

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: GRAFIKA I SYSTEMY MULTIMEDIALNE

PRACA DYPLOMOWA
INŻYNIERSKA

Interactive 3D visualization of geographical data
based on open sources using web technologies.

Interaktywna wizualizacja 3D danych
geograficznych z otwartych źródeł
z wykorzystaniem technologii webowych.

AUTOR:

Damian Koper

PROWADZĄCY PRACĘ:

Dr inż. Marek Woda

OCENA PRACY:

Spis treści

1. Wstęp	8
1.1. Istota rzeczy	8
1.1.1. Podejścia do tworzenia wizualizacji	9
1.2. Cel projektu i zawartość pracy	11
2. Wymagania	12
2.1. Twórca wizualizacji	13
2.1.1. Wymagania funkcjonalne	13
2.1.2. Wymagania niefunkcjonalne	13
2.2. Odbiorca wizualizacji	14
2.2.1. Wymagania funkcjonalne	14
2.2.2. Wymagania niefunkcjonalne	14
2.3. Aplikacja	14
2.3.1. Wymagania niefunkcjonalne	14
3. Silnik	15
3.1. WebGL i ESSL	15
3.1.1. Three.js	17
3.2. Praca kamery	19
3.2.1. Orbita globalna	19
3.2.2. Orbita lokalna	20
3.2.3. Parametry wspólne dla obu orbit	20
3.2.4. Obrót orbity globalnej	21
3.2.5. Obrót orbity lokalnej	23
3.2.6. Ograniczenia ruchu orbit	23
3.2.7. Tryb kompasu	25
3.2.8. Animacje - płynność ruchów	25
3.2.9. Macierze transformacji obiektów na podstawie orbit	26
3.3. Implementacja	26
3.3.1. GeoVisCore	26
3.4. Wizualizacja	30
3.4.1. Vue.js i Webpack	36
3.5. Podsumowanie	39
4. Wizualizacje	41
4.1. Gwiazdy	41
4.2. Atmosfera	42
4.3. Ziemia	46
4.4. Wybrane satelity	49
4.4.1. TLE	49

4.5. Aktywne satelity	51
4.6. Kafelki i Radar pogodowy	53
4.6.1. Kafelki	53
4.6.2. Implementacja	54
Literatura	59

Spis rysunków

1.1. Widok wizualizacji dwuwymiarowej na stronie <i>windy.com</i> wyświetlający informacje pogodowe na dwuwymiarowej mapie	9
1.2. Widok wizualizacji trójwymiarowej na stronie <i>earth.google.com</i>	10
3.1. Ogólny graf sceny wizualizacji	20
3.2. Schemat orbit w specyficznym przypadku dwóch wymiarów	21
3.3. Funkcja wygładzająca <i>cubicOut</i>	25
3.4. Zależności głównych komponentów Silnika	27
3.5. Diagram klas dla klasy <i>GeoVisCore</i>	28
3.6. Diagram klas dla klasy <i>TrackballController</i> i najważniejszych zależności	29
3.7. Diagram klas dla klasy <i>Visualization</i> i najważniejszych zależności	31
3.8. Diagram aktywności cyklu życia wizualizacji	34
3.9. Diagram sekwencji cyklu życia wizualizacji	35
3.10. Uproszczony łańcuch transformacji plików *.vue	37
3.11. Struktura komponentów <i>Vue.js</i> Silnika	38
3.12. Elementy kontrolne wizualizacji	38
3.13. Elementy kontrolne wizualizacji - otwarty panel dodatkowy	39
3.14. Elementy kontrolne wizualizacji - panel kontrolny wizualizacji	39
4.1. Wizualizacja gwiazd - klasa <i>StarsVis</i>	41
4.2. Wizualizacja atmosfery - klasa <i>AtmosphereVis</i>	43
4.3. Schemat elementów kluczowych dla wyliczenia parametrów atmosfery	44
4.4. Funkcja wygładzająca <i>expoIn</i>	45
4.5. Widok początkowy wizualizacji Ziemi o godzinie 19:50, 27.09.2020r.	46
4.6. Zachód słońca nad Europą o godzinie 18:54, 27.07.2020r.	47
4.7. Zależności pomiędzy klasami wizualizacji Ziemi	48
4.8. Wybrane satelity - klasa <i>IssVis</i>	51
4.9. Aktywne satelity - klasa <i>ActiveSatellitesVis</i>	52
4.10. Opady deszczu nad Europą - klasa <i>OsmTilesVis</i>	53
4.11. Opady deszczu nad Polską - klasa <i>OsmTilesVis</i>	54
4.12. Zależności pomiędzy klasami wizualizacji <i>OsmTilesVis</i>	55
4.13. Przykładowe rozwinięcie drzewa kafelków	57
4.14. Rozwinięcie drzewa kafelków - testowe kafelki	57

Spis tabel

2.1. Wymagania funkcjonalne zdefiniowane dla twórcy wizualizacji	13
2.2. Wymagania niefunkcjonalne zdefiniowane dla twórcy wizualizacji	13
2.3. Wymagania funkcjonalne zdefiniowane dla odbiorcy wizualizacji	14
2.4. Wymagania funkcjonalne zdefiniowane dla odbiorcy wizualizacji	14
2.5. Wymagania funkcjonalne zdefiniowane dla aplikacji	14

Spis listingów

1.1.	Konfiguracja podstawowej wizualizacji w bibliotece Cesium. Źródło [2].	10
3.1.	Pobranie kontekstu API WebGL do zmiennej	15
3.2.	Hello World w świecie grafiki 3D	18
3.3.	Fragmenty vertex shadera materiału MeshBasicMaterial	18
3.4.	Fragmenty części <code>project_vertex</code> vertex shadera	19
3.5.	Obsługa zdarzenia w języku JavaScript	30
3.6.	Obsługa zdarzenia w języku TypeScript z wykorzystaniem biblioteki <code>strongly-typed-events</code>	30
3.7.	Pusta klasa wizualizacji <code>EmptyVis</code> rozszerzająca klasę <code>Visualization</code>	33
4.1.	Fragmenty klasy <code>StarsVis</code>	42
4.2.	Fragmenty klasy <code>StarsVis</code>	43
4.3.	Modyfikacja fragment shadera materiału <code>MeshPhongMaterial</code>	49
4.4.	Fragmenty klasy <code>ActiveSatellitesVis</code>	52
4.5.	Fragmenty klasy <code>OsmTilesVis</code>	56

Skróty

GIS (ang. *Geographic Information System*)

API (ang. *Application Programming Interface*)

CPU (ang. *Central Processing Unit*)

GPU (ang. *Graphics Processing Unit*)

ESSL (ang. *OpenGL ES Shading Language*)

NDC (ang. *Normalized Device Coordinates*)

SFC (ang. *Single File Component*)

TLE (ang. *Two-Line Elements*)

NOARD (ang. *North American Aerospace Defense Command*)

Rozdział 1

Wstęp

Rzeczywistość otaczająca człowieka i jej aspekty są bardzo złożonym zagadnieniem. Człowiek w procesie jej poznawania może postawić się w różnych punktach odniesienia. Może obserwować rzeczywistość w skali wszechświata badając i poszerzając wiedzę na temat galaktyk oraz innych ciał niebieskich, gdzie Ziemia jest pomijalnie małym elementem. Może również obserwować świat w skali makro i mikroskopowej skupiając się na organizmach zamieszkujących i strukturach budujących planetę, schodząc również na poziom atomów i kwarków.

Większość obserwacji nie może być dokonana bezpośrednio przez człowieka. Nie może on bowiem objąć wzrokiem całej galaktyki, albo dostrzec poszczególnych atomów. Obrazowanie takich zjawisk musi być zaprezentowane w formie przystępnej dla człowieka wizualizacji zbudowanej z uwzględnieniem konkretnych aspektów danego przypadku.

Dobrze zbudowana wizualizacja danych, jaką jest chociażby prosty wykres punktowy, pozwala na ich analizę w lepszym stopniu i ułatwia wyciąganie wniosków. Dobrze skonstruowana wizualizacja, w przypadku prezentacji jej większemu gronu odbiorców, pozwala również na skuteczniejsze zainteresowanie grupy tematem oraz pomaga w opowiadaniu historii, a co za tym idzie, pozwala na wyciągnięcie przez odbiorców właściwych wniosków [36].

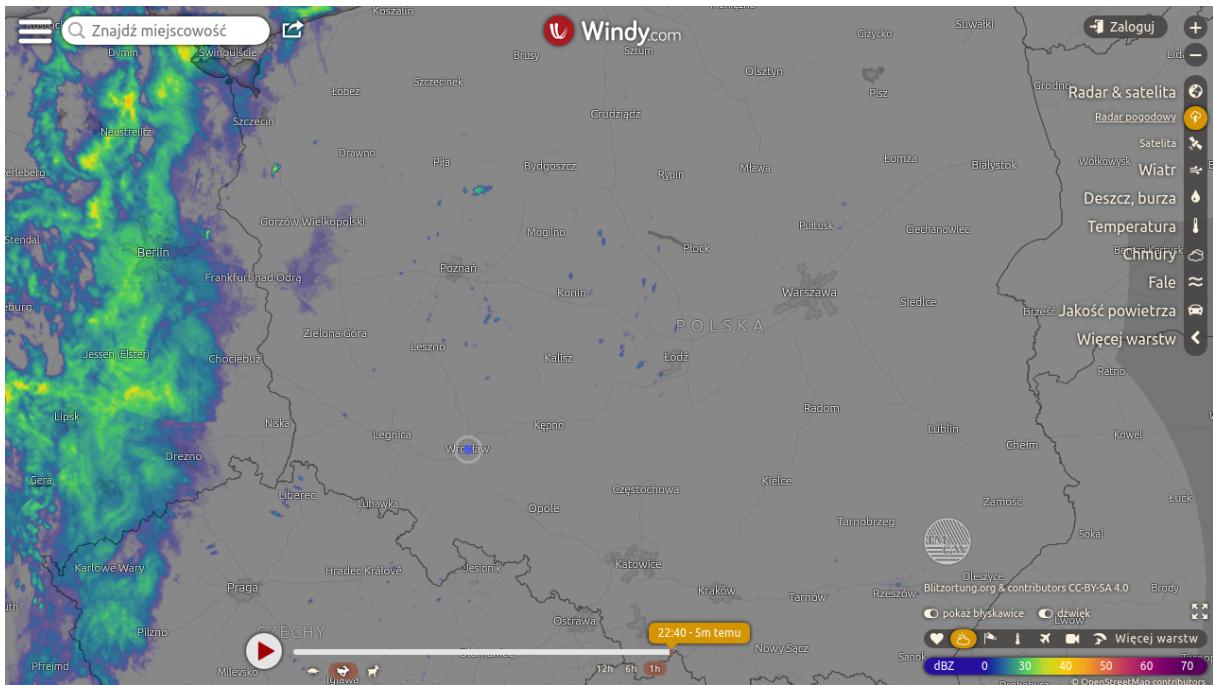
1.1. Istota rzeczy

Jednym z obszarów, w którym wizualizacje pełnią istotną rolę są reprezentacje zjawisk geograficznych oraz tych w bliskim sąsiedztwie Ziemi. Prezentowane dane mogą być związane z działalnością człowieka, bądź z obiektami i zjawiskami fizycznymi, którymi planeta się cechuje.

System, który zajmuje się wprowadzaniem, analizowaniem i wizualizacją danych geograficznych jest nazywany się Systemem informacji geograficznej (ang. geographic information system, **GIS**). Może on wyświetlać informacje z wielu źródeł ujęte w warstwy, które wyświetlane razem w różnych kombinacjach mogą nadawać danym różnego kontekstu. Każda wyświetlana informacja jest ściśle powiązana z pozycją na powierzchni Ziemi. [34, Rozdział 1.6].

Wizualizacje danych mogą dotyczyć się dowolnych zjawisk. GIS może obrazować podział terytorialny państw świata, jak i położenie obiektów kosmicznych w bliskim sąsiedztwie Ziemi. Korzystając z aktualizowanych na bieżąco źródeł danych wizualizacje mogą obrazować zjawiska zmienne, pokazywać stan obecny, przeszły (Rysunek 1.1), jak i prognozować przyszłość.

Ważnym czynnikiem odbiorze wizualizacji, jest jej przystępność dla użytkownika. Profesjonalne, skomplikowane systemy nierzadko cechują się złożonym interfejsem użytkownika. Duża liczba opcji pomaga łatwo uzyskać pożądane dane przez doświadczonego użytkownika, ale odstraszyć może niezagłębianego w temat odbiorcę. Przystępność odbioru wiąże się również z szybkością uzyskania dostępu do samej platformy obsługującej wizualizację. Alternatywą dla



Rys. 1.1: Widok wizualizacji dwuwymiarowej na stronie *windy.com* wyświetlający informacje pogodowe na dwuwymiarowej mapie

instalowanych aplikacji desktopowych jest przeglądarka internetowa. Tworzy ona środowisko, które może być uruchomione na wielu systemach operacyjnych, również na urządzeniach mobilnych, a zaimplementowane wspomagane sprzętowe generowanie grafiki i interfejsy takie jak *HTML5 Canvas*[6] i *WebGL*[31] czynią ją potężnym narzędziem do wydajnego wyświetlania złożonych grafik. Aplikacje webowe oczywiście nie będą nigdy dorównywać profesjonalnym aplikacjom dedykowanym konkretnej platformie, jednak stanowią ich dobrą i ogólnodostępną alternatywę.

Innym kryterium definiującym wizualizację jest jej interaktywność. Definiuje ono w jakim stopniu użytkownik może dostosować wyświetlany widok, zarządzać warstwami, sterować położeniem kamery, czy też wyszukiwać informacje. Dwuwymiarowy widok mapy (Rysunek 1.1) pozwala jednoznacznie odnieść informacje z różnych warstw do konkretnego miejsca na planecie. Z kolej widok trójwymiarowy (Rysunek 1.2) pozwala na obserwację sceny z różnych perspektyw, pokazuje kulistość Ziemi i redukuje efekty zniekształcenia danych związanych z techniką rzutowania sfery na płaszczyznę. Przy kamerze skierowanej prostopadle do płaszczyzny powierzchni, oraz w bliskim powiększeniu widok taki jest porównywalny do widoku dwuwymiarowego. Czynniki te zdaniem autora pracy czynią taką wizualizację bardziej atrakcyjną dla ogólnego odbiorcy. Oczywiście wybór techniki wizualizacji zawsze zależy od konkretnego przypadku, jak i od oczekiwanej wydajności, gdyż złożoność generowania grafiki w przypadku wizualizacji trójwymiarowych jest z reguły większa.

1.1.1. Podejścia do tworzenia wizualizacji

Zadaniem twórcy wizualizacji jest zebranie i przetworzenie danych na formę grafiki dwu lub trójwymiarowej. Od używanego systemu informacji przestrzennej zależy w jaki sposób definiowana jest wizualizacja i skutkiem tego, jaki poziom wiedzy i umiejętności z danej dziedziny jest potrzebny do jej stworzenia. System w definicji wizualizacji opiera się na swoich założeniach. Aplikacje uruchamiane bezpośrednio w środowisku systemu operacyjnego mogą być wyposażone w rozbudowane kreatory i edytory, które zaspokajają wymagania użytkowników.



Rys. 1.2: Widok wizualizacji trójwymiarowej na stronie earth.google.com

Pozwalają skupić się na zagadnieniach domenowych, na poprawności i dokładności wizualizacji zamiast na aspektach generowania grafiki.

W środowisku przeglądarki internetowej do tworzenia wizualizacji nie stosuje się zwykle rozbudowanych edytorów graficznych i formularzy. Biblioteki wyświetlające dane geoprzezstrzenne konfigurowalne są zwykle z poziomu języka JavaScript. Przykładem takiej biblioteki jest Cesium [3]. Potrafi ona generować wizualizacje dwu i trójwymiarowej różnego rodzaju danych, a jej konfiguracja następuje poprzez jej API, które dostarcza, ale też ogranicza jej możliwości (listing 1.1).

Listing 1.1: Konfiguracja podstawowej wizualizacji w bibliotece Cesium. Źródło [2].

```
<script>
    Cesium.Ion.defaultAccessToken = 'your_access_token';
    var viewer = new Cesium.Viewer('cesiumContainer', {
        terrainProvider: Cesium.createWorldTerrain()
    });

    var tileset = viewer.scene.primitives.add(
        new Cesium.Cesium3DTileset({
            url: Cesium.IonResource.fromAssetId(your_asset_id)
        })
    );
    viewer.zoomTo(tileset);
</script>
```

Jeszcze innym podejściem, możliwym do zastosowania w przypadku aplikacji i desktopowych, i webowych, jest dostarczenie twórcy tylko podstawowych abstrakcji (najczęściej interfejsów programistycznych) wizualizacji takich jak sterowanie kamerą, przekazywanie zdarzeń pochodzących od odbiorcy, czy interfejs służący do generowania obiektów na scenie dwu lub trójwymiarowej. Podejście daje to najwięcej możliwości, ale z drugiej strony wymaga posiadania największej wiedzy o funkcjonowaniu dostarczonych interfejsów.

W każdym wypadku istotnym czynnikiem ułatwiającym tworzenie wizualizacji jest dostarczona przez narzędzia i biblioteki interaktywna dokumentacja. Powinna ona dobrze opisywać

dostarczone rozwiązania ze strony praktycznej i przez swoją interaktywność ułatwiać poruszanie się po niej użytkownikowi.

1.2. Cel projektu i zawartość pracy

Celem opisywanego projektu jest stworzenie biblioteki umożliwiającej definiowanie i wyświetlanie trójwymiarowych wizualizacji w środowisku przeglądarki internetowej. Projekt zakłada również stworzenie aplikacji webowej, która za pomocą osadzonej w niej stworzonej biblioteki, umożliwia zarządzanie wyświetlaniem dostarczonych wizualizacji.

Rozdział drugi pracy opisuje szczegółowe wymagania postawione przed poszczególnymi komponentami aplikacji. Rozdział trzeci opisuje projekt i implementację komponentu Silnika wyświetlającego wizualizację, a rozdział czwarty przedstawia implementację przykładowych wizualizacji, które możliwe są do zdefiniowania korzystając z interfejsów dostarczonych przez Silnik. W rozdziale piątym opisana jest Aplikacja korzystająca z komponentu Silnika zbierająca wizualizacje i umożliwiająca filtrowanie i przełączanie się pomiędzy nimi. Rozdział szósty opisuje sposoby testowania zaimplementowanych rozwiązań, a rozdział siódmy przedstawia używane w projekcie biblioteki pomocnicze wraz z ich krótkim opisem. Rozdział ósmy podsumowuje całość projektu i zwraca uwagę na problemy napotkane podczas implementacji, możliwości optymalizacji i alternatywne rozwiązania poruszanych wcześniej kwestii projektowych i implementacyjnych.

Rozdział 2

Wymagania

Ze względu na możliwy podział funkcjonalności projektu na wiele typów, zdefiniowano następujące pojęcia:

1. Silnik - komponent odpowiedzialny za definicję i wyświetlenie wizualizacji.
2. Wizualizacja - konfigurowalny widok przedstawiający obiekty, których położenie zdefiniowano za pomocą współrzędnych geograficznych, na powierzchni sfery.
3. Aplikacja - uruchomiona w przeglądarce użytkownika strona umożliwiająca wybór i wyświetlenie wizualizacji.

Silnik dostarcza komponenty i interfejs programistyczny, dzięki którym można definiować, wyświetlać i zarządzać wizualizacją. Pozwala także na zdefiniowane wielu niezależnych wizualizacji. Z tego powodu można wyróżnić dwa typy użytkowników:

1. Twórcę wizualizacji,
2. Odbiorcę wizualizacji.

Wymagania aplikacji zostały zdefiniowane z podziałem na typ użytkownika. Struktura danych definiująca renderowany obraz, zwana dalej sceną.

2.1. Twórca wizualizacji

2.1.1. Wymagania funkcjonalne

Numer	Wymaganie
RA_1	Twórca może zdefiniować metadane wizualizacji określone przez interfejs Silnika.
RA_2	Twórca może zdefiniować statyczną scenę określając położenie obiektów na sferze z wykorzystaniem długości i szerokości geograficznej.
RA_3	Twórca do definicji sceny może wykorzystać interfejs tworzenia obiektów dostarczony przez aplikację lub załadować obiekty, materiały i tekstury z zewnętrznego źródła.
RA_4	Twórca może zagnieźdzać sceny predefiniowane w silniku, oraz sceny wcześniejsze stworzone przez siebie.
RA_5	Twórca może parametryzować sceny w celu określonej ich modyfikacji w procesie zagnieźdzania.
RA_6	Twórca może określić parametry początkowe obserwatora, dynamikę i zakres jego ruchów: 1. położenie, 2. prędkość i przyspieszenie ruchu, 3. ograniczenie przybliżenia, 4. ograniczenie pozycji.
RA_7	Twórca może zdefiniować wygląd i funkcjonalność panelu kontrolnego. Panel ten służyć będzie do zmiany parametrów wizualizacji i obsługiwany będzie przez odbiorcę.
RA_8	Twórca, poprzez interfejs programistyczny dostarczony przez silnik, może aktualizować scenę w dowolnym momencie, określonym przez siebie w definicji wizualizacji.
RA_9	Twórca może definiować zachowania, które będą odpowiedziały na zdarzenia związane z poruszaniem się po scenie generowane przez odbiorcę.

Tab. 2.1: Wymagania funkcjonalne zdefiniowane dla twórcy wizualizacji

2.1.2. Wymagania niefunkcjonalne

Numer	Wymaganie
RA_10	Silnik powinien definiować i w sposób jasny przekazywać potencjalnemu twórcy akceptowalną strukturę danych, plików i katalogów, określającą jedną wizualizację.
RA_11	Włączenie zdefiniowanej wizualizacji do ich zbioru w aplikacji powinno ustalone być tylko w jednym miejscu poprzez prosty interfejs.
RA_12	Dane wizualizacji muszą być ładowane asynchronicznie. Dane źródłowe definiujące scenę mogą być przetwarzane po stronie odbiorcy lub być przetworzone wcześniej i pobrane.

