Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Biblioteka do tworzenia grafiki trójwymiarowej w przeglądarce internetowej

Bartosz Rajczyk
Nr indeksu: 244928

Praca inżynierska napisana
pod kierunkiem
dr. inż. Marcina Zawady

Politechnika
Wrocławska

Wrocław 2020

Spis treści

Wstęp

Przeglądarki internetowe regularnie wzbogacają się o nowe funkcjonalności, a jako środowisko aplikacji nieustannie rosną na popularności, umożliwiając uruchomienie tych samych programów na komputerze, telefonie, telewizorze czy jakimkolwiek innym urządzeniu wyposażonym w jedną z nich. Programy uruchamiane w przeglądarkach robią się coraz bardziej złożone, przez co pojawia się zapotrzebowanie na narzędzia umożliwiające tworzenie wydajnej i zaawansowanej grafiki trójwymiarowej.

Praca ma na celu zaprojektowanie i implementację biblioteki, której główna funkcjonalność może być opisana właśnie w ten sposób. W tym celu potrzeba przeanalizować narzędzia udostępniane przez platformę docelową, zapoznać się z zasadami i matematyką stojącą za tworzeniem obrazu na ekranie komputera oraz zaplanować tworzony kod w ten sposób, aby był on odpowiednio uniwersalny i prosty w użyciu. Następnie w celu prezentacji możliwości stworzonej biblioteki potrzeba utworzyć kilka przykładów użycia.

Poza samym wyświetlaniem grafiki, utworzony kod ma także umożliwić korzystającemu z niego programiście kontrolowanie i edytowanie obiektów, które znajdują się na ekranie. Celem jest utworzenie wielopoziomowych abstrakcji, które obejmować będą zarówno podstawowe operacje, jak dodawanie nowych przedmiotów do wyświetlenia czy przesuwanie elementów, przez bardziej złożone, jak kontrola wielu kamer i podział na sceny, po takie o wysokim poziomie, jak interakcje ze strony użytkownika czy tworzenie animacji z odpowiednimi interpolacjami wartości pomiędzy klatkami kluczowymi. Ma to na celu zwiększenie uniwersalności biblioteki, która pozwoli tworzyć zarówno proste elementy dekoracyjne, jak i np. gry wideo.

W pierwszym rozdziale pracy zostaną opisane jej podstawy teoretyczne; w jaki sposób działa grafika trójwymiarowa na komputerze, jakie sposoby jej opisywania i realizowania zostały wybrane w tej pracy, jak reprezentowane są obiekty czy jak dokonuje się ich przekształceń. Rozdział drugi zostanie poświęcony środkom technicznym, które zostały wybrane do realizacji zadanych celów - zostaną porównane różne dostępne opcje oraz dokładnie opisane te, które wykorzystane zostaną w finalnej wersji tworzonego projektu. Trzeci rozdział obejmie szczegóły implementacyjne; strukturę programu, interakcje klas, zastosowane algorytmy. W czwartym rozdziale zaprezentuję efekty w postaci opisów przypadków użycia i przykładowych innych zastosowań. Ostatni rozdział zawierać będzie podsumowanie wraz z napotkanymi problemami oraz możliwymi poprawkami.

Analiza problemu

2.1 Grafika komputerowa

W szerokim rozumieniu grafika komputerowa jest wszystkim związanym z tworzeniem, modyfikowaniem, wyświetlaniem i przetwarzaniem treści wizualnych przy pomocy komputera. W definicji tej zawierają się pojęcia zarówno czysto artystyczne, jak sztuka cyfrowa, jak i techniczne, które to są tematem tej pracy. Głównym rozróżnieniem w tematyce pracy jest podział ze względu na sposób zapisywania informacji o obrazie - czyli grafika rastrowa oraz grafika wektorowa.

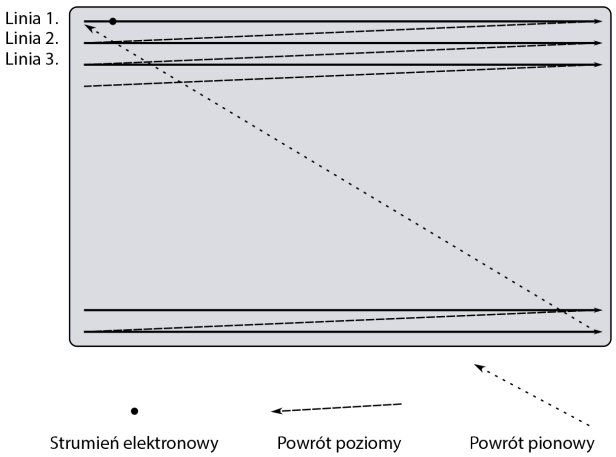
2.1.1 Grafika rastrowa

Zwana także bitmapową, jest współcześnie podstawowym typem grafiki dwuwymiarowej. Jej nazwa pochodzi z angielskiego słowa *raster*, pochodzącego z łacińskiego **rastrum**, oznaczającego grabie, używanego w kontekście *raster scan* - jest to metoda wytwarzania obrazu znana z monitorów kineskopowych, polegająca kolejnym rysowaniu poziomych linii [3]. Sposób ten opisuje dokładniej schemat 2.1.

Dzisiaj to pojęcie rozumie się jako grafikę, która reprezentowana jest przez macierz punktów obrazu, zawierającą krotki z informacją o kolorze czy przezroczystości każdego z nich. Każdy z tych punktów nazywany jest *pikselem*. Nazwa *bitmapa* związana jest z faktem, że każdy z pikseli potrzebuje jakiejś liczby bitów na zapisanie informacji o swoim kolorze.

Grafika tworzona w ten sposób charakteryzuje się określoną *rozdzielczością*, czyli liczbą pikseli pionowych oraz poziomych. Konsekwencją tego faktu jest to, że ma ona skończoną dokładność reprezentowanego obiektu - powiększanie, obracanie o kąty inne niż wielokrotności kąta prostego czy nawet pomniejszanie wiązać się będzie z utratą informacji i potrzebą interpolowania ich na podstawie sąsiadujących pikseli.

Rysunek 2.1: Rysowanie kolejnych poziomych linii w skanowaniu rastrowym



2.1.2 Grafika wektorowa

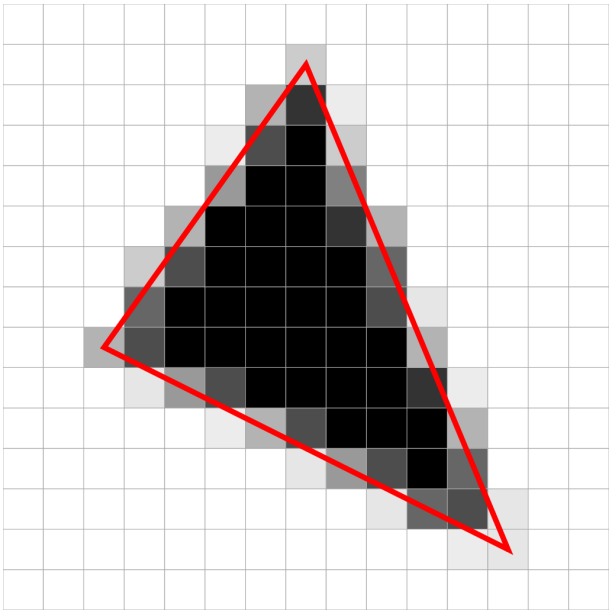
Różni się od grafiki rastrowej sposobem przechowywania informacji o obrazie; nie jest ona zbiorem dyskretnych punktów obrazu, a raczej matematycznym opisem obiektów, które się na nim znajdują. Grafika wektorowa składa się z linii, wielokątów, krzywych, elips oraz wielu innych rzeczy, które dają się zawrzeć w równaniu czy układzie równań. W ten sam sposób dodawana jest informacja o kolorach -

może być to jedna barwa opisująca cały obiekt, może być to gradient czy funkcja przypisująca jakiś kolor współrzędnym.

Grafika wektorowa do wyświetlenia zawsze wymaga *interpolacji* obrazu na podstawie podanych równań; chcąc wyświetlić linię idącą po przekątnej ekranu, należy obliczyć, jakie fragmenty obrazu należy aktywować. Proces ten nazywa się *rasteryzacją* - pojęcie to będzie ważne w kolejnych rozdziałach pracy. Przykładową rasteryzację obrazuje schemat 2.2.

Ze względu na charakterystykę grafiki wektorowej, umożliwia ona dowolne przekształcenia bez utraty na jakości; obiekty matematyczne pozwalają bowiem na użycie liczb dowolnej dokładności i dowolnego rzędu wielkości, zawsze zwracając precyzyjne dane - w takim wypadku limitem są jedynie dokładności reprezentacji liczb konkretnych implementacjach w komputerze.

Rysunek 2.2: Rasteryzacja trójkąta z *antyaliasingiem*



2.2 Reprezentacja obiektów trójwymiarowych

Najpowszechniejszym sposobem reprezentowania obiektów w grafice trójwymiarowej jest aproksymacja ich kształtu przy pomocy siatek wielokątów (najczęściej trójkątów). Metoda ta ma wiele zalet:

- **łatwość reprezentacji** - każdy trójkąt jest zbiorem trzech jego wierzchołków opisywanych przy pomocy trzech współrzędnych
- **łatwość przekształcania** - każdy trójkąt można łatwo poddać przekształceniom; wystarczy zastosować je do każdego z jego wierzchołków
- **łatwość rasteryzacji** - istnieją wydajne algorytmy umożliwiające szybką i równoległą rasteryzację trójkątów [6]; są one implementowane przez współczesne karty graficzne [4] [2]

2.3 Matematyka grafiki trójwymiarowej

2.3.1 Reprezentacja

Wiedząc już, że obiekty trójwymiarowe przedstawiane są przy pomocy zbiorów trójkątów, należy przeanalizować podstawy matematyczne takiego podejścia. Każdy trójkąt może zostać opisany przy pomocy trzech punktów będących jego wierzchołkami. Każdy z tych wierzchołków charakteryzuje się pozycją w przestrzeni trójwymiarowej, euklidesowej, będzie to więc trójka (x, y, z), taka że $x, y, z \in \mathbb{R}$, gdzie każda z liczb oznacza przesunięcie na jednej z płaszczyzn układu.

2.3.2 Przekształcenia

Podczas pracy z grafiką komputerową wyróżnia się kilka podstawowych przekształceń. Są nimi:

- **translacja** - przesunięcie punktu o wektor
- **rotacja** - obrót punktu wokół jakiejś osi o zadany kąt
- **skalowanie** - przeskalowanie współrzędnej punktu na zadanej osi o podany współczynnik

Każde ze zdefiniowanych przekształceń posiada swoją *macierz przekształceń*, która pomnożona przez wektor będący reprezentacją punktu poddawanemu temu przekształceniu, zwróci nam wektor wynikowy. Przed zdefiniowaniem tych macierzy, warto zaznaczyć, że w grafice komputerowej stosuje się *współrzędne jednorodne*. Polegają one na dodaniu dodatkowego wymiaru do reprezentacji punktów i wykorzystywanych macierzy; punkt (x, y, z) we współrzędnych jednorodnych ma postać $(x, y, z, 1)$. Jednocześnie umożliwia to odróżnienie punktów od wektorów; wektory w takiej reprezentacji mają czwartą współrzędną (nazywaną powszechnie w) równą 0.

Taka reprezentacja ma wiele zalet:

1. umożliwia łączenie wielu przekształceń w jedną uniwersalną macierz wykonującą jednocześnie translację, rotację oraz skalowanie współrzędnych punktu
2. umożliwia wykorzystanie macierzy projekcji, które służą do przedstawienia trójwymiarowych obiektów na płaszczyźnie w perspektywie; w przypadku grafiki komputerowej płaszczyzną tą jest oczywiście obszar ekranu, na który rzutowany jest realistyczny obraz manipulowanych obiektów

Translacja

Translacja punktu $(x, y, z, 1)$ o wektor $t = (t_x, t_y, t_z, 1)$ odbywa się przez mnożenie wektora $[x, y, z, 1]$ przez macierz translacji:

$$T_t = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ma to ten sam efekt, co dodanie wektorów t oraz v, ale posiada tę zaletę, że takie przekształcenie macierzowe umożliwia wymnożenie macierzy przekształceń z inną macierzą przekształceń, otrzymując w wyniku macierz osiągającą jednocześnie oba efekty.

Rotacja

Do rotacji służą trzy macierze obracające punkt wokół każdej z osi. Oznaczmy je R_x, R_y oraz R_z. Wyglądają one następująco:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta_x) & -sin(\theta_x) & 0 \\ 0 & sin(\theta_x) & cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} cos(\theta_y) & 0 & sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta_y) & 0 & cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} cos(\theta_z) & -sin(\theta_z) & 0 & 0 \\ sin(\theta_z) & cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

gdzie θ_x oznacza obrót o kąt θ wokół osi x i analogicznie.

Skalowanie

Macierz skalowania o współczynniki s_x, s_y i s_z na odpowiadających im osiach przestawia się następująco:

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Istnieje też prostsza macierz skalowania o współczynniki $s = s_x = s_y = s_z$ wykorzystująca dodatkowy wymiar współrzędnych jednorodnych.

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{s} \end{bmatrix}$$

2.3.3 Uniwersalna macierz przekształceń

Jak było już wspominane, podane macierze przekształceń można mnożyć, aby otrzymać kolejne macierze przekształceń realizujące kilka z nich jednocześnie. Dla przykładu, chcąc przesunąć punkt (x, y, z) o wektor $(1, 2, 3)$, a następnie obrócić go o kąt $\frac{\pi}{6}$ wokół osi z, możemy obliczyć najpierw

$$cos(\frac{\pi}{6}) = \frac{\sqrt{3}}{2}$$

$$sin(\frac{\pi}{6}) = \frac{1}{2}$$

by następnie obliczyć

$$\begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} - 1 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 & \sqrt{3} + \frac{1}{2} \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

następnie możemy użyć macierzy wynikowej i pierwotnego punktu z dodaną współrzędna jednorodną, aby obliczyć współrzędne punktu po obu przekształceniach

$$\begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} - 1 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 & \sqrt{3} + \frac{1}{2} \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

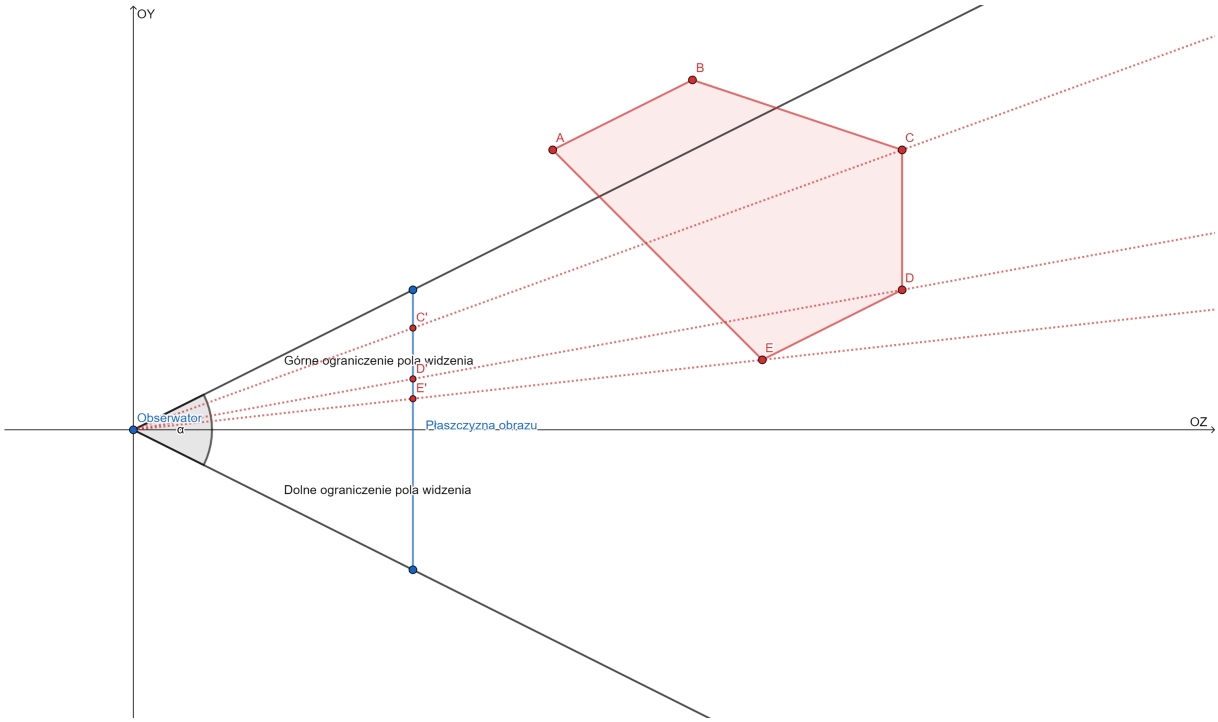
2.3.4 Perspektywa

By wyświetlić punkty na ekranie, potrzeba narzędzia, które pozwoli przekształcić pozycję punktu w trzech wymiarach na pozycję punktu na płaszczyźnie reprezentującej pole widzenia obserwatora. Wizualizuje to rysunek 2.3.

