Tworzenie aplikacji desktop z wykorzystaniem bibliotek Electron oraz vtk.js
Projekt indywidualny

Autor: Weronika Pijagin
Opiekun naukowy: mgr inż. Bartosz Chaber

24 stycznia 2017

1 Biblioteka Electron

Biblioteka **Electron** umożliwia tworzenie aplikacji desktop w bardzo prosty i szybki sposób. Oprogramowanie tworzymy za pomocą JavaScrpit, HTML i CSS w taki sposób, w jaki tworzymy stronę internetową. Biblioteka zapewnia obsługę takich zdarzeń jak przeładowanie, zamknięcie okna, zwiększanie-/zmniejszanie rozmiaru zawartości.

Właściwą część aplikacji tworzymy za pomocą strony HTML wyświetlanej przez bibliotekę Electron. Umieszczamy w niej skrypty, które odpowiadają za wyświetlanie potrzebnych elementów oraz logikę aplikacji. Podczas tworzenia mojej aplikacji, zagnieździłam w stronie HTML skrypt, który wykorzystuje bibliotekę vtk.js.

2 Biblioteka vtk.js

Vtk.js jest biblioteką open-source, która służy do wizualizacji danych. Do wyświetlania grafiki 3D wykorzystuje WebGL. Jest to wersja biblioteki VTK (napisanej głównie w C++) zaimplementowana w języku JavaScript. Oprogramowanie dostarcza wiele algorytmów, które możemy wykorzystać do przetwarzania danych.

3 Instalacja bibliotek

Electron zbudowany jest z wykorzystaniem **Node.js**. Do jej pobrania wykorzystujemy narzędzie do zarządzania pakietami **npm**. W ten sposób pobieramy również **vtk.js**.

4 Niezgodność wersji JavaScript

Przy tworzeniu swojej aplikacji natknęłam się na problem różnych wersji JavaScript, w którym napisane są biblioteki. Electron napisany jest w standardzie ES5, a vtk.js w ES6. Żeby oba frameworki mogły ze sobą współpracować, skompilowałam kod źródłowy vtk.js do starszej wersji za pomocą narzędzia **babel.js**.

4.1 Babel

Biblioteka Babel umożliwia zmianę nieobsługiwanych elementów składni nowszej wersji na ich starsze odpowiedniki. Jedną z najważniejszych różnic jest zmiana importowania modułów w plikach.

Jeżeli chcemy uniknąć korzystania z jakichkolwiek poleceń biblioteki, możemy skorzystać z narzędzia udostępnionego na stronie internetowej https://babeljs.io/, w zakładce *Try it out*. Możemy w ten sposób przekompilować każdy plik pojedynczo, jednak jest to bardzo czasochłonne.

Inny sposób to skorzystanie z polecenia:

```
1  babel <nazwa_pliku_es6> --presets es2015
2    --out-file <nazwa_pliku_es5>
```

Listing 1: Użycie biblioteki Babel

Oraz podobnie dla katalogu:

```
1  babel <nazwa_kat_es6> --presets es2015
2    --out-dir <nazwa_kat_es5>
```

Listing 2: Użycie biblioteki Babel dla katalogu

Aby możliwe było użycie tego polecenia, musimy zainstalować poniższe rozszerzenia:

```
1  npm install babel-preset-es2015 --save --no-bin-links
2  npm install babel-preset-react --save --no-bin-links
```

Listing 3: Instalacja dodatkowych rozszerzeń

Musimy też pamiętać, aby katalog, który chcemy przekompilować oraz katalog *node_ modules* (do którego npm instaluje dodatkowe moduły) znajdowały się na tym samym poziomie.

5 Tworzenie aplikacji

5.1 Konfiguracja

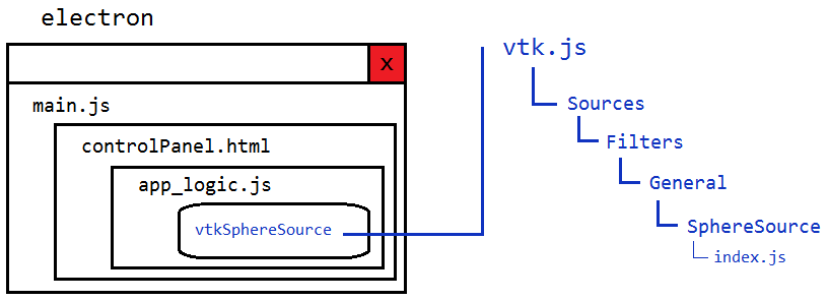
Konfiguracja aplikacji zapisana jest w pliku **package.json**. Tam podajemy nazwę procesu głównego **main**, który odpowiedzialny jest za wyświetlanie dokumentu HTML stanowiącego najważniejszą część aplikacji.