Tab. 2.2: Wymagania niefunkcjonalne zdefiniowane dla twórcy wizualizacji

2.2. Odbiorca wizualizacji

2.2.1. Wymagania funkcjonalne

Numer	Wymaganie
RU_1	Odbiorca może zobaczyć dane dostępnych wizualizacji.
RU_2	Odbiorca może wyświetlić wybraną wizualizację.
RU_3	Odbiorca może poruszać się po wizualizacji, zmieniając położenia kamery, używając myszki lub klawiatury.
RU_4	Odbiorca może zobaczyć orientację kamery relatywnie do kierunku północnego i ją zresetować.
RU_5	Odbiorca może wyświetlić lub ukryć panel sterujący wizualizacją dostarczony przez twórcę.

Tab. 2.3: Wymagania funkcjonalne zdefiniowane dla odbiorcy wizualizacji

2.2.2. Wymagania niefunkcjonalne

Numer	Wymaganie
RU_6	Każda akcja użytkownika związana ze sterowaniem kamerą może zostać wykonana używając myszki lub równolegle klawiatury.

Tab. 2.4: Wymagania funkcjonalne zdefiniowane dla odbiorcy wizualizacji

2.3. Aplikacja

2.3.1. Wymagania niefunkcjonalne

Numer	Wymaganie
RU_7	Aplikacja powinna być stroną typu <i>Single Page Application</i> .
RU_8	Jeśli to możliwe aplikacja powinna wykorzystywać sprzętową akcelerację obliczeń graficznych.
RU_9	Aplikacja powinna ustawiać i obsługiwać adres URL w przeglądarce definiujący wyświetlana wizualizację.

Tab. 2.5: Wymagania funkcjonalne zdefiniowane dla aplikacji

Rozdział 3

Silnik

Rozdział ten opisuje główny komponent tworzonego systemu nazwanego Silnikiem. Odpowiadający jest on za dostarczenie interfejsu definiowania trójwymiarowej wizualizacji i jej późniejsze wyświetlanie. Narzuca również sposób pracy kamery i umożliwia konfigurację jej parametrów.

Wszystkie obiekty wyświetlane na scenie, razem z definicją ich wyglądu, tekstur i dynamiki ruchów dostarcza wizualizacja. Jej obiekty mogą reagować na zdarzenia, które generuje użytkownik. Wywołanie zdefiniowanych procedur obsługi tych zdarzeń również leży w gestii komponentu Silnika.

Najpierw w sposób uproszczony opisany zostanie sposób renderowania grafiki z wykorzystaniem API WebGL oraz biblioteki Three.js. Następnie przedstawione zostaną mechanizmy sterujące pracą kamery, a następnie implementacja komponentu Silnika i opisanych mechanizmów.

3.1. WebGL i ESSL

WebGL jest dostępnym z poziomu języka JavaScript API pozwalającym na renderowanie grafiki trójwymiarowej w przeglądarce. Złożone obiekty rysowane są tylko za pomocą punktów, linii i trójkątów. WebGL działa w trybie *immediate*, który to wymusza na aplikacji wykonywanie bezpośrednio niskopoziomowych komend rysujących podstawowe obiekty 3D. Aplikacja korzystająca z WebGL musi sama definiować abstrakcje podstawowych obiektów takich jak scena, kamera, czy światło. Podejście to jest bardzo elastyczne i pozwala na optymalizację implementowanych rozwiązań w zależności od potrzeb[35, Rozdział 1]. WebGL korzysta z akceleracji sprzętowej podczas renderowania grafiki - działa na GPU. W przypadku kart graficznych bez wsparcia dla tej technologii przeglądarki Google Chrome i Internet Explorer 11 umożliwiają rysowanie z użyciem CPU.

Drugim podejściem do renderowania grafiki jest podejście *retained*, gdzie biblioteki z niego korzystające implementują swoją abstrakcję sceny i same zajmują się jej rysowaniem. Przykładem takiej biblioteki jest Windows Presentation Foundation[16].

Dostęp do API WebGL uzyskać można poprzez kontekst elementu Canvas. Na listingu 3.1 pokazano pobranie kontekstu API WebGL do zmiennej gl. Wszystkie interakcje związane z użyciem API będą odbywały się z użyciem pobranego obiektu kontekstu. Numer w identyfikatorze 'webgl2' mówi, że używamy WebGL w wersji drugiej.

Listing 3.1: Pobranie kontekstu API WebGL do zmiennej

```
const canvas = document.getElementById('vis-container');
const gl = canvas.getContext('webgl2');
```

Obiekt kontekstu działa jak maszyna stanów. Przechowuje ustwiony stan do czasu jego zmiany przez aplikację. Wszystkie operacje renderowania grafiki korzystają z globalnie ustawionych parametrów, które definiują stan kontekstu i mają bezpośredni wpływ na efekt końcowy[35, Rozdział 1].

Rysowanie sceny

Rysowanie obiektu rozpoczyna się od utworzenia buforów danych i umieszczenia w nich współrzędnych wierzchołków oraz kolejności, według której wierzchołki mają brać udział w procesie rysowania. Kolejność ma istotne znaczenie w przypadku różnych trybów rysowania oraz, Cullingu czyli określania widocznej strony rysowanego trójkąta. Bufory są reprezentowane zewnętrznie jako tablice `TypedArray`. Przechowują one jedynie surowe dane w postaci binarnej [8]. W języku JavaScript występuje jeden typ `number` przechowujący liczby, które wewnętrznie reprezentowane są jako 64b liczba zmienoprzecinkowa. Dodatkowo każda zmienna numeryczna jest obiektem typu `Number` z własnymi metodami. Użycie buforów z interfejsem tablicy przyspiesza masowe operacje na danych.

Shadery

W WebGL'u *programem* nazywane są skompilowane przez kontekst shadery. Są to krótkie programy napisane w specjalistycznym języku, którym w przypadku WebGL'a jest ESSL (ang. OpenGL ES Shading Language). Przypomina on składnię języka C/C++ [21] i zawiera wbudowane funkcje wymagane do operacji matematycznych takich jak iloczyn skalarny wektorów, czy mnożenie macierzy. Na wspomniany *program* składają się dwa podprogramy (shadery) - `vertex shader` i `fragment shader`. `Vertex shader`, uruchamiany jako pierwszy, pobiera dane o wierzchołkach z buforów, oraz korzystając ze stałych (`uniforms`) oblicza finalną pozycję wierzchołka. W większości przypadków shader ten odpowiada również za obliczenie innych parametrów wierzchołka takich jak kolor, jego wektor normalny, czy też współrzędne tekstur. Dla każdego wierzchołka wyliczone wartości wysyłane są dalej do shadera `fragment shader`.

`Fragment shader` odpowiada za wyliczenie koloru pojedynczego pixela. Dane wysłane z `vertex shader`'a w zmiennych typu `varying` są automatycznie interpolowane dla każdego punktu w renderowanym trójkącie na podstawie trzech wierzchołków.

W shaderach, po dostarczeniu odpowiednich danych, realizowane są abstrakcje takie jak kamera, oświetlenie, czy materiały.

Obliczanie finalnej pozycji wierzchołków

W grafice 3D każdy model reprezentowany jest przez zbiór punktów i informacji o kolejności ich rysowania. Model może mieć swoją pozycję w świecie 3D, a obserwator może znajdować się w różnych miejscach sceny. WebGL sam w sobie nie posiada abstrakcji kamery i do wyświetlenia sceny z konkretnej perspektywy konieczne jest przemieszczenie wszystkich wierzchołków geometrii. Transformacja pozycji wierzchołków odbywa się za pomocą przekształceń afanicznych, które transformują pozycję zbioru wierzchołków i nie zaburzają relacji przestrzennych pomiędzy nimi. Efektywnie transformacja taka jest mnożeniem macierzy transformacji o wymiarach 4×4 i wektora z dodaną czwartą współrzędną równą 1, co daje nowy wektor współrzędnych wierzchołka.

Przekształcenia związane z pozycją modelu i kamery w świecie wyrażane są za pomocą macierzy. Macierzowy opis przekształceń możliwy jest dzięki zastosowaniu współrzędnych jednorodnych[33]. Transformacja pozycji modelu odbywa się z pomocą macierzy M , a trans-

formacja pozycji związana z położeniem kamery z pomocą macierzy widoku V . Wyliczanie współrzędnych wierzchołka w układzie współrzędnych świata pokazano w równaniu 3.1.

Aby uzyskać wyjściową pozycję piksela na ekranie konieczne jest pomnożenie macierzy projekcji i wektora pozycji wierzchołka w układzie współrzędnych świata (równanie 3.2). Macierz projekcji odpowiada za transformację współrzędnych wierzchołka do sześcianu o wymiarach $2 \times 2 \times 2$ i środku w punkcie $(0, 0, 0)$. Transformacja ta może być perspektywiczna, gdzie przekształceniu ulega przestrzeń w kształcie ostrosłupa ściętego. Może być też ortograficzna, gdzie przekształceniu ulega przestrzeń w kształcie prostopadłościanu. Punkty leżące poza tą przestrzenią nie są rysowane. Współrzędne (x, y) transformowanych wierzchołków są współrzędnymi NDC (ang. Normalized Device Coordinates), niezależnymi od urządzenia. Dzięki temu mogą być one łatwo przekształcone na piksele elementu `Canvas`, gdzie punkt $(0, 0)$ znajduje się w lewem górnym rogu. Podejście to uniezależnia generowanie pikseli od elementu wyświetlającego, do którego trzeba dostosować tylko sposób przekształcenia współrzędnych NDC .

$$p' = VM \cdot \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \quad (3.1)$$

gdzie:

M — macierz transformacji pozycji modelu

V — macierz transformacji widoku

p' — wektor pozycji wierzchołka w układzie współrzędnych widoku

p — wektor pozycji modelu w układzie współrzędnych świata

$$v = P \cdot \begin{bmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{bmatrix} \quad (3.2)$$

gdzie:

P — macierz projekcji

p' — wektor pozycji wierzchołka w układzie współrzędnych widoku

Kalkulacja pozycji modeli oraz kamery ma szczególne znaczenie przy złożonym zachowaniu kamery oraz sceny w komponencie Silnika.

3.1.1. Three.js

Three.js[22] jest biblioteką 3D, która domyślnie do renderowania grafiki używa WebGL. Ułatwia ona rozpoczęcie pracy z grafiką 3D i jednocześnie nie nakłada ograniczeń związanych z niskopoziomową konfiguracją wyświetlanej sceny. Pozwala ona na opisanie sceny, obiektów, świata i materiałów w postaci obiektoowej. Posiada rozbudowany system animacji oraz wsparcie dla systemów wirtualnej rzeczywistości. Na listingu 3.2 pokazano kod aplikacji, która wyświetla zielony sześcian.

Na początku tworzony jest obiekt sceny, który jest kontenerem na pozostałe wyświetlane obiekty oraz światła. Następnie tworzony jest obiekt kamery, który definiuje właściwości, w tym wypadku, projekcji perspektywicznej. Utworzony dalej obiekt `THREE.WebGLRenderer` odpowiedzialny jest za utworzenie i przechowywanie referencji do obiektu `Canvas`, na którym, w głównej pętli programu, rysuje dostarczoną scenę z perspektywy wybranej kamery. Odpowiada za to wywołanie `renderer.render(scene, camera)`.

Geometrię kostki definiuje obiekt `THREE.BoxGeometry`, która z domyślnymi argumentami konstruktora jest sześcianem o wymiarach $1 \times 1 \times 1$. Obiekt ten posiada atrybuty ułatwiające zarządzanie wygenerowaną geometrią. Zwykłe obiekty geometrii są konwertowane do typu

Listing 3.2: Hello World w świecie grafiki 3D

```

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
    ↪ window.innerHeight, 0.1, 1000 );

const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );

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

camera.position.z = 5;

function animate() {
    requestAnimationFrame( animate );
    renderer.render( scene, camera );
}
animate();

```

BufferGeometry w procesie renderowania. Wtedy dane wierzchołków są umieszczane w buforach, które mogą być bezpośrednio wykorzystane w interakcji z WebGL'em. Three.js pozwala tworzyć geometrię w sposób bardziej efektywny, jednak gorzej zarządzalny, wykorzystując klasy pochodne klasy BufferGeometry, takie jak BoxBufferGeometry.

Elementy wyglądu rysowanych geometrii określa materiał. W Three.js obiektami je reprezentujące są pochodne klasy Material. Umożliwiają ustawienie koloru, tekstur, różnego rodzaju map, a w przypadku światła parametry jego interakcji z powierzchnią obiektu. W procesie rysowania obiektu, atrybuty jego materiału, oraz atrybuty obiektów ważnych dla wyglądu rysowanego obiektu, na przykład światel, są wysyłane do shaderów w postaci stałych (**uniforms**). Sam materiał definiuje jednoznacznie działanie shaderów, wykorzystanych w procesie jego rysowania. Przykład shadera dla materiału MeshBasicMaterial pokazano na listingu 3.3.

Listing 3.3: Fragmenty vertex shadera materiału MeshBasicMaterial

```

#include <common>
/* ... */

void main() {
    /* ... */
    #include <color_vertex>
    /* ... */

    #include <begin_vertex>
    /* ... */
    #include <project_vertex>
    /* ... */
}

```

Shadery różnych materiałów współdzielą pomiędzy sobą wiele swoich części. Zastosowana dyrektywa `#include` pozwala na umieszczenie w kodzie wspólnych ich części. Na listingu 3.4, w części `project_vertex`, widać właściwy proces obliczania pozycji wierzchołka przedstawiony na równaniach 3.1 i 3.2. Macierz projekcji mnożona jest przez połączoną macierz modelu i widoku oraz zmienną wektorową `mvPosition`. Wynikowy wektor wpisywany jest

do specjalnej zmiennej globalnej `gl_Position`, której zawartość informuje resztę składowych procesu generowania grafiki o wyniku kalkulacji.

Listing 3.4: Fragmenty części `project_vertex` vertex shadera

```
vec4 mvPosition = vec4( transformed, 1.0 );
/* ... */
mvPosition = modelViewMatrix * mvPosition;
gl_Position = projectionMatrix * mvPosition;
```

Three.js dostarcza również wiele narzędzi ułatwiających operacje matematyczne na wektorach oraz macierzach. Pozwala między innymi na interpolację liniową i sferyczną wektorów, generowanie macierzy transformacji, czy reprezentowanie obrotów za pomocą kątów Eulera lub kwaternionów.

3.2. Praca kamery

Komponent Silnika wyświetla scenę, w której kamera orbituje wokół jednego punktu. Dodatkowo użytkownik może zmienić orientację kamery względem punktu na powierzchni sfery. Opis pracy kamery odnosić się będzie do obiektu sfery i jej powierzchni, jednak nic nie stoi na przeszkodzie, aby kamera orbitowała wokół innego obiektu. W tym podrozdziale opis mechanizmów jest przedstawiony w oderwaniu od ich implementacji w projekcie. Finalnie, pozycję kamery relatywnie do środka sfery opisują dwie orbity. Orbitą, w kontekście pracy kamery, nazwana została para wektorów określająca obrót od wektora odniesienia i odległość od punktu jego zaczepienia.

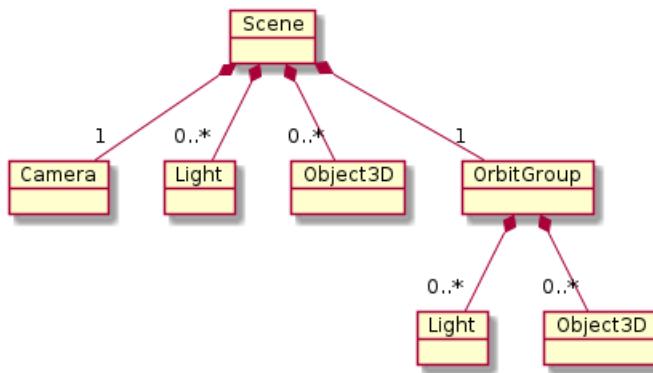
Wspomniane orbity nazwano orbitą lokalną i orbitą globalną. Orbita globalna odpowiedzialna jest za pozycję kamery nad punktem obiektu sfery. Orbita lokalna określa orientację kamery względem punktu, nad którym się znajduje. Taki podział sceny wprowadza również dwa układy odniesienia.

1. Układ obserwatora - układ, w którym znajduje się obserwator i wszystkie obiekty są pozyjonowane relatywnie do obiektu kamery. Jest to domyślny układ renderowanej sceny.
2. Układ wizualizacji - układ, w którym obiekty pozyjonowane są relatywnie do mogącej obracać się sfery. Obiekty umieszczane są w obracającej się grupie.

Żeby przybliżyć zależności pomiędzy tymi układami, można posłużyć się przykładem. Aby symulować cykl dnia i nocy, światło musi być pozyjonowane w układzie wizualizacji, ponieważ jest niezależne od ruchu kamery. Aby światło oświetlało zawsze widoczną stronę planety, musi być ono pozyjonowane w układzie obserwatora. Ogólny graf sceny przedstawiono na diagramie 3.1.

3.2.1. Orbita globalna

Orbitę globalną definiują dwa wektory - \vec{g}_v i \vec{g}_{up} na rysunku 3.2. Pierwszy rozciągnięty jest od środka s sfery do punktu c_g , wokół którego orbituje kamera. Drugi jest wektorem jednostkowym do niego prostopadłym określającym orientację sfery w osi pierwszego wektora. W późniejszym opisie działanie *na orbicie*, na przykład obrót orbity, oznacza wykonanie tej samej transformacji na obu wektorach. W ten sposób oba wektory nigdy nie zmieniają swojej wzajemnej orientacji.



Rys. 3.1: Ogólny graf sceny wizualizacji

Użytkownik za pomocą myszy lub klawiatury może obrócić sferę, a konkretne grupę obrotu (OrbitGroup na diagramie 3.1) i zawierane przez nie obiekty. Jest to efektywnie zmianą punktu, nad którym znajduje się kamera, pomimo tego, że jej pozycja się nie zmienia. Jako, że obrót sfery definiowany jest abstrakcją orbity, cała operacja sprowadza się do jej odpowiedniego obrócenia. Parametrami specyficznymi dla orbity globalnej są:

1. Tryb pracy orbity - określa czy podczas przesuwania orbity ma ona zachowywać swoją orientację w kierunku północnym. Wydzielono tryb *swobodny* i *kompas*.

3.2.2. Orbita lokalna

Orbitę lokalną, podobnie jak globalną, definiują dwa wektory - \vec{l}_v i \vec{l}_{up} na rysunku 3.2. Pierwszy rozciągnięty jest od punktu c_g , wokół którego orbituje kamera, do kamery (punkt c_l). Drugi jest wektorem jednostkowym do niego prostopadłym określającym orientację kamery w osi pierwszego wektora.

Użytkownik za pomocą myszy lub klawiatury może zmienić punkt orbitowania kamery. Jest to efektywnie zmianą położenia kamery w układzie obserwatora. Jako, że pozycja kamery definiowany jest przez abstrakcję orbity, cała operacja sprowadza się do jej odpowiedniego obrócenia. Długość wektora \vec{l}_v reprezentuje odległość obserwatora do punktu na powierzchni sfery.

Parametrami specyficznymi dla orbity lokalnej są:

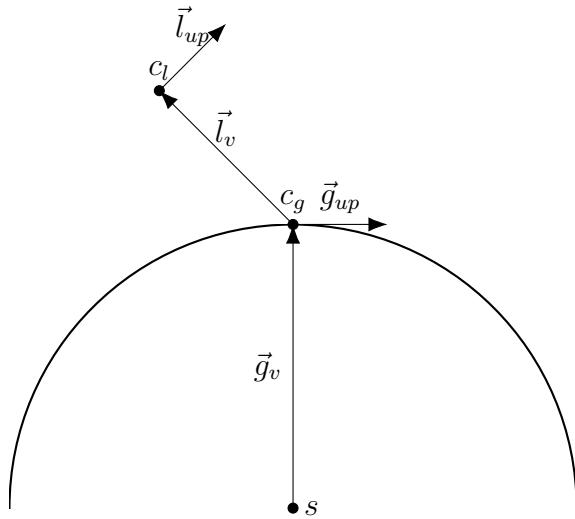
1. współczynnik przybliżenia - jak powinna zmienić się odległość kamery od punktu na powierzchni sfery podczas jednej akcji przybliżenia.
2. granice przybliżenia - minimalna i maksymalna odległość kamery od punktu na powierzchni sfery.

3.2.3. Parametry wspólne dla obu orbit

Dla obu orbit wyróżniono wspólne parametry. Są nimi:

1. granice - wyrażony w radianach zakres współrzędnych geograficznych definiujący fragment sfery, nad którym kamera może się znaleźć. Może służyć na przykład do ograniczenia obszaru poruszania się użytkownika tylko do jednej półkuli lub jednego miasta.
2. prędkość obrotu - współczynnik sterujący prędkością obrotu danej orbity.

Wszystkie parametry, ogólne i te specyficzne dla każdej z orbit mogą być konfigurowane przez wizualizację.



Rys. 3.2: Schemat orbit w specyficzny przypadku dwóch wymiarów

3.2.4. Obrót orbity globalnej

Algorytm obrotu orbity globalnej wykonywany jest dla każdego zdarzenia przesunięcia myszy użytkownika podczas gestu chwycenia, przeciągnięcia i upuszczenia wygenerowanym przez element Canvas. Obrót wymaga obliczenia jego chwilowej osi i kąta.

Kwaterniony

Kwaterniony są rozszerzeniem liczb zespolonych [14]. Mają one postać:

$$q = a + bi + cj + dk : a, b, c, d \in \mathbb{R} \quad (3.3)$$

gdzie, podobnie jak w przypadku liczb zespolonych, zachodzi zależność:

$$i^2 = j^2 = k^2 = ijk = -1 \quad (3.4)$$

Ich mnożenie następuje, z uwzględnieniem zależności z równania 3.4, tak jak mnożenie wielomianów.

Kwaternion może być interpretowany jak suma skalaru z wektorem, gdzie współczynnik a jest skalarem, a b, c i d są współrzędnymi wektora. Kwaterniony jednostkowe służą między innymi do reprezentacji obrotów w przestrzeni 3D. Rozwiązuje ono problemy związane z reprezentacją obrotów poprzez kąty Eulera. Obrót taki jest zdefiniowany poprzez obroty wokół każdej z osi układu współrzędnych. Obracanie z ich użyciem, w pewnych kombinacjach obrotu może skutkować efektem gimbal-lock, który powoduje zanikiem stopnia swobody obrotu. Dlatego w tym przypadku ważna jest kolejność obrotów. Stosując kąty Eulera niemożliwa jest bezpośrednią interpolację sferyczną pomiędzy dwoma zdefiniowanymi obrotami.