Schemat ten jest rzutem bocznym (widoczne są na nim dwie osie z trzech), reprezentującym przykładowy trójwymiarowy obiekt z zaznaczonymi wierzchołkami A, B, C, D oraz E. Niebieski odcinek to rzut boczny płaszczyzny obrazu; zaznaczone są na niej rzuty trzech widocznych wierzchołków obiektu, C', D' oraz E'. Wierzchołki A i B znajdują się poza zasięgiem wzroku obserwatora; obserwator posiada pionowy kąt widzenia α, oznaczony na schemacie dwiema półprostymi. Poziomy kąt widzenia jest zależny od pionowego kąta widzenia oraz stosunku szerokości i wysokości obrazu wynikowego. Stosunek ten nazywa się *formatem obrazu* i przyjęto dla niego oznaczenie fo.

Znalezienie współrzędnych punktu na płaszczyźnie obrazu znając jego współrzędne oraz kąt widzenia jest trywialne; wystarczy użyć podobieństwa tworzonych w ten sposób trójkątów prostokątnych. W tym momencie obliczeń należy jednak przerwać rozważania teoretyczne i wprowadzić do nich szczegóły implementacyjne.

Rysunek 2.3: Przykładowe działanie perspektywy



OpenGL (czy także *WebGL*) zostaną dokładnie omówione w kolejnych rozdziałach, teraz należy jedynie zaznaczyć, jakiej reprezentacji w przestrzeni oczekuje ten standard. Rysunek 2.4 obrazuje przestrzeń, którą rysuje OpenGL; jej nazwa to *clip space* i zawiera wartości $[-1, 1]$ na każdej osi (łącznie z osią "głębokości", OZ - mimo tego że obraz docelowo jest dwuwymiarowy, współrzędna z jest wykorzystywana do sprawdzania, który obiekt powinien zostać narysowany jeżeli kilka z nich posiada te same współrzędne pionową i poziomą).

Mając to na uwadze, można rozpatrzyć obliczenia związane z perspektywą dla przykładowego, pojedynczego punktu. Reprezentuje je rysunek 2.5.

Obserwator znajduje się zawsze w punkcie $(0, 0, 0)$ i skierowany jest w dodatnie z. Płaszczyzna obrazu zawsze ma wysokość 2, zaczynając się w 1 i kończąc w -1 - pokrywa ona więc *clip space*. Aby wykonać kolejne obliczenia, potrzebna jest znajomość wartości oznaczonej d, czyli odległości tejże płaszczyzny od obserwatora. Jest ona parametryzowana przez α, pionowy kąt widzenia obserwatora. Korzystając z trygonometrii, można zapisać:

$$tg(\frac{\alpha}{2}) = \frac{1}{d}$$

$$d = \frac{1}{tg(\frac{\alpha}{2})}$$

Składową y (oznaczoną E'_y) punktu E' można obliczyć korzystając z podobieństwa tworzonych przy pomocy E oraz E' trójkątów prostokątnych:

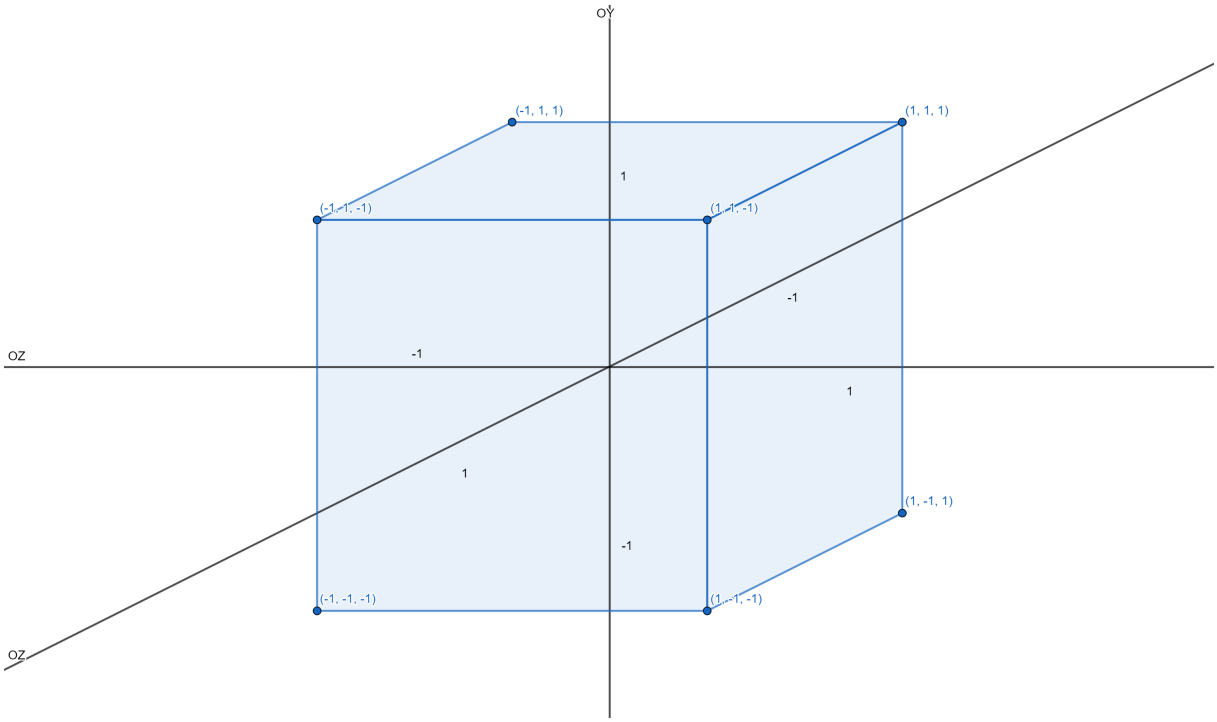
$$\frac{E'_y}{d} = \frac{E_y}{E_z}$$

$$E'_y = \frac{E_y \cdot d}{E_z}$$

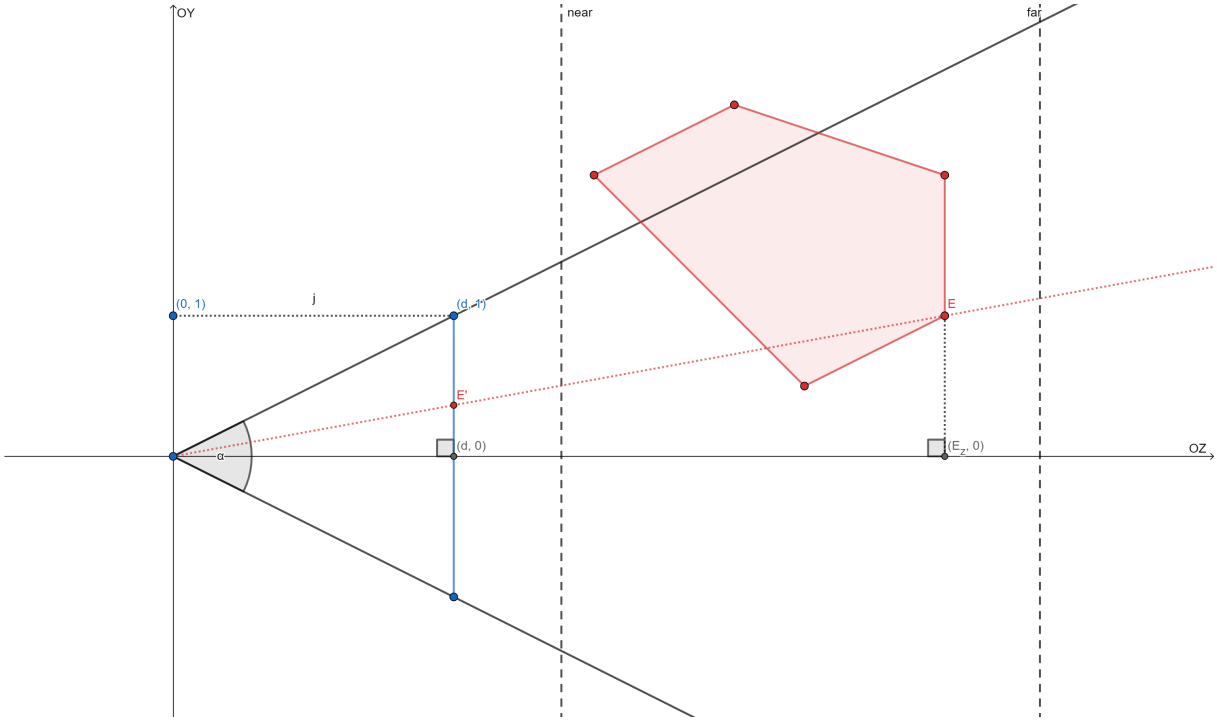
$$E'_y = \frac{E_y}{E_z \cdot tg(\frac{\alpha}{2})}$$

Analogiczna sytuacja zachodzi dla składowej x tego punktu; w tym wypadku należy jednak przywołać zdefiniowaną wcześniej wartość fo. Jeżeli obraz wynikowy ma wymiary R_x szerokości i R_y wysokości, to $fo = \frac{R_x}{R_y}$. Dodatkowo w poprzednim punkcie ustalono, że wysokość płaszczyzny obrazu, na którą

Rysunek 2.4: *Clip space* z OpenGL



Rysunek 2.5: Perspektywa jednego punktu w kontekście OpenGL



rzutowane są punkty, zawsze wynosi 2 i zawiera się w $[-1, 1]$. Aby uzyskać ten sam efekt dla składowej x, należy w takim wypadku dodatkowo podzielić jej równanie przez fo:

$$E'_x = \frac{E_x}{fo \cdot E_z \cdot tg(\frac{\alpha}{2})}$$

Na podstawie tych równań można otrzymać równania przeskalowujące współrzędne E_x i E_y wierzchołka w przestrzeni na współrzędne na płaszczyźnie obrazu. Analizując jednak to zagadnienie dokładniej, napotyka się na pewien problem. Każdy punkt jest reprezentowany jako wektor długości cztery; (E_x, E_y, E_z, E_w). Szukanym rozwiązaniem jest coś pokroju współczynników w równaniu:

$$a \cdot E_x + b \cdot E_y + c \cdot E_z + d \cdot E_w = \frac{E_x}{E_z \cdot tg(\frac{\alpha}{2})}$$

Nie jest niestety możliwe znalezienie takich wartości a oraz c, aby zachować równość obu stron. Rozwiązanie przyjęte przez OpenGL polega na porzuceniu dzielenia przez E_z na początku i zrobienie tego dopiero w kroku rasteryzacji. Jeżeli jednak wszystkie wartości zostaną podzielone przez E_z, to współrzędna z wszystkich punktów wynosić będzie $\frac{E_z}{E_z} = 1$. Aby temu zaradzić, budowana macierz musi kopiować wartość E_z dopiero po wymnożeniu.

Na podstawie aktualnie przeprowadzonego rozumowania, jej pierwsza wersja wyglądać będzie następująco:

$$\begin{bmatrix} \frac{1}{fo \cdot tg(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{tg(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Jest to macierz skalowania z obliczonymi wcześniej współczynnikami, kopiująca wartość E_z do wynikowego wektora poprzez jedynkę w ostatnim wierszu.

Ostatnią rzeczą, jaka pozostała do utworzenia w pełni funkcjonalnej macierzy perspektywy jest normalizacja współrzędnej z do $[-1, 1]$ przyjmowanego przez clip space. W tym celu na rysunku 2.5 zostały wyprowadzone płaszczyzny $near$ oraz far o współrzędnych $z = near$ oraz $z = far$. Są to ograniczenia głębokości obiektów, które będą rozpatrywane do rysowania. Do odnalezienia funkcji przekształcającej liniowo wartości z na $[-1, 1]$, musimy znaleźć parametry funkcji

$$f(z) = a \cdot z + b$$

pamiętając jednak, że wartości te zostaną podzielone przez z w kroku rasteryzacji

$$a + \frac{b}{z}$$

W celu odnalezienia wartości a, b należy rozwiązać układ równań:

$$a + \frac{b}{near} = -1$$

$$a + \frac{b}{far} = 1$$

Szukaną własnością jest bowiem to, aby obiekty na końcu pola widzenia posiadały najniższą wartość w clip space, a obiekty na początku - najwyższą. Po kilku przekształceniach otrzymujemy:

$$a = \frac{-(near + far)}{near - far}$$

$$b = \frac{2 \cdot near \cdot far}{near - far}$$

Są to współczynniki, które w równaniu $a \cdot z + b$ przerzucają wartości z z $[near, far]$ na $[-1, 1]$. Korzystając z faktu, że współrzędna w punktu ustawiona jest zawsze na 1 oraz mając na uwadze, że x oraz y mają nie wpływać na szukane przekształcenie, można uzupełnić macierz perspektywy w trzecim rzędzie:

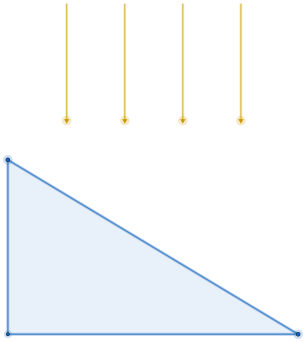
$$\begin{bmatrix} \frac{1}{fo \cdot tg(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{tg(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & \frac{-(near+far)}{near-far} & \frac{2 \cdot near \cdot far}{near-far} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Jest to jej finalna postać używana w implementacjach przekształceń perspektywy wedle standardu OpenGL.

2.3.5 Oświetlenie

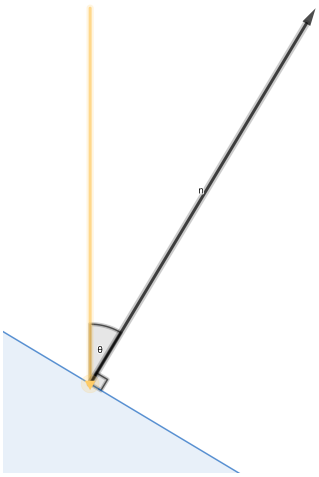
Projektowana biblioteka wspierać ma podstawowe oświetlenie obiektów. Prostym typem oświetlenia jest globalne oświetlenie kierunkowe rozproszone, symulujące np. słońce. Przykładowy rzut takiego oświetlenia reprezentuje rysunek 2.6.

Rysunek 2.6: Oświetlenie kierunkowe - na dole rzut oświetlanej figury, wyżej kierunek promieni światła



Szukanym efektem jest spadek jasności powierzchni wraz z odchylaniem się od kąta prostego względem płaszczyzny, na której leży. Wektor prostopadły do płaszczyzny nazywa się *wektorem normalnym*.

Rysunek 2.7: Wektor normalny i kąt pomiędzy nim oraz kierunkiem padania światła



Na rysunku 2.7 wektor n to wektor normalny niebieskiej płaszczyzny. Żółty wektor oznacza kierunek padania światła; kąt między nimi oznaczono θ. Ilość światła docierającą do obserwatora, jeżeli światło pada pod kątem θ względem wektora normalnego płaszczyzny, która je rozprasza, dana jest przez prawo Lamberta i wyraża się wzorem

$$I(\theta) = I_0 cos(\theta)$$

gdzie $I(\theta)$ to szukana wartość, a I_0 to oryginalna intensywność światła. Aby obliczyć tę wartość, należy przygotować wektor normalny rozpatrywanej płaszczyzny, normalizować go (do długości 1, tj. podzielić

każdą składową wektora przez jego długość), przygotować znormalizowany wektor przeciwny wektorowi kierunku światła i obliczyć ich iloczyn wektorowy. Wynikiem takiej operacji jest liczba z $[0, 1]$ oznaczająca intensywność tak odbitego światła.

