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:

WROCŁAW, 2020

Spis treści

Spis rysunków

Spis tabel

Spis listingów

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*)

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 całek 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 analizę danych w lepszym stopniu i łatwiejsze wyciągnięcie wniosków. Dobrze skonstruowana wizualizacja, w przypadku prezentacji jej większemu gronu odbiorców, pozwala również na skuteczniejsze zainteresowanie grupy tematem oraz pomóc w opowiedzeniu historii, a co za tym idzie, na wyciągnięcie przez odbiorców właściwych wniosków [11].

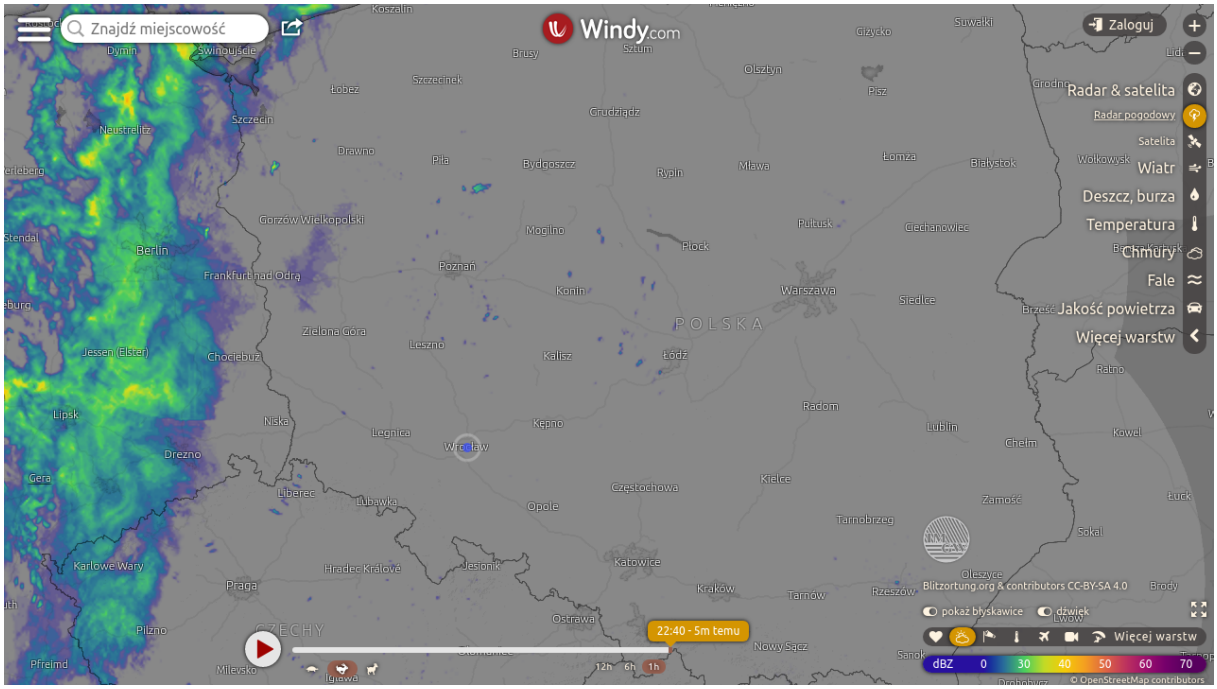
1.1. Istota rzeczy

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 nazywa 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. [9, Rozdział 1.6].

Wizualizacje danych mogą tyczyć 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 profesjonalnego użytkownika, ale odstraszyć może niezagłębionego w temat odbiorcę. Przystępność odbioru wiąże się również z szybkością uzyskania dostępu do samej platformy obsługującej wizualizację. Alternatywą



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

dla instalowanych aplikacji desktopowych jest przeglądarka internetowa. Tworzy ona środowisko, które może być uruchomione na wielu systemach operacyjnych, również na urządzeniach mobilnych, a zaimplementowane wspomaganie sprzętowe generowania grafiki i interfejsy takie jak *HTML5 Canvas*[2] i *WebGL*[7] 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 związany 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 2D lub 3D. 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. Pozwalają skupić