Konstrukcja kwaternionu definiującego dany obrót jest następująca:

$$q = \cos \frac{\theta}{2} + (u_x i + u_y j + u_z k) \sin \frac{\theta}{2} \quad (3.5)$$

gdzie:

θ — kąt obrotu

u — wektor jednostkowy definiujący oś obrotu

i, j, k — jednostki ujęte kwaternionu

Obrót wektora jest sprowadzeniem wektora do kwaternionu ze współczynnikiem $a = 0$ i pomnożeniem kwaternionu q , wektora i sprzężenie tego kwaternionu q^{-1} (równania 3.6 i 3.7).

$$q^{-1} = \cos \frac{\theta}{2} - (u_x i + u_y j + u_z k) \sin \frac{\theta}{2} \quad (3.6)$$

gdzie: θ — kąt obrotu

u — wektor jednostkowy definiujący oś obrotu

i, j, k — jednostki urojone kwaternionu

$$v' = q v q^{-1} \quad (3.7)$$

gdzie: v' — wektor obrócony

q — kwaternion definiujący obrót

q^{-1} — sprzężenie kwaternionu q

Składanie obrotów w przypadku zapisu macierzowego transformacji następuje poprzez ich pomnożenie. To samo ma miejsce w przypadku kwaternionów, gdzie kolejność mnożenia decyduje o odwrotnej kolejności złożenia obrotów.

Obrót orbity

Obrót orbity globalnej odbywa się relatywnie do pozycji kamery na orbicie lokalnej. W wyliczaniu osi obrotu muszą wziąć udział więc wektory obu orbit. Niech \vec{g}_v i \vec{g}_{up} oraz \vec{l}_v i \vec{l}_{up} będą wektorami orbit kolejno globalnej i lokalnej zgodnymi z rysunkiem 3.2. Końcowy obrót orbity globalnej jest złożeniem obrotu w kierunku pionowym i poziomym relatywnym do widoku obserwatora. Osie tych obrotów wyrażają wektory:

$$\vec{h}_g = \vec{l}_{up} \times \vec{l}_v \quad (3.8)$$

$$\vec{v}_g = \vec{l}_{up} \quad (3.9)$$

gdzie: \vec{h}_g — oś obrotu zorientowana poziomo
 \vec{v}_g — oś obrotu zorientowana pionowo

Niech $\vec{d} \in \mathbb{R}^2$ będzie przesunięciem widoku pochodząącym ze zdarzenia wygenerowanego przez użytkownika, wyrażonym w pikselach. Współczynnik warunkujący kąt przesunięcia wyrażony jest wzorem:

$$s = 0.001 \cdot G_s \cdot \|\vec{l}_v\| \cdot \|\vec{g}_v\| \quad (3.10)$$

gdzie: G_s — konfigurowany współczynnik obrotu orbity

Kąty obrotu wyrażone w radianach wyrażone są wzorami:

$$\theta_x = s \cdot d_x \quad (3.11)$$

$$\theta_y = s \cdot d_y \quad (3.12)$$

gdzie: θ_x — kąt obrotu w osi pionowej, przesuwa widok w osi OX

θ_y — kąt obrotu w osi poziomej, przesuwa widok w osi OY

Niech $Q : (\mathbb{R}, \mathbb{R}^3) \rightarrow \mathbb{H}$ będzie funkcją konstrującą kwaternion na podstawie kąta i osi obrotu wyrażoną wektorem jednostkowym. Kwaternion chwilowego obrotu orbity globalnej wyrażony jest wtedy wzorem 3.13.

$$q = Q(\theta_x, \frac{\vec{v}}{\|\vec{v}\|}) \cdot Q(\theta_y, \frac{\vec{h}}{\|\vec{h}\|}) \quad (3.13)$$

Dla każdego zdarzenia przeciągnięcia widoku wygenerowane przez użytkownika dostarczany jest nowy wektor \vec{d} . Orbita globalna jest obracana na podstawie obliczonego kwaternionu q . Następnie zachodzi potrzeba korekty owego obrotu związana z trybem kamery i ustalonymi ograniczeniami jej ruchu.

3.2.5. Obrót orbity lokalnej

Obrót orbity lokalnej przebiega podobnie co obrót orbity globalnej. Zmianie ulegają wektory, które biorą udział w wyznaczaniu osi obrotu orbity (równanie 3.14) oraz współczynniki prędkości obrotu (równanie 3.16).

$$\vec{h}_l = \vec{l}_{up} \times \vec{l}_v \quad (3.14)$$

$$\vec{v}_l = [0, 0, 1]^T \quad (3.15)$$

gdzie: \vec{h}_l — oś obrotu zorientowana poziomo
 \vec{v}_l — oś obrotu zorientowana pionowo

W przypadku orbity lokalnej konieczne było rozdzielenie współczynników s dla kierunku pionowego i poziomego przesuwania. Obliczane są one według wzorów 3.16 i 3.17.

$$s_x = -0.008 \cdot L_s \quad (3.16)$$

$$s_y = 0.004 \cdot L_s \quad (3.17)$$

gdzie: L_s — konfigurowany współczynnik obrotu orbity

Przyspieszenia mają inne znaki dlatego, że przeciągnięcie w osi OX skutkować musi obrotem kamery w przeciwną stronę. Różne moduły współczynników mają na celu spowolnienie podnoszenia i opuszczanie kamery względem obracania jej wokół punktu na powierzchni sfery. Z powodu tych zmian kąty obrotu obliczane są następująco:

$$\theta_x = s_x \cdot d_x \quad (3.18)$$

$$\theta_y = s_y \cdot d_y \quad (3.19)$$

gdzie: θ_x — kąt obrotu w osi pionowej, przesuwa widok w osi OX
 θ_y — kąt obrotu w osi poziomej, przesuwa widok w osi OY

Tak jak w przypadku orbity globalnej, końcowy kwaternion obrotu obliczany jest według wzoru 3.13. Po wykonaniu obrotu o kwaternion q , następuje jego korekta związana z ograniczeniami ruchu kamery zdefiniowanymi dla orbity lokalnej.

3.2.6. Ograniczenia ruchu orbit

Obliczanie współrzędnych geograficznych na podstawie wektorów orbity.

Powiązanie współrzędnych geograficznych z orbitą ma sens tylko w przypadku orbity globalnej, ponieważ reprezentuje ona obrót części ruchomej wizualizacji. Obie orbity jednak wspólnie dzielą ustawienia definiowane za pomocą owych współrzędnych.

Niech $P : (\mathbb{R}^3, \mathbb{R}^3) \rightarrow \mathbb{R}^3$ będzie funkcją rzutującą wektor na płaszczyznę określona jej wektorem normalnym i normalizującą go, a $A : \mathbb{H} \rightarrow \mathbb{R}$ funkcją ekstrahującą kąt obrotu z kwaterionu. Niech $Q : (\mathbb{R}^3, \mathbb{R}^3) \rightarrow \mathbb{H}$ będzie funkcją konstruującą kwaterion obrotu na podstawie dwóch wektorów jednostkowych. Obliczenie długości i szerokości geograficznej na podstawie wektorów orbity globalnej wyrażone jest wzorami 3.21 i 3.23.

$$q = Q\left(P(\vec{g}_v, \vec{g}_{up}), P([0, 0, 1]^T, \vec{g}_{up})\right) \quad (3.20)$$

$$long = A(q) \cdot sgn([q_b, q_c, q_d]^T \cdot [0, 0, 1]^T) \quad (3.21)$$

$$p = Q\left(P(\vec{g}_{up}, [0, 0, 1]^T), [0, 0, 1]^T\right) \quad (3.22)$$

$$lat = A(p) \cdot sgn([0, 0, 1]^T \cdot \vec{g}_{up}) \quad (3.23)$$

gdzie:

$long$ — długość geograficzna w przedziale $\langle -180^\circ; 180^\circ \rangle$

lat — szerokość geograficzna w przedziale $\langle -90^\circ; 90^\circ \rangle$

Obliczenie długości i szerokości geograficznej na podstawie wektorów orbity lokalnej wyrażone jest wzorami 3.25 i 3.27.

$$q = Q\left(P(\vec{l}_v, [0, 0, 1]^T), P([0, -1, 0]^T, [0, 0, 1]^T)\right) \quad (3.24)$$

$$long = A(q) \cdot sgn([q_b, q_c, q_d]^T \cdot [0, 0, 1]^T) \quad (3.25)$$

$$p = Q\left(P(\vec{l}_v, [0, 0, 1]^T), \vec{l}_v\right) \quad (3.26)$$

$$lat = A(p) \cdot sgn([0, 0, 1]^T \cdot \vec{l}_v) \quad (3.27)$$

gdzie:

$long$ — długość geograficzna w przedziale $\langle -180^\circ; 180^\circ \rangle$

lat — szerokość geograficzna w przedziale $\langle -90^\circ; 90^\circ \rangle$

Ograniczenia pozycji sprawdza się do skonstruowania kwaterionu, który redukuje nadmiarowy obrót do ostatniej dozwolonej pozycji. Potrzebny obrót musi być złożeniem obrotów w osiach długości i szerokości geograficznej. Dla orbity globalnej i lokalnej osie te wyrażone są wektorami:

$$\vec{lat}_g = [0, 0, 1]^T \times \vec{g}_{up} \quad (3.28)$$

$$\vec{long}_g = \vec{g}_{up} \quad (3.29)$$

$$\vec{lat}_l = \vec{l}_{up} \times \vec{l}_v \quad (3.30)$$

$$\vec{long}_l = [0, 0, 1]^T \quad (3.31)$$

gdzie:

\vec{lat}_g — osь obrotu orbity globalnej dla szerokości geograficznej

\vec{long}_g — osь obrotu orbity globalnej dla długości geograficznej

\vec{lat}_l — osь obrotu orbity lokalnej dla szerokości geograficznej

\vec{long}_l — osь obrotu orbity lokalnej dla długości geograficznej

Podczas korekty ruchu orbita jest obracana wokół wyliczonych osi o kąt, który wynika z różnic pierwotnych współrzędnych i współrzędnych ograniczających widok.

3.2.7. Tryb kompasu

Tryb kompasu dla orbity globalnej wymaga wykonania jeszcze jednego obrotu korekcyjnego. W momencie włączenia trybu kompasu zapisany zostaje wektor \vec{c}_n wyliczony ze wzoru 3.32, który definiuje obecną orientację kierunku północnego.

Niech $P : (\mathbb{R}^3, \mathbb{R}^3) \rightarrow \mathbb{R}^3$ będzie funkcją rzutującą wektor na płaszczyznę określona jej wektorem normalnym. Wtedy:

$$\vec{c}_n = P(\vec{g}_{up}, \vec{l}_v) \quad (3.32)$$

$Q : (\mathbb{R}^3, \mathbb{R}^3) \rightarrow \mathbb{H}$ jest funkcją konstruującą kwaternion obrotu na podstawie dwóch wektorów jednostkowych. Konstrukcja kwaternionu korekcji dla trybu kompasu kamery przedstawiona jest na równaniu 3.33.

$$q = Q\left(\frac{\vec{c}_n}{\|\vec{c}_n\|}, \frac{\overrightarrow{lat_g}}{\|\overrightarrow{lat_g}\|}\right) \quad (3.33)$$

3.2.8. Animacje - płynność ruchów

Sterowanie kamerą jest przyjemniejsze w odbiorze i bardziej intuicyjne, jeśli poszczególne automatyczne operacje zmiany widoku są płynne, a nie skokowe. Komponent Silnika obsługuje następujące animacje:

1. Wytracanie prędkości obrotu orbity globalnej. Kiedy użytkownik zwolni przycisk myszy podczas obracania widoku, ruch nie zatrzymuje się od razu.
2. Animacja przybliżania i oddalania kamery.
3. Orientowanie kamery w kierunku północnym.

Animacje posiadają konfigurowalny czas trwania i w celu wyliczenia pozycji pośredniej orbit korzystają z interpolacji liniowej i sferycznej[18] wektorów (równanie 3.34).

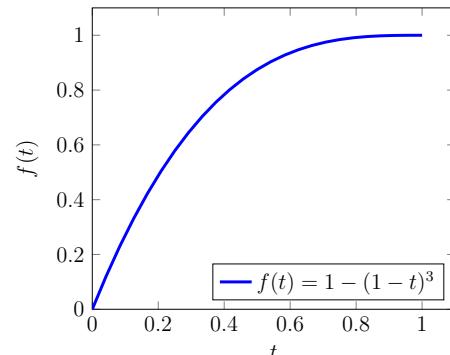
$$Slerp(q_0, q_1, t) = q_0(q_0^{-1}q_1)^t \quad (3.34)$$

gdzie: q_0 — kwaternion początkowy

q_1 — kwaternion końcowy

t — postęp interpolacji w przedziale $[0; 1]$

Postęp interpolacji t modyfikowany jest przez konfigurowalną funkcję wygładzającą $f : X \rightarrow Y$ gdzie $X, Y \in [0; 1]$. Domyślnie używaną funkcją jest *cubicOut*, której wzór i wykres przedstawiony jest na rysunku 3.3.



Rys. 3.3: Funkcja wygładzająca *cubicOut*

3.2.9. Macierze transformacji obiektów na podstawie orbit

Niech $T : \mathbb{R}^3 \rightarrow \mathbb{R}^{4 \times 4}$ będzie funkcją konstruującą macierz translacji na podstawie wektora w przestrzeni trójwymiarowej. Niech $R : (\mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3) \rightarrow \mathbb{R}^{4 \times 4}$ będzie funkcją konstruującą macierz obrotu obserwatora, aby patrzyć na dany punkt w odpowiedniej orientacji. Macierz transformacji grupy obrotu, czyli ruchomej części wizualizacji, wyrażona jest równaniem 3.35. Macierz transformacji obiektu kamery wyrażona jest równaniem 3.36.

$$M_g = T([0, 0, -\|\vec{g}_v\|]^T) \cdot R([0, 0, 0]^T, -\vec{g}_v, \vec{g}_{up}) \quad (3.35)$$

$$M_l = T(\vec{l}_v) \cdot R(\vec{l}_v, [0, 0, 0]^T, \vec{l}_{up}) \quad (3.36)$$

3.3. Implementacja

Komponent Silnika zaimplementowany został w języku TypeScript[26]. Jest on nadzbiorem języka JavaScript i umożliwia korzystanie ze statycznego typowania, które w JavaScriptie nie jest możliwe. Udostępnia tworzenie unii typów, zawiera mechanizmy ich inferencji. Wprowadza wzorzec dekoratorów, zawiera obsługę formatu JSX i dodaje zmienne wyliczeniowe. Rozszerza możliwości programowania obiektowego o typy generyczne. Przed uruchomieniem musi być transpilowany do języka JavaScript. Walidacja utworzonego kodu pod względem poprawności typowania następuje w momencie transpilacji i nie jest dokonywana podczas jego wykonywania.

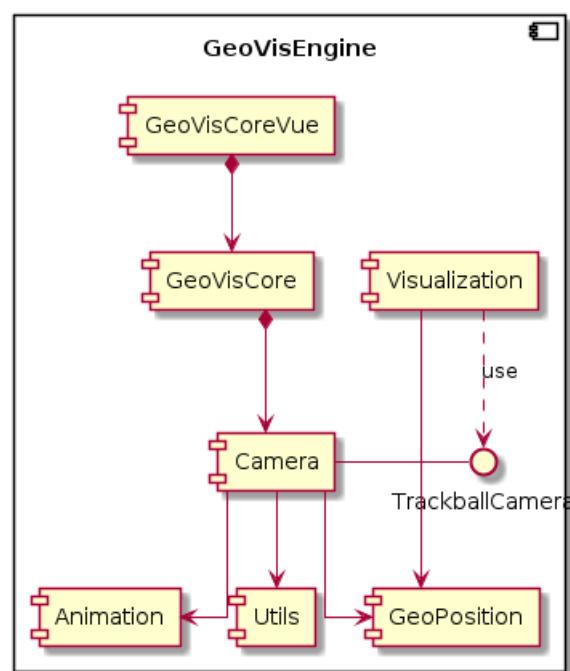
Zaletą stosowania statycznego typowania jest zniwelowanie możliwości pomyłek programisty związanych z nieznajomością interfejsów używanych klas i modułów. Statycznie typowany kod sam w sobie stanowi źródło swojej dokumentacji, a mechanizmy refleksji pozwalają na jeszcze bardziej rozległą walidację typów i dynamiczne generowanie dokumentacji.

Systemu wizualizacji danych geograficznych jako całość nazwany został *GeoVis*, co jest skrótem wyrażenia *Geographic Visualization*. Komponent Silnika przyjął nazwę *GeoVisEngine* i składa się z modułu *GeoVisCore*, który osadzony jest w eksportowanym komponencie *GeoVisCoreVue*. Relację pomiędzy ogólnymi komponentami przedstawiona jest na diagramie 3.4. Kod silnika podzielony został na domeny, które realizują zadania według odkreślonej odpowiedzialności. Są to:

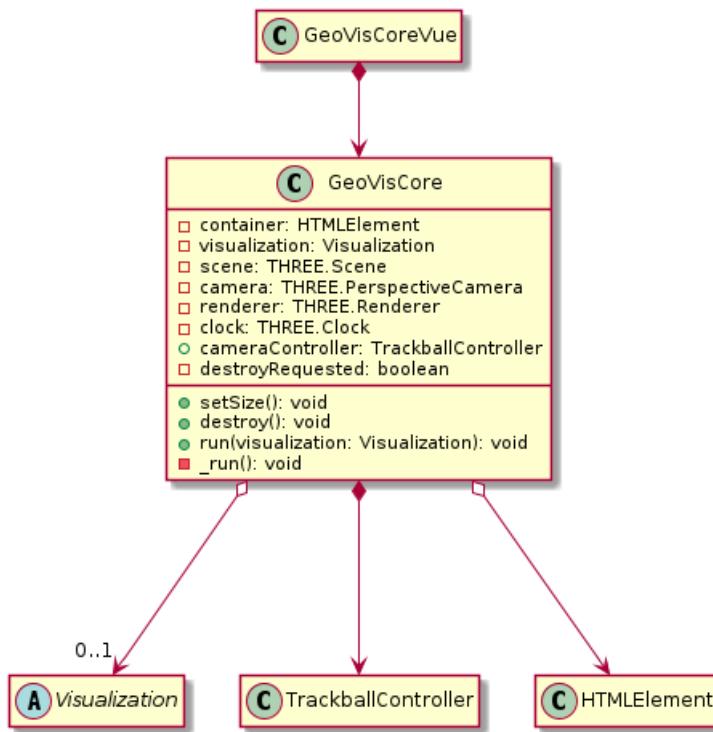
1. **Animation** - obsługuje animacje, udostępnia mechanizmy definiowania transformacji pomiędzy dwoma obiekttami z użyciem wybranej funkcji wygładzającej i zdefiniowanym czasem.
2. **Camera** - zarządza ruchami kamery, udostępnia interfejs `TrackballCamera` dostępny od strony wizualizacji.
3. **GeoPosition** - zawiera definicję obiektów orbit i współrzędnych geograficznych. Odpowiedzialna jest również za transformacje pomiędzy współrzędnymi geograficznymi, a trójwymiarową sceną.
4. **Utils** - zawiera funkcje pomocnicze.
5. **Visualization** - zawiera klasy bazowe definiujące wizualizację oraz ich przykłady.

3.3.1. GeoVisCore

GeoVisCore jest komponentem odpowiedzialnym za dostarczenie elementu *Canvas*, który następnie osadzony jest w komponencie *GeoVisCoreVue* opisany w dalszej części pracy. *GeoVisCore* obsługuje zdarzenia wygenerowane przez użytkownika pochodzące z elementu *Canvas*. Zdarzeniami tymi są te, związane z myszką i klawiaturą. Komponent realizuje cykl życia wizualizacji, wyświetlając opisane przez nią obiekty i propagując zdarzenia pochodzące



Rys. 3.4: Zależności głównych komponentów Silnika



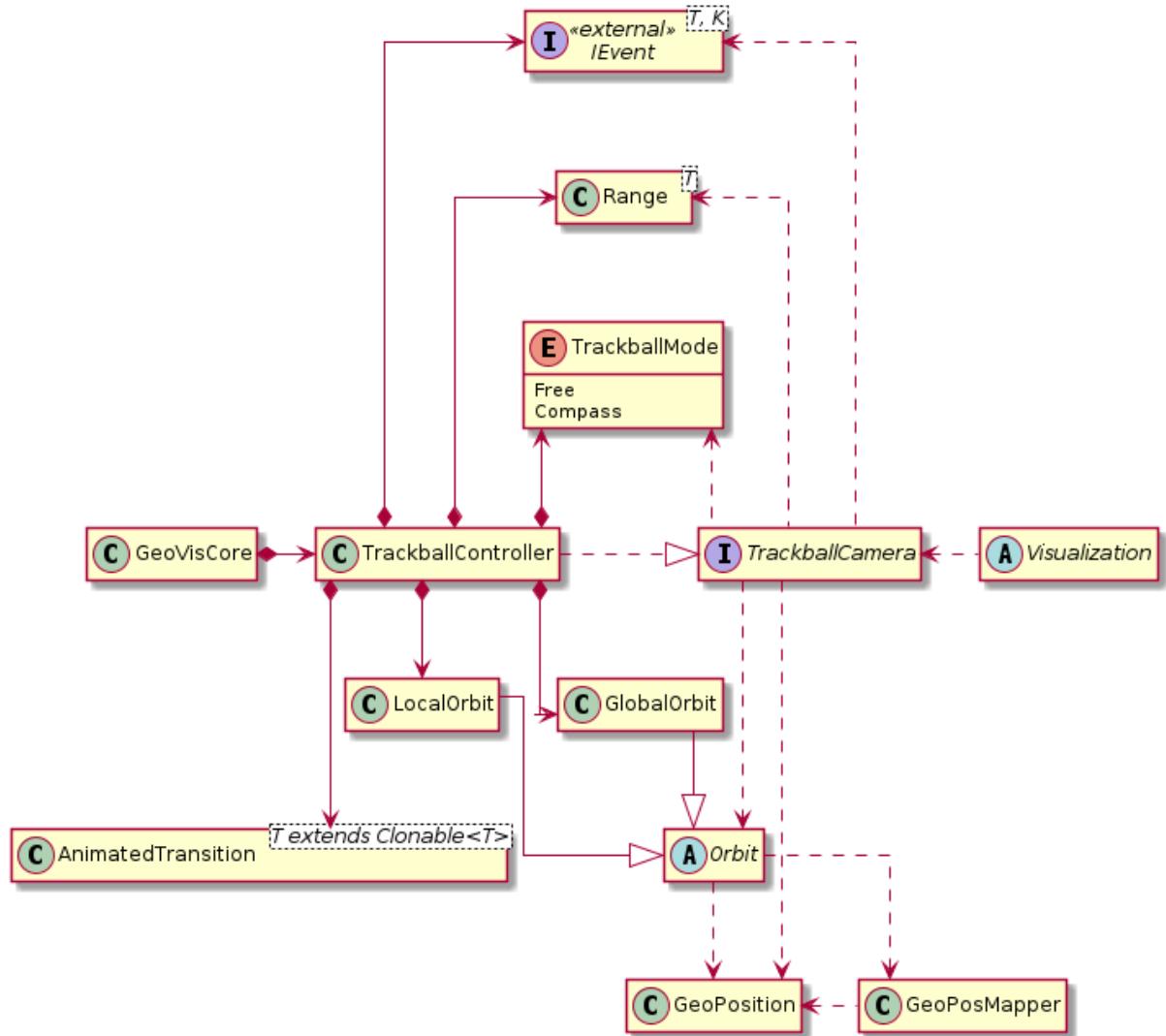
Rys. 3.5: Diagram klas dla klasy GeoVisCore

od użytkownika. To w tym komponencie osadzone jest środowisko biblioteki Three.js. W opisie implementacji najpierw zostanie przedstawiona struktura i zależności każdej z klas i komponentów, a następnie opisany zostanie cykl życia wizualizacji.