Dostępne technologie

3.1 Przeglądarki internetowe

Platformą docelową prezentowanego rozwiązania są przeglądarki internetowe. Są to programy umożliwiające użytkownikowi dostęp do treści internetowych oraz wchodzenie z nimi w interakcje, powszechnie wspierające rekomendowane przez *W3C* (*World Wide Web Consortium*, organizację standaryzującą dla takich zagadnień jak np. HTML, CSS, DOM czy XML) czy *ECMA* (tworząca standard ECMAScript, z którym zgodny jest Javascript) standardy. Pośród zalet tego wyboru można wymienić:

- dostępność na wielu urządzeniach, od komputerów stacjonarnych po smartphone'y
- duża społeczność programistów[5]
- wysokie zapotrzebowanie rynkowe na aplikacje webowe[5]

3.2 Kod programu

Współcześnie przeglądarki internetowe nie oferują dużego wyboru w kwestii wyboru technologii, których można użyć do programowania. Jedynym powszechnie wspieranym językiem jest Javascript [10].

3.2.1 JavaScript

Skracany *JS*, Jest to skryptowy, dynamicznie i słabo typowany, język wysokiego poziomu, umożliwiający programowanie obiektowe, proceduralne oraz funkcyjne. Każda przeglądarka internetowa posiada silnik języka, który zajmuje się uruchamianiem dołączanego do niej kodu; pośród różnych implementacji można wymienić V8 używany przez przeglądarki oparte na Chromium, SpiderMonkey używany przez Mozilla Firefox oraz JavaScriptCore używany przez Safari. Pierwotnie kod języka był zaledwie interpretowany, lecz współczesne silniki kompilują go *JIT* (*just-in-time*, to jest bezpośrednio przed jego wykonaniem), co pozytywnie wpływa na prędkość jego wykonywania [8].

Kod źródłowy 3.1 Przykładowy kod programu w JavaScript korzystający z globalnego obiektu Math dostępnego w powszechnie wykorzystywanych środowiskach uruchomieniowych

```
1  class Point {
2      constructor(x, y) {
3          this.x = x;
4          this.y = y;
5      }
6
7      distance(point) {
8          return Math.sqrt((point.x - this.x)**2 + (point.y - this.y)**2);
9      }
10 }
```

3.2.2 TypeScript

Z punktu widzenia wytwarzania oprogramowania, Javascript jako standard posiada wiele wad. Najpowszechniejszą z nich jest przyjęty system typów, który pozwala na dynamiczne zmienianie typu zmiennej, porównywanie zmiennych dowolnych typów wykonując niejawne konwersje. Kod w JS nie ma także

możliwości posiadania żadnych *type hints* (“wskazań co do typów”), tak jak ma to miejsce np. Pythonie >=3.5.

Powstało wiele projektów mających adresować te problemy; dziś najpopularniejszym z nich jest TypeScript, skracany TS. Jego głównym wyróżnikiem jest dodatnie możliwości zaawansowanego typowania wykorzystywanych zmiennych. Składniowo jest nadzbiorem JS; każdy program napisany w JS jest także poprawnym programem w TS, patrząc jednak z perspektywy praktycznej - dodając typowanie do każdego poprawnego programu w JS, nie wszystkie staną się poprawnymi programami w TS - czyni go to podzbiorem.

Kod napisany w TS jest poddawany kompilacji (konkretniej transpilacji) do JS, który dopiero potem wykonywany jest przez przeglądarkę. Pozwala to zachować pełną zgodność z istniejącym standardem, dodatkowo dodając wiele narzędzi przydatnych w tworzeniu oprogramowania, takich jak:

- specyfikatory dostępu w polach klas
- system podziału kodu źródłowego na wiele plików
- typy generyczne
- typu *enum*
- interfejsy

Te zalety spowodowały, że Typescript został wybrany jako język do implementacji biblioteki będącej tematem pracy.

Kod źródłowy 3.2 Kod 3.1 przepisany do TS

```
1  class Point {
2      constructor(
3          public x: number,
4          public y: number
5      ) { }
6
7      public distance(point: Point): number {
8          return Math.sqrt((point.x - this.x)**2 + (point.y - this.y)**2);
9      }
10 }
```

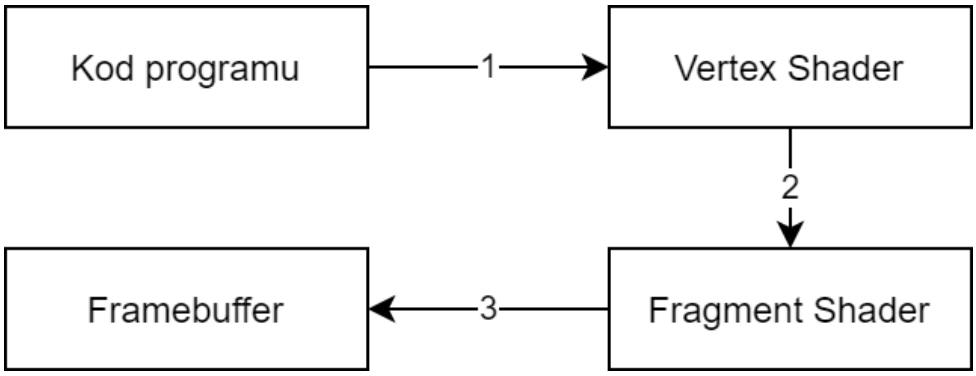
3.2.3 Budowanie biblioteki

Przed użyciem w przeglądarce, biblioteka musi być każdorazowo kompilowana do kodu w JS. Sam proces kompilacji wykonywany jest przez *TSC*, *TypeScript Compiler*, czyli po prostu kompilator TS. Aby umożliwić wygodną pracę z kodem, całe repozytorium jest przygotowane jako projekt *npm*, *Node Packet Manager*. Node (właściwie Node.js) to środowisko uruchomieniowe JS uruchamiane poza przeglądarką internetową, npm to menadżer zależności dla Node.js, umożliwiający tworzenie plików konfiguracyjnych projektów, zawierających listy zależności, skrypty uruchomieniowe oraz podstawowe informacje o projekcie. Do przygotowywania pojedynczej *paczki* (pliku zawierającego cały wynikowy kod źródłowy) kodu użyto narzędzia webpack. Poza tworzeniem paczek, narzędzie to umożliwia także optymalizację oraz minifikację kodu.

3.3 Grafika w przeglądarkach

Przeglądarki internetowe umożliwiają tworzenie grafiki komputerowej na wielu poziomach; od podstawowych transformacji obrazów i dokumentów SVG, przez rysowanie przy pomocy *Canvas API*, po niskopoziomowe, akcelerowane sprzętowo (czyli korzystające z karty graficznej komputera) rozwiązanie nazywane *WebGL* i będące przedmiotem tej implementacji.

Rysunek 3.1: Schemat działania programu w WebGL



3.3.1 Canvas

Umożliwia tworzenie nieskomplikowanych rysunków przy pomocy prostego API przeglądarkowego. Przy pomocy Canvas można rysować między innymi linie, okręgi, elipsy, wielokąty, kwadraty oraz gotowe obrazy. Ze względu na charakterystykę tworzenia obrazu oraz sposób działania odbiegający technicznie od powszechnego sposobu tworzenia grafiki przy pomocy kart graficznych, Canvas nie jest najlepszym wyborem w przypadku tworzenia zaawansowanej grafiki komputerowej.

3.3.2 WebGL

To tak naprawdę adaptacja API OpenGL ES 2 do użycia w przeglądarce. Pozwala ono na bezpośrednią interakcje z procesorem graficznym (GPU) komputera, w tym przygotowywanie *shaderów* (prostych programów komputerowych uruchamianych na GPU, które odpowiedzialne są za wybranie m.in. pozycji i kolory rysowanych pikseli) oraz ładowanie danych do pamięci graficznej. API to jest niskopoziomowe, to znaczy że opiera się na prawie atomowych wywołaniach odpowiedzialnych np. za tworzenie buforów, ustawianie wskaźników czy rysowanie odpowiedniej liczby figur. W tym sensie WebGL nie umożliwia tworzenie nam gotowej grafiki dwuwymiarowej czy trójwymiarowej; jest to jedynie wysokowydajny pipeline rasteryzacji.

3.4 Zasada działania WebGL

Przed samą czynnością rysowania przez WebGL musimy przygotować *program* WebGL składający się z *shaderów* - *Vertex Shadera* oraz *Fragment Shadera*. Ich zadania opisane zostaną później. Przy pomocy kilku wywołań w API kompiluje się oba shadery, a następnie tworzy przy ich pomocy program. Istnieje możliwość tworzenia wielu programów opartych na wielu różnych shaderach; shadery także mogą być tworzone dynamicznie przy pomocy prostej konkatenacji ciągów znaków lub podmienianiu zmiennych, shadery potem dostarczane są do kompilacji w formie zwykłego ciągu znaków w kodzie JS.

Każde narysowanie obiektu przy użyciu WebGL składa się z kilku kroków. Bardzo uproszczone działanie przedstawia diagram 3.1.

Elementy tego diagramu to:

- **kod programu** - kod napisany w JS przy użyciu API WebGL. Odpowiada on między innymi za:
  - przygotowywanie *atrybutów*, czyli danych w postaci tablic o pozycjach wierzchołków, kolorach i wektorach normalnych rysowanych obiektów w formie buforów WebGL i ładowanie ich do pamięci graficznej
  - przygotowywanie *uniformów*, czyli wartości stałych podczas rysowania obiektu; są to między innymi macierze transformacji, macierze kamery i perspektywy czy informacje o globalnym oświetleniu
  - tworzenie macierzy przekształceń dla obiektów
  - całą logikę odpowiedzialną za strukturyzowanie programu; ustawianie danych o obiektach, tworzenie nowych obiektów, usuwanie obiektów

– obliczenia związane z animacjami, interakcje z użytkownikiem

- **przejście numer 1.** - załadowanie do pamięci graficznej wszystkich atrybutów oraz uniformów, które wykorzystywane są w tym wywołaniu funkcji `drawArrays` odpowiedzialnej za uruchomienie procesu rysowania
- **Vertex Shader**, czyli "shader wierzchołka" - odpowiedzialny za przekształcenia związane z pojedynczym wierzchołkiem rysowanego obiektu; dane w buforach atrybutów ładowane są w sposób liniowy, a następnie wczytywane w grupach (np. w przypadku atrybutu pozycji - po cztery, jako cztery współrzędne pojedynczego punktu reprezentującego pozycję we współrzędnych jednorodnych). Vertex Shader uruchamiany jest dla każdej takiej grupy. Zajmuje się on obliczaniem pozycji każdego wierzchołka przy pomocy wykonywania mnożeń z macierzami przekształceń i ustawienia obliczanej pozycji w zmiennej `gl_Position`, oraz ustawaniem zmiennych *varying* dla każdego wierzchołka, które przekazywane są do Fragment Shadera
- **przejście numer 2.** - rasteryzacja; WebGL może rysować trzy typy *prymitywów*, czyli podstawowych obiektów: punkty, linie oraz trójkąty. Typ prymitywu rysowanego w aktualnym przejściu ustawiany jest jako argument funkcji `drawArrays`. Rasteryzacja zwraca piksele będące częścią aktualnie rysowanego prymitywu
- **Fragment Shader** - uruchamiany dla każdego piksela uzyskanego w wyniku rasteryzacji w poprzednim kroku. Przy pomocy danych dostarczonych przez zmienne *varying* oraz informacje o prymitywach z poprzedniego kroku, jego rolą jest interpolacja koloru każdego z pikseli i ustawienie otrzymanej wartości do zmiennej `gl_FragColor`
- **przejście numer 3.** - ostatnie przetwarzanie przed utworzeniem gotowej klatki obrazu, chociażby testy głębokości pikseli, czyli ustalanie, który z pikseli na tych samych współrzędnych powinien zostać ostatecznie wyświetlony
- **Framebuffer** - bufor ramki, zajmuje się przechowywaniem informacji o gotowym obrazie wygenerowanym przez GPU przed jego wyświetleniem

Ten ciąg wywołań dotyczy każdego rysowanego obiektu; chcąc narysować wiele obiektów równocześnie, musimy wykonać go wielokrotnie. To sprawia, że rysowanie wielu mało skomplikowanych (to znaczy posiadających mniej wierzchołków) obiektów jest wolniejsze od rysowania jednego obiektu bardziej skomplikowanego (posiadającego wiele wierzchołków).

Implementacja biblioteki

4.1 Zarys koncepcyjny

Biblioteka miała umożliwiać tworzenie prostej grafiki trójwymiarowej w przeglądarce, jednocześnie będą funkcjonalną i umożliwiającą manipulację obiektami trójwymiarowymi przy pomocy możliwie najmniejszej ilości kodu pisanego przez jej użytkownika.

4.1.1 Cele

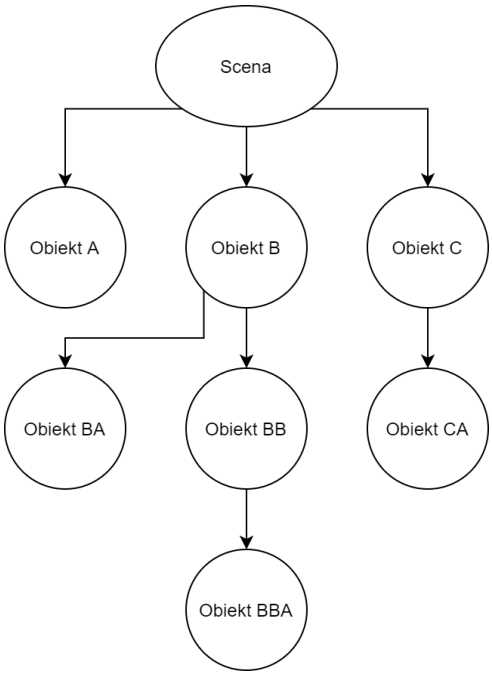
Poszukiwanymi własnościami biblioteki w kontekście graficznym są:

- **podział na sceny** - czyli osobne konteksty zawierające różne obiekty trójwymiarowe, umożliwiające przełączanie się między nimi bez usuwania aktualnie wyświetlanych obiektów i dodawania nowych
- **scene graph** ("drzewo sceny", konkretnej "drzewo obiektów sceny") - graf (konkretnej drzewo) zależności obiektów i ich przekształceń. Każdy z obiektów w takim drzewie posiada swojego rodzica oraz listę potomków (także pustą). Przekształcenia aplikowane do obiektu w drzewie aplikowane są w przestrzeni *lokalnej*, która jednak przekształcana jest także przez przestrzeń lokalną rodzica obiektu i wynikowe przekształcenie jest dopiero przekształceniem w przestrzeni *globalnej* aplikacji. Obiekty będące bezpośrednimi potomkami obiektu sceny w takim drzewie jako jedyne mają tożsame przestrzenie lokalne oraz globalne. Przykładowy scene graph prezentuje rysunek 4.1
- **kamery** - podstawowo silniki graficzne rysują rzeczy tak, jakby perspektywa obserwatora znajdowała się w ćentrum świata"(zwykle na koordynatach $(0, 0, 0)$), a następnie przesuwają je, by nadać wrażenie ruchu kamery. Projektowany silnik graficzny ma posiadać możliwość przełączania się pomiędzy wieloma różnymi kamerami w scenie i oglądania obrazu z ich perspektywy
- **oświetlenie globalne** - globalne światło rozproszone umożliwia lepsze oddanie głębi i kształtu obiektów
- **wczytywanie modeli z plików .obj** - *.obj* to prosty format tekstowy zawierający opis wierzchołków, koordynatów tekstur oraz opcjonalnie wektorów normalnych modeli trójwymiarowych. Dzięki temu obiekt utworzony w oprogramowaniu do modelowania trójwymiarowego będzie można wyświetlić w bibliotece

Natomiast poszukiwanymi charakterystykami tworzonego interfejsu programistycznego biblioteki są:

- **prostota użycia** - użytkownik biblioteki powinien być w stanie zorientować się, jak działa system i zrealizować jego najprostsze funkcjonalności przy użyciu kilku wywołań funkcji
- **precyzyjne otypowanie** - kod powinien zapewniać dokładne informacje o używanych typach, aby umożliwić wygodny rozwój oprogramowania bez żadnych nieścisłości
- **wielopoziomowość** - możliwość użycia biblioteki w wielu kontekstach; zarówno jako gotowego rozwiązania z zestawem oferowanych funkcjonalności, jak i pojedynczych jej elementów
- **rozszerzalność** - elementy biblioteki powinny realizować tylko podstawowe zadania, które potem mogą zostać rozszerzone do własnych potrzeb przed programistę do konkretnego zastosowania
- **wydajność** - biblioteka powinna być tworzona mając na uwadze złożoność obliczeniową oraz narzuty pamięciowe używanych struktur danych i algorytmów tak, aby osiągnąć możliwie dużą szybkość wykonywanego kodu

Rysunek 4.1: Diagram reprezentujący przykładowy *scene graph*; wszystkie przekształenia obiektu B są aplikowane także na obiekty BA, BB i BBA; przekształcenia obiektu BB są aplikowane na obiekt BBA



4.1.2 Założenia programistyczne

TypeScript nie narzuca paradygmatu programowania; do projektu wybrano **programowanie obiektowe**. Umożliwia on organizację kodu odpowiedzialnego za konkretne funkcjonalności w klasach oraz tworzenie abstrakcji przy pomocy interfejsów. Jednocześnie duża elastyczność TS umożliwiła wydzieleni czystych funkcji odpowiedzialnych np. za tworzenie podstawowych konfiguracji czy przygotowywanie wywołań z API WebGL na podstawie danych shaderów do osobnych plików, bez tworzenia pustych klas ze statycznymi polami.