5.2 Zależności pomiędzy plikami

Poniżej przedstawiłam jak poszczególne pliki zależą od siebie. W aplikacji najważniejszym plikiem jest **main.js**, który następnie ładuje dokument HTML **controlPanel.html**. W dokumencie wykonywany jest skrypt **app_logic.js**, który jest w stanie korzystać z modułu **vtk.js**.

```
- main.js
- app_logic.js
- controlPanel.html
- package.json
-- node_modules
   |
   - vtk.js
   - babel
```

Listing 4: Struktura plików i katalogów



Rysunek 1: Schemat budowy aplikacji

5.3 Procesy w aplikacji

5.3.1 Main

Z poziomu tego procesu mamy dostęp jedynie do elementów dotyczących okna przeglądarki. W tym miejscu tworzone są instancje **BrowserWindow** wyświetlające strony HTML. Każda taka instancja może uruchomić swój własny proces renderujący. W procesie main możemy m.in. zamknąć, czy zminimalizować okno aplikacji. Przy tworzeniu instancji BrowserWindow określamy jego wymiary.

```
1  mainWindow = new BrowserWindow({width: 1200, height: 900,
       offscreen: true});
```

Listing 5: Tworzenie instancji BrowserWindow

5.3.2 Renderer

Proces ten odpowiedzialny jest za całą logikę naszej aplikacji. Ma on dostęp do elementów DOM, więc za pomocą skryptów JavaScript możemy nimi dowolnie manipulować. Ja wykorzystałam tylko jeden taki proces, który wyświetla zarówno scenę z renderowanymi komponentami graficznymi, jak i panel służący do sterowania nimi.

5.3.3 Komunikacja między procesami

Jeśli chcemy przesłać informację pomiędzy procesem głównym, a renderującym, musimy skorzystać z odpowiednio **ipcMain** oraz **ipcRenderer**. Komunikacja może odbywać się w sposób synchroniczny lub asynchroniczny. Mamy możliwość wysyłania oraz odbierania komunikatów z obu procesów. Tylko dzięki zastosowaniu tej komunikacji jesteśmy w stanie wpłynąć z poziomu dokumentu HTML na aplikację Electron (np. zamknięcie okna aplikacji, przeładowanie strony).

```
1  \\ Renderer:
2  function reload_scene() {
3    (...)
4    const {ipcRenderer} = require('electron');
5    ipcRenderer.sendSync('synchronous-message', 'reload');
6  }
7
8  \\ Main:
9  const {ipcMain} = require('electron');
10 ipcMain.on('synchronous-message', function(event, arg) {
11   if (arg == "reload") {
12     event.sender.send('asynchronous-reply', '');
```

```
13        event.returnValue = '';
14        mainWindow.reload();
15      }
16    });
```

Listing 6: Komunikacja za pomocą IPC

5.4 Użycie biblioteki vtk.js

Vtk.js używamy jako jednego z modułów Node.js. Mamy dostęp do przejrzystego API oraz dokumentacji tej biblioteki, więc znalezienie odpowiedniej funkcji nie powinno być trudne. Użytkownicy VTK napisanej w C++ nie powinni mieć problemu ze stosowaniem się do zasad nowej wersji. Z pomocą często przychodzi badanie zawartości oraz funkcjonalności vtk.js za pomocą narzędzi przeglądarki w oknie aplikacji. Tutaj znajdziemy między innymi odpowiednie nazwy szukanych przez nas klas. W przypadku gdy nieodpowiednio skompilowaliśmy kod źródłowy za pomocą *babel*, będziemy w stanie zauważyć brak niektórych klas z biblioteki vtk.js.

Aby wykorzystać moduł vtk.js w aplikacji, wystarczy umieścić kod w skrypcie w wyświetlanym dokumencie HTML. Dokument ten przekazujemy w procesie main za pomocą metody *loadURL*.

```
1   mainWindow.loadURL(url.format({
2     pathname: path.join(__dirname, 'controlPanel.html'),
3     protocol: 'file:',
4     slashes: true
5   }));
```