Klasa `GeoVisCore`, której despośrednie zależności przedstawia diagram 3.5, jest głównym elementem komponentu Silnika. Inicjalizuje one zależności ekosystemu biblioteki Three.js, odpowiada za obsługę elementu Canvas oraz za uruchamianie i aktualizowanie wizualizacji. Sterowanie wyświetlana wizualizacją odbywa się za pomocą metody `GeoVisCore.run(v: Visualization)`. Przy kolejnych jej wywołaniach instancje obiektów starej wizualizacji są usuwane. Ma to miejsce również podczas wywołania metody `GeoVisCore.destroy()`, która powoduje wyjście z głównej pętli animacji i zamknięcie instancji komponentu.

Najważniejszą klasą, odpowiedzialną za sterowanie kamerą, jest klasa `TrackballController`. Implementuje ona interfejs `TrackballCamera`, który definiuje metody sterowania kamerą dostępne dla twórcy wizualizacji. Diagram najważniejszych zależności klasy `TrackballController` pokazano na rysunku 3.6. W celu zachowania czytelności, na diagramie nie uwzględniono metod i atrybutów klas i interfejsów. Na opisany diagramie nie występują cykle, które oznaczałyby, że zależności pomiędzy klasami są niepoprawnie skonstruowane. Zmiany w jednej klasie pociągnąć mogą za sobą zmiany innych, od niej zależnych klas, a cykl mógłby skutkować niepotrzebnymi, dodatkowymi iteracjami zmian struktury kodu.

Ze względów bezpieczeństwa kod wykonujący się w przeglądarce nie ma dostępu do niczego, poza obsługiwana stroną. Wykonuje się on również tylko na jednym wątku, aby uniknąć typowych problemów programów wielowątkowych takich jak wyścigi czy trudniejsze zarządzenie i współdzielenie pamięci. W typowych przypadkach aplikacje webowe przez większość czasu są bezczynne i większość kodu, który wykonuje się na stronie, wyzwalaany jest za pomocą zdarzeń wysyłanych przez przeglądarkę. Dlatego JavaScript opiera swoje działania na tak zwanym EventLoop. Jest to pętla, która, w dużym uproszczeniu, obsługuje zdarze-

Rys. 3.6: Diagram klas dla klasy `TrackballController` i najważniejszych zależności

nia z określonym priorytetem, kładąc nacisk na responsywność interfejsu użytkownika. Przykładem takich zdarzeń może być naciśnięcie przycisku myszy, ale też żądanie wygenerowanie klatki animacji. Wszystko sprowadza się do asynchronicznego wykonania zdefiniowanej procedury obsługi takiego zdarzenia. JavaScript definiuje zaimplementowany w obiektach DOM interfejs `EventTarget`[7], który umożliwia zdefiniowanie funkcji wykonywanej po zajściu zdarzenia, które identyfikowane są jako ciąg znaków. Wykorzystując przewagę języka TypeScript możemy tworzyć, emitować i obsługiwać zdarzenia, gdzie każdy aspekt będzie posiadał stałe typowanie, włączając w to samą abstrakcję zdarzenia i dane jakie są z nim powiązane. W tym celu wykorzystano bibliotekę o nazwie `strongly-typed-events`[20] implementującą to podejście. Interface `TrackballCamera` korzysta z interfejsu `IEvent` dostarczonego przez tę bibliotekę (diagram 3.6). Różnica pomiędzy obsługą zdarzeń w JavaScriptie i z możliwościami TypeScriptu przedstawiają listingi 3.5 i 3.6.

Listing 3.5: Obsługa zdarzenia w języku JavaScript

```
const event = new Event('event');
event['payload'] = "testData";
elem.addEventListener('event', (data) => {
    console.log(data.payload)
}, false);

elem.dispatchEvent(event); // prints: testData
```

Listing 3.6: Obsługa zdarzenia w języku TypeScript z wykorzystaniem biblioteki `strongly-typed-events`

```
const onEvent = new SimpleEventDispatcher<{payload: string}>();
onEvent.asEvent().sub((data) => {
    console.log(data.payload);
})

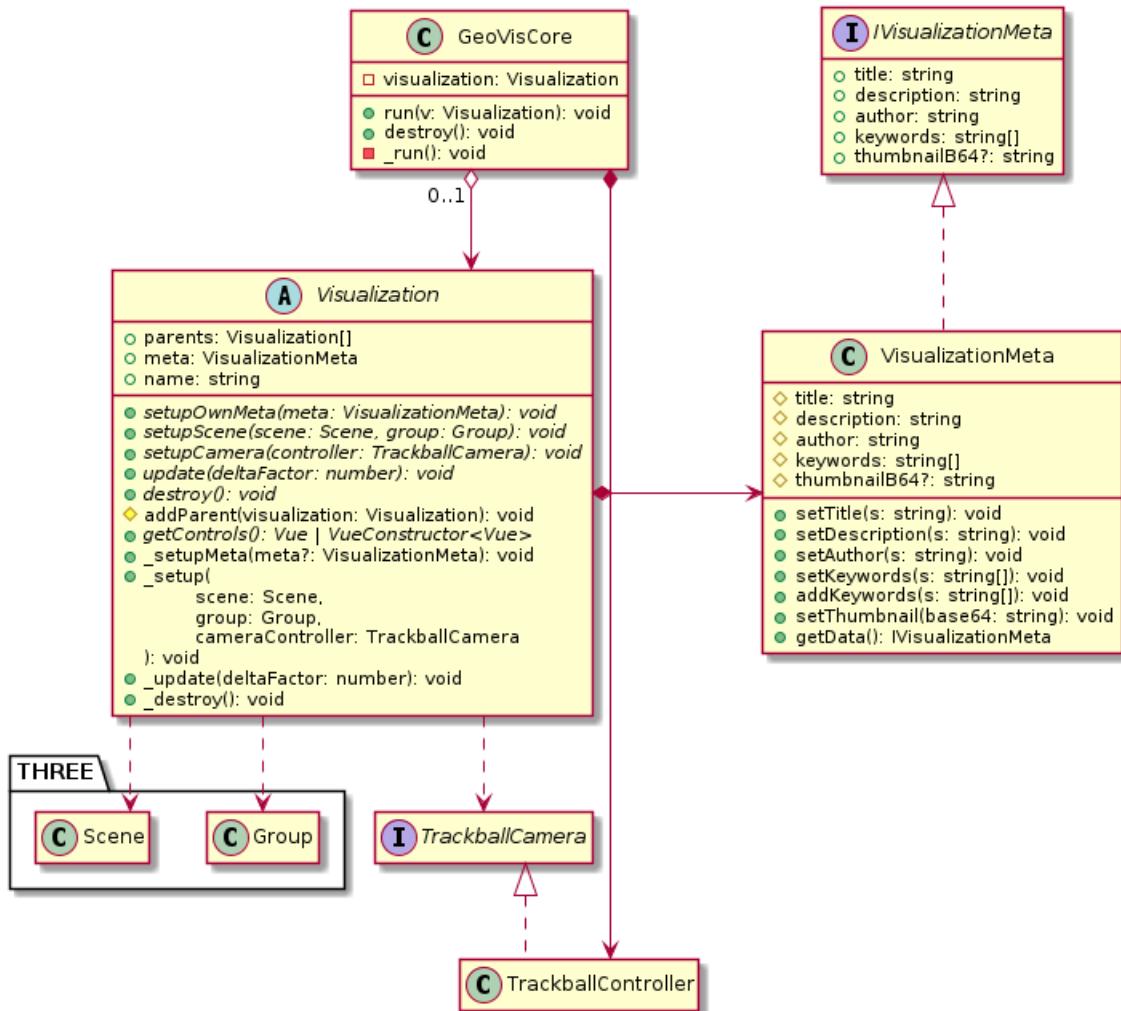
onEvent.dispatch({payload: 'testData'}); // prints: testData
```

`TrackballController` realizuje operacje obrotów orbit opisane w rozdziale 3.2. Orbita reprezentowane są przez obiekty `GlobalOrbit` i `LocalOrbit` dziedziczące po klasie `Orbit`, które realizuje wspólną operację dla obu z nich. Są to między innymi wykonywanie obrotów korekcyjnych dla pozycji kamery i orientacji w kierunku północnym. Wspólną funkcjonalność różni układ odniesienia obu orbit i dlatego wydzielono metody abstrakcyjne klasy `Orbit`. Jej klasy pochodne implementują ją inaczej, zwracając różne wektory. Klasa `AnimatedTransition` reprezentuje animację pojedynczego obiektu. Zawiera ona swój własny zegar i pozwala zdefiniować długość oraz funkcję wygładzającą.

Ukrycie faktycznej implementacji kontrolera kamery za interfejsem `TrackballCamera` daje późniejszą możliwość zastąpienia jej inną. Realizując ten sam interfejs, nie będzie ona wpływać na już utworzone wizualizacje i pozostałą część aplikacji.

3.4. Wizualizacja

Kwestia definiowana wizualizacji wyświetlanych przez złożony system, poruszona we wstępnie, jest ważna z punktu widzenia twórcy wizualizacji. Projektowany system proponuje definicję wizualizacji jako obiektu implementującego dostarczony interfejs. Abstrakcyjna klasa `Visualization` definiuje metody niezbędne do osadzenia obiektów sceny, definicję metadanych, aktualizacji wizualizacji i jej usuwania. Jest to najważniejsza klasa przy definiowaniu wizualizacji, więc każda jej część wymaga dokładnego opisu. Klasa i jej najważniejsze zależności zostały przedstawione na diagramie 3.7.

Rys. 3.7: Diagram klas dla klasy `Visualization` i najważniejszych zależności

Wizualizacje, mogą zawierać osadzone inne wizualizacje tworząc strukturę drzewiastą zależnych wizualizacji. Przykładem może być prosta wizualizacja przestrzeni kosmicznej, która może być rodzicem dla wizualizacji Ziemi, ale też innych planet. Pola klasy `Visualization` dzielą się na abstrakcyjne i te definiowane przez klasę bazową, wywoływanie wewnętrznie:

- `parents` - tablica mieszcząca obiekty wizualizacji - rodziców,
- `meta` - obiekt typu `VisualizationMeta` przechowujący informacje o wizualizacji widoczne w panelu kontrolnym aplikacji i dostępnym z poziomu interfejsu użytkownika,
- `name` - identyfikator wizualizacji, używany do jednoznacznej identyfikacji obiektu wizualizacji,
- `setupOwnMeta` - metoda umożliwiająca wizualizacji ustawienie swoich metadanych,
- `setupScene` - metoda umożliwiająca wizualizacji osadzenie na scenie i w grupie obiektów 3D,
- `setupCamera` - metoda umożliwiająca wizualizacji modyfikację domyślnym ustawień kamery,
- `update` - metoda wywoływana co każdą klatkę animacji, służy do aktualizacji obiektów wizualizacji,
- `destroy` - metoda wykorzystywana do usuwania obiektów i czyszczenia pamięci,
- `addParent` - metoda umożliwiająca dodanie wizualizacji - rodzica, wywoływana typowo w konstruktorze,
- `getControls` - metoda umożliwiająca wizualizacji zdefiniowanie swojego panelu kontrolnego w postaci komponenty frameworka Vue,
- `_setupMeta` - metoda wywoływana wewnętrznie dla całego drzewa wizualizacji obsługująca ustawianie metadanych,
- `_setup` - metoda wywoływana wewnętrznie dla całego drzewa wizualizacji, odpowiedzialna za wywołanie metod `setupCamera` i `setupScene`,
- `_update` - metoda odpowiedzialna za aktualizowanie całego drzewa wizualizacji,
- `_destroy` metoda odpowiedzialna za niszczenie całego drzewa wizualizacji.

Metadane wizualizacji przechowywane są w obiekcie typu `VisualizationMeta` implementującego interfejs `IVisualizationMeta`, który to dostępny jest od strony aplikacji. Interfejs metadanych definiuje następujące pola:

- `title` - tytuł wizualizacji,
- `description` - opis wizualizacji,
- `author` - autor wizualizacji,
- `keywords` - słowa kluczowe zapisane w tablicy,
- `thumbnailBase64` - miniaturka wizualizacji zapisana jako ciąg znaków kodujący obraz kodowaniem Base64 [9].

Wszystkie pola, poza polem `keywords`, są nadpisywane przez dziedziczące wizualizacje. Tablica `keywords` za każdym wywołaniem metody `addKeywords` jest rozszerzana o nowe słowa kluczowe, a ich ewentualne duplikaty są usuwane.

Opisane wcześniej klasy i interfejsy obsługują cykl życia wizualizacji. Składają się na niego utworzenie obiektu wizualizacji i umieszczenie jej w komponencie `GeoVisCore`. Komponent ten dalej zajmuje się inicjalizacją wszystkich obiektów sceny definiowanych przez wizualizację oraz modyfikacją ustawień kamery. Podczas każdej iteracji pętli głównej animacji aktualizowane są położenie kamery oraz sama wizualizacja. Biblioteką obsługiwana w głównej pętli animacji, ułatwiającą tworzenie animacji, jest `Tween.js` [24]. Po zmianie wizualizacji na inną lub po wyłączeniu komponentu, stara wizualizacja jest niszczona. Uogólniony diagram cyklu życia animacji przedstawiono na rysunku 3.8. Szczegółowy diagram sekwencji przedstawiono na rysunku 3.9.

Listing 3.7: Pusta klasa wizualizacji EmptyVis rozszerzająca klasę Visualization

```

import { VueConstructor } from "vue/types/umd";
import * as THREE from "three";
import TrackballCamera from "../../../../../core/domain/Camera/interfaces/
    ↪ TrackballCamera";
import Visualization from "../../../../../core/domain/Visualization/models/
    ↪ Visualization";
import VisualizationMeta from "../../../../../core/domain/Visualization/
    ↪ models/VisualizationMeta";

/**
 * Example of empty visualization
 * @category VisualizationExamples
 */
export default class EmptyVis extends Visualization {
    constructor() {
        super("emptyVis");
        Object.seal(this);
    }

    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    public setupCamera(camera: TrackballCamera): void {
        //
    }

    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    public setupScene(scene: THREE.Scene, group: THREE.Group): void {
        //
    }

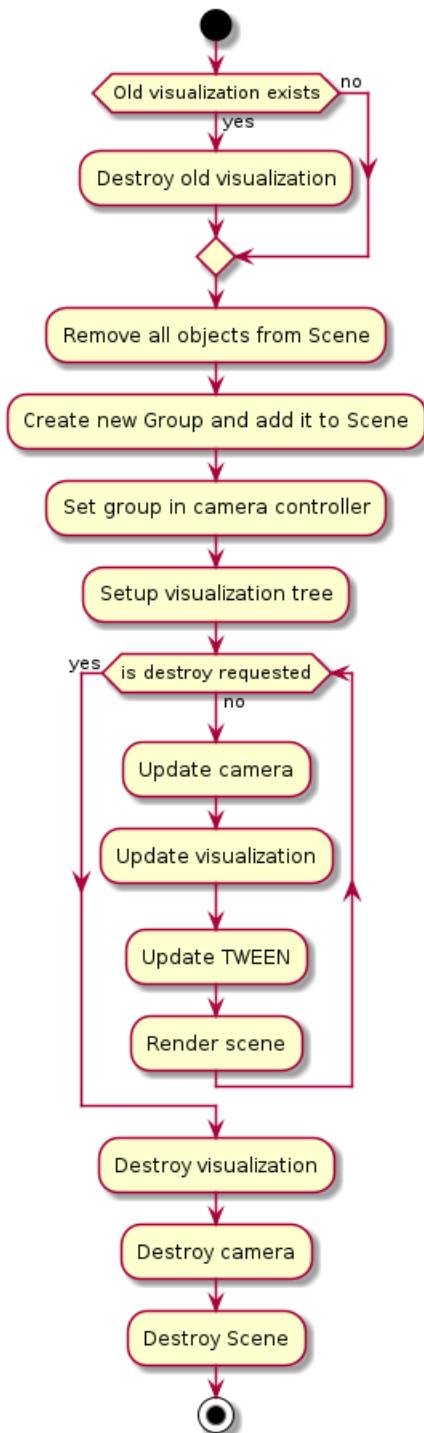
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    public update(deltaFactor: number): void {
        //
    }

    public destroy(): void {
        //
    }

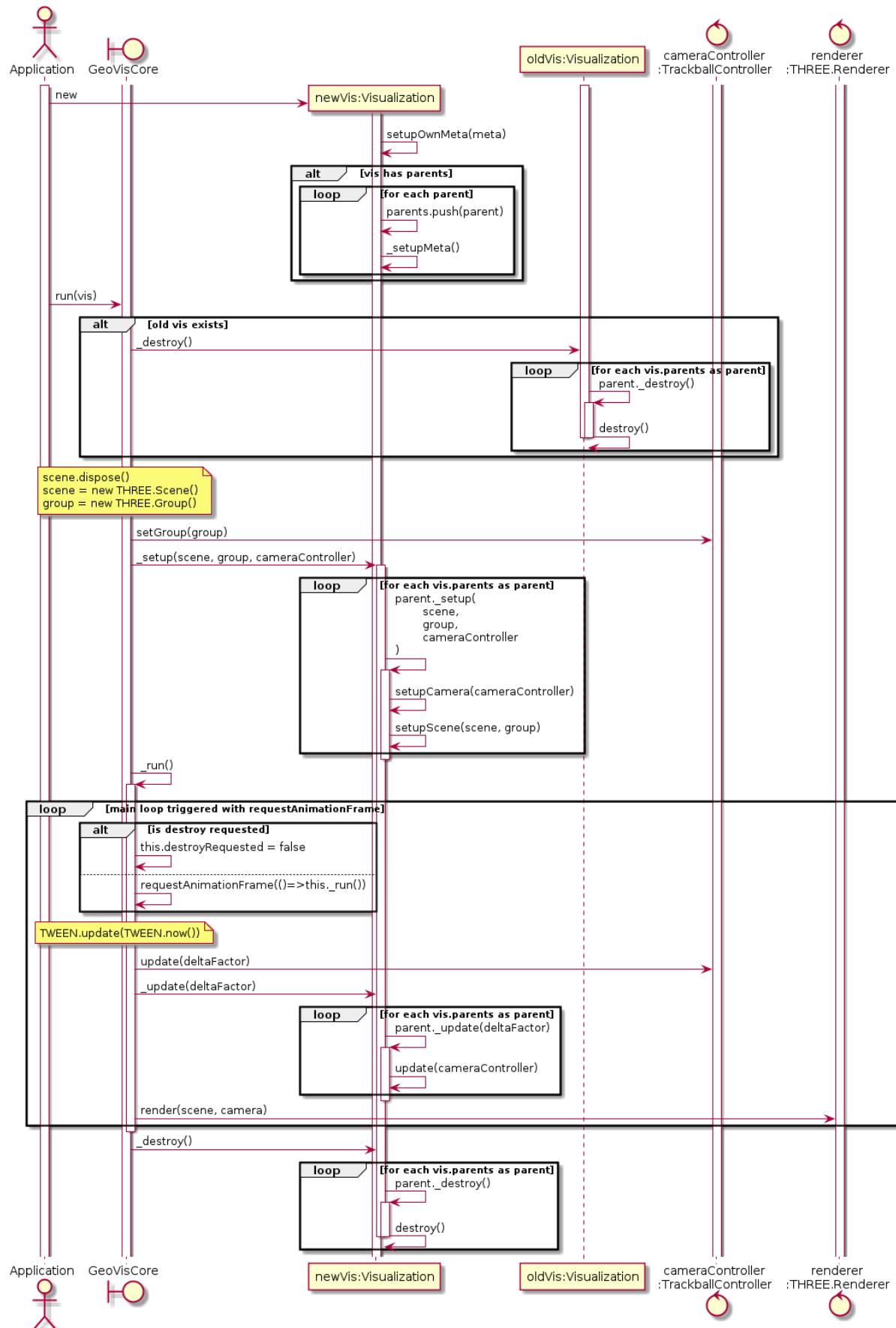
    public getControls(): Vue | VueConstructor<Vue> | null {
        return null;
    }

    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    public setupOwnMeta(meta: VisualizationMeta): void {
        //
    }
}

```



Rys. 3.8: Diagram aktywności cyklu życia wizualizacji



Rys. 3.9: Diagram sekwencji cyklu życia wizualizacji

3.4.1. Vue.js i Webpack

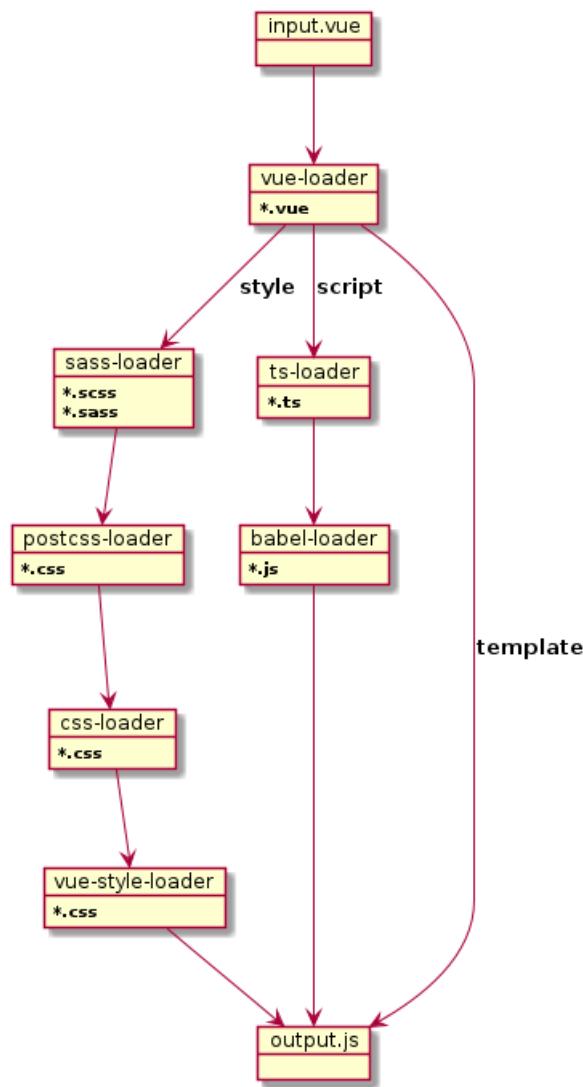
Dotychczas opisane elementy służyły tylko i wyłącznie do obsługi grafiki wyświetlanej za pomocą elementu `Canvas`. Oprócz tego Silnik nakłada na niego komponenty odpowiedzialne za obsługę sterowania kamery oraz wyświetlające informacje o jej położeniu. Wyświetlają one również metadane wizualizacji oraz obsługują dostarczony przez nią komponent sterujący.