4.2 Omówienie architektury

Biblioteka została zaprojektowana z poszanowaniem dobrych praktyk programowania obiektowego, w szczególności zasad *SOLID*. W tej sekcji omówione są klasy przedstawione na diagramie relacji z rysunku 4.2.

S3e

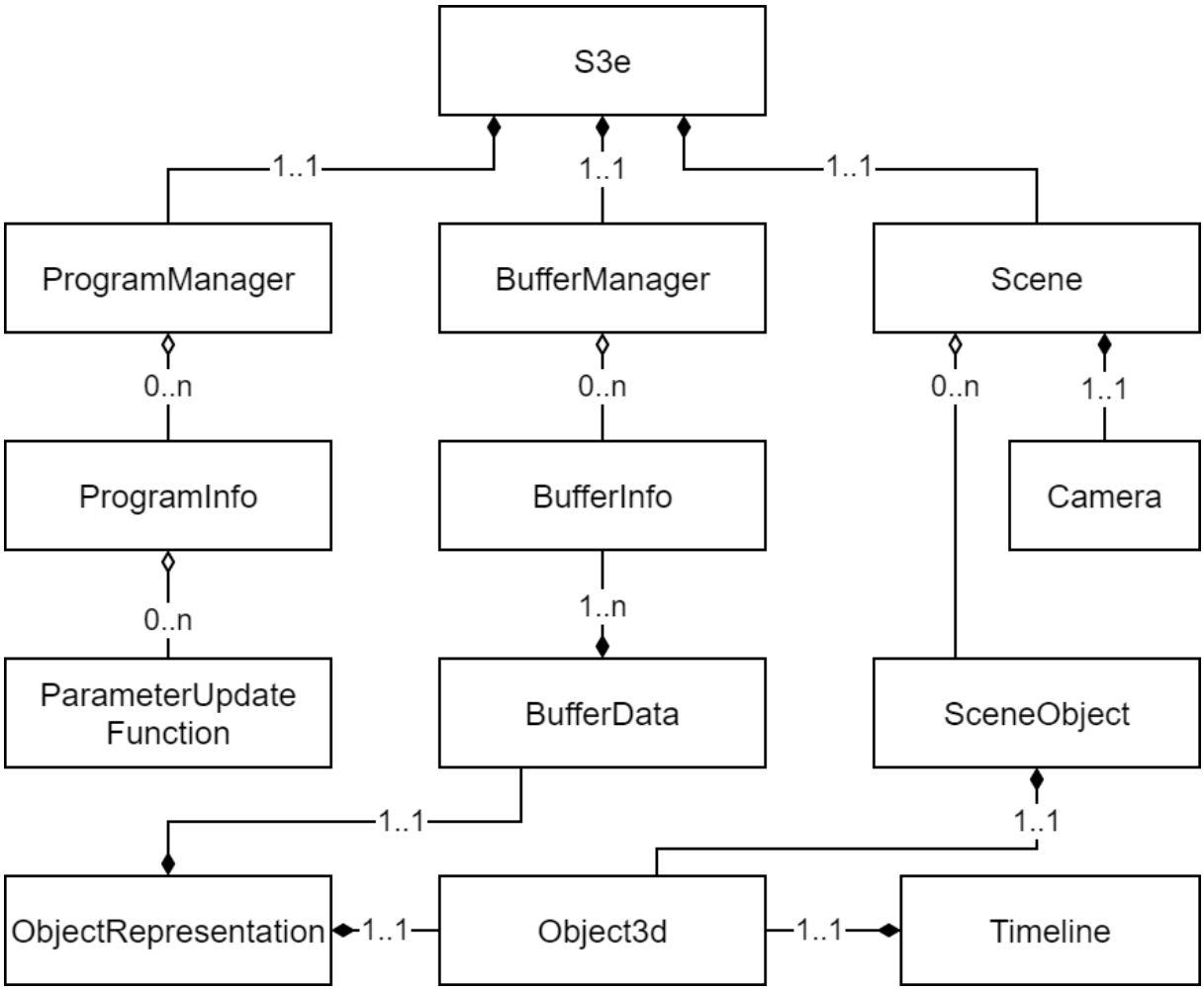
Główna klasa reprezentująca silnik graficzny. Zarządza ona technicznym procesem rysowania obrazu; na podstawie danych ze Scene i wywołań metod z BindingsManager umożliwia ona narysowanie pojedynczej klatki obrazu. Zmiana aktualnie rysowanej sceny następuje przez zmianę referencji pola `currentScene`. Jako argumenty konstruktora przyjmuje obiekt HTML elementu `canvas`, na którym silnik ma rysować.

W silniku wykonywana jest główna pętla rysująca - korzysta ona z danych o elementach zawartych w aktualnej scenie oraz z klasy ProgramManager, aby uzyskać odpowiedni program z informacjami.

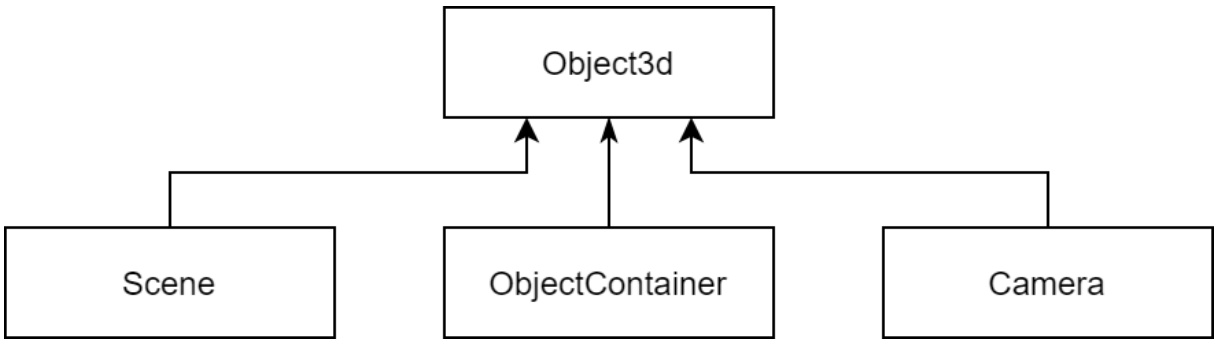
ProgramManager

Dosłownie menadżer programów. Programy to skompilowane i gotowe do użycia pary vertex shaderów oraz fragment shaderów. Klasa ta odpowiada za przygotowywanie interakcji z API WebGL w programach o różnych funkcjonalnościach, co przedstawia kod 4.1, oraz tworzenie informacji o programie dla silnika. Program identyfikowany jest poprzez maskę bitową, w której kolejne bity opisują kolejne funkcje (dla przykładu - kolory to `0x1`). Jeżeli jakieś funkcje nie są potrzebne, program przygotowywany jest bez nich, co przekłada się na lepszą wydajność biblioteki.

Rysunek 4.2: Diagram relacji



Rysunek 4.3: Diagram zależności



Kod źródłowy 4.1 Wycinek kodu odpowiedzialnego za tworzenie wywołania funkcji, które aktualizuje wartość atrybutu w programie WebGL. Obiekt `mappings` zawiera nazwy typów z przypisanymi im rozmiarami. Linia 16. pokazuje zwracanie funkcji, która następnie wywołuje kilka innych, realizujących aktualizację zawartości bufora. Dzięki temu wiele skomplikowanych wywołań można uprościć do jednego.

```
1  const createAttributeUpdateCall = (
2    signature: DataSignature,
3    gl: WebGLRenderingContext,
4    customSize?: number,
5    normalize = false,
6    stride = 0,
7    offset = 0
8  ): DataUpdateCall<WebGLBuffer> => {
9    const mapping = typeMappings[signature.type];

11   if (mapping === undefined)
12     throw new Error(`Cannot create a binding call for ${signature.type} type`);

14   const size = customSize ?? mapping.size;

16   return (newValue) => {
17     gl.enableVertexAttribArray(signature.location as number);
18     gl.bindBuffer(gl.ARRAY_BUFFER, newValue);
19     gl.vertexAttribPointer(
20       signature.location as number,
21       size,
22       gl.FLOAT,
23       normalize,
24       stride,
25       offset
26     );
27   };
28 };
```

ProgramInfo

Obiekt opisujący pojedyczny program. Zawiera informację o masce bitowej jego funkcjonalności oraz agreguje funkcje aktualizujące atrybuty i uniformy w WebGL, które potem wywoływane są podczas rysowania kolejnych obiektów.

Scene

Obiekt pojedynczej sceny. Scena posiada pojedynczą aktywną kamerę ustawioną w polu `currentCamera`, wartość siły *ambient light*, czyli podstawowego, globalnego oświetlenia bez kierunku oraz kierunek *directional light*, globalnego rozproszonego światła kierunkowego. Jak wynika z rysunku 4.3, dziedziczy z klasy Object3d. Wynika to z tego, że scena sama w sobie jest także obiektem trójwymiarowym ze swoją macierzą przekształceń, która umożliwia przemieszczanie wszystkich znajdujących się na niej obiektów, czyli zgodnie z rysunkiem 4.1 - jej potomków.

Poza pełnieniem w drzewie sceny teoretycznej roli korzenia, obiekt sceny odgrywa także bardziej przyziemną rolę przygotowywania danych do rysowania obiektów. Scena podczas dołączania do niej potomka dodaje utworzony SceneObject do tablicy `elements`.

Tablica `elements` jest ważna z punktu widzenia optymalizacji programu; podczas rysowania pojedynczej sceny byłoby możliwe odwiedzenie wszystkich znajdujących się w niej potomków używając po prostu algorytmu przechodzącego drzewo i w teorii miałoby to tę samą złożoność obliczeniową, $O(n)$, gdzie n to liczba wierzchołków w drzewie. Układając jednak wszystkie obiekty sceny w tablicy, niektóre silniki JS (jak na przykład najbardziej popularny, V8 [7]) mogą wykonywać optymalizacje mające na celu szybszy dostęp do ich elementów (jak chociażby optymalizacje pod cache procesora).

Dodatkową optymalizacją jest także sortowanie tej tablicy wedle masek bitowych funkcjonalności obiektów oraz następnie hashów buforów obiektów. Proces ten jest opisany dokładniej w dalszej części pracy.

SceneObject

Jest reprezentacją abstrakcyjnego Object3d w konkretnym kontekście; zawiera obiekt i informację, czy ma być on w ogóle rysowany przy pomocy silnika (dla przykładu kamery są całkowicie pomijane podczas rysowania).

Object3d

Bazowa klasa reprezentująca pojedynczy obiekt wyświetlany na ekranie. Posiada ona trzy pola reprezentujące jej najważniejsze własności, czyli pozycję, rotację oraz skalę, każde rozbite na trzy współrzędne. Obiekty posiadają też pole zwracające ich aktualną macierz przekształceń oraz macierz normalną dla ich macierzy przekształceń. Posiadają referencję na scenę, w której się znajdują, swojego rodzica w drzewie sceny oraz listę swoich potomków. Object3d stanowi w tym rozumieniu abstrakcyjną reprezentację obiektu; jest to tak naprawdę lista własności z funkcjami umożliwiającymi ich modyfikowanie oraz generowanie odpowiednich macierzy na ich podstawie. Polem które definiuje właściwy kształt rysowany przez silnik jest `ObjectRepresentation`.

ObjectRepresentation

Opisuje cechy "fizyczne" obiektu; posiada instancję `BufferData` oraz informację o używanych przez ten obiekt funkcjonalnościach silnika w postaci maski binarnej w polu `featuresMask`. Dodatkowo definiuje "skalę podstawową" obiektu; jej zastosowanie opisane jest w dalszej części pracy.

Z funckjonalności dodatkowych, klasa ta przechowuje także informacje o teksturach wykorzysytwanych przez obiekt.

BufferData

Jest to słownik z kluczami będącymi ciągami znaków i wartościami w postaci `BufferInfo`. Każdy wpis w takim słowniku opisuje pojedynczy bufor używany przez obiekt; buforami mogą być np. dane o pozycjach wierzchołków, o kolorach czy o wektorach normalnych.

BufferInfo

Zawiera informacje dotyczące pojedynczego bufora WebGL. Są nimi referencja na sam bufor, *hash* identyfikujący ten bufor, liczba elementów bufora do pobierania na każde wywołanie vertex shadera oraz długość bufora.

BuffersManager

Czyli dosłownie "menadżer buforów", bowiem nazwa dobrze określa jego funkcjonalność. Jest to klasa odpowiedzialna za zarządzanie buforami w silniku; alokowanie nowych buforów jest kosztowne, ponieważ składa się przesyłania danych pomiędzy silnikiem wykonaniowym przeglądarki a pamięcią GPU, a jeżeli silnik zawiera wiele obiektów o identycznym kształcie, nie ma potrzeby tworzyć dla nich osobnych buforów. Menadżer buforów sprawdza, czy odpowiedni bufor już istnieje i jeżeli tak, to przydziela go obiektowi; w przeciwnym wypadku zostaje on dopiero zaalokowany. Dodatkowo menadżer liczy, ile obiektów korzysta z bufora i jeżeli liczba ta spadnie do zera, bufor jest automatycznie dealokowany.

Camera

Reprezentacja pojedynczej kamery, czyli punktu z którego może oglądana być scena. Kamera dziedziczy po Object3d, jak to przedstawia rysunek 4.3, ponieważ dzieli z obiektem wiele własności, takich jak pozycja czy rotacja. Kamera konfigurowalna jest w zakresie wartości kąta widzenia, formatu obrazu oraz wartości $near$ i far wspomnianych w poprzednich rozdziałach pracy. Obiekt ten nie posiada nigdy swojej reprezentacji trójwymiarowej; podczas jego tworzenia ustawiana jest ona na pustą, kamery są też całkowicie pomijane podczas renderowania sceny. Jeżeli jednak użytkownik chciałby reprezentować kamerę przez obiekt w scenie lub uzależnić od jej ruchu ruch innego elementu, kamery mogą tak jak każdy inny obiekt posiadać swoich potomków w drzewie sceny.

ObjectContainer

Obiekt, którego jedyną rolą jest posiadanie potomków w drzewie sceny - nie może posiadać swojej reprezentacji, gdyż podczas konstruowania jest ona ustawiana na pustą, a podczas rysowania sceny obiekt ten jest pomijany. Umożliwia jednak grupowanie obiektów i kontrolowanie ich własności (pozycji, rotacji, skali) niczym pojedynczej instancji.

