimplementujmy więc ten mechanizm dla mapy sześciennej x1y0. Folder `public` powinien wyglądać następująco

```
public
├── x0y0
│   ├── nx.png
│   ├── ny.png
│   ├── nz.png
│   ├── px.png
│   ├── py.png
│   └── pz.png
├── x1y0
├── x2y0
└── x2y-1
```

Zacznijmy od zaimplementowania tej mapy w kodzie. Stworzymy zmienną przechowującą `id` kostki w której będziemy startować oraz tablicę z łańcuchów znaków symbolizującymi mapę sześcienną, będącymi jej *id* i nazwą folderu z zdjęciami.

index.js

```
const startingCubeId = "x1y0";
const map = ["x0y0", "x1y0", "x2y0", "x2y-1", "x3y-1", "x4y0"];
```

Teraz, stwórzmy funkcję która będzie zwracała wszystkich sąsiadów wybranej przez nas mapy sześciennej.

index.js

```
function findNeighbors(point) {
  const x = Number.parseInt(point.match(/x(-?\d+)/)[1], 10);
  const y = Number.parseInt(point.match(/y(-?\d+)/)[1], 10);
  const directions = [];
  let px, nx, py, ny;
  let pxy, nxy, pxny, nxny;
  for (let step = 1; step <= 3; step++) {
    if (px && nx && py && ny && pxy && nxy && pxny && nxny) break;
    if (!px && map.includes(`x${x + step}y${y}`)) px = `x${x + step}y${y}`;
    if (!nx && map.includes(`x${x - step}y${y}`)) nx = `x${x - step}y${y}`;
    if (!py && map.includes(`x${x}y${y + step}`)) py = `x${x}y${y + step}`;
    if (!ny && map.includes(`x${x}y${y - step}`)) ny = `x${x}y${y - step}`;
    if (!pxy && map.includes(`x${x + step}y${y + step}`)) {
      pxy = `x${x + step}y${y + step}`;
    }
    if (!nxy && map.includes(`x${x - step}y${y + step}`)) {
      nxy = `x${x - step}y${y + step}`;
    }
    if (!pxny && map.includes(`x${x + step}y${y - step}`)) {
      pxny = `x${x + step}y${y - step}`;
    }
    if (!nxny && map.includes(`x${x - step}y${y - step}`)) {
      nxny = `x${x - step}y${y - step}`;
    }
  }
  if (px) directions.push({ cubemapId: px, loc: [450, 0, 0] });
  if (nx) directions.push({ cubemapId: nx, loc: [-450, 0, 0] });
  if (py) directions.push({ cubemapId: py, loc: [0, 0, 450] });
  if (ny) directions.push({ cubemapId: ny, loc: [0, 0, -450] });
  if (pxy) directions.push({ cubemapId: pxy, loc: [450, 0, 450] });
  if (nxy) directions.push({ cubemapId: nxy, loc: [-450, 0, 450] });
  if (pxny) directions.push({ cubemapId: pxny, loc: [450, 0, -450] });
  if (nxny) directions.push({ cubemapId: nxny, loc: [-450, 0, -450] });
  return directions;
}
```

Następnie, stwórzmy logikę, która na podstawie otrzymanych lokalizacji sąsiadów z funkcji `findNeighbors` utworzy obiekty pozwalające przemieścić się do kolejnych map sześciennych

index.js

```
let neighborsIds = findNeighbors(startingCubeId);
let neighborsObjects = createNeighborObjects(neighborsIds);
```

Aby zaimplementować obsługę kliknięć myszy oraz możliwość lokalizacji tych kliknięć, musimy dodać obiekt klasy `Raycaster` oraz `Vector2`, do przechowywania informacji o współ-

rzędnych kliknięcia myszą. Obsługa kliknięcia będzie odbywać się w funkcji `onMouseClick`. Żeby zdobyć informację, czy kliknięcie użytkownika powinno go przenieść do innej mapy sześciennej, wykorzystamy dwie metody klasy `Raycaster`, pierwsza to `setFromCamera`, która otrzymuje informacje o współrzędnych kliknięcia myszą oraz pozycji kamery i tworzy prostą przecinającą te 2 punkty. Następnie przy wykorzystaniu drugiej metody wspomnianej klasy - `intersectObject`, sprawdzimy czy prosta przecina dowolny obiekt sceny. Jeśli użytkownik naciśnie w obiekt który służy do przeniesienia go dalej w naszym wirtualnym spacerze, odkryjemy to i odpowiednio załadujemy kolejne zdjęcia mapy sześciennej. Dodatkowo, musimy pamiętać o usuwaniu poprzednio znalezionych obiektów sąsiadów i znajdować nowych, dla aktualnej mapy sześciennej