Vue.js jest frameworkiem języka JavaScript skupiający się w dużej mierze na wyświetlaniu i aktualizowaniu zdefiniowanych widoków [28]. Określany jest przez twórców jako *progressywny*, ponieważ posiada wysokie zdolności do adaptacji i integracji z innymi bibliotekami oraz tworzony jest z myślą o niskim progu wejścia. Inne funkcjonalności, takie jak obsługa routera, czy przechowywanie globalnego stanu aplikacji, oferują oficjalne biblioteki poboczne. W tworzeniu interfejsu użytkownika, tak jak inne popularne biblioteki, proponuje podejście oparte na komponentach, gdzie rozwój i testowanie skupia się na zawartych w pojedynczych plikach, w pełni funkcjonalnych częściach interfejsu. Podejście to pozwala na izolację testowanych komponentów, łatwe zarządzanie nimi oraz ich ponowne użycie w innych aplikacjach. Vue.js nie oferuje jednoznacznie zdefiniowanego monolitycznego środowiska uruchomieniowego. Nie stoi to na przeszkodzie, aby takie zdefiniowane środowisko powstało. Przykładem takiego środowiska jest framework Nuxt.js oparty na Vue.js [11]. Vue.js w wersji drugiej nie był pisany tak, aby w pełni wspierać język TypeScript, choć mimo to typy dostarczonych obiektów są eksportowane razem z nimi. Brak kompatybilności jest widoczny między innymi w przypadku eksportu typów podczas budowania biblioteki komponentu oraz w samej konfiguracji projektu i oficjalnych bibliotek pobocznych. Wersja trzecia frameworka została kompletnie oparta na języku TypeScript.

Komponent Silnika eksportowany jest jako komponent Vue.js, który posiada osadzony element `Canvas` razem z modułem `GeoVisCore`. Eksportowane są również przykładowe wizualizacje i wszystkie potrzebne użytkownikowi Silnika klasy i interfejsy. Komponent ten może być później zimportowany i osadzony w dowolnym innym komponencie aplikacji chcącym wyświetlać wizualizacje. Konstrukcja struktury projektu w taki sposób daje wszystkie korzyści, jakie daje rozwój aplikacji opartej na komponentach. Są to między innymi opisane wcześniej izolowane testowanie i możliwość wielokrotnego użycia komponentów.

Vue do definicji komponentów proponuje format plików zwany SFC (ang. Single File Component). Jeden taki plik w pełni definiuje zachowanie i wygląd komponentu. Zawiera on trzy sekcje: `template`, przechowującą kod HTML komponentu opatrzony nieraz specjalnymi dyrektywami frameworka, `script`, przechowującą właściwą definicję obiektu komponentu i model jego danych, oraz `style`, gdzie umieszczany jest kod kaskadowych arkuszy stylów, których zakres może być globalny lub ograniczony tylko do właściwego sobie komponentu. Pliki w taka formie nie są rozumiane przez przeglądarkę i przed uruchomieniem muszą być odpowiednio przetworzone. Sposobem na przetworzenie tych plików jest użycie tzw. *module bundlera*, który potrafi zebrać wiele plików w różnych formatach do zasobów rozumianych przez przeglądarkę - kodu HTML, JavaScript i CSS. W ekosystemie Vue.js tego typu rozwiązanie oferuje Webpack [32]. Opiera się on na koncepcji *loaderów* - skryptów, które z użyciem właściwych sobie bibliotek tworzą łańcuch kolejnych transformacji dla plików konkretnych formatów. Dzięki temu aplikacja może być zbudowana z wielu plików źródłowych, które mogą być pisane w różnych formatach i mieć różne odpowiedzialności, a i tak finalnie złożonych do jednego skryptu wynikowego. Alternatywami Webpacka są między innymi Parcel [13] i Rollup [17].

Przykładem plików przetwarzanych przez Webpacka są pliki `*.vue`, gdzie uproszczony łańcuch transformacji, właściwy dla konfiguracji komponentu Silnika, przedstawiony jest na rysunku 3.10. Loader `vue-loader` rozdziela pliki `*.vue` na trzy wcześniej opisane części i każdą z nich traktuje jako odzielny moduł wymagający własnego przetworzenia. Loader `ts-loader` dokonuje transpilacji kodu z języka TypeScript na JavaScript, a `sass-loader` transpiluje style



Rys. 3.10: Uproszczony łańcuch transformacji plików *.vue

z notacji SCSS i SASS na CSS. Pozostałe *loadery* dbają o kompatybilność wsteczną kodu oraz o osadzenie kodu w odpowiednim miejscu pliku wynikowego.

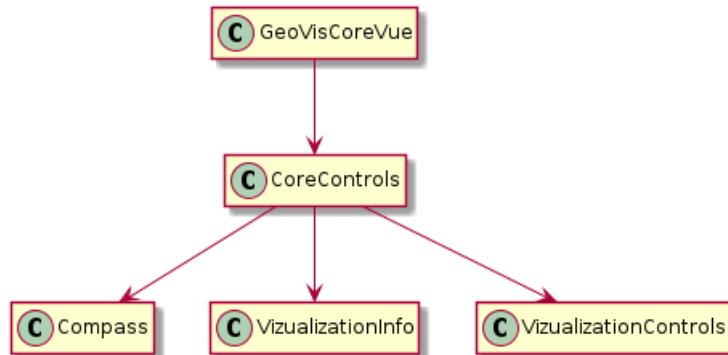
Komponenty Vue.js interfejsu użytkownika

Struktura komponentów przedstawiona została na rysunku 3.11. Komponent GeoVisCoreVue jest komponentem eksportowanym i przeznaczonym do osadzenia w aplikacji używającej komponentu Silnika. Zarządza on elementem Canvas oraz instancją klasy GeoVisCore i zagnieździła pozostałe komponenty interfejsu użytkownika. Eksportowany jest on jako transpilowany kod w języku TypeScript zawierający poprawne adresy do również skopiowanych plików zasobów, takich jak tekstury i ikony. Równolegle eksportowane są typy w plikach *.d.ts, które umożliwiają korzystanie ze statycznego typowania przy wykorzystaniu komponentu Silnika.

Komponenty tworzą strukturę drzewiastą. Przekazywanie danych pomiędzy nimi może zachodzić w dwie strony. Od góry do dołu dane, zmienne i stałe, mogą być przekazywane poprzez tzw. *properties*, definiowane w szablonie tak jak atrybuty znacznika HTML. W górę drzewa dane przekazywane są za pomocą zdarzeń, które to komponent musi wyemitować, a komponent go zagnieżdzający może je obsłużyć. Komponent GeoVisCoreVue do poprawnego dzia-

łania potrzebuje obiektu wyświetlanej wizualizacji, który jest przekazywany przez *property visualization*.

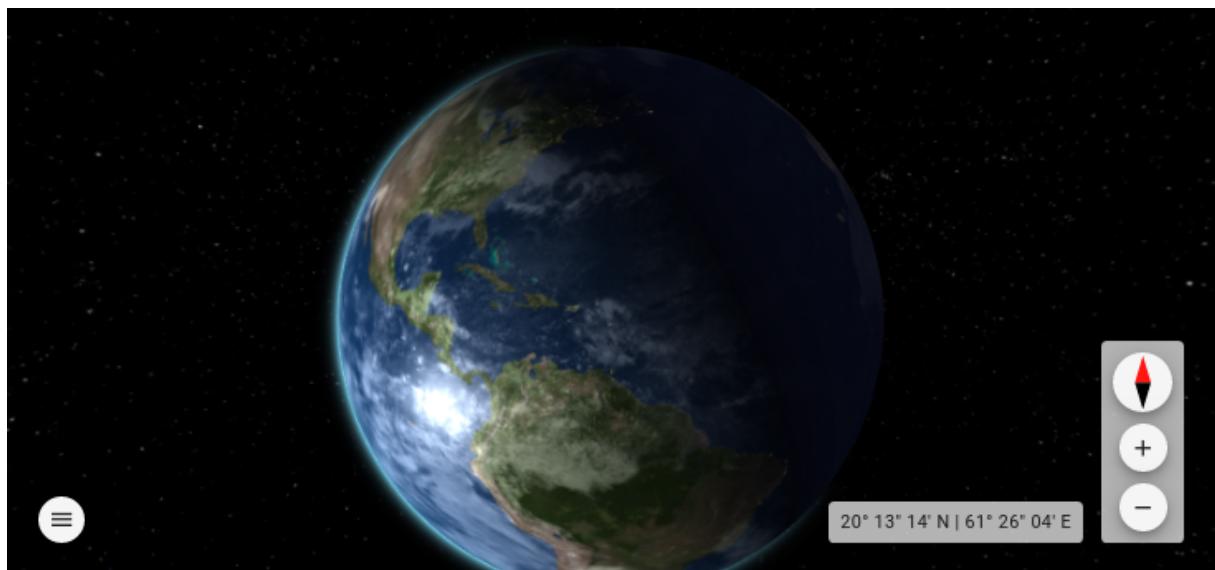
Do podstawowych komponentów, takich jak przyciski, czy też komponenty siatki, użyty został framework Vuetify [29]. Dostarcza on wysoce konfigurowalne komponenty w stylu Material Design. Nazwy komponentów tego framework'a oznaczone przedrostkiem v, na przykład v-btn.



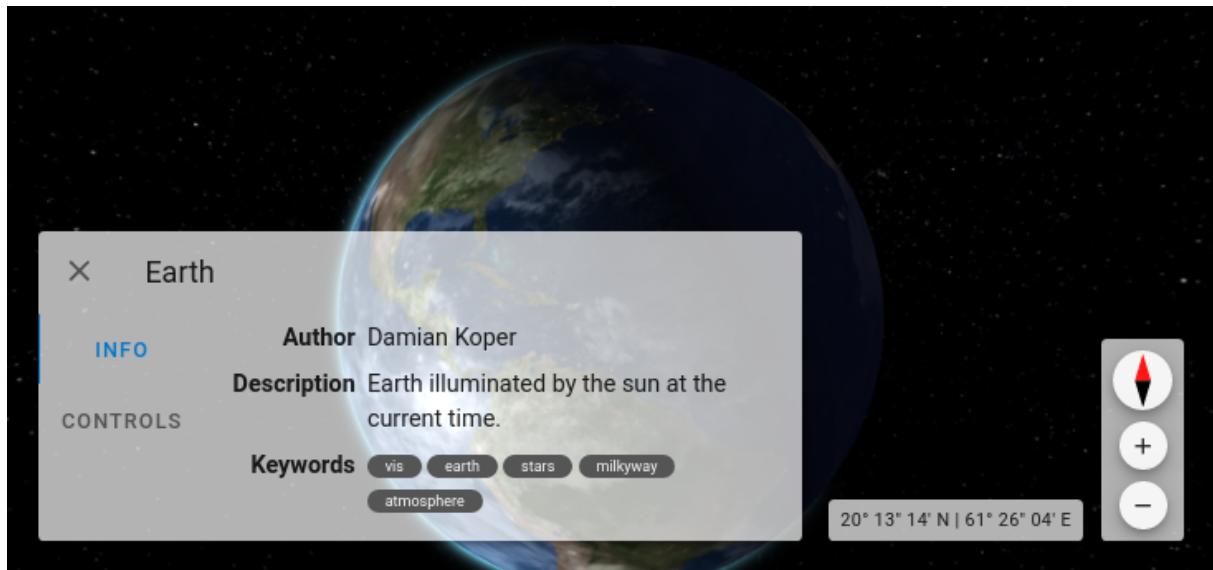
Rys. 3.11: Struktura komponentów Vue.js Silnika

Komponent CoreControls, nałożony na element Canvas, pokazany na rysunku 3.12, odpowiedzialny jest głównie za pracę prawego panelu kontrolnego. Zagnieżdża on komponent Compass, po którego kliknięciu wywoływana jest metoda rotateNorth kontrolera kamery. Przyciski poniżej odpowiedzialne są za przybliżanie i oddalanie kamery - metody zoomIn i zoomOut kontrolera kamery. Kompas oraz współrzędne geograficzne widoczne na lewo od skrajnego prawego panelu aktualizowane są na skutek wyemitowania zdarzenia onNorthAngleChange i onGlobalOrbitChange przez kontroler kamery. Aktualizacja wyświetlanych danych z każdą emisją odpowiadającym im zdarzeń nie jest potrzebna i powodowałaby straty wydajności. Wykorzystywany został tutaj mechanizm throttle, który dostarcza biblioteka Lodash [10]. Zdarzenia obsługiwane są minimalnie co ok. 142,85 ms.

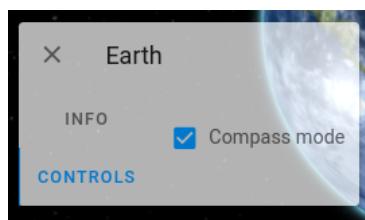
CoreControls zagnieżdża też panel wyświetlający metadane oraz panel kontrolny dostarczany przez wizualizację. Po lewej stronie ekranu znajduje się przycisk pokazujący i ukrywający



Rys. 3.12: Elementy kontrolne wizualizacji



Rys. 3.13: Elementy kontrolne wizualizacji - otwarty panel dodatkowy



Rys. 3.14: Elementy kontrolne wizualizacji - panel kontrolny wizualizacji

dodatkowy panel (rysunki 3.12 i 3.13). Po kliknięciu na zakładkę *Controls*, ukazuje się panel kontrolny wizualizacji (rysunek 3.14).

3.5. Podsumowanie

Komponent Silnika jest najważniejszym komponentem aplikacji. To on odpowiada za wyświetlanie i kontrolę nad wizualizacją. Rozdział ten opisał technologie użyte do definiowania oraz wyświetlania sceny. W pełni też opisał zasady pracy kamery, która może orbitować wokół środka grupy obrotu, oraz niezależnie nad punktem na powierzchni sfery. Wykorzystuje w tym celu abstrakcję orbity globalnej i lokalnej. Orbity mogą mieć zdefiniowane ograniczenia obrotu, a kamera może utrzymywać stałą orientację w kierunku północnym w trybie kompasu. Opisano również mechanizmy odpowiedzialne za animację transformacji widoku kamery.

Następnie przedstawiona została implementacja systemu z wykorzystaniem języka TypeScript, biblioteki Three.js i frameworka Vue.js. Dokładnie opisany został również sposób definiowania wizualizacji w postaci klasy dziedziczącej po klasie `Visualization`. Opisano też sposób budowania eksportowanej wersji komponentu Silnika z użyciem narzędzia Webpack.

Jak się okazało na końcowym etapie rozwoju projektu, użycie bazowej konfiguracji Webpacka proponowanej przez Vue.js nie wspiera eksportowania typów języka TypeScript, co uniemożliwia korzystanie ze statycznego typowania w momencie zimportowania komponentu w innej aplikacji. Konicznym było eksportowanie ich poza głównym procesem budowania komponentu. Było to możliwe jedynie dla plików `*.ts`. Pliki `*.vue` zostały w tym procesie pominięte, ponieważ nie mogą być przetworzone natywnie przez narzędzie `tsc`. Nie stanowi to

dużego problemu, ponieważ najważniejszym elementem wymagającym statycznego typowania jest interfejs `TrackballCamera` oraz klasy przykładowych wizualizacji.

Użycie plików zasobów eksportowanych wraz z komponentem Silnika w docelowej aplikacji, takich jak tekstury, czy też ładowane asynchronicznie skrypty, wymaga ręcznego ich skopiowania w odpowiednie miejsce, aby były dostępne możliwe do pobrania z użyciem serwera plików statycznych. Jest to spowodowane tym, że Webpack, ładując pliki, kopiuje je do zdefinowanej lokalizacji. Dla tych plików, w transpilowanym kodzie, ustawia on odpowiedni adres, który nie zmieni się po zimportowaniu komponentu do innej aplikacji. Obsługa plików statycznych leży w gestii użytkownika komponentu.

Wiele z tych problemów spowodowane jest użyciem domyślnej, z pewnymi modyfikacjami, konfiguracji narzędzia Webpack dla Vue.js. Brak możliwości swobodnej konfiguracji utrudnia rozwiązywanie problemów, które mogą wyniknąć przy budowaniu biblioteki wykorzystującej język TypeScript i pliki statyczne. Z kolei całkowicie ręczna konfiguracja tego narzędzia jest trudna i czasochłonna. Przed przystąpieniem do implementacji, warto zatem przetestować procedurę budowania na minimalnym i kompletnym przykładzie.

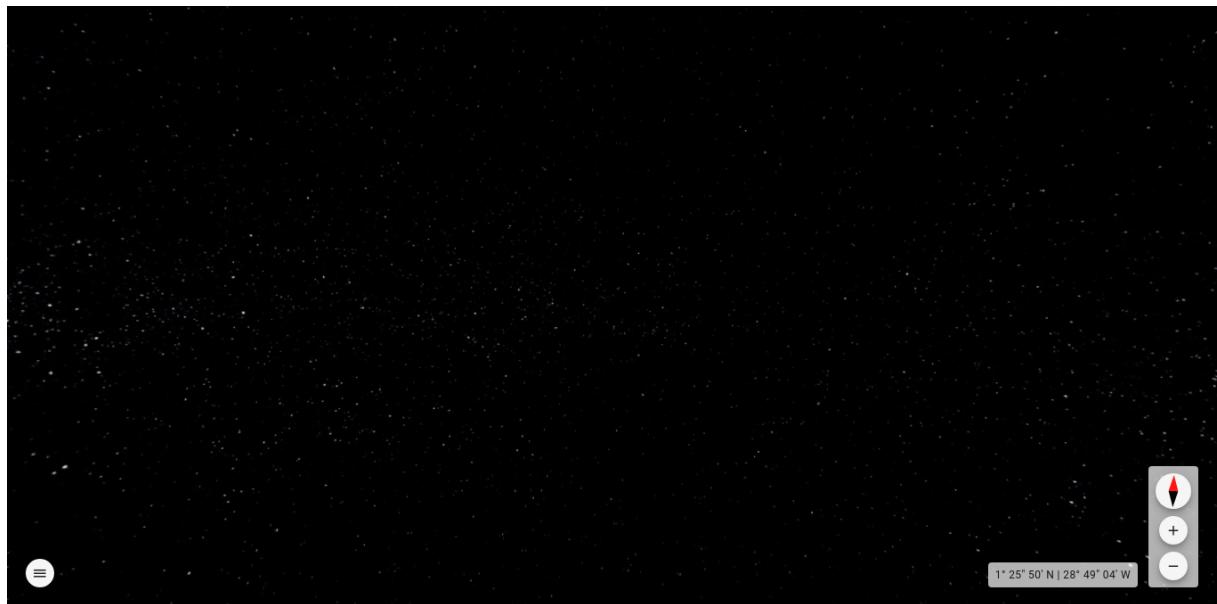
Rozdział 4

Wizualizacje

Rozdział ten opisuje obiekty wizualizacji, które powstały podczas rozwoju komponentu Silnika. Ich opis będzie postępował w kolejności od najprostszych, do tych bardziej skomplikowanych, których te prostsze stanowią podstawę. Opisane zostaną również bardziej złożone aspekty renderowania grafiki, jeśli wizualizacje z takich korzystają. Wizualizacje te są po części demonstracją możliwości Silnika i nie były tworzone z zachowaniem stuprocentowej dokładności odwzorowania zjawisk.

4.1. Gwiazdy

Wizualizacja ta wyświetla teksturę kosmosu nałożoną na wewnętrzną część kuli - rysunek 4.1. Kamera znajduje się w jej środku, więc przeciągnięcie widoku w jedną stronę skutkuje przesunięciem się tekstuury kosmosu w przeciwną. Wizualizacja ta nie modyfikuje ustawień kamery oraz nie definiuje swojego panelu kontrolnego. Tekstura gwiazd pochodzi ze strony <https://www.solarsystemscope.com/> [19].