Timeline

Czyli w tłumaczeniu "oś czasu", służy do reprezentacji zmian własności obiektu w czasie. Oś czasu ma określoną liczbę klatek na sekundę (*fps - frames per second*), będącymi dyskretnymi, równomiernie rozmieszczonymi i najmniejszymi w kontekście osi okresami czasu, i umożliwia precyzyjne kontrolowanie w czasie wartości jednej własności obiektu - pozycji, rotacji lub skalowania - poprzez użycie klatek kluczowych. Każda klatka kluczowa to krotka wartość - numer klatki - sposób interpolacji. Może istnieć wiele osi czasu kontrolujących różne własności. Obiekt osi czasu pozwala na wywołanie metody `evaluateAt`, która ustawia kontrolowaną przez nią własność do wartości interpolowanej na podstawie klatek kluczowych ustawionych w podanym czasie.

4.2.1 Wybrane detale implementacyjne

Ta sekcja ma na celu przybliżenie technicznych aspektów ważnych elementów biblioteki.

Obiekty w scenie

Biblioteka była projektowana z myślą o wykorzystywaniu jej w animacjach w czasie rzeczywistym. Istniejące w tym celu API przeglądarkowe nazywa się Window.requestAnimationFrame, czyli w dosłownym tłumaczeniu "zażądaj klatki animacji". Funkcja ta przyjmuje jeden argument będący *callbackiem* (czyli inną funkcją) wykonywanym kiedy przeglądarka gotowa jest narysować kolejna klatkę obrazu - rekomendacje sugerują, aby była to liczba odpowiedzialna częstotliwości odświeżania ekranu [9]. W ten sposób można wywoływać logikę odpowiedzialną za tworzenie obrazu wtedy i tylko wtedy, kiedy przeglądarka gotowa jest to obsłużyć.

Wartości obiektów zmieniać się jednak mogą pomiędzy tymi oknami renderowania dowolną liczbę razy, zarówno z wejść od użytkownika, jak i na przykład używanych animacji. To wymaga wypracowania systemu, który ogranicza najbardziej czasochłonne operacje do minimum. W tym wypadku takimi operacjami jest tworzenie macierzy przekształceń dla każdego obiektu; mnożenie dwóch macierzy 4×4 reprezentujących przekształcenia to 64 mnożenia ich elementów oraz 48 dodawań.

Rozwiązaniem tego problemu są dwa elementy:

1. śledzenie zmian, czyli sprawdzanie, czy jakaś własność obiektu zmieniła się od ostatniego rysowania
2. generowanie nowej macierzy przekształceń dopiero po wykryciu zmian obiektu

Punkt pierwszy został zrealizowany w sposób prezentowany w kodzie 4.2. Istnieje zmienna `changed`, która ustawiana jest na wartość dodatnią, kiedy któraś z wartości obiektu `position` zostanie zmieniona. Właściwa wartość pozycji jest ukryta przed użytkownikiem jako pole prywatne klasy, aktualizowane przez zdefiniowany *setter* (funkcję wyglądającą "z zewnątrz" jak pole klasy, wykonywaną przy przypisaniu wartości do pola o jej nazwie).

Kod źródłowy 4.2 Kod odpowiedzialny za utworzenie obiektu z *getterami* i *setterami* zmieniającymi właściwe wartości i zapamiętujące informację o zmianach. Kod jest częścią definicji klasy Object3d.

```
1  private _pos: Position3d = { x: 0, y: 0, z: 0 };

3  public changed = false;
4  public updatedByChild = true;

6  public readonly position: Position3d = Object.defineProperties(
7    {},
8    {
9      x: {
10       get: () => this._pos.x,
11       set: (value: number) => {
```

```
12          this._pos.x = value;
13          this.changed = true;
14          this.updatedByChild = false;
15        },
16      },
17      y: {
18          // ...analogicznie
19      },
20      z: {
21          // ...analogicznie
22      },
23    }
24 );
```

Drugą część tego założenia realizuje kod przedstawiony w 4.3. Pierwsza linijka to chronione pole klasy reprezentujące alokację aktualnej macierzy przekształceń, zainicjalizowaną do macierzy równoważności. Pole `absoluteMatrix` jest *getterem* odpowiedzialnym za sprawdzenie, czy obiekt lub jego rodzic uległ zmianie; jeżeli tak, tworzona jest nowa macierz przekształceń na podstawie aktualnych własności obiektu; w przeciwnym wypadku krok ten jest pomijany, na koniec zwracana jest macierz przekształceń. Funkcje postaci `move` czy `rotateX` to funkcje czyste wykonujące mnożenie macierzy podanej jako pierwszy argument z macierzą przekształceń utworzoną na podstawie kolejnych argumentów. Ich ostatnim argumentem jest cel operacji, czyli macierz, do której ma być zapisany wynik mnożenia. Kod dodatkowo sprawdza, czy każde z przekształceń jest w ogóle potrzebne; jeżeli skala ustawiona jest na 1, a pozycja czy rotacja na 0, wykonywanie mnożeń nie będzie miało żadnego efektu poza marnowaniem czasu.

Kod źródłowy 4.3 Kod odpowiedzialny za zwracanie macierzy przekształceń obiektu. Kod jest częścią definicji klasy Object3d.

```
1  protected _absMat: Mat4 = identity();

3  public get absoluteMatrix() {
4    if (this.changed || this.parent.changed) {
5      identity(this._absMat);

7      if (this._sca.x !== 1 || this._sca.y !== 1 || this._sca.z !== 1)
8        scale(
9          this._absMat,
10         this._sca.x,
11         this._sca.y,
12         this._sca.z,
13         this._absMat
14       );

16     if (this._pos.x || this._pos.y || this._pos.z)
17       move(this._absMat, this._pos.x, this._pos.y, this._pos.z, this._absMat);

19     if (this._rot.x) rotateX(this._absMat, this._rot.x, this._absMat);
20     if (this._rot.y) rotateY(this._absMat, this._rot.y, this._absMat);
21     if (this._rot.z) rotateZ(this._absMat, this._rot.z, this._absMat);

23     if (this.parent.changed && !this.parent.updatedByChild) {
24       multiply(this.parent.absoluteMatrix, this._absMat, this._absMat);
25       this.parent.updatedByChild = true;
26     } else {
27       multiply(this.parent._absMat, this._absMat, this._absMat);
28     }
29   }
30   return this._absMat;
31 }
```

Wspomniana własność - zapisywanie wyniku mnożenia w podanej macierzy zamiast zwracania nowej macierzy - jest kolejną użytą optymalizacją. Każdy obiekt alokuje potrzebne mu macierze tylko jeden raz, a następnie nadpisuje je, jeżeli mają ulec zmianie. Pozwala to oszczędzić czas potrzebny na alokowanie nowej pamięci i zwalnianie nieużywanej.

Należy zwrócić też uwagę na linijkę 23 prezentowanego kodu; jeżeli zmienił się rodzic, ale jego macierz nie została nadal zaktualizowana przez dziecko, do mnożenia zostaje użyta wartość z gettera `absoluteMatrix` - pozwala to rodzicowi na obliczenie swojej macierzy przekształceń na nowo i zwrócenie jej. W przeciwnym wypadku używana jest po prostu jej bezpośrednia wartość.

Zarządzanie buforami

Funkcjonalność menadżera buforów podzielona jest na trzy części:

1. zarządzanie uniwersalnymi buforami
2. zarządzanie buforami kolorów
3. zarządzanie teksturami

Każda z tych części oparta jest przy pomocy tablic asocjacyjnych (nazywanych też *mapami*), przypisującymi liczbom lub ciągom znaków obiekty `BufferCounter`, które są niczym innym jak parą liczby i instacji `BufferInfo`.

Pierwsza opisana funkcjonalność realizowana jest przy pomocy liczenia *hashu* (zwanego też funkcją skrótu czy funkcją hashującą) charakteryzującego bufor. W przypadku ładowania obiektów z pliku, hash liczony jest na podstawie jego zawartości. W przypadku programistycznego tworzenia kształtu, funkcja generująca kształt zwraca dane o jego buforach, z góry ustalony, stały hash oraz informację o skali, która używana jest do finalnego przekształcenia obiektu do rozmiaru, w którym ma być wyświetlony na ekranie - umożliwia to utworzenie pojedynczego bufora pozycji wierzchołków dla wszystkich prostopadłościanów (opisującego sześcian jednostkowy), a następnie podawanie jedynie wartości skalowania we wszystkich trzech kierunkach, aby uzyskać dowolny inny prostopadłościan. Oszczędza to pamięć GPU.

Przykładowy fragment programu odpowiedzialny za zarządzanie uniwersalnymi buforami 4.4.

Kod źródłowy 4.4 Kod odpowiedzialny za zarządzanie uniwersalnymi buforami.

```
1  private buffersMap: Record<string, BufferMap | undefined> = {
2    colors: {},
3  };

5  private bufferNewData(
6    data: Float32Array,
7    itemsPerVertex: number,
8    at: BufferMap,
9    hash: Hash
10 ): BufferCounter {
11   const buffer = this.gl.createBuffer();
12   this.gl.bindBuffer(this.gl.ARRAY_BUFFER, buffer);
13   this.gl.bufferData(this.gl.ARRAY_BUFFER, data, this.gl.STATIC_DRAW);
14   return (at[hash] = {
15     instances: 1,
16     bufferInfo: { itemsPerVertex, buffer, length: data.length, hash },
17   });
18 }

20 private resolveUniversalBuffer(
21   buffersType: string,
22   dataSource: () => Float32Array,
23   itemsPerVertex: number,
24   hash: Hash
25 ): BufferCounter {
26   if (this.universalBuffersMap[buffersType] === undefined) {
27     this.universalBuffersMap[buffersType] = {};
28   }
29   if (this.universalBuffersMap[buffersType][hash] !== undefined) {
30     this.universalBuffersMap[buffersType][hash].instances++;
31     return this.universalBuffersMap[buffersType][hash];
32   } else {
33     return this.bufferNewData(
34       dataSource(),
35       itemsPerVertex,
```

```
36        this.universalBuffersMap[buffersType],
37        hash
38      );
39    }
40  }
```

Druga funkcjonalność związana z buforami kolorów różni się znacząco od poprzedniej - obrazuje ją kod 4.5. Kolory przekazywane są jako tablica liczb dowolnej długości. Po połączeniu jej elementów w jeden ciąg znaków, przy jego pomocy liczony jest hash, co przedstawia linijka 2. Jeżeli obliczony hash już znajduje się w mapie, oznacza to, że podana kombinacja kolorów istnieje, jednak nie oznacza to jeszcze, że istniejący bufor może zostać wykorzystany w tym przypadku. Sprawdzana jest długość bufora i dopiero jeżeli jest ona większa lub równa potrzebnej długości, bufor jest wykorzystywany. W przeciwnym wypadku istniejący bufor jest zastępowany nowym o odpowiedniej długości. W tym celu przekazana tablica jest rozszerzania modularnie do wymaganej długości, co obrazują linijki 7 oraz 19. Takie potraktowanie kwestii kolorów pozwala utworzyć jednokolorowy obiekt przekazując tablicę długości 4 o zawartości $[r, g, b, a]$ od r - *red* - zawartość czerwieni, g - *green* - zawartość zieleni, b - *blue* - zawartość niebieskiego oraz a - *alpha* - przezroczystość (każda z tych wartości musi zawierać się w przedziale $[0, 1]$) albo np. obiekt w kratkę, kolorujący co drugi trójkąt na inny kolor, przekazując najpierw kolor trzech wierzchołków, a następnie inny kolor trzech kolejnych, to jest $[r_1, g_1, b_1, a_1, r_1, g_1, b_1, a_1, r_1, g_1, b_1, a_1, r_2, g_2, b_2, a_2, r_2, g_2, b_2, a_2, r_2, g_2, b_2, a_2]$, co zostanie rozszerzone modularnie na wszystkie wierzchołki reprezentacji obiektu.

Kod źródłowy 4.5 Kod odpowiedzialny za zarządzanie buforami kolorów.

```
1  private resolveColorsBuffer(color: number[], length: number): BufferCounter {
2    const hash = hashString(color.join(""));
3    if (this.colorsBuffers[hash] !== undefined) {
4      this.colorsBuffers[hash].instances++;
5      if (this.colorsBuffers[hash].bufferInfo.length < length) {
6        const data = new Float32Array(
7          Array.from({ length }).map((_, i) => color[i % color.length])
8        );
9        this.gl.bindBuffer(
10         this.gl.ARRAY_BUFFER,
11         this.colorsBuffers[hash].bufferInfo.buffer
12       );
13       this.gl.bufferData(this.gl.ARRAY_BUFFER, data, this.gl.STATIC_DRAW);
14     } else {
15       return this.colorsBuffers[hash];
16     }
17   } else {
18     const data = new Float32Array(
19       Array.from({ length }).map((_, i) => color[i % color.length])
20     );
21     return this.bufferNewData(data, 4, this.colorsBuffers, hash);
22   }
23 }
```

Trzecia funkcjonalność jest łatwiejsza i nie wymaga obrazowania fragmentem kodu; tworzona jest nowa tekstura przy pomocy API WebGL, wypełniana tymczasowo szarością, a po załadowaniu obrazka (obrazki ładowane są asynchronicznie, ponieważ przeglądarka internetowa musi wykonać po nie zapytanie) aktualizowane przy pomocy jego zawartości.

Zarządzanie programami

Klasa `ProgramManager` ma za zadanie zapewniać programy realizujące oczekiwane funkcjonalności; robi to ona na podstawie masek bitowych opisujących funkcjonalność programu. Definicje masek w bibliotece prezentuje kod 4.6.

Kod źródłowy 4.6 Maski funkcjonalności biblioteki

```
1  const FEATURES = {
2    COLOR: 1 << 0,
3    AMBIENT_LIGHTING: 1 << 1,
```

```
  DIFFUSE_LIGHTING: 1 << 2,
  TEXTURES: 1 << 3,
  NORMAL_MAP: 1 << 4,
};

// Maska dla programu z wszystkimi funkcjonalnościami
const UNIVERSAL_MASK = Object.values(FEATURES).reduce((acc, e) => acc | e, 0);
```

Maski te wykorzystywane są przez obiekty `ShaderPart`, na podstawie których generowane są kody źródłowe shaderów. Kod 4.7 obrazuje tablicę tychże wraz z funkcją, która konkatenuje fragmenty w celu uzyskania finalnego kodu shadera; w linijce 26 odfiltrowywane są te elementy tablicy, których maska bitowa nie zawiera się w szukanej masce.

Kod źródłowy 4.7 Generowanie kodu źródłowego shadera

```
const shaderParts: ShaderPart[] = [
  {
    featureMask: UNIVERSAL_MASK,
    sourceCode: "precision mediump float;",
  },
  {
    featureMask: FEATURES.COLOR,
    sourceCode: "varying vec4 v_color;",
  },
  {
    featureMask: FEATURES.TEXTURES,
    sourceCode: "varying vec2 v_uv;",
  },
  {
    featureMask: FEATURES.AMBIENT_LIGHTING,
    sourceCode: "uniform float u_ambient;",
  },
  // ...i tak dalej
];

const getFragmentShader = (gl: WebGLRenderingContext, featuresMask: number) =>
  compileShader(
    gl,
    gl.FRAGMENT_SHADER,
    shaderParts
      .filter((part) => (part.featureMask & featuresMask) !== 0)
      .map((part) => part.sourceCode)
      .join("\n")
  );
```

Dodatkowo wygenerowane muszą zostać funkcje odpowiedzialne za aktualizowanie atrybutów i uniformów wykorzystywanych przez różne funkcjonalności. Dane te przechowuje tablica przechowująca obiekty `FeaturesParameters`, które posiadają funkcję zwracającą funkcję do aktualizowania tych danych na podstawie stanu renderowania (omówionego w kolejnej sekcji). Fragment tej dla funkcjonalności kolorowania obiektów pokazuje kod 4.8.