index.js

```
const raycaster = new THREE.Raycaster();
const mouse = new THREE.Vector2();
function onMouseClick(event) {
  mouse.x = (event.clientX / width) * 2 - 1;
  mouse.y = -(event.clientY / height) * 2 + 1;
  raycaster.setFromCamera(mouse, camera);
  const intersects = raycaster.intersectObjects(scene.children);
  const firstCubemapId = intersects
    .map((item) => item.object.userData.cubemapId)
    .find((id) => id !== undefined);
  if (firstCubemapId) {
    const newMaterials = [
      new THREE.MeshBasicMaterial({
        map: texureLoader.load(`${firstCubemapId}/px.png`),
        side: THREE.BackSide,
      }),
      new THREE.MeshBasicMaterial({
        map: texureLoader.load(`${firstCubemapId}/nx.png`),
        side: THREE.BackSide,
      }),
      new THREE.MeshBasicMaterial({
        map: texureLoader.load(`${firstCubemapId}/py.png`),
        side: THREE.BackSide,
      }),
      new THREE.MeshBasicMaterial({
        map: texureLoader.load(`${firstCubemapId}/ny.png`),
        side: THREE.BackSide,
      }),
      new THREE.MeshBasicMaterial({
        map: texureLoader.load(`${firstCubemapId}/pz.png`),
        side: THREE.BackSide,
      }),
      new THREE.MeshBasicMaterial({
        map: texureLoader.load(`${firstCubemapId}/nz.png`),
        side: THREE.BackSide,
      }),
    ];
```

index.js

```
    cubemap.material = newMaterials;
    for (const neighbor of neighborsObjects) { scene.remove(neighbor); }
    neighborsObjects = [];
    neighborsIds = [];
    neighborsIds = findNeighbors(firstCubemapId);
    neighborsObjects = neighborsIds.map((neighbor) => {
      const geometry = new THREE.BoxGeometry(50, 50, 50);
      const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
      const cube = new THREE.Mesh(geometry, material);
      cube.position.set(...neighbor.loc);
      cube.userData.cubemapId = neighbor.cubemapId;
      scene.add(cube);
      return cube;
    });
  }
}
window.addEventListener("click", onMouseClick);
```

Uzyskany finalnie kod dostępny będzie w załączniku do niniejszej pracy. Po wpisaniu komendy uruchamiającej lokalny serwer Vite zobaczyć będziemy mogli podobny widok jak na zdjęciu poniżej, a po kliknięciu dowolnego zielonego kwadratu, powinniśmy zostać przeniesieni do następnej mapy sześciennej.

Rysunek 4.12: Widok z kamery w mapie sześciennej oznaczonej x1y0. *Materiał własny.*

Rysunek 4.13: Widok z kamery w mapie sześciennej oznaczonej x2y0. *Materiał własny.*

Co istotne, aplikacja pozwala na łatwe rozszerzenie swojego rozmiaru, dodając poprawnie oznaczone foldery z zdjęciami. Wirtualny spacer po Instytucie Matematyki dostępny jest na stronie internetowej dostępnej pod adresem `https://dejwodejo.github.io/cube`, a cały kod aplikacji webowej dostępny jest w załączniku do pracy dyplomowej oraz w zdalnym repozytorium na platformie Github pod adresem `https://github.com/dejwodejo/cube/`.

Podsumowanie

Praca dyplomowa koncentruje się na teoretycznych i praktycznych aspektach modelowania 3D na podstawie zdjęć. W części teoretycznej wprowadzono kluczowe pojęcia, takie jak przestrzeń wektorowa, baza standardowa, iloczyn skalarny i rzut środkowy, oraz przeanalizowano operacje skalowania, rotacji i translacji. Przedstawiono przykłady obliczeniowe – od transformacji sześcianu po wyznaczanie rzutu punktu na płaszczyznę – które zobrazowały praktyczne zastosowanie teorii i ułatwiły zrozumienie złożonych procesów matematycznych.

Główny nacisk położono na analizę hierarchii układów współrzędnych w potoku graficznym od przestrzeni lokalnej, przez światową, widoku i przycinania, aż do rzutni. Przedstawiono matematyczne podstawy transformacji między tymi układami, w tym strukturę macierzy projekcji perspektywicznej, oraz przeanalizowano wpływ parametrów kamery takich jak kąt widzenia czy płaszczyzny przycinania – na finalny obraz. Do wizualizacji tych zależności wykorzystaliśmy bibliotekę Three.js, demonstrując, jak teoria przekłada się na kontrolę nad renderowaniem sceny.

Zwieńczeniem naszej pracy jest implementacja aplikacji webowej umożliwiającej wirtualny spacer po Instytucie Matematyki Uniwersytetu Zielonogórskiego. Wykorzystano zdjęcia panoramiczne 360°, przekształcone w mapy sześcienne, oraz zaimplementowaliśmy mechanizm nawigacji oparty na wykrywaniu kliknięć, który umożliwił płynne przejścia między punktami. Modularna struktura kodu pozwala nam na łatwe dodawanie nowych lokalizacji, czyniąc rozwiązanie uniwersalnym narzędziem do wizualizacji dowolnej przestrzeni.