Rys. 4.1: Wizualizacja gwiazd - klasa StarsVis

Na listingu 4.1 pokazano część klasy `StarsVis` definiującą tę wizualizację. Kula tworzona jest z użyciem klasy `THREE.SphereGeometry`, a jej materiał `THREE.MeshBasicMaterial`

Listing 4.1: Fragmenty klasy StarsVis

```

/* ... */
export default class StarsVis extends Visualization {
  private stars = new THREE.SphereGeometry(40000, 10, 10);
  private starsMaterial = new THREE.MeshBasicMaterial({
    side: THREE.BackSide,
    map: new THREE.TextureLoader().load(starsMap),
    depthWrite: false,
    depthFunc: THREE.NeverDepth,
  });
  private mesh = new THREE.Mesh(this.stars, this.starsMaterial);
  /* ... */
  public setupCamera(camera: TrackballCamera): void {
    //
  }
  public setupScene(scene: THREE.Scene, group: THREE.Group): void {
    this.mesh.renderOrder = 0;
    group.add(this.mesh);
  }
  public update(deltaFactor: number): void {
    this.mesh.rotation.y = TimeService.getHourAngle();
  }
  /* ... */
}

```

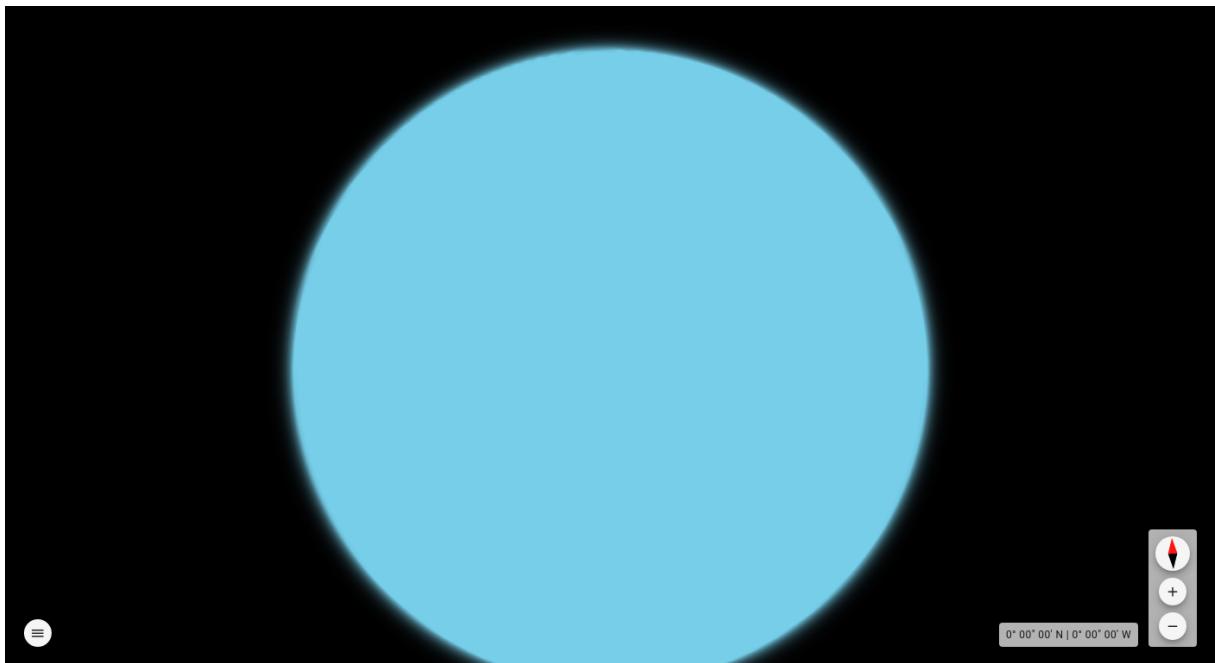
zawiera ustawienia definiujące jej wyświetlanie. Ustawienie `side:THREE.BackSide` sprawia, że teksturowane są odwrotna niż zwykle strona rysowanego trójkąta. Ustawienia `depthWrite:false` oraz `depthFunc:THREE.NeverDepth` sprawiają, że obiekt nie będzie wpływał na wartość z-bufora, oraz będzie rysowany zawsze za innymi obiekty, co jest oczekiwane od obiektu tła.

Obiekt 3D `THREE.Mesh` w metodzie `update` obracany jest w osi *OY* o pewien kąt. Kąt ten wynika z czasu słonecznego, ponieważ wizualizacja ta domyślnie ma stanowić tło dla wizualizacji Ziemi w czasie rzeczywistym. Może być ona również rozszerzona, aby obsługiwać każdy inny dowolny czas. Kiedy kamera jest nieruchoma względem punktu na Ziemi, jej obrót wokół własnej osi widocznego jest jako obrót tła w przeciwnym kierunku. Klasa `TimeService`, dokładniej opisana w dalszej części pracy, zawiera metodę `getHourAngle`, która dla danej strefy czasowej oblicza kąt obrotu słońca od danej długości geograficznej o danym czasie [4]. Punktem odniesienia jest południk 0° i strefa czasowa $+00:00$.

4.2. Atmosfera

Wizualizacja atmosfery stanowi wizualną dekorację dla innych wizualizacji. Składa się ona z dwóch osobno generowanych części. Wizualnie atmosfera to poświata widoczna nad powierzchnią planety, która zanika wraz ze wzrostem wysokości punktu nad powierzchnią. Na efekt też wpływa sama grubość atmosfery, jej skład chemiczny, oraz gęstość w poszczególnych jej partiach.

Utworzona wizualizacja nie posiada rozbudowanych możliwości konfiguracji i została stworzona do współpracy z wizualizacją Ziemi w dużej skali. Na efekt poświaty składają się dwa obiekty. Pierwszym jest kula, której średnica odpowiada średnicy planety razem z grubością atmosfery. Wyświetlana jest jej wewnętrzna część i znając pozycje obserwatora, wyświetlana jest właściwie zanikająca poświata. Drugim obiektem jest kula rozmiarów planety, która zawsze generowana jest przed nią i odpowiada za poświatę widoczną bezpośrednio nad planetą.



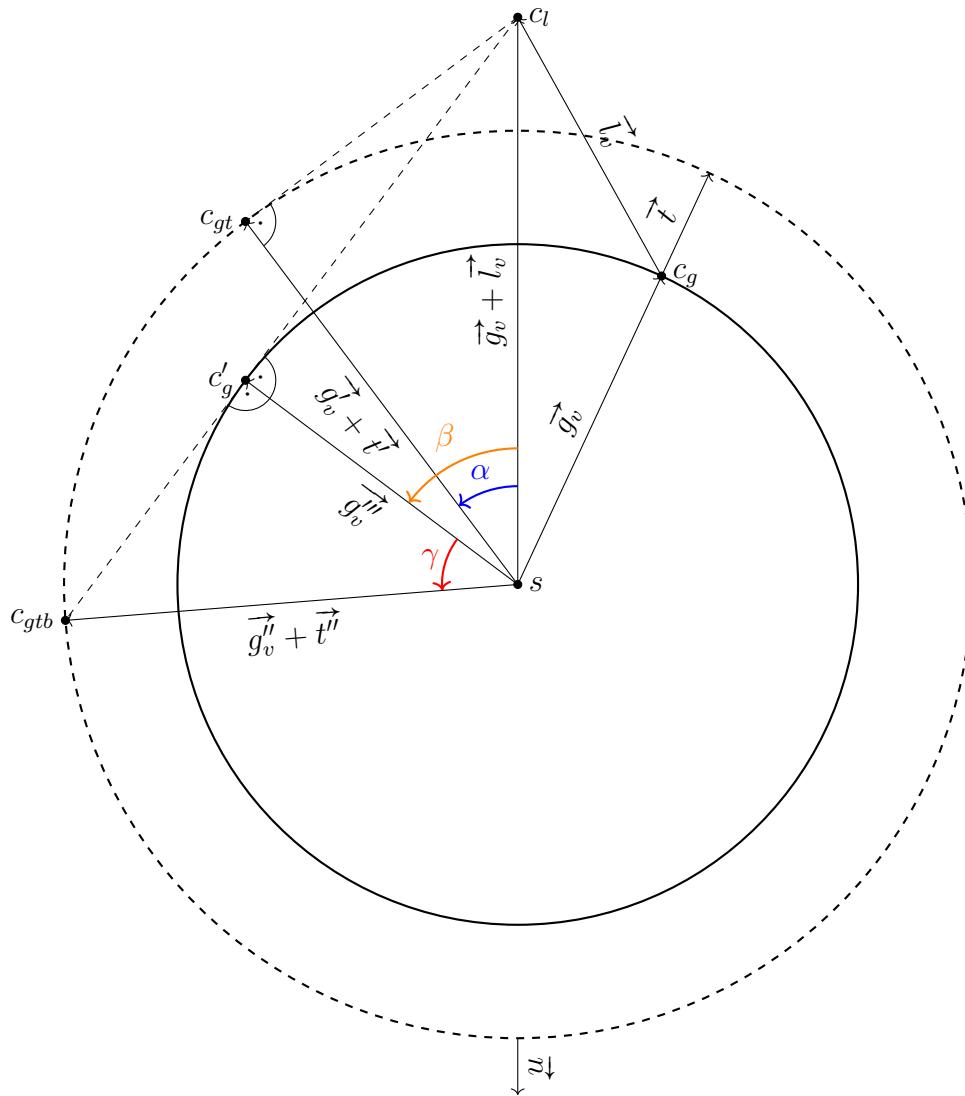
Rys. 4.2: Wizualizacja atmosfery - klasa `AtmosphereVis`

Listing 4.2: Fragmenty klasy `StarsVis`

```
public atmosphereMaterial = new THREE.ShaderMaterial({
  vertexShader: vertexShader,
  fragmentShader: fragmentShader,
  uniforms: {
    start: { value: 1 },
    stop: { value: 0.6 },
    fadeOut: { value: 0 },
    light: { value: 0 },
    power: { value: 1.25 },
    glowColor: { value: new THREE.Color(0x87ceeb) },
    viewVector: { value: new THREE.Vector3() },
    ...THREE.UniformsLib.lights,
  },
  depthFunc: THREE.NeverDepth,
  lights: true,
  transparent: true,
  side: THREE.BackSide,
  depthWrite: false,
});
```

Na rysunku 4.2 przedstawiono efekt atmosfery bez planety. Istotne są tutaj jedynie krawędzie widocznego okręgu, ponieważ jego środek ukryty będzie za planetą. Na rysunku 4.3 przedstawiono schemat elementów kluczowych dla wyliczenia parametrów atmosfery. Kamera znajduje się w punkcie c_l . Okrąg rysowany linią ciągłą symbolizuje powierzchnię planety, a linią przerwaną, zasięg atmosfery. Wektor \vec{t} stanowi przedłużenie wektora $\vec{g_v}$ o grubość atmosfery. Widoczny dla obserwatora fragment atmosfery jest łukiem pomiędzy punktami c_{gtb} i c_{gt} .

Na listingu 4.2 przedstawiono inicjalizację materiału odpowiedzialnego za poświatę nadplanetą. Klasa `THREE.ShaderMaterial` pozwala kontrolować cały proces rysowania punktów, ponieważ wymaga dostarczenia obydwu typów shaderów. Tak jak w przypadku materiału w wizualizacji `StarsVis`, materiał definiuje też ustawienia modyfikacji z-bufora i strony wyświetlanego trójkąta. Materiał definiuje również stałe dla jednego procesu rysowania (`uniforms`).



Rys. 4.3: Schemat elementów kluczowych dla wyliczenia parametrów atmosfery

Stałe te są aktualizowane w każdym cyklu animacji. W procesie rysowania poświata generowana jest w zależności od kąta pomiędzy wektorem normalnym płaszczyzny dla wierzchołka, a wektorem określającym kierunek obserwacji. Niżej opisano stałe przekazywane do materiału.

Uniform start

Uniform **start** to ułamek liczby π w zakresie $[0; 1]$, który stanowi kąt wektora obserwatora z wektorem normalnym wierzchołka, od którego rozpoczyna się rysowanie poświaty. Wartość ta zawsze wynosi 1, co oznacza, że poświata rysowana jest od wierzchołków najdalej od kamery. Jego wektor normalny na rysunku 4.3 oznaczony został symbolem \vec{n} . Kąt między nim, a wektorem $\vec{g}_v + \vec{l}_v$ wynosi 180° , czyli $1 \cdot \pi$.

Uniform stop

Uniform **stop** to ułamek liczby π w zakresie $[0; 1]$, który stanowi kąt wektora obserwatora z wektorem normalnym wierzchołka, od którego kończy się rysowanie poświaty o pełnej przezroczystości. Ostatnim widocznym z kamery punktem jest punkt c_{gtb} . Dalsza część schowana jest za

planetą. Kąt ten wyliczany jest z zależności $\beta + \gamma$. Sposób wyliczenia poszczególnych kątów pokazano na równaniach 4.2 i 4.3.

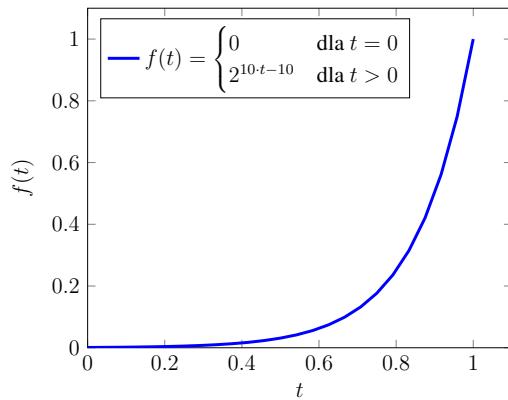
$$\alpha = \arccos\left(\frac{\|\vec{g}_v + \vec{t}\|}{\|\vec{g}_v + \vec{l}_v\|}\right) \quad (4.1)$$

$$\beta = \arccos\left(\frac{\|\vec{g}_v\|}{\|\vec{g}_v + \vec{l}_v\|}\right) \quad (4.2)$$

$$\gamma = \arccos\left(\frac{\|\vec{g}_v\|}{\|\vec{g}_v + \vec{t}\|}\right) \quad (4.3)$$

Uniform `fadeOut`

`Uniform fadeOut` to ułamek liczby π w zakresie $[0; 1]$, który stanowi kąt wektora obserwatora z wektorem normalnym wierzchołka, od którego kończy się rysowanie poświaty o zanikającej przezroczystości. Jest ona interpolowana z wykorzystaniem funkcji wygładzającej `expoIn`, którą pokazano na wykresie na rysunku 4.4. Kąt ten wyliczany jest z zależności $\beta + \gamma - \alpha$.



Rys. 4.4: Funkcja wygładzająca `expoIn`

Uniform `light`

`Uniform light` to ułamek liczby π w zakresie $[0; 1]$, który stanowi kąt przesunięcia oświetlanej powierzchni dla światła. Jeśli do sceny zostanie dodane światło kierunkowe to wpływa ono na wygląd atmosfery. W normalnej sytuacji światło kierunkowe oświetla dokładnie pół kuli. Założymy sytuację, w której kąt padania światła wyznacza wektor rozciagnięty pomiędzy punktami c'_q i c_{gtb} . W takiej sytuacji atmosfera oświetlona by była na łuku pomiędzy wektorami \vec{g}'_v i $\vec{g}'_v + \vec{t}'_v$. Żeby oświetlić pozostałą, widoczną część atmosfery (łuk pomiędzy punktami c_{gtb} i c_{gt}) w obliczeniach, trzeba uwzględnić kąt γ .

Uniform `power`

`Uniform power` to wykładnik potęgi, do której podniesiona zostaje finalna przezroczystość materiału atmosfery, przed kalkulacją oświetlania. Wartość ta nie jest aktualizowana i wynosi 1.25.

Uniform `glowColor`

`Uniform glowColor` to bazowy kolor materiału atmosfery. Nie ulega on zmianie i ma wartość `0x87ceeb`. Możliwą poprawą zachowania atmosfery byłaby dynamiczna zmiana koloru atmos-

fery powiązana z kątem padania światła, co symulowałoby zmianę koloru w miejscach zachodu i wschodu słońca.

Uniform viewVector

Uniform `viewVector` to wektor jednostkowy skierowany od środka s grupy obrotu do kamery c_l . Odpowiada on za orientację wyświetlanej atmosfery zawsze w kierunku punktu, w którym znajduje się kamera. Uzyskanie wektora przed normalizacją przedstawiono na równaniu 4.4.

$$\vec{v} = q^{-1}([0, 0, \|\vec{g}_v\|]^T + \vec{l}_v)q \quad (4.4)$$

gdzie:

\vec{v} — wektor `viewVector`

q — kwaternion uzyskany z macierzy układu odniesienia wizualizacji.

W równaniu obrót następuje w kierunku odwrotnym.

Uniform `fadeOut` bierze też udział w ujednoliceniu koloru nieba, kiedy kamera schodzi poniżej granicy atmosfery. Zachowanie to nie zostało jednak wystarczająco rozwinięte, żeby odzwierciedlać realistycznie przejście pomiędzy czernią kosmosu, a niebieskim niebem z użyciem tego samego materiału w procesie rysowania. Pozostałe stałe w opisywanym materiale są uzupełnieniem z obiektu `THREE.UniformsLib.lights`. Zawierają one dane związane ze światłami obecnymi na scenie.

Materiał opisujący wygląd drugiej kuli, atmosfery widocznej nad ziemią, ma podobną budowę, korzysta z wyliczonych kątów i podobnych mechanizmów. Biorąc to pod uwagę, nie zostanie on tutaj opisany.

4.3. Ziemia

Opisane wcześniej wizualizacje kosmosu i atmosfery pozwalają na stworzenie wizualizacji planety. Wizualizacja `EarthVis` jest wizualizacją planety Ziemi, która prezentuje jej aktualne oświetlenie przez Słońce. Widok początkowy wizualizacji przedstawiono na rysunku 4.5.



Rys. 4.5: Widok początkowy wizualizacji Ziemi o godzinie 19:50, 27.09.2020r.



Rys. 4.6: Zachód słońca nad Europą o godzinie 18:54, 27.07.2020r.

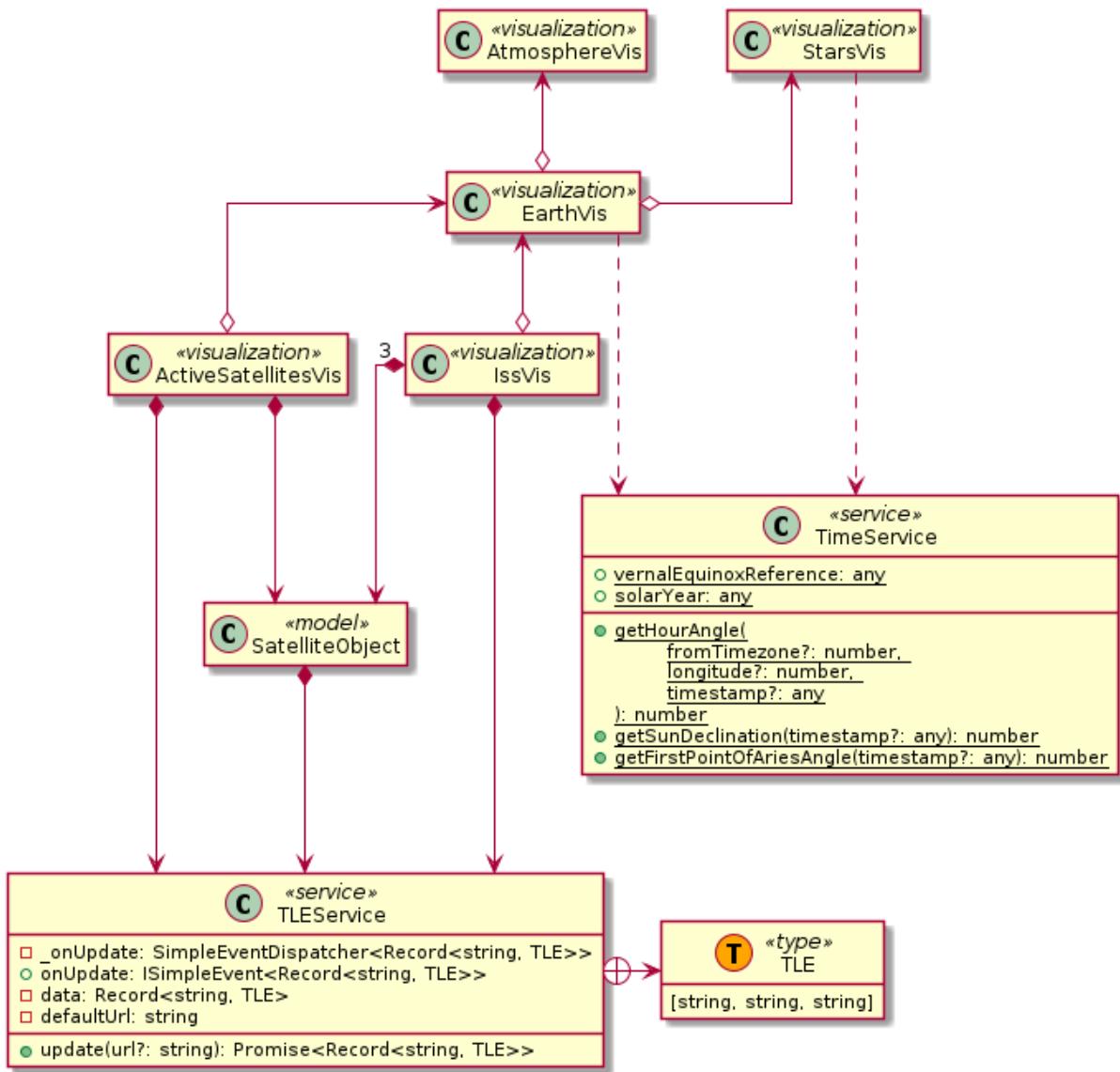
Podczas każdej aktualizacji animacji obliczane jest nowe położenie słońca. Poruszając się w układzie odniesienia Ziemi konieczne jest obliczenie deklinacji Słońca, czyli kąta pomiędzy nim, a płaszczyzną równika [5]. Drugą wartością konieczną do wyliczenia jest kąt godzinny [4]. Jest to kąt, o jaki obróciło się Słońce od wybranego punktu odniesienia dla wybranej strefy czasowej w konkretnej chwili. Wizualizacja Ziemi stanowi podstawę do wizualizacji połączonych z nią zjawisk dużej skali. Bazując na niej wizualizacje ActiveSatellitesVis oraz IssVis opisane w dalszej części pracy. Na rysunku 4.7 przedstawiono diagram zależności pomiędzy poszczególnymi wizualizacjami i klasami pomocniczymi.