Kod źródłowy 4.8 Tworzenie funkcji aktualizujących atrybuty kolorów

```
const FEATURES_PARAMETERS: FeatureParameters[] = [
  {
    featureMask: UNIVERSAL_MASK,
    createFeatureCall: createBasicCall,
  },
  {
    featureMask: FEATURES.COLOR,
    createFeatureCall: (gl, program) => {
      const location = gl.getAttribLocation(program, "a_color");
      const call = createAttributeUpdateCall(gl, {
        type: "vec4",
        location,
```

```
13        });

15        return (state) => {
16          if (
17            state.hashes.color !==
18            state.renderedObject.representation.bufferData.colors.hash
19          ) {
20            call(state.renderedObject.representation.bufferData.colors.buffer);
21            state.hashes.color =
22              state.renderedObject.representation.bufferData.colors.hash;
23          }
24        };
25      },
26    },
27    // ...i tak dalej
28  ];
```

To wszystko wykorzystywane jest przez `ProgramManager`, aby zarządzać programami potrzebnymi do rysowania sceny. Kluczową logikę działania tej klasy prezentuje kod 4.9.

Kod źródłowy 4.9 Logika klasy `ProgramManager`

```
1  private programs: Record<number, ProgramInfo | undefined> = {};

3  private prepareProgram(featuresMask: number): ProgramInfo {
4    const vertex = getVertexShader(this.gl, featuresMask);
5    const fragment = getFragmentShader(this.gl, featuresMask);
6    const program = createProgram(this.gl, vertex, fragment);
7    return (this.programs[featuresMask] = {
8      program,
9      featuresMask,
10     updateFunctions: FEATURES_PARAMETERS.filter(
11       (parameter) => (parameter.featureMask & featuresMask) !== 0
12     )
13       .map((parameter) => parameter.createFeatureCall(this.gl, program))
14       .reduce<ParameterUpdateFunction[]>((acc, e) => [...acc, e], []),
15   });
16 }

18 public requestProgram(featureMask: number): ProgramInfo {
19   return this.programs[featureMask] ?? this.prepareProgram(featureMask);
20 }
```

Rysowanie sceny

Jest to tak naprawdę główna funkcjonalność silnika. Jest ona zoptymalizowana na kilka sposobów.

Pierwszy z nich odbywa się podczas dodawania nowych elementów do sceny. W funkcji `Scene.connectScene` tworzona jest wcześniej wspominana tablica `elements`, która sortowana jest po wielu polach przy pomocy biblioteki `thenby`. Na samym początku obiekty sortowane są wedle używanych przez nie funkcjonalności, ponieważ proces przełączania programu WebGL jest kosztowny - celem jest pogrupowanie obiektów korzystających z tego samego programu obok siebie. Następnie sortowane są one po buforach o najmniejszej entropii; zliczane są unikalne hashe występujące w każdym typie bufora i obiekty sortowane są kolejno po hashach buforów w rosnącej tejże liczbie. Proces ten przedstawia fragment kodu 4.10.

Kod źródłowy 4.10 Sortowanie obiektów sceny

```
1 // Policzenie wystąpień unikalnych hashów w każdym typie bufora; obiekt Set działa
      niczym matematyczny zbiór - nie duplikuje wartości
2 const buckets: Record<string, Set<Hash>> = {};
3 this.elements
4   .filter((element) => element.drawable)
5   .forEach((element) => {
6     Object.entries(element.object.representation.bufferData).forEach(
```

```
7         ([key, value]) => {
8           buckets[key] = buckets[key]
9             ? buckets[key].add(value.hash)
10            : new Set([value.hash]);
11        }
12      );
13    });
14  // Utworzenie funkcji sortującej, zaczynając od maski funkcjonalności, na podstawie
        której dobierany jest odpowiedni program do rysowania obiektu
15  let sortingFn = firstBy<SceneObject>(
16    (a, b) => a.object.representation.featuresMask -
          b.object.representation.featuresMask
17  );
18  // Dodanie do funkcji sortującej kolejnych poziomów sortowania po buforach o
        najmniejszej liczbie unikalnych hashów
19  Object.entries(buckets)
20    .sort(([_, a], [__, b]) => a.size - b.size)
21    .forEach(([key], index) => {
22      sortingFn = sortingFn.thenBy((a, b) => {
23        a.object.representation.bufferData[key]?.hash.toString()
24          .localeCompare(b.object.representation.bufferData[key]?.hash.toString());
25      );
26    });
27  this.elements = this.elements.sort(sortingFn);
```

Następną kwestią jest sama implementacja funkcji odpowiedzialnej za narysowanie pojedynczej klatki obrazu. Kolejne etapy funkcji `S3e.draw` wraz z przykładami kodu to kolejno:

1. Wyczyszczenie buforów kolorów oraz głębokości oraz inicjalizacja stanu renderowania; stan przechowuje informacje o aktualnie rysowanym obiekcie, słownik hashy umożliwiający sprawdzenie, czy należy zmienić wskaźnik na bufor oraz informacje o aktualnie wykorzystywanym programie wraz z tablicą zawierającą referencje na `ParameterUpdateFunction` (początkowo pustą), które to są funkcjami przyjmującymi właśnie stan renderowania za parametr i korzystającymi z niego, aby aktualizować odpowiednio wartości atrybutów i uniformów przed narysowaniem kolejnego elementu sceny. Kod 4.11 prezentuje tę część funkcji.

2. Iteracja po wszystkich obiektach aktualnej sceny; jeżeli element jest zdatny do narysowania (niezdatne są np. kamery oraz kontenery scen) i jego maska funkcjonalności różni się od maski aktualnie wykorzystywanego programu, program zastępowany jest inną instancją uzyskiwaną z `ProgramManager`. Czyszczone są także hashe, ponieważ po zmianie programu wszystkie wcześniej używane atrybuty i uniformy muszą być ustawione ponownie. Ustawione są także parametry rysowania; bufor głębokości oraz *face culling*, czyli rysowanie trójkątów tylko z jednej strony. Kod 4.12 prezentuje tę część funkcji.

3. Przeskalowanie obiektu wedle wspomnianego wcześniej `defaultScale` używanego przy rysowaniu kształtów geometrycznych oraz przygotowanie finalnej macierzy widoku z kamery. Kod 4.13 prezentuję tę część funkcji.

4. Wywołanie wszystkich funkcji aktualizujących atrybuty oraz uniformy związanych z aktualnym programem, narysowanie trójkątów i oznaczenie, że wszystkie elementy nie zostały zmienione od ostatniego narysowania. Kod 4.14 prezentuje tę część funkcji.

Kod źródłowy 4.11 Początek funkcji `draw`

```
1  public draw() {
2    this.gl.clear(this.gl.COLOR_BUFFER_BIT | this.gl.DEPTH_BUFFER_BIT);
3    const renderState: RenderState = {
4      engine: this,
5      renderedObject: null,
6      drawable: false,
7      currentProgram: {
8        program: null,
9        featuresMask: null,
```

```
10        basicCall: null,
11        updateFunctions: [],
12      },
13      hashes: {},
14    };
```

Kod źródłowy 4.12 Początek pętli rysującej

```
1  for (const element of this.currentScene.elements) {
2    renderState.renderedObject = element.object;
3    renderState.drawable = element.drawable;

5    if (
6      element.drawable &&
7      element.object.representation.featuresMask !==
8        renderState.currentProgram.featuresMask
9    ) {
10     renderState.currentProgram = this.programManager.requestProgram(
11       element.object.representation.featuresMask
12     );
13     renderState.hashes = {};
14     this.gl.useProgram(renderState.currentProgram.program);
15   }
16   this.gl.enable(this.gl.DEPTH_TEST);
17   this.gl.enable(this.gl.CULL_FACE);
```

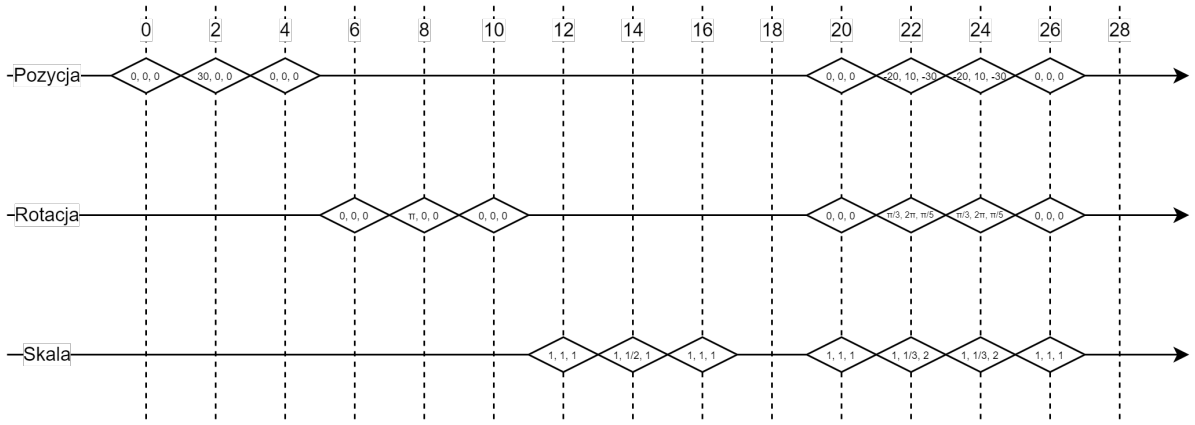
Kod źródłowy 4.13 Przeskalowanie obiektu oraz przygotowanie macierzy worldView - widoku z aktualnej kamery

```
1  if (element.object.representation.defaultScale !== undefined) {
2    scale(
3      element.object.absoluteMatrix,
4      element.object.representation.defaultScale.x,
5      element.object.representation.defaultScale.y,
6      element.object.representation.defaultScale.z,
7      this.worldView
8    );
9    multiply(
10     this.currentScene.currentCamera.viewProjection,
11     this.worldView,
12     this.worldView
13   );
14 } else {
15   multiply(
16     this.currentScene.currentCamera.viewProjection,
17     element.object.absoluteMatrix,
18     this.worldView
19   );
20 }
```

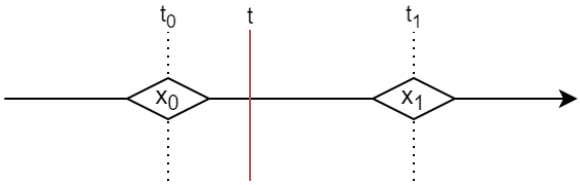
Kod źródłowy 4.14 Aktualizacje danych oraz narysowanie trójkątów

```
1      for (let updateFunction of renderState.currentProgram.updateFunctions) {
2        updateFunction(renderState);
3      }
4      if (element.drawable) {
5        this.gl.drawArrays(
6          this.gl.TRIANGLES,
7          0,
8          element.object.representation.bufferData.positions.length /
9            element.object.representation.bufferData.positions.itemsPerVertex
10       );
```

Rysunek 4.4: Przykładowe trzy osie czasu - translacji, rotacji oraz skali - wraz z zaznaczonymi klatkami kluczowymi w konkretnych sekundach



Rysunek 4.5: Dwie klatki i czas pomiędzy nimi




```
11      }
12    }
13    for (const element of this.currentScene.elements) {
14      element.object.changed = false;
15    }
16  }
```

Animacje

Tworzenie animacji oparte jest o osie czasu, czyli obiekty kontrolujące określoną własność obiektu o określonej liczbie klatek na sekundę z możliwością ustawienia na nich klatek kluczowych. Klatka kluczowa jest trójką - numer klatki, wartość własności oraz interpolacja. Przykładową wizualizację osi czau przedstawia rysunek 4.4.

O ile numer klatki i jej wartość mają naturalne znacznie z samej ich nazwy, o tyle interpolacja wymaga precyzyjniejszego wyjaśnienia. Zadany przypadek obrazuje rysunek 4.5.

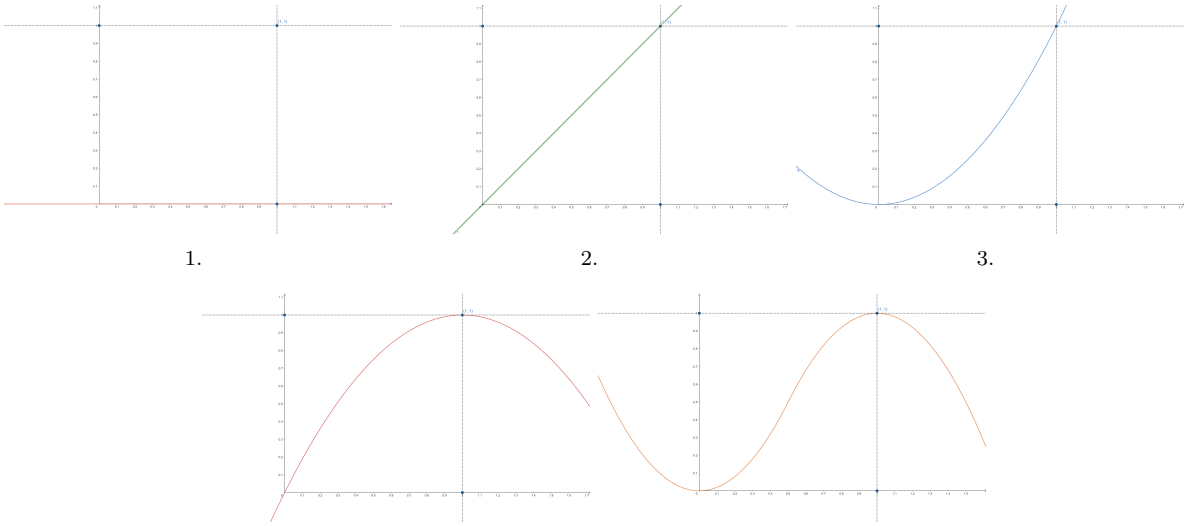
Przedstawia on dwie klatki kluczowe w dwóch momentach w czasie, t_0 i t_1, oraz moment w czasie, w którym chcemy uzyskać wartość zmiennej interpolowanej pomiędzy tymi klatkami, oznaczony t. t może być dowolną liczbą, ale klatki kluczowe przypisane są do konkretnej klatki będącej liczbą naturalną; ich czas możemy obliczyć dzieląc ich klatkę przez liczbę klatek na sekundę osi czasu czyli $t_0 = \frac{f_0}{fps}$ i $t_1 = \frac{f_1}{fps}$.

Wartości w tych klatkach to x_0 oraz x_1 i są to jedyne dyskretnie zdefiniowane wartości w tej sytuacji; można zapisać, że $T(t_0) = x_0$ oraz $T(t_1) = x_1$, gdzie T to funkcja przypisująca czasowi wartość. $T(t)$ wymaga jednak zdefiniowania i do tego celu służą metody interpolacji.

W sytuacji, kiedy szukany czas znajduje się pomiędzy dwiema klatkami kluczowymi, pierwszą częścią interpolacji jest znalezienie w jakiej części czasu pomiędzy nimi znajduje się szukane t; można to zrobić korzystając z prostego stosunku:

$$p = \frac{t - t_0}{t_1 - t_0}$$

oznaczonego p od słowa *postęp* (lub angielskiego *progress*). $p = 0.5$ oznacza nic innego, jak "t jest równo w połowie pomiędzy t_0 i t_1.



1. 2. 3. 4. 5.

Rysunek 4.6: Wykresy funkcji interpolujących

Funkcja interpolacyjna $I : [0,1] \to [0,1]$ będzie służyła do zdecydowania, w jakich stosunkach wybrać wartości klatek kluczowych. Ostateczne równanie na wartość wygląda następująco:

$$T(t) = I(p) \cdot x_0 + (1 - I(p)) \cdot x_1$$

co można rozwinąć do:

$$T(t) = I(\frac{t - t_0}{t_1 - t_0}) \cdot T(t_0) + (1 - I(\frac{t - t_0}{t_1 - t_0})) \cdot T(t_1)$$

W tej sytuacji pozostaje jedynie zdefiniować funkcję interpolacyjną I. Jest ona całkowicie zależna od zastosowania i może przybierać różne postaci. Zaimplementowane w pracy to:

1. $I(p) = 0$ - klatka kluczowa typu **HOLD**, utrzymująca wartość poprzedniej klatki
2. $I(p) = p$ - klatka kluczowa typu **LINEAR**, liniowo zmieniająca wartość pomiędzy klatkami
3. $I(p) = p^2$ - klatka kluczowa typu **SQUARE IN**, rozpoczynająca przejście kwadratowo
4. $I(p) = 2p^2 - 2p$ - klatka kluczowa typu **SQUARE OUT**, kończąca przejście kwadratowo
5. $I(p) = \begin{Bmatrix} 2p^2, & \text{dla } p < \frac{1}{2} \\ -2p^2 + 4p - 1, & \text{dla } p \geqslant \frac{1}{2} \end{Bmatrix}$ - klatka kluczowa typu **SQUARE IN OUT**, rozpoczynająca i kończąca przejście kwadratowo

Wykresy funkcji można zobaczyć na rysunku 4.6

Tak zdefiniowane klatki kluczowe można dowolnie mieszać pomiędzy sobą; interpolacja wybierana jest na podstawie wcześniejszej klatki (w przytoczonym przykładzie - klatki w t_0).