Dodatek

A Kod źródłowy sceny do przedstawienia wpływu parametrów kamery na wyświetlanie obrazu.

Zaprojektowana scena składa się z dziewięciu obiektów oraz podłogi w stylu szachownicy, by lepiej wizualizować odległości między obiektami. Do stworzenia sceny użyto biblioteki three.js.

scene.js

```
import * as THREE from "three"; // Import biblioteki Three.js

const scene = new THREE.Scene(); // Tworzenie sceny 3D

// Ustawianie wymiarów okna
const windowWidth = window.innerWidth;
const windowHeight = window.innerHeight;

// Konfiguracja kamery (perspektywa, pozycja, punkt obserwacji)
const lookAtTarget = new THREE.Vector3(0, 0, 0);
const camera = new THREE.PerspectiveCamera(
  50,
  windowWidth / windowHeight,
  0.1,
  2000,
);
camera.position.set(3, 1.5, 4);
camera.lookAt(lookAtTarget);

// Tworzenie obiektu służącego do wyświetlenia sceny
// i dołączenie go do elementu w dokumencie
const renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(windowWidth, windowHeight);
renderer.setClearColor(0xffffff);
document.getElementById("app").appendChild(renderer.domElement);

// Generowanie szachownicy oraz rozmieszenie jej w scenie
const gridSize = 10;
const squareSize = 1;
const gridGroup = new THREE.Group();

// Dodanie oświetlenia w scenie
const ambientLight = new THREE.AmbientLight(0xffffff, 1);
scene.add(ambientLight);
```

scene.js

```
for (let i = 0; i < gridSize; i++) {
  for (let j = 0; j < gridSize; j++) {
    const geometry = new THREE.PlaneGeometry(squareSize, squareSize);
    const color = (i + j) % 2 === 0 ? 0xffffff : 0x000000;
    const material = new THREE.MeshBasicMaterial({
      color,
      side: THREE.DoubleSide,
    });
    const square = new THREE.Mesh(geometry, material);

    square.position.x = (i - gridSize / 2) * squareSize;
    square.position.y = (j - gridSize / 2) * squareSize;

    gridGroup.add(square);
  }
}

gridGroup.rotation.x = -Math.PI / 2;
gridGroup.position.y = -0.01;
scene.add(gridGroup);

// Definicja i rozmieszczenie różnych brył
const blocks = [
  { geometry: new THREE.BoxGeometry(1, 3, 1),
     material: new THREE.MeshStandardMaterial({ color: 0x0099ff }),
     position: { x: -4, y: 1, z: -4 }},
  { geometry: new THREE.BoxGeometry(2, 2, 1.5),
     material: new THREE.MeshStandardMaterial
      ({ color: 0x0000ff, metalness: 0.5, roughness: 0.1 }),
     position: { x: 3.5, y: 0.5, z: -4 }},
  { geometry: new THREE.BoxGeometry(1, 1.2, 2),
      material: new THREE.MeshStandardMaterial
       ({ color: 0xffd700, emissive: 0x555500 }),
     position: { x: -4, y: 1, z: 4 }},
  { geometry: new THREE.BoxGeometry(1, 1, 1),
     material: new THREE.MeshStandardMaterial({ color: 0xff00ff }),
     position: { x: 3.5, y: 0.75, z: 1.4 }},
  { geometry: new THREE.BoxGeometry(3, 2, 2),
     material: new THREE.MeshStandardMaterial({ color: 0xffa500 }),
     position: { x: -2, y: 1.5, z: -3.5 }},
  { geometry: new THREE.BoxGeometry(1.5, 1.5, 0.5),
     material: new THREE.MeshStandardMaterial
      ({ color: 0x800080, emissive: 0x220022 }),
     position: { x: -3, y: 1, z: 2 }},
  { geometry: new THREE.BoxGeometry(2, 1.3, 2),
     material: new THREE.MeshStandardMaterial({ color: 0x00ffff }),
     position: { x: -0.5, y: 0.2, z: -0.5 }},
  { geometry: new THREE.CylinderGeometry(1, 1.5, 1.3, 8),
     material: new THREE.MeshStandardMaterial({ color: 0xff3300 }),
     position: { x: 3, y: 0.5, z: -1.5 }},
  { geometry: new THREE.CylinderGeometry(1, 1, 1, 12),
     material: new THREE.MeshStandardMaterial({ color: 0x0033ff }),
     position: { x: -1.4, y: 0.5, z: 3.5 }},
];
```