Na wizualizację składają się wszystkie obiekty ustawiane przez wizualizacje `AtmosphereVis` i `StarsVis`. Na scenie w układzie współrzędnych wizualizacji znajdują się dwie kule. Pierwsza z nich odpowiada za wyświetlanie tekstury Ziemi, a druga, z promieniem większym o 4 km, wyświetla chmury. Materiał pierwszej kuli implementuje model oświetlenia z cieniowaniem Phonga [35, Rozdział 3] za pomocą obiektu THREE.MeshPhongMaterial i używa trzech map.

1. `map` - mapa zawierająca kolor teksceli tekstury
2. `specular map` - mapa zawierająca poziom odbicia kierunkowego światła dla każdego tekscela tekstury
3. `normalMap` - mapa zawierająca wektory normalne w przestrzeni stycznej wierzchołka modelu.

Materiał kuli wyświetlające chmury zawiera tylko mapę definiującą sam kolor teksceli. Jako, że ta mapa jest obrazem chmur na czarnym tle i nie zawiera kanału `alpha`, materiał w procesie renderowania stosuje mieszanie addytywne. Grupa obrotu zawiera również światło kierunkowe, które zawsze skierowane jest do jej środka. Pozycja źródła światła obracana jest zgodnie z wartościami wyliczonymi przez serwis `TimeService`, kolejno o kąt godzinny w osi `OY` i deklinację w osi `OX`.

Na rysunku 4.6 pokazane zostało płynne przejście pomiędzy tekstyurą nocną i dzienną, która jest wyświetlana w zależności od oświetlenia planety. Zachowanie to wymagało zmodyfikowania domyślnego fragmentu shadera i uzależnienia wyświetlanej tekstyury od kąta pomiędzy inter-



Rys. 4.7: Zależności pomiędzy klasami wizualizacji Ziemi

polowanym wektorem normalnym dla tekscela, a kierunkiem padania światła. Na listingu 4.3 pokazano najważniejszą część modyfikacji shadera. Potrzebny kąt wyliczany jest z wykorzystaniem iloczynu skalarnego wspomnianych wektorów. Wynik ten musi być zmapowany z przedziału $[-1; 1]$ na przedział $[0; 1]$. Obliczenie wartości, która trafia do funkcji wygładzającej, a następnie do funkcji `mix`, która miesza wartości teksceli, pokazano na rysunku 4.5.

Listing 4.3: Modyfikacja fragment shadera materiału MeshPhongMaterial

```
#if NUM_DIR_LIGHTS > 0
    float dotL = dot(vNormal, directionalLights[0].direction);
    vec4 texelColorNight = texture2D( nightMap, vUv );
    texelColorNight = mapTexelToLinear( texelColorNight );

    outgoingLight = mix(
        vec3(texelColorNight) + ambientLightColor,
        outgoingLight,
        easeInOutExpo(dotL*0.5+0.5)
    );
#endif
```

$$x = (\vec{n} \cdot \vec{d}) \cdot 0.5 + 0.5 \quad (4.5)$$

gdzie:
 \vec{n} — wektor normalny
 \vec{d} — kierunek padania światła
 x — wartość w przedziale $[0; 1]$

Wizualizacje dostarcza również panel kontrolny, który pokazany został na rysunku 3.14 i pozwala na przełączanie się pomiędzy trybami kamery - swobodnym i kompas.

4.4. Wybrane satelity

Wizualizacją stworzoną na podstawie wizualizacji Ziemi jest wizualizacja wybranych satelitów orbitujących wokół niej. Klasą definiującą tę wizualizację jest klasa `IssVis` obecna na rysunku 4.7. Znajdują się na nim również klasy wspomagające wyświetlanie i wyliczanie pozycji obiektów. Na scenie znajdują się trzy satelity. Są nimi Międzynarodowa Stacja Kosmiczna, Kosmiczny Teleskop Hubble'a oraz Eutelsat Hot Bird 13C. Każda satelita reprezentowana jest za pomocą trzech obiektów, którymi jest elipsa reprezentująca kształt orbity, obiekt satelity będący złożonym modelem 3D lub sferą oraz linia łącząca obiekt satelity ze środkiem planety. Wizualizacja dostarcza komponent panelu kontrolnego, który umożliwia zmianę trybu pracy kamery oraz zarządzenie widocznością poszczególnych satelitów. Pozycja satelitów wyświetlana jest w czasie rzeczywistym, a ich pozycja i orbita kalkulowana jest z wykorzystaniem danych w formacie TLE (ang. Two-Line Elements).

4.4.1. TLE

TLE jest formatem zapisu informacji o satelicie pozwalającym wyznaczyć z dużym przybliżeniem jej pozycję relatywnie do ciała orbitowanego [25]. Pierwsza linia zawiera nazwę satelity i może być pomijana w zapisie. Druga linia jednoznacznie identyfikuje satelitę i zawiera informacje o punkcie w czasie, dla którego określone są parametry orbity - epokę. Zawiera również informacje kontrolne o samym TLE oraz pierwszą i drugą pochodną prędkości ruchu. Trzecia linia zawiera parametry orbity. Najważniejszymi z nich są inklinacja, kąt węzła wstępującego,

ekscentryczność i argument perycentrum, który dla Ziemi nazywa się argumentem perygeum. Poniżej przedstawiono przykładowe dane dla Międzynarodowej Stacji Kosmicznej.

ISS (ZARYA)

```
1 25544U 98067A 08264.51782528 - .00002182 00000-0 -11606-4 0 2927
2 25544 51.6416 247.4627 0006703 130.5360 325.0288 15.72125391563537
```

W stworzonej wizualizacji TLE pobierane są ze strony Celestrak [1], która zbiera i analizuje dane otrzymane od jednostki NORAD (ang. North American Aerospace Defense Command). Dane otrzymywane są w formacie tekstowym. Serwis TLEService odpowiedzialny jest za pobranie danych TLE wszystkich aktywnych satelitów, sparsowanie je, a następnie wyemitowanie zdarzenia `onUpdate`, które może być obsłużone w procesieinicjalizacji obiektów na scenie. Serwis ten umożliwia również dostęp do danych satelity z wykorzystaniem jej identyfikatora. Dla satelitów, na których pozycję wpływać może atmosfera i inne nieprzewidziane czynniki, dane TLE mogą z czasem stawać się nieaktualne. Dzieje się to jednak na przestrzeni dni. Założyć można, że pobranie najnowszej ich wersji w momencie uruchamiania wizualizacji pozwala na wystarczająco dokładne wyliczenia ich pozycji.

Za obliczenie parametrów orbity, wygenerowanie obiektów sceny i odpowiednią transformację ich pozycji odpowiada obiekt `SatelliteObject` reprezentujący pojedynczą satelitę. Oblicza on parametry elipsy i generuje reprezentującą ją obiekt klasy `THREE.EllipseCurve`. Równania 4.6 - 4.8 opisują proces wyliczenia parametrów orbity. Wyliczenia półosi wielkiej a wynika z trzeciego prawa Kelpera, które łączy jej długość zależnością z okresem obiegu satelity wokół Ziemi, który to dostarcza TLE. Transformacja aktualizowana jest w każdym przebiegu pętli animacji.

$$a = \frac{G^{\frac{1}{3}}}{\left(\frac{2n\pi}{86400}\right)^{\frac{2}{3}}} \quad (4.6)$$

$$b = a\sqrt{1 - e^2} \quad (4.7)$$

$$c = e \cdot a \quad (4.8)$$

gdzie:

a — półosie wielka orbity

G — standardowy parametry grawitacyjny Ziemi

n — średnia liczba obiegów Ziemi w ciągu 24 godzin

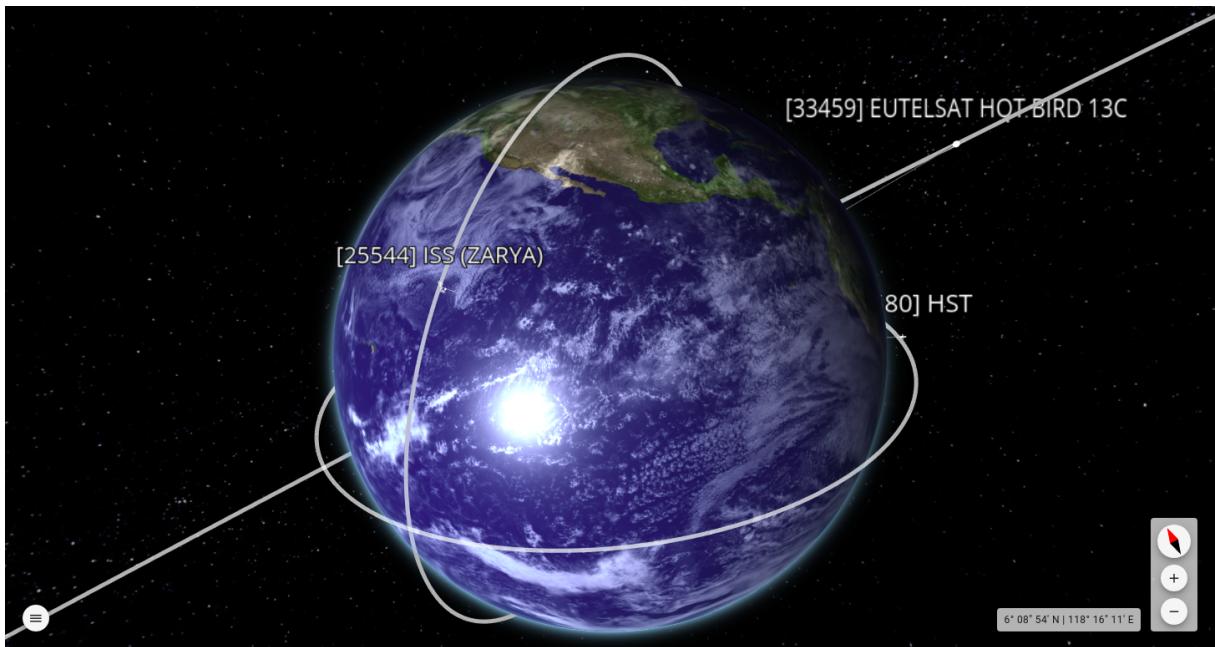
b — półosie mała orbity

e — ekscentryczność orbity

c — przesunięcie od środka do ogniska orbity

Finalna transformacja obiektu elipsy jest złożeniem obrotów i translacji. Najpierw wykonywana jest translacja elipsy tak, aby środek Ziemi pokrywał się z jej ogniskiem. Następnie w osi OX elipsa obracana jest o kąt wynikający z danych o inklinacji orbity. Potem wykonywany jest obrót w osi OY o kąt wynikający z położenia punktu Barana, czyli punktu służącego do orientacji w odniesieniu do równikowego układu współrzędnych oraz ekliptyki. W tej samej osi elipsa obracana jest o kąt wynikający z kąta węzła wstępującego oraz o kąt przeciwny do kąta godzinnego, aby uwzględnić obrót Ziemi w czasie. Na końcu następuje obrót uwzględniający argument perygeum.

Funkcja `TimeService.getFirstPointOfAriesAngle` odpowiedzialna jest za obliczanie kąta od punktu Barana. W obecnej implementacji może ona jednak być czynnikiem wpływającym na brak pokrycia orientacji orbity w osi OY z jej faktyczną trajektorią. Sztuczna korekcja



Rys. 4.8: Wybrane satelity - klasa IssVis

staje się również nieaktualna po jakimś czasie. Autor wizualizacji nie był w stanie dostatecznie zbadać przyczyny tego problemu.

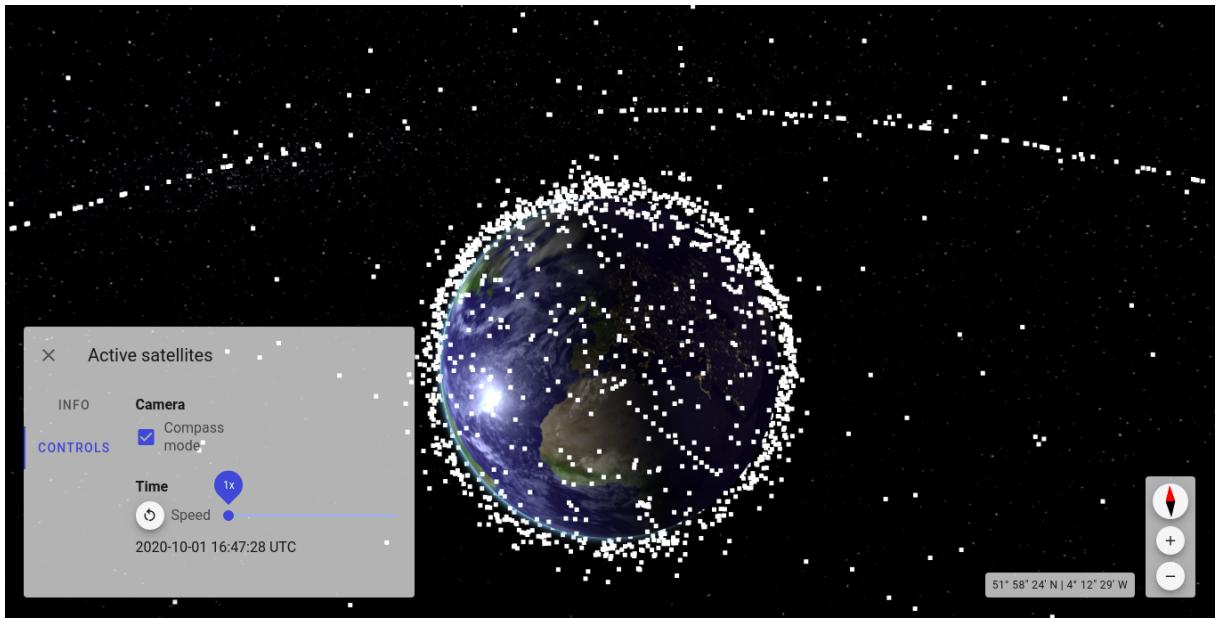
Za obliczanie pozycji samej satelity na orbicie odpowiada biblioteka `tle.js` [23], która jest używana w procesie aktualizacji obiektu `SatelliteObject`. Pozwana ona, na postawie TLE, uzyskać położenie satelity o podanym czasie. Etykiety generowane są z użyciem obiektu `THREE.Sprite`, który wyświetla kwadratową teksturę zawsze zorientowaną w kierunku obserwatora. Tekstura jest napisem, który rysowany jest dynamicznie na elemencie `Canvas` przetworzonym przez obiekt `THREE.CanvasTexture`. Rozmiar etykiet jest wprost proporcjonalny do odległości obiektu od obserwatora. Zachowanie to utrzymuje taki sam rozmiar etykiet niezależnie od orientacji kamery na scenie.

4.5. Aktywne satelity

Wizualizacja, którą opisuje klasa `ActiveSatellitesVis`, która pokazana jest na rysunku 4.7, wyświetla wszystkie aktywne satelity na podstawie danych ze strony CelesTrak [1]. Widok początkowy wizualizacji przedstawia za pomocą punktów pozycje satelitów w chwili obecnej. Wizualizacja dostarcza panel kontrolny, dzięki któremu można zmienić tryb ruchu kamery oraz przyspieszyć upływ czasu. Można dzięki temu zobaczyć przybliżoną pozycję satelitów w przyszłości. Panel kontrolny posiada również możliwość resetu czasu wizualizacji do chwili obecnej. Na rysunku 4.9 pokazana została wizualizacja oraz jej panel kontrolny. Widać na nim dobrze łuk, który tworzą satelity na orbicie geostacjonarnej.

Podobnie jak w przypadku wizualizacji opisanej klasą `IssVis` dane TLE pobierane są i zarządzane poprzez klasę `TLESERVICE`. Kalkulacja pozycji satelitów również wykonywana jest z użyciem mechanizmów klasy `SatelliteObject`. Różnicą pomiędzy tymi wizualizacjami jest to, że klasa `SatelliteObject` nie tworzy i nie dodaje do sceny obiektów reprezentujących satelity.

Dostępne dane zawierają informacje o około 8000 aktywnych satelitów. Wywołanie polecenia rysowania dla każdego punktu oddzielenie skutkowałoby długim czasem rysowania, ponieważ w procesie generowania grafiki najczęściej czasu tracone jest na komunikacji z GPU. Dlatego



Rys. 4.9: Aktywne satelity - klasa ActiveSatellitesVis

Listing 4.4: Fragmenty klasy ActiveSatellitesVis

```

private pointsMaterial = new THREE.PointsMaterial({
  transparent: true,
  color: 0xffffffff,
  size: 5,
  sizeAttenuation: false,
});
private points = new THREE.Points(
  new THREE.BufferGeometry(),
  this.pointsMaterial
);

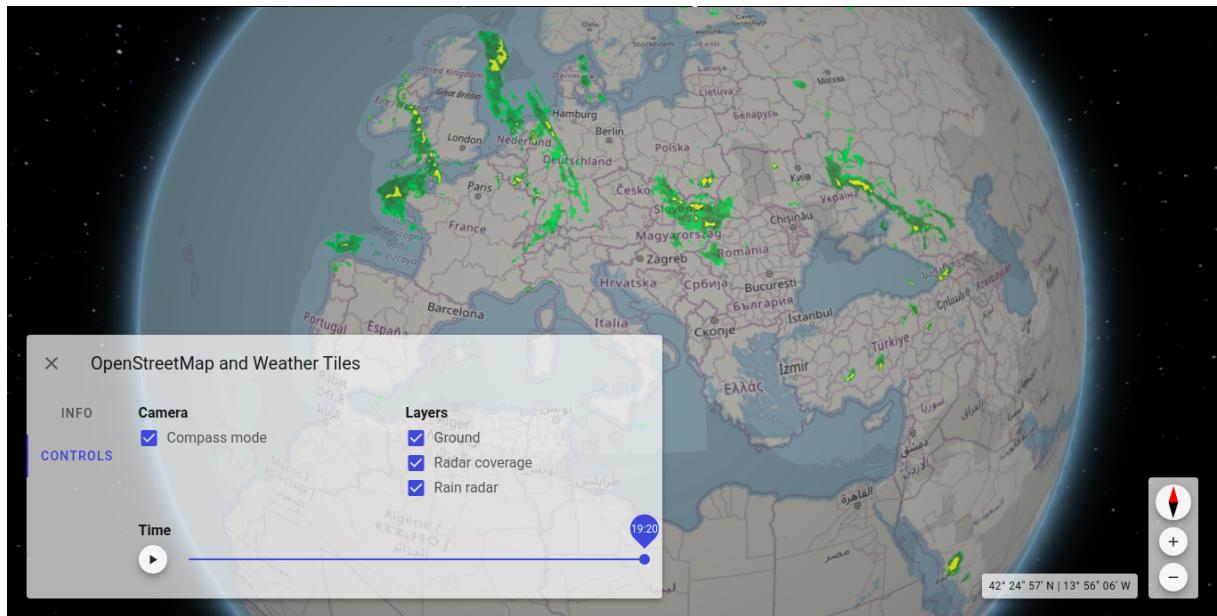
public update(deltaFrac: number) {
  /* ... */
  const points: number[] = [];
  this.satelliteObjects.forEach((o) => {
    const p = o.getPosition(this.timestamp);
    points.push(p.x, p.y, p.z);
  });
  this.points.geometry.setAttribute(
    "position",
    new THREE.BufferAttribute(new Float32Array(points), 3)
);
  /* ... */
}
  
```

wizualizacja wywołuje tylko jedno polecenie rysowania dla wszystkich punktów. Listing 4.4 zawiera inicjalizację obiektu materiału THREE.PointsMaterial oraz obiektu THREE.Points odpowiedzialnego za reprezentację punktów na scenie. Inicjalizowany jest on z pustym buforem reprezentującym pozycje wierzchołków. Rola buforów w procesie rysowania opisana została w rozdziale 3.1. W pokazanej na listingu 4.4 metodzie update, wykonywanej co każde przejście pętli głównej, bufor uzupełniany jest nowymi współrzednymi punktów. Wszystkie punkty rysowane są za pomocą jednego polecenia, a rysowanie każdego z osobna dzieje się równolegle w GPU. Największy narzut obliczeniowy spowodowany jest kalkulacją pozycji w obiektach SatelliteObject, który przy dużych liczbach jest już dostrzegalny.

4.6. Kafelki i Radar pogodowy

Wizualizacja, którą opisuje klasa OsmTiles, przedstawia Ziemię, na której nałożono wiele warstw tekstur. Pierwszą z nich jest mapa przeglądowa powierzchni, drugą jest pokrycie terenu zasięgiem radarów meteorologicznych, a trzecią z nich jest wizualizacja danych z owych radarów w danej chwili. Szczegółowość mapy zależy od przybliżenia kamery. Wizualizacja składa się z wcześniej opisanych wizualizacji gwiazd oraz atmosfery. Nie zawiera ona dynamicznego oświetlenia, a światło kierunkowe znajduje się w układzie odniesienia obserwatora, zawsze oświetlając widoczną stronę planety.