Pod względem implementacyjnym, funkcjonalność ta jest bardzo analogiczna do przedstawionego modelu matematycznego. Klatki w osi czasu przechowywane są w tablicy, która sortowana jest od najwcześniejszych do najpóźniejszych. W ten sposób iterując w poszukiwaniu pary klatek kluczowych, pomiędzy którymi potrzebujemy interpolować wartość, nie trzeba sprawdzać wszystkich par i wykonywane jest to w czasie liniowym. Funkcje interpolujące są bezpośrednio przełożone z zapisu matematycznego na język programowania, co przedstawia kod 4.15.

Kod źródłowy 4.15 Kod funkcji interpolacyjnych.

```
1 export const KeyframeType = {
2   HOLD: (_: number) => 0,
3   LINEAR: (value: number) => value,
```

```
4    SQUARE_IN: (value: number) => value ** 2,
5    SQUARE_OUT: (value: number) => 2 * value - value ** 2,
6    SQUARE_IN_OUT: (value: number) =>
7      value < 0.5 ? 2 * value ** 2 : -2 * value ** 2 + 4 * value - 1,
8  };
```

Kod posiada dodatkowo jedną optymalizację, która sprawia, że kod nie jest niepotrzebnie wywoływany, jeżeli wszystkie klatki kluczowe animacji zostały już wykonane. Wartość `evaluatedLast` w kodzie 4.16 jest ustawiana na prawdziwą, kiedy zostanie odtworzona ostatnia klatka kluczowa animacji (jest to sprawdzane w warunkach w późniejszej części kodu). Jeżeli tak jest i czas przekroczył czas ostatniej klatki, funkcja kończy wykonywanie. Jest to rozwiązanie częstego scenariusza, w którym animacja ma wykonać się na początku, a potem nie jest już aktywna; zapobiega to niepotrzebnemu przeszukiwaniu jej klatek kluczowych.

Kod źródłowy 4.16 Początek funkcji sprawiającej, że oś czasu ustawia kontrolowaną wartość na zadany w jej argumencie czas.

```
1  public evaluateAt(time: number) {
2    if (
3      time > this.keyframes[this.keyframes.length - 1].frame / this.fps &&
4      this.evaluatedLast
5    ) {
6      return;
7    } else {
8      this.evaluatedLast = false;
9    }

11   // reszta logiki
12 }
```

Ładowanie modeli z pliku

Podstawowa struktura pliku *.obj* jest złożona z linii postaci `identyfikator argument1 argument2 argument3...` , między innymi:

- `v x y z` - od *vertex*, czyli "wierzchołek". Opisuje pojedyncze współrzędne jednego wierzchołka modelu. Nie jest to konkretny wierzchołek w kolejności, jedynie jego definicja, do której odnieść się może reszta pliku. Kolejne definicje wierzchołków indeksowane są od jedynki.
- `vt x y` - od *vertex texture*, czyli "tekstury wierzchołka". Działa na tej samej zasadzie, co poprzednie, opisuje jednak współrzędne tekstury. Ta sekcja jest opcjonalna.
- `vn x y z` - od *vertex normal*, czyli "wektor normalny wierzchołka". Działa analogicznie jak poprzednie sekcje. Ta sekcja jest opcjonalna.
- `f v1/t1/n1 v2/t2/n2 ...` - od *face*, czyli "ściany". Opisuje jedną ścianę figury trójwymiarowej. Ściana ta jest wielokątem, którego wierzchołki, pozycje tekstur oraz wektory normalne są opisane przy pomocy kolejnych argumentów. Pozycje wierzchołków opisują parametry `v`, koordynaty tekstur `t`, a wektory normalne - `n`. Warto zaznaczyć, że nie są to wartości, a jedynie indeksy, które odnoszą się do konkretnych wierzchołków, koordynatów i wektorów zdefiniowanych wcześniej w pliku. Umożliwia to użycie tych samych wartości wielokrotnie odwołując się do nich jedynie indeksem, co w rezultacie zmniejsza rozmiar pliku. Warto zaznaczyć, że parametry koordynatów i wektorów normalnych są opcjonalne w tym zapisie. Ważne w opisywanym przykładzie jest to, że wielokąt w tym poleceniu może mieć dowolną liczbę wierzchołków - nie musi być zawsze trójkątem. Rozwiązanie tego problemu jest opisane poniżej, w szczegółach implementacyjnych.

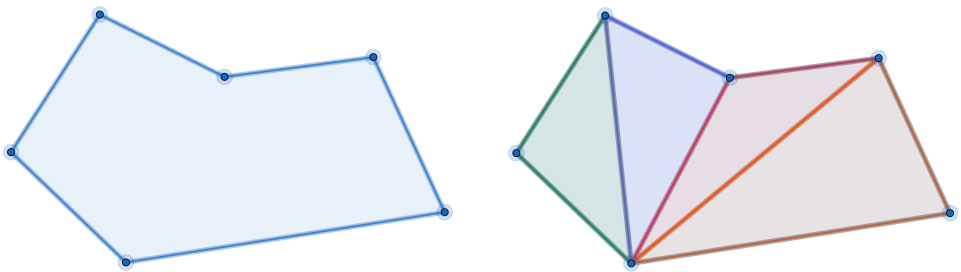
Kod odpowiedzialny za parsowanie jest prosty; dzieli on wczytany tekst na wiersze, inicjalizuje stan, w którym zapisuje dane o wierzchołkach i wektorach normalnych, a następnie iteruje po wszystkich wierszach sprawdzając ich pierwsze polecenie - przedstawia to kod 4.17. Warto zwrócić uwagę na linię 14, która odpowiedzialna jest za tworzenie trójkątów z dowolnych wielokątów; pierwszy wierzchołek się nie zmienia, a iteracja przeprowadzana jest po parach dwóch przyległych do siebie.

Kod źródłowy 4.17 Główny kod parsujący plik .obj.

```
const parsePart = (part: string[], state: ParsingState) => {
  const [command, ...args] = part;

  switch (command) {
    case "v":
      state.vertices.push(args.map(Number.parseFloat));
      break;
    case "vn":
      state.normals.push(args.map(Number.parseFloat));
      break;
    case "vt":
      state.uvs.push(args.map(Number.parseFloat));
    case "f":
      for (let i = 0; i < args.length - 2; i++) {
        parseVertex(args[0], state);
        parseVertex(args[i + 1], state);
        parseVertex(args[i + 2], state);
      }
      break;
    default:
      break;
  }
};
```

Podział będący wynikiem takiej iteracji prezentuje rysunek 4.7. Istnieje jeden warunek konieczny, aby ten sposób zadziałał; dzielony wielokąt musi być wypukły. Mimo że specyfikacja formatu *.obj* nie wspomina nigdzie o tym warunku, oprogramowanie eksportujące w tym obiekty go spełnia.

Rysunek 4.7: Podział wielokąta na trójkąty



4.3 Funkcje dodatkowe

Poza celami opisanymi w zarysie koncepcyjnym pracy, biblioteka realizuje także cele wykraczające poza początkowy jej zakres. W tej sekcji zostały one pokrótce opisane.

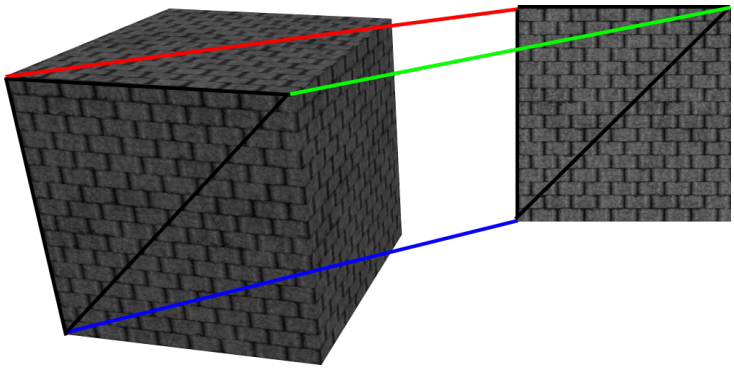
4.3.1 Tekstury

Tekstury są dwuwymiarowymi obrazami nakładanymi na rysowane przez silnik wielokąty. WebGL posiada specjalne bufory przystosowane do ładowania i wykorzystywania tekstur, metodę do generowania mipmap (pomniejszonych rozmiarów teksury do wykorzystania, kiedy jej pełna rozdzielczość nie jest potrzebna) oraz umożliwia automatyczną interpolację tekstury na podstawie danych o współrzędnych.

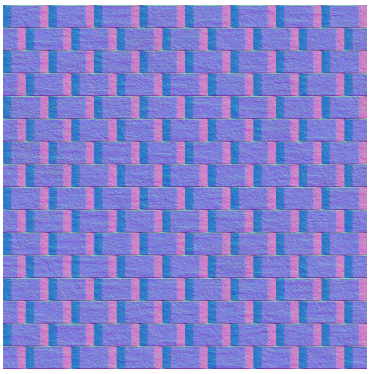
Zasada działania

Aby poprawnie oteksturować obiekt trójwymiarowy, potrzebne są dwa elementy: tekstura oraz współrzędne teksturowania, nazywane powszechnie UVs (angielska liczba mnoga od UV; przez (u, v) zwykle

Rysunek 4.8: Mapowanie tekstur



Rysunek 4.9: Mapa normalnych dla tekstury z rysunku 4.8



określa się współrzędne tekstury, ponieważ (x, y, z, w) używane są do określania współrzędnych wierzchołków trójkątów). Dla każdego wierzchołka określane są współrzędne tekstury jako para liczb (u, v), $u, v \in [0, 1]$, gdzie $(0, 0)$ to lewy dolny róg tekstury, a $(1, 1)$ - prawy górny. Rysunek 4.8 przedstawia sześcian oteksturowany przy pomocy faktury ceglanej ściany z kamienia - schematycznie przedstawiono, w jaki sposób mapowane są współrzędne na teksturze przedstawionej po prawej stronie do wierzchołków trójwymiarowej kostki po lewej.

4.3.2 Mapy normalnych

Jest to specjalnych rodzaj tekstur, które odpowiadają za przekształcenie wektorów normalnych powierzchni tak, aby symulować na niej nierówności - stąd też jedna z angielskich nazw, *bump mapping*.

Zasada działania

Tekstury te ładowane są na analogicznej zasadzie co zwykłe tekstury; korzystają też z tych samych współrzędnych, aby przypisać pozycje na nich do wierzchołków. Rysunek 4.9 przedstawia przykładową mapę normalnych; kanały RGB każdego kodują wektory normalnych - jasność R odpowiada za wartość x, G - y, a B - z. Mapa ta zakłada, że wektor normalny do powierzchni to $(0, 0, 1)$, stąd też wynika jej niebieski odcień. Aby zastosować taką mapę do dowolnej powierzchni, należy wykonać zmianę bazy wyznaczonego z niej wektora normalnego do takiej, w którym odpowiada on wektorowi normalnemu rozważanego trójkąta.

Więcej wyników uzyskanych przy pomocy opisanych funkcjonalności znajduje się w kolejnym rozdziale.

Rezultaty

5.1 Przykład minimalny

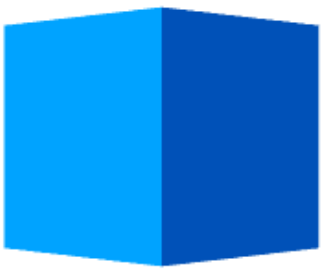
Biblioteka jest budowana do pojedynczego pliku, który potem musi zostać załadowany w kodzie strony, na której chcemy go użyć. Można także wykorzystać ją bezpośrednio w autorskich projektach, które dopiero potem budowane są do załączenia na stronie.

Niezależnie od sposobu użycia, minimalny przykład przedstawia kod 5.1.

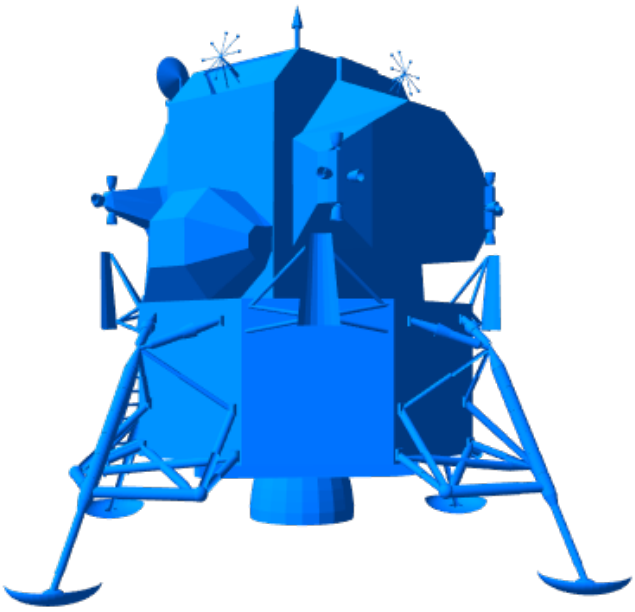
Kod źródłowy 5.1 Przykład minimalny.

```
1  const canvas = document.getElementById("example");


4  const engine = new S3e(canvas); // silnik
5  const shape = createCuboid(20); // abstrakcyjy kształt
6  const color = [0, 0.44, 1, 1]; // kolor
7  const bufferData = engine.bufferManager.loadShape(shape, { color }); // bufory
8  const cube = new Object3d(bufferData); // obiekt

10 cube.rotation.y = Math.PI/4;

12 engine.currentScene.addChild(cube);
13 engine.currentScene.currentCamera.position.z = 100;

15 engine.draw();
```

Kod ten zakłada, że na stronie istnieje element `canvas` o id *example*, inicjalizuje silnik, tworzy niebieski sześcian o rozmiarze 20, obraca go, dodaje do sceny, odsuwa kamerę i rysuje jedną klatkę obrazu. Wynik uruchomienia tego kodu przedstawia rysunek 5.1.



Rysunek 5.1: Przykład minimalny



Rysunek 5.2: Model lądownika wyświetlany w silniku

5.2 Ładowanie modelu

Aby lepiej zaprezentować grafikę generowaną przez bibliotekę, można użyć opcji ładowania modelu z pliku .obj. Do przykładu wykorzystano plik z otwartej bazy modeli 3d udostępnianej przez NASA - lądownik księżycowy misji Apollo [1]. Model ten zawiera ponad 100 000 wielokątów. Kod potrzebny do załadowania modelu przedstawia 5.2.

Kod źródłowy 5.2 Ładowanie pliku .obj i tworzenie obiektu do wyświetlenia.

```
1 const response = await fetch("LunarLander.obj");
2 const objFileContents = await response.text();

4 const color = [0, 0.44, 1, 1];
5 const bufferData = engine.bufferManager.loadObj(objFileContents, { color });

7 const landerObject = new Object3d(bufferData);

9 engine.currentScene.addChild(landerObject);
```

Obiektowi nadawany jest jednolity kolor. Wynik po odsunięciu kamery i obróceniu modelu przedstawia rysunek 5.2.

5.3 Tekstury

Do oteksturowania obiektu potrzeba dodatkowego pliku z teksturami, do którego ścieżkę należy przekazać podczas tworzenia obiektu. Kod 5.3 generuje obrazek 5.3.

Kod źródłowy 5.3 Ładowanie pliku .obj i teksturowanie go.