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

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 geoprzestrzenne konfigurowalne są zwykle z poziomu języka JavaScript. Przykładem takiej biblioteki jest Cesium[1]. Potrafi ona generować wizualizacje 2D i 3D 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: `https://cesium.com/docs/tutorials/getting-started/`

```
<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 2D lub 3D. 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 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 - zbiór komponentów odpowiedzialnych 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 będzie sceną. Opis zachowań i implementacji projektu w dalszej części pracy będzie odnosił się do numeru wymagania w celu wskazania jego spełnienia.

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 zagnieżdżać sceny predefiniowane w silniku, oraz sceny wcześniej stworzonych przez siebie.
RA_5	Twórca może parametryzować sceny w celu określonej ich modyfikacji w procesie zagnieżdżania.
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 odpowiedzią 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 ustanowione 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 *Single Page Application*.
RU_8	Jeśli to możliwe aplikacja powinna wykorzystywać sprzętową akcelerację obliczeń graficznych.
RU_9	Aplikacja powinna ustawiać i obsługiwać adres URL w przeglądarce definiujący wyświetlaną wizualizację.

Tab. 2.5: Wymagania funkcjonalne zdefiniowane dla aplikacji

Rozdział 3

Silnik

Rozdział ten opisuje główny komponent tworzonego systemu nazwanego Silnikiem. Odpowiedzialny 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. Zainicjowanie procedury obsługi tych zdarzeń również komponent Silnika.

Najpierw w sposób uproszczony opisany został sposób renderowania grafiki z wykorzystaniem API WebGL oraz biblioteki Three.js. Następnie przedstawiono 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 3D w przeglądarce. Złożone obiekty rysowane są tylko za pomocą punktów, linii i trójkątów. WebGL działa w trybie *immidiate*, 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[10, Rozdział 1]. WebGL korzysta z akceleracji sprzętowej podczas renderowania grafiki - działa na GPU. W przypadku kart graficznych bez wsparcia dla ten 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[4].

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 ustawiony stan do czasu jego zmiany przez aplikację. Wszystkie operacje renderowania grafiki korzystają z globalnie ustawio-

nych parametrów, które definiują stan kontekstu i mają bezpośredni wpływ na efekt końcowy[10, 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 [3]. W języku JavaScript występuje jeden typ `number` przechowujący liczby, które wewnętrznie reprezentowane są jako 64b liczba zmiennoprzecinkowa. Dodatkowo każda zmienna numeryczna jest obiektem typu `Number` z własnymi metodami. Użycie buforów z interfejsem tablicy przyspiesza 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 składnią język C/C++[5] i zawiera wbudowane funkcje wymagane do operacji matematycznych takich jak iloczyn skalarny wektorów, czy mnożenie macierzy. Na wspomniany *program* składają się dwa 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. Dale 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 afinicznych, które transformują pozycję zbioru wierzchołków i nie zaburzają relacji pomiędzy nimi. Efektywnie transformacja taka jest mnożeniem macierzy transformacji o wymiarach 4x4 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[8]. Transformacja pozycji modelu odbywa się z pomocą macierzy M, a transformacja 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

2x2x2 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*.

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

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

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

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[6] 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, świateł i materiałów w postaci obiektowej. Posiada rozbudowany system animacji oraz wsparcie dla systemów wirtualnej rzeczywistości. Na listingu 3.2 pokazano kod aplikacji, która wyświetla zielony sześcian.

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();
```

Na początku tworzony jest obiekt sceny, która 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.WegGLRenderer` 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 1x1x1. Obiekt ten posiada atrybuty ułatwiające zarządzanie wygenerowaną geometrią. Zwykłe obiekty geometrii są konwertowane do typu `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 świateł, 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. Dlatego zastosowano dyrektywę `#include` w celu umieszczenia w kodzie wspólnych 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

3.2.1. Orbita globalna

3.2.2. Orbita lokalna

3.2.3. Animacje - płynność ruchów

3.3. Implementacja

Literatura

[1] Cesium. https://github.com/CesiumGS/cesium/. Na dzień: 2020-09-04.

[2] HTML Living Standard - The canvas element. https://html.spec.whatwg.org/multipage/canvas.html. Na dzień: 2020-09-03.

[3] JavaScript typed arrays. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays. Na dzień 2020-09-07.

[4] Retained mode versus immediate mode. https://docs.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode. Na dzień: 2020-09-05.

[5] 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.

[6] Three.js. https://github.com/mrdoob/three.js/. Na dzień 2020-09-08.

[7] WebGL 2.0 Specification. https://www.khronos.org/registry/webgl/specs/latest/2.0/. Na dzień: 2020-09-03.

[8] Współrzędne jednorodne. https://en.wikipedia.org/wiki/Homogeneous_coordinates. Na dzień 2020-09-08.

[9] D. Dorrell, J. Henderson, T. Lindley, and G. Connor. *Introduction to Human Geography (2nd Edition)*. GALILEO, University System of Georgia, 2019.

[10] F. Ghayour and D. Cantor. *Real-Time 3D Graphics with WebGL 2, Second Edition*. Packt Publishing, 2018.

[11] C. N. Knaflic. *Storytelling with Data*. Wiley John&Sons Inc, 2015.