scene.js

```
// Tworzenie siatek i krawędzi dla zdefiniowanych brył
for (const { geometry, material, position } of blocks) {
  const mesh = new THREE.Mesh(geometry, material);
  mesh.position.set(position.x, geometry.parameters.height / 2, position.z);

  if (material.color.getHex() === 0xffd700) {
    mesh.position.y = geometry.parameters.height / 2;
  }

  const edges = new THREE.EdgesGeometry(geometry);
  const lineMaterial = new THREE.LineBasicMaterial({ color: 0x000000 });
  const line = new THREE.LineSegments(edges, lineMaterial);
  line.position.copy(mesh.position);
  scene.add(line);

  scene.add(mesh);
}

// Funkcja animacji z pojedynczym wywołaniem renderowania
function animate() {
  renderer.render(scene, camera);
}

animate(); // Start renderowania
```

Uruchomienie sceny

Aby uruchomić scenę, należy w pliku `index.html` wskazać na lokalizację powyższego kodu za pomocą znacznika `script`. Przykładowo, jeśli kod źródłowy sceny znajduje się w pliku `script.js` oraz znajduje się w tym samym folderze co plik `index.html`, załączenie kodu powinno wyglądać następująco

index.html

```
    <script src="/scene.js" type="module"></script>
```

Bibliografia

[1] Microsoft Corporation A. Hejlsberg. Typescript. https://www.typescriptlang.org, 2012. Visited: 2024-12-05.

[2] S. Ho Ahn. Opengl transformation. http://www.songho.ca/, 2012. Visited: 2024-11-15.

[3] Netscape B. Eich. Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript, 1995. Visited: 2024-12-05.

[4] W. C. Brown. Learn webgl. https://learnwebgl.brown37.net/, 2016. Visited: 2025-01-13.

[5] R. Cabello and Contributors. https://threejs.org/, 2025. Visited: 2024-10-16.

[6] Microsoft Corporation. Visual studio code. https://code.visualstudio.com, 2015. Visited: 2025-01-07.

[7] H. S. M. Coxeter and S. L. Greitzer. *Geometry revisited, Fifth printing.* The Mathematical Association of America, Washington, D.C., 1967.

[8] L. Crane. Panorama to cubemap. https://jaxry.github.io/panorama-to-cubemap/, 2017. Visited: 2025-01-18.

[9] J. de Vries. https://learnopengl.com/, 2016. Visited: 2024-11-18.

[10] D. Ho. Notepad++. https://notepad-plus-plus.org, 2003. Visited: 2025-01-07.

[11] JetBrains. Webstorm. https://www.jetbrains.com/webstorm, 2010. Visited: 2025-01-07.

[12] T. Jurlewicz and Z. Skoczylas. *Algebra liniowa 1 Definicje, twierdzenia, wzory. Wydanie 7 rozszerzone.* Oficyna Wydawnicza GiS, Wrocław, 2000.

[13] T. Jurlewicz and Z. Skoczylas. *Algebra liniowa 2 Definicje, twierdzenia, wzory. Wydanie 5 rozszerzone.* Oficyna Wydawnicza GiS, Wrocław, 2000.

[14] J. Kunimune. Plate carrée with tissot's indicatrices of distortion. https://en.wikipedia.org/wiki/Equirectangular_projection#/media/File:Plate_Carr%C3%A9e_with_Tissot's_Indicatrices_of_Distortion.svg, 2015. Visited: 2025-01-17.

[15] Canonical Ltd. Ubuntu 22.04.5 lts (jammy jellyfish). https://ubuntu.com, 2023. Visited: 2024-11-05.

[16] T. Marrinan. Cubemap to equirectangular converter. https://github.com/tmarrinan/cube2equirect, 2023-05-18. Visited:2025-01-20.

[17] S. Marschner and P. Shirley. *Fundamentals of computer graphics (4th ed).* CRC press, Taylor and Francis group, 2016.

[18] J. C. Prunier. Mathematics, physics for computer graphics. https://www.scratchapixel.com, 2015. Visited: 2025-01-04.

[19] Microsoft Corporation R. Brodie. Notepad. https://support.microsoft.com/en-us/windows, 1983. Visited: 2025-01-07.

[20] Joyent R. Dahl. Node.js. https://nodejs.org, 2009. Visited: 2024-12-05.

[21] P. Richardus and R. K. Adler. *Map Projections For Geodists, Cartographers and Geographers*. North-Holland Publishing Company, Amsterdam, 1972.

[22] J. P. Snyder. *Map Projections - A Working Manual*. United States Government Printing Office, Washington, D. C., 1987.

[23] J. Sumner and Contributors. Bun. https://bun.sh, 2022. Visited: 2024-12-05.

[24] E. You and Contributors. Vite. https://vitejs.dev, 2020. Visited: 2023-10-05.