Rysunek 5.3: Oteksturowany model

```
const objFileContents = await fetch("HDU_lowRez_part1.obj").then(async (data) =>
  data.text()
);

const base = new Object3d(
  engine.bufferManager.loadObj(objFileContents, { texture: { main: "HDU_01.jpg" } })
);
```

5.4 Mapy normalnych

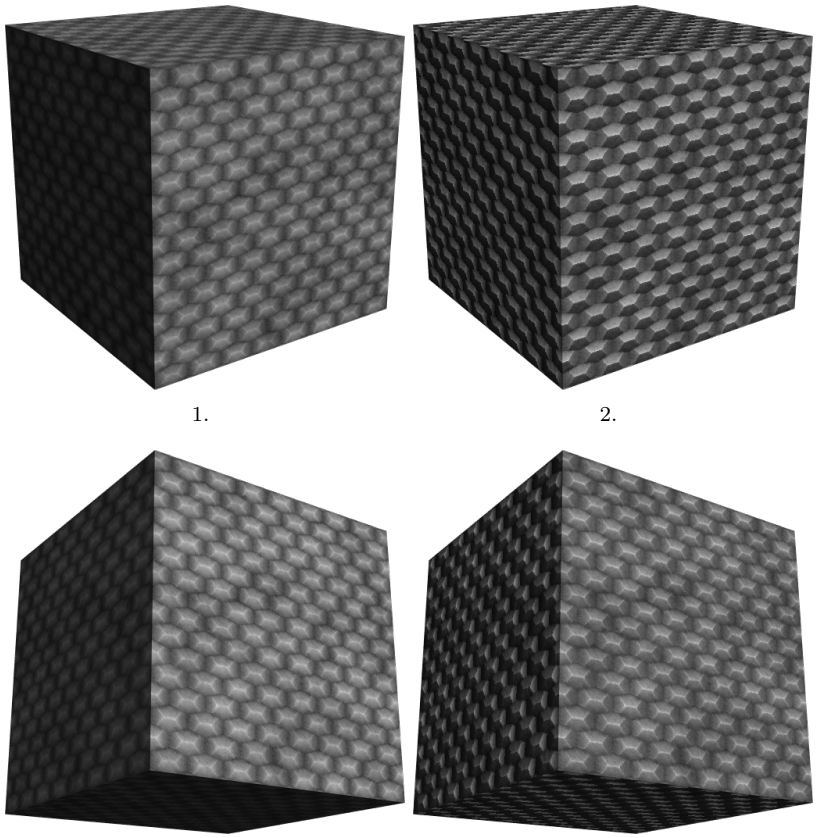
Aby dodać do oteksturowanego modelu mapę normalnych, przy tworzeniu go należy przekazać do niej ścieżkę. Kod 5.4 tworzy sześcian używający tekstur oraz mapy normalnej.

Kod źródłowy 5.4 Tworzenie oteksturowanego sześcianu z mapą normalnych

```
const cube = new Object3d(
  engine.bufferManager.loadShape(createCuboid(30), {
    texture: {
      main: "blocks_colors.jpg",
      normalMap: "blocks_normals.jpg",
    },
  })
);
```

5.5 Animowanie sceny

Aby sprawić, by tworzony obraz się poruszał, należy wywołać odpowiednio często metodę `S3e.draw` . By nie marnować zasobów wywoływaniem jej częściej niż jest to w stanie oddać monitor (np. w nieskończonej pętli) lub psuć doświadczenia użytkownikowi przez za wolne wołanie (i w efekcie obniżoną liczbę klatek animacji na sekundę), można wykorzystać funkcję z API przeglądarek nazwaną `Window.requestAnimationFrame`. Przyjmuje ona za argument funkcję nie zwracającą niczego z jednym opcjonalnym argumentem liczbowym. Wywołanie `Window.requestAnimationFrame` wkłada do kolejki modelu asyn-



Rysunek 5.4: Po lewej stronie znajdują się sześciany z samymi teksturami, a po prawej z teksturami i mapami normalnych

chronicznego przeglądarki wywołanie funkcji podanej w jej argumencie; ta wywoływana jest, kiedy przeglądarka gotowa jest wygenerować kolejna klatkę obrazu do komputera. W argumencie opcjonalnym przekazywany jest aktualny czas.

Pozwala to pisać kod, który generuje obraz wtedy i tylko wtedy, kiedy przeglądarka jest na to gotowa, a dzięki dostępowi do aktualnego, precyzyjnego czasu można uniezależnić prędkość animacji od liczby klatek na sekundę.

Kod źródłowy 5.5 Przykładowe wykorzystanie `Window.requestAnimationFrame`.

```
function draw(t) {
  requestAnimationFrame(draw);
  landerObject.rotation.z = t / 1000;
  engine.draw();
}

draw(0);
```

Kod 5.5 przedstawia podstawowy sposób użycia tego API. Definiujemy funkcje `draw` przyjmującą za argument czas, następnie wywołujemy ją z zerem - czas liczony jest w milisekundach od zera, gdzie zero to moment uruchomienia aktualnej karty przeglądarki. W tej funkcji najpierw wywoływane jest dodanie do kolejki wywołań jej samej przy pomocy `Window.requestAnimationFrame`, następnie zależnie od czasu ustawiana jest rotacja obiektu lądownika z poprzedniego przykładu, a potem silnik rysuje nową klatkę obrazu.

W ten sam sposób można użyć klasy `Timeline` służących do tworzenia precyzyjnych animacji. Prosty przykład pokazuje kod 5.6. Używa on funkcji pomocniczej `getPositionTimeline`, która inicjalizuje oś czasu dla pozycji podanego obiektu w podanej liczbie klatek na sekundę. Następnie do osi czasu dodawane są trzy klatki kluczowe, w 0. klatce = 0. sekundzie, w 60. klatce = 1. sekundzie i w 120. klatce = 2. sekundzie. Pierwsza i ostatnia klatka ustawiają pozycję obiektu na początkową wartość, $(0, 0, 0)$, środkowa natomiast przesuwa obiekt w górę o 30 jednostek. Następnie pokazana jest nowa definicja funkcji `draw`, w której oś czasu ewaluowana jest dla aktualnego czasu w sekundach.

Wynikiem tego przykładu będzie model lądownika unoszący się płynnie w górę przez sekundę, a następnie opadający do oryginalnej pozycji tak samo długo.

Kod źródłowy 5.6 Animacja z wykorzystaniem klasy `Timeline`

```
const positionTimeline = getPositionTimeline(landerObject, 60);

positionTimeline.addKeyframes(
    {
      frame: 0,
      value: { x: 0, y: 0, z: 0 },
      interpolation: KeyframeType.SQUARE_IN_OUT,
    },
    {
      frame: 60,
      value: { x: 0, y: 30, z: 0 },
      interpolation: KeyframeType.SQUARE_IN_OUT,
    },
    {
      frame: 120,
      value: { x: 0, y: 0, z: 0 },
      interpolation: KeyframeType.HOLD,
    }
);

function draw(t) {
  requestAnimationFrame(draw);
  positionTimeline.evaluateAt(t / 1000);
  engine.draw();
}

draw(0);
```

Podsumowanie

Głównym celem pracy było utworzenie biblioteki umożliwiającej tworzenie grafiki trójwymiarowej w przeglądarce internetowej. Dodatkowo miała ona posiadać szereg funkcjonalności zdefiniowanych w poprzednich rozdziałach oraz spełniać narzucone wymagania projektowe od strony technicznej oraz konceptualnej. Można uznać, że wszystkie ustalone cele zostały spełnione.

Utworzona biblioteka pozwala realizować wszystkie swoje funkcjonalności dostatecznie intuicyjnie i łatwo, co potwierdzają przytoczone przykładowe fragmentu kodu. Jej zakres użycia jest bardzo szeroki; umożliwia wyświetlanie zarówno prostych obrazków podstawowych figur geometrycznych, jak i tworzenie zaawansowanych animacji używając modele ładowane z plików pochodzących z oprogramowania do modelowania trójwymiarowego.

Podczas pracy nad projektem pojawiły się także problemy. Niektórymi z nich były:

- Budowanie biblioteki do modułów, które mogą być wykorzystywane uniwersalnie na wielu platformach. Rozwiązaniem była zmiana koncepcji budowania biblioteki i użycie Webpacka jako narzędzia do paczkowania.
- Optymalizacja wywołań aktualizujących atrybuty i uniformy w programie WebGL. W pierwotnej koncepcji wywołania te miały być uniwersalne i niezależne od programu WebGL używanego przez silnik. Zdecydowano jednak, że takie rozwiązanie będzie mało wydajne w kontekście scen, które muszą być rysowane kilkadziesiąt razy na sekundę i składających się z kilkuset obiektów, dla których zmienne te muszą być aktualizowane.
- Poprawne aktualizowanie macierzy przekształceń przy zmianach u rodzica obiektu. Ten aspekt biblioteki wielokrotnie generował błędy, które musiały być naprawiane

Możliwym rozwiązaniem części z napotkanych problemów byłoby wprowadzenie testów do projektu. Poprawnie napisane testy jednostkowe do różnych funkcjonalności skracałyby czas poszukiwań błędów kodu przy kolejnych jego zmianach.

Poza samymi celami podstawowymi, w bibliotece zostały zrealizowane także drobne funkcjonalności wykraczające poza oryginalne założenia, w tym:

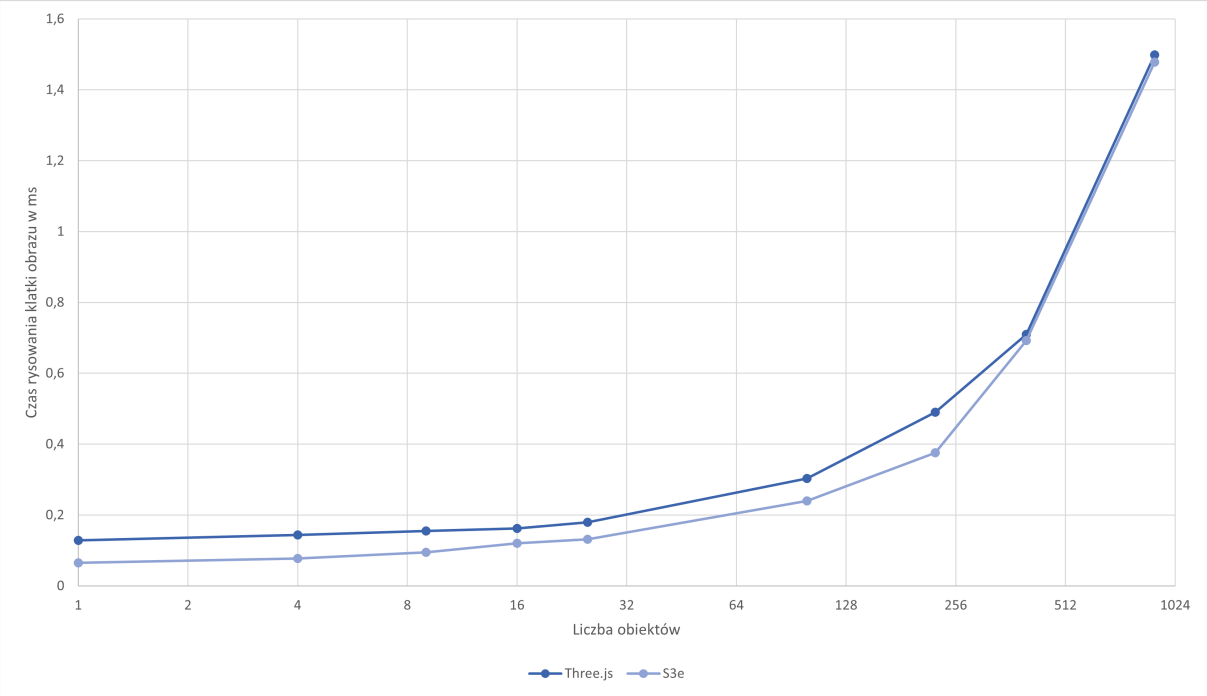
- Teksturowanie, wsparcie dla map normalnych.
- Funkcje tworzące proste kształty, jak np. sześciany czy ostrosłupy prostokątne.
- Funkcje inicjalizujące poprawnie osie czasu dla różnych własności obiektu, ograniczające ilość powtarzalnego kodu, który musiałby być pisany przez użytkownika.
- Automatyczne generowanie wektorów normalnych w przypadku ich braku przy modelach ładowanych z plików.

Biblioteka jest nastawiona także na rozwój. Funkcjonalności możliwe do implementacji w przyszłości to między innymi:

- Możliwość dodawania do sceny określonej z góry liczby lokalnych źródeł światła. Aktualnie projekt posiada jedynie oświetlenie globalne.
- Rzucanie przez obiekty cieni.

Wydajność otrzymanego rozwiązania jest bardzo trudna do oszacowania; najprostszym wskaźnikiem byłaby liczba trójkątów możliwych do narysowania w ciągu jednej sekundy, jednak z kilku powodów (między innymi złożoność shaderów, sposób rysowania trójkątów, liczba pokrywających się trójkątów) nie jest to miarodajna statystyka.

Rysunek 6.1: Wydajność w porównaniu z Three.js



W celu sprawdzenia, czy utworzony silnik jest pod tym względem konkurencyjny, można porównać jego wydajność z innym silnikiem w możliwie zbliżonym zastosowaniu. Do porównania użyto silnika *Three.js*. W obu silnika przygotowano możliwie identyczną scenę składającą się z różnej liczby modeli lądowników użytych w poprzednich przykładach, bez tekstur, z tymi samymi przekształceniami, w tej samej rozdzielczości. Zostały obliczone średnie czasy narysowania pierwszych 1000 klatek obrazu. Uzyskane wyniki prezentuje wykres na rysunku 6.1.

Widać na nim, że przygotowana biblioteka (nazwana "S3e" od *Simple 3d engine*) otrzymuje wyniki porównywalne z Three.js. Przy małej liczbie rysowanych obiektów są one nawet lepsze; prawdopodobnie jest to rezultatem narzutów obliczeniowych bardziej rozbudowanej biblioteki względem prostszej, które zmniejszają się wraz z dodawaniem kolejnych obiektów do sceny.

Wykonany projekt spełnił postawione założenia i wymagania projektowe oraz okazał się być konkurencyjny wydajnościowo z istniejącymi rozwiązaniami. Biblioteka dodatkowo przystosowana jest pod rozwój i umożliwia proste użycie na wielu poziomach. Dodatkowo praca nad projektem pozwoliła na przyswojenie przydatnej wiedzy z zakresu grafiki komputerowej.

Bibliografia

[1] M. Carbajal. Apollo lunar module. https://nasa3d.arc.nasa.gov/detail/lunarlandernofoil-c.

[2] D. Kanter. Tile-based rasterization in nvidia gpus. https://web.archive.org/web/20201122125549/https://www.realworldtech.com/tile-based-rasterization-nvidia-gpus/.

[3] H. S. Michael Bach, Thomas Meigen. Raster-scan cathode-ray tubes for vision research-limits of resolution in space, time and intensity, and some solutions. *Spatial Vision*, 10:403–414, 1997.

[4] R. Shrout. Amd vega gpu architecture preview: Redesigned memory architecture. https://web.archive.org/web/20201122125916/https://pcper.com/2017/01/amd-vega-gpu-architecture-preview-redesigned-memory-architecture/2/.

[5] Stackoverflow. Stackoverflow developer survey 2020. https://insights.stackoverflow.com/survey/2020#most-popular-technologies.

[6] C.-H. Sun, Y.-M. Tsao, K.-H. Lok, S.-Y. Chien. *Universal Rasterizer with edge equations and tile-scan triangle traversal algorithm for graphics processing units.* 2009.

[7] v8.dev blog. Elements kinds in v8. https://v8.dev/blog/elements-kinds.

[8] v8.dev blog. Firing up the ignition interpreter. https://v8.dev/blog/ignition-interpreter.

[9] W3C. *HTML 5.1 2nd Edition.*

[10] W3Techs. Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript/.

Zawartość płyty CD

Dołączona płyta CD zawiera pliki kodu źródłowego projektu. Aby móc objrzeć dołączone przykłady, będą potrzebne:

- środowisko uruchomieniowe Node.js 14
- menadżer paczek npm 6

Następnie należy wykonać kolejno polecenie `npm i` w celu zainstalowania potrzebnych zależności.

Aby zbudować projekt i zobaczyć dołączone przykłady, należy użyć `npm run examples`. Zbuduje ono projekt, uruchomi prosty serwer HTTP serwujący potrzebne pliki i otworzy listę przykładów w przeglądarce.

Chcąc zbudować bibliotekę do użycia we własnych projektach, należy użyć `npm run build`. Zbuduje ono projekt, a pliki wynikowe umieści w folderze `dist`.