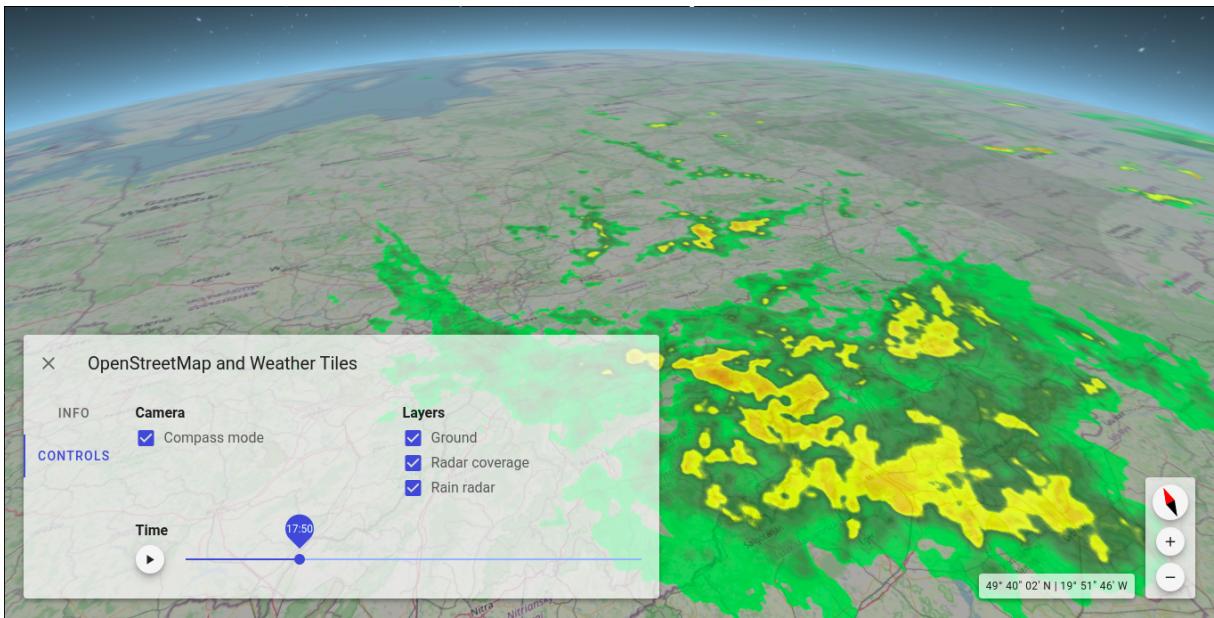
Panel kontrolny dostarczony przez wizualizację pozwala na zmianę trybu pracy kamery, na zmianę widocznych warstw oraz na zmianę czasu dla wizualizowanych opadów. Wizualizacja pozwala również na animację warstwy opadów, pozwalając na zaobserwowanie ich dynamiki w ostatnim czasie. Na rysunkach 4.10 oraz 4.11 przedstawiono mapy w małej i średniej skali.



Rys. 4.10: Opady deszczu nad Europą - klasa OsmTilesVis

4.6.1. Kafelki

Kafelki są sposobem reprezentacji danych mapy. Pozwalają na optymalizację jej wyświetlenia, ponieważ dzielą obszar na niezależne fragmenty, z których można skomponować pożądany widok. Dzięki nim można pobierać te dane, które aktualnie są potrzebne. Kafelki mogą być dowolnymi danymi, które mapują pewne wartości na powiązaną z nimi pozycję na mapie. W przy-



Rys. 4.11: Opady deszczu nad Polską - klasa OsmTilesVis

padku opisu wyglądu obiektów na mapie mogą występować w postaci grafiki wektorowej jak i rastrowej.

W celu odwzorowania powierzchni Ziemi na płaską grafikę zaszła potrzeba dobrania odpowiedniej metody jej projekcji. Odwzorowanie walcowe równokątne, zwane odwzorowaniem Merkatora jest metodą projekcji, w której kąty pomiędzy równoleżnikami i południkami są zachowane. Wszystkie południki na mapie są równoległe do siebie, a co za tym idzie, odwzorowanie to na biegunach dąży do nieskończoności. Web Mercator [27] to modyfikacja odwzorowania Merkatora wykorzystywana przez zdecydowaną większość map dostępnych w internecie. Od swojego pierwowzoru różni się tym, że obszar mapowany jest na powierzchni sfery, a nie tak jak pierwotnie, na elipsoidzie. Projekcja ta nie odwzorowuje też terenu dla szerokości geograficznej większej od $85.051\ 129^\circ$. Pozwala to na umieszczenie odwzorowanego terenu na kwadratowej powierzchni. Pojedynczy kafelek rastrowy zwykle jest kwadratem o boku 256 pikseli.

Kafelki generowane są dla różnych poziomów szczegółowości, co skutkuje różną wymaganą do ich wyświetlenia rozdzielczością przyjmując ten sam punkt odniesienia. Aby zachować stały rozmiar kafelka, kolejne poziomy szczegółowości zawierają odpowiednio więcej kafelków. Aby ujednolicić poruszanie się po kafelkach wprowadzono system współrzędnych, który jednoznacznie identyfikuje kafelek. Powiązanie kafelków ze współrzędnymi geograficznymi ich lewego górnego rogu opisane zostało za pomocą równań 4.9 - 4.10. Z równań tych wynika, że dla przybliżenia $z = 0$ mapa składa się z jednego kafelka. Liczba kafelków n , z których składa się mapa, w zależności od przybliżenia z opisana może zostać zależnością $n = 2^{2z}$.

4.6.2. Implementacja

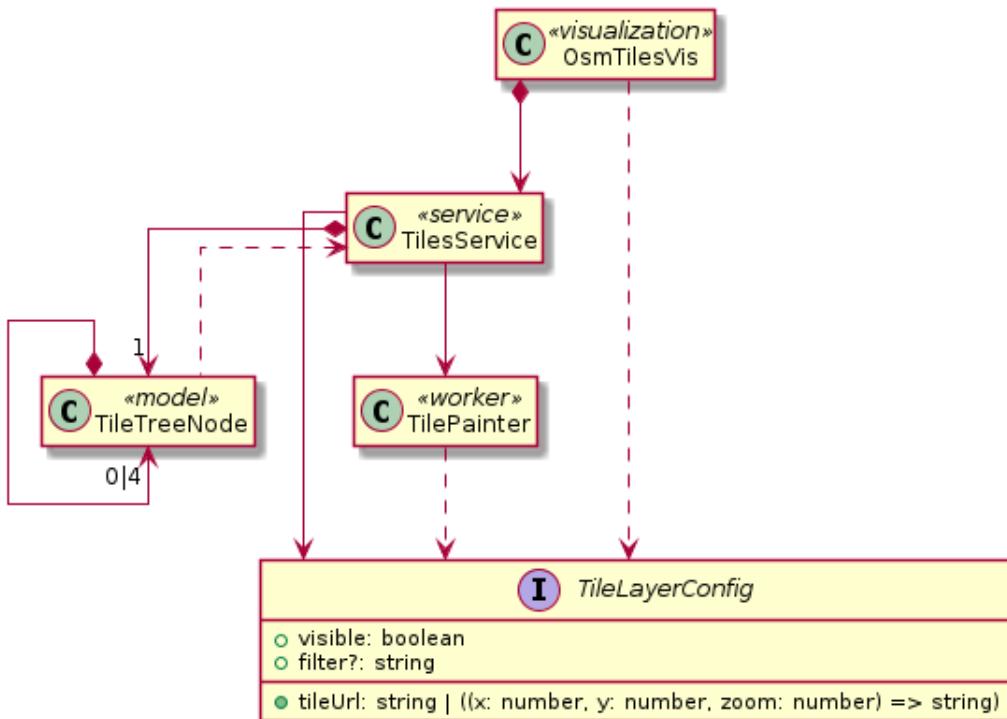
Wizualizacja, pobierając potrzebne dane kafelków, korzysta z domyślnego zestawu warstw map OpenStreetMap [12]. Dane dotyczące radaru i deszczu udostępniane są przez aplikację RainViewer [15]. Kafelki z obu tych źródeł udostępniane są w tym samym, pożądanych formacie. Na diagramie na rysunku 4.12 przedstawiono klasę `TilesService`, która odpowiedzialna jest za generowanie i zarządzanie geometrią sceny związanej z kafelkami. Wywołanie konstruktora tej klasy, pokazane na listingu 4.5 zawiera przekazany obiekt konfiguracji, który zawiera metodę zwracającą adres URL kafelka dla pożądanej pozycji i przybliżenia, ustawienie widoczności i filtry CSS, modyfikujące oryginalny wygląd pobranej grafiki.

$$x = \left\lfloor \frac{lon + 180}{360} \cdot 2^z \right\rfloor \quad (4.9)$$

$$y = \left\lfloor \left(1 - \frac{\ln \left(\tan \left(lat \cdot \frac{\pi}{180} \right) + \frac{1}{\cos(lat \cdot \frac{\pi}{180})} \right)}{\pi} \right) \cdot 2^{z-1} \right\rfloor \quad (4.10)$$

gdzie:

- x — pozycja odpowiadająca długości geograficznej
- lon — długość geograficzna w stopniach
- y — pozycja odpowiadająca szerokości geograficznej
- lat — szerokość geograficzna w stopniach
- z — stopień szczegółowości (przybliżenia)



Rys. 4.12: Zależności pomiędzy klasami wizualizacji OsmTilesVis

Każdy kafelek jest wycinkiem sfery, za którego utworzenie odpowiada konstruktor klasy THREE.SphereGeometry. Wygenerowana geometria może być współdzielona dla każdego kafelka na tej samej szerokości geograficznej i z tym samym przybliżeniem, dlatego serwis TileService zapisuje raz wygenerowane geometrię do map, których kluczami są współrzędne y i z kafelków, a następnie z niej pobierane w przypadku potrzeby ponownego użycia. Serwis ten definiuje również metody przeliczające długość i szerokość geograficzną na współrzędne x i y dla określonego przybliżenia z . Definiuje też przekształcenie odwrotne.

Kafelki tworzą strukturę drzewiastą. Korzeniem drzewa jest kafelek o współrzędnych $(x, y, z) = (0, 0, 0)$. Po zwiększeniu przybliżenia dzielony jest on na cztery mniejsze kafelki o współrzędnych $(0, 0, 1)$, $(1, 0, 1)$, $(0, 1, 1)$ i $(1, 1, 1)$. Każdy kafelek w drzewie może posiadać czterech potomków. Drzewo kafelków rozwijane jest w sposób możliwie optymalny, a wyświetlane są zawsze kafelki będące liśćmi tego drzewa. Nie ma potrzeby rozwijać poddrzewa kafelków dla obszarów będących po drugiej stronie planety, ponieważ nie jest on widoczny. Nie

Listing 4.5: Fragmenty klasy OsmTilesVis

```

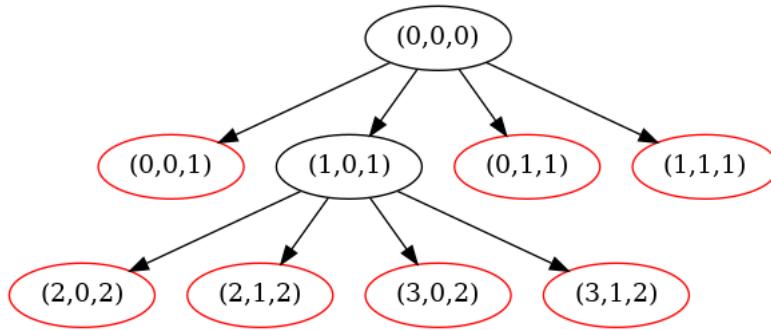
/* ... */
this.osmTilesService = new TilesService(
  [
    {
      tileUrl: (x, y, z) =>
        'https://tile.openstreetmap.org/${z}/${x}/${y}.png',
      visible: true,
      filter: "brightness(60%)",
    },
    {
      tileUrl: (x, y, z) =>
        'https://tilecache.rainviewer.com/v2/coverage/0/256/${z}/${x}/${y}
          ↴ .png',
      visible: true,
      filter: "opacity(10%)",
    },
    {
      tileUrl: (x, y, z) =>
        'https://tilecache.rainviewer.com/v2/radar/${
          this.timestamps[this.timestampIndex]
        }/256/${z}/${x}/${y}/4/1_1.png',
      visible: true,
      filter: "opacity(60%)",
    },
  ],
  100
);
/* ... */

```

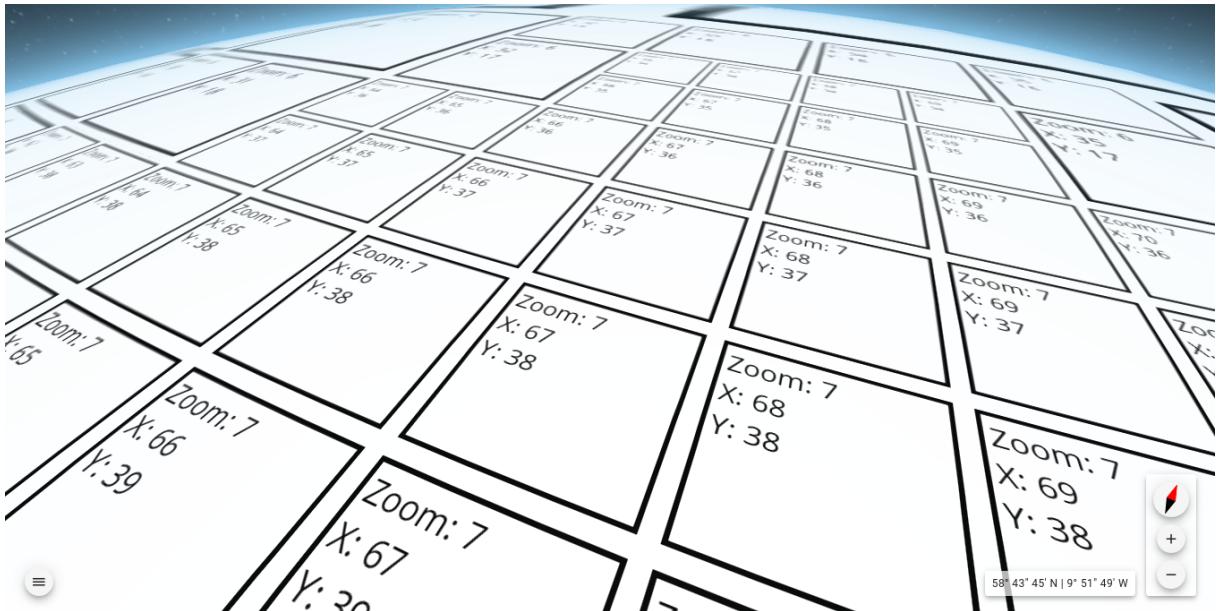
ma również potrzeby wyświetlania kafelków wysokiej rozdzielczości na horyzoncie, ponieważ znajdują się daleko, wyświetlane są one z użyciem mniejszej ilości pikseli, a co za tym idzie wymagają mniejszej dokładności. Serwis `TilesService` zawiera instancję klasy `TileTreeNode` reprezentującą węzeł drzewa, który w tym przypadku jest jego korzeniem. Węzeł drzewa odpowiedzialny jest za generowanie swoich potomków oraz niszczenie samego siebie i rysowanie swojej tekstury.

Drzewo analizowane jest pod kątem pożądanego stopnia szczegółowości z . W rekurencyjnym rozwijaniu drzewa brana jest pod uwagę odległość kafelka od kafelka, nad którym znajduje się kamera w metryce taksówkowej na tym samym poziomie szczegółowości oraz kąt pod jakim kamera spogląda na kafelek. Im większa owa odległość i kąt, tym rozwijanie kafelków zakończy się wcześniej, nie osiągając danego stopnia z w obszarach, które nie są ważne z perspektywy generowanego widoku. Działanie to jest główną metodą optymalizacji wyświetlanych kafelków. Rozwinięte wcześniej węzły drzewa, których stopień szczegółowości jest większy niż pożądany, są niszczone. Przykładowe rozwinięcie węzłów drzewa pokazano na diagramie na rysunku 4.13. Liście oznaczone kolorem czerwonym są wyświetlane. Działanie w praktyce zaobserwować można na rysunku 4.14, gdzie wszystkie kafelki zawierają grafikę z ich współrzędnymi. Na horyzoncie widać wyświetlane kafelki z mniejszym poziomem szczegółowości.

Na obiekt kafelka w procesie rysowania nałożona zostaje dynamicznie tekstura. Pochodzi ona z obiektu `Canvas`, który obsługiwany jest przez obiekt `THREE.CanvasTexture`. Pobieranie obrazów, ich późniejsze dekodowanie i późniejsza kompozycja warstw w jedną teksturową kafelkę wymaga pewnej mocy obliczeniowej. Przy mapie złożonej z dużej liczby kafelków czynności te, jeśli zajmowałby się nimi wątek główny JavaScriptu, mogły stanowić obciążenie dla urządzenia wyświetlającego wizualizację. Kod wykonujący się domyślnie w przeglądarce internetowej



Rys. 4.13: Przykładowe rozwinięcie drzewa kafelków



Rys. 4.14: Rozwinięcie drzewa kafelków - testowe kafelki

może wykonać się tylko w jednym wątku. Aby wprowadzić możliwość akceleracji obliczeń dla zadań, które mogą wykonywać się w tle i nie mają nic wspólnego z interfejsem użytkownika, stworzone zostały Web Workery [30]. Aby rozwiązać problem bezpieczeństwa w komunikacji pomiędzy wątkami, workery korzystają ze ścisłe określonego interfejsu. Nie współdzielą one między sobą pamięci, którą można przekazywać jedynie w jedną stronę poprzez operację transferu. Sama komunikacja opera się na zdarzeniach, których dane są serializowane i deserializowane na styku wątków. Aby odciążyć główny wątek w procesie rysowania sceny, operacje pobierania i kompozycji kafelków odbywają się w workerze, czyli całkowicie niezależnie od głównej pętli animacji.

Klasa `TilePainter` reprezentuje obiekt workera. Standardowy obiekt `Canvas` jest elementem DOM, do którego nie ma dostępu w workerach, ponieważ nie kontrolują one interfejsu użytkownika. Na potrzeby rysowania grafik w tle zaproponowano interfejs `OffscreenCanvas`, który zachowuje się dokładnie tak jak element `Canvas` dostępny w głównym wątku.

Kiedy użytkownik zmieni widoczność warstwy lub przesunie kamerę pokazując nowe kafelki wywołane zostają zdarzenia, które po przejściu przez mechanizm `throttle` wywołują przeliczenie drzewa kafelków. Każdy kafelek, jeśli nigdy nie miał wygenerowanej tekstury lub jeśli wymagania co do wyglądu tekstuły uległy zmianie, wysyła, poprzez serwis `TilesService`, zdarzenie do workera `TilePainter`. Następnie worker pobiera wymagane kafelki z ich źródeł i nakłada je na siebie wykorzystując zdefiniowane filtry. W zdarzeniu zwrotnym obiekt bitmapy

zostaje przetransferowany do wątku głównego i tam narysowany na elemencie `Canvas` tekstury z użyciem szybkiego kontekstu `bitmaprenderert`. Worker, w celu zaoszczędzenia pamięci i transferu sieciowego, prowadzi cache już narysowanych kafelków i omija proces pobierania i kompozycji jeśli kafelek został już wcześniej przetworzony.

Literatura

- [1] CelesTrak. <https://celesttrak.com/NORAD/elements>. Na dzień 2020-09-29.
- [2] Cesium. <https://cesium.com/docs/tutorials/getting-started/>. Na dzień: 2020-09-15.
- [3] Cesium. <https://github.com/CesiumGS/cesium/>. Na dzień: 2020-09-04.
- [4] Czas słoneczny. <https://www.pveducation.org/pvcdrrom/properties-of-sunlight/solar-time>. Na dzień 2020-09-23.
- [5] Deklinacja. <https://www.pveducation.org/pvcdrrom/properties-of-sunlight/declination-angle>. Na dzień 2020-09-23.
- [6] HTML Living Standard - The canvas element. <https://html.spec.whatwg.org/multipage/canvas.html>. Na dzień: 2020-09-03.
- [7] Interface EventTarget. <https://dom.spec.whatwg.org/#interface-eventtarget>. Na dzień 2020-09-14.
- [8] JavaScript typed arrays. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays. Na dzień 2020-09-07.
- [9] Kodowanie Base64. <https://en.wikipedia.org/wiki/Base64>. Na dzień 2020-09-18.
- [10] Lodash. <https://lodash.com/>. Na dzień 2020-09-21.
- [11] NuxtJS. <https://nuxtjs.org/>. Na dzień 2020-09-18.
- [12] Openstreetmap tiles. https://wiki.openstreetmap.org/wiki/Standard_tile_layer. Na dzień 2020-10-03.
- [13] Parcel. <https://parceljs.org/>. Na dzień 2020-09-22.
- [14] Quaternions and spatial rotation. https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation. Na dzień 2020-09-10.
- [15] Rainviewer. <https://www.rainviewer.com/api.html>. Na dzień 2020-10-03.
- [16] Retained mode versus immediate mode. <https://docs.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode>. Na dzień: 2020-09-05.
- [17] Rollup. <https://rollupjs.org/guide/en/>. Na dzień 2020-09-22.
- [18] Sferyczna interpolacja liniowa. https://en.wikipedia.org/wiki/Slerp#Quaternion_Slerp. Na dzień 2020-09-13.
- [19] Solar textures. <https://www.solarsystemscope.com/textures/>. Na dzień 2020-09-23.

-
- [20] Strongly typed events. <https://www.npmjs.com/package/strongly-typed-events>. Na dzień 2020-09-13.
 - [21] The OpenGL ES® Shading Language. https://www.khronos.org/registry/OpenGL/specs/es/3.0/GLSL_ES_Specification_3.00.pdf. Na dzień 2020-09-07.
 - [22] Three.js. <https://github.com/mrdoob/three.js/>. Na dzień 2020-09-08.
 - [23] tle.js. <https://www.npmjs.com/package/tle.js>. Na dzień 2020-09-29.
 - [24] tween.js. <https://www.npmjs.com/package/@tweenjs/tween.js>. Na dzień 2020-09-17.
 - [25] Two-Line Element. https://en.wikipedia.org/wiki/Two-line_element_set. Na dzień 2020-09-29.
 - [26] Typescript. <https://www.typescriptlang.org/>. Na dzień 2020-09-13.
 - [27] https://en.wikipedia.org/wiki/Web_Mercator_projection. Web Mercator projection. Na dzień 2020-10-03.
 - [28] Vue.js. <https://v3.vuejs.org/guide/introduction.html>. Na dzień 2020-09-18.
 - [29] Vuetify. <https://vuetifyjs.com/en/>. Na dzień 2020-09-21.
 - [30] Web Workers. <https://html.spec.whatwg.org/multipage/workers.html#workers>. Na dzień 2020-10-03.
 - [31] WebGL 2.0 Specification. <https://www.khronos.org/registry/webgl/specs/latest/2.0/>. Na dzień: 2020-09-03.
 - [32] Webpack. <https://webpack.js.org/concepts/>. Na dzień 2020-09-21.
 - [33] Współrzędne jednorodne. https://en.wikipedia.org/wiki/Homogeneous_coordinates. Na dzień 2020-09-08.
 - [34] D. Dorrell, J. Henderson, T. Lindley, and G. Connor. *Introduction to Human Geography (2nd Edition)*. GALILEO, University System of Georgia, 2019.
 - [35] F. Ghayour and D. Cantor. *Real-Time 3D Graphics with WebGL 2, Second Edition*. Packt Publishing, 2018.
 - [36] C. N. Knaflic. *Storytelling with Data*. Wiley John&Sons Inc, 2015.