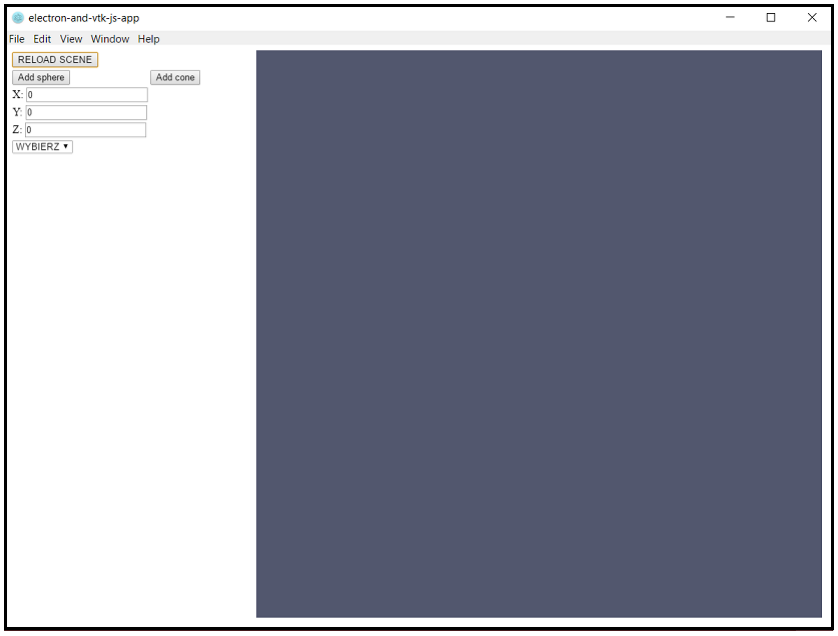
Listing 7: Wczytanie dokumentu HTML

Vtk.js staje się w tym momencie procesem renderującym. Biblioteka tworzy element HTML *canvas*, w którym następnie wyświetla odpowiednie komponenty. Po zaimportowaniu vtk.js za pomocą *require* otrzymujemy dostęp do zawartości tej biblioteki.

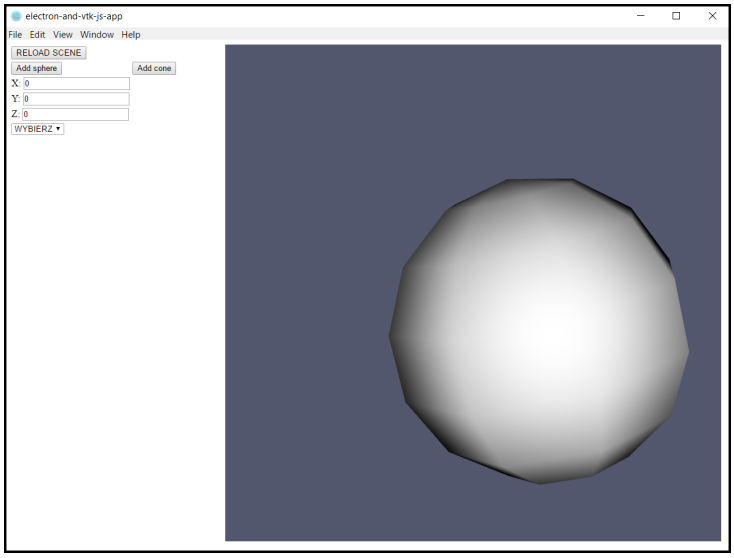
5.5 Interfejs graficzny aplikacji

Po pokonaniu wszelkich trudności i dostosowaniu do siebie bibliotek Electron oraz vtk.js, byłam w stanie stworzyć aplikację, której wygląd przedstawiłam na Rysunku 2.

Na Rysunku 3 przedstawiłam jak wygląda aplikacja po wyrenderowaniu kształtu - w tym przypadku sfery.

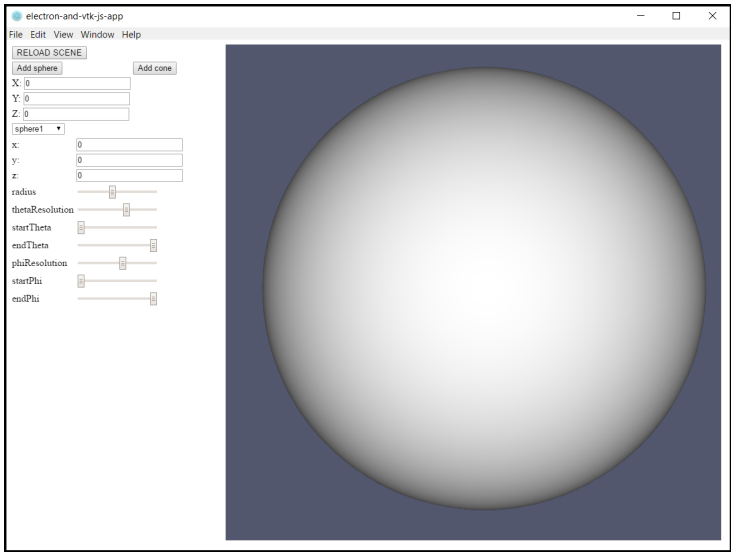


Rysunek 2: Podstawowy ekran aplikacji



Rysunek 3: Ekran przedstawiający wyrenderowaną sferę

Rysunek 4 przedstawia zmianę właściwości wyrenderowanego kształtu za pomocą różnych opcji w menu.



Rysunek 4: Ekran przedstawiający zmianę właściwości sfery

6 Ogólne wrażenia

Tworzenie aplikacji z wykorzystaniem wyżej wymienionych bibliotek jest, po pokonaniu niewielu początkowych trudności, proste i szybkie. Kod źródłowy potrzebny do uruchomienia aplikacji Electron może zająć niewiele ponad kilkanaście linijek, a otwiera bardzo wiele możliwości. Instalacja modułów za pomocą npm nie stanowi problemu. Oprogramowanie vtk.js również nie jest skomplikowane w użyciu, a pozwala na zaawansowane operacje na zbiorach danych oraz ich wizualizację. WebGL renderuje komponenty graficzne sprawnie i bez powodowania spowolnienia czy zacinania się aplikacji.