Witold Alda

Grafika komputerowa. Laboratorium 1.

WebGL: Grafika 3D w przeglądarkach internetowych

WebGL jest okrojoną wersją biblioteki OpenGL, która służy do prezentowania grafiki 3D w przeglądarkach internetowych. Naturalnym środowiskiem dla WebGL jest Javascript, który to kod jest z kolei zanurzony w pliku HTML5.

Większość współczesnych przeglądarek automatycznie obsługuje polecenia WebGL, choć efekty ich działania mogą się nieco różnić między sobą

Programowanie w WebGL może odbywać się „niskopoziomowo" – z bezpośrednim wykorzystaniem wierzchołków, buforów, macierzy transformacji i rzutowania oraz shaderów – programów cieniujących wykonywanych bezpośrednio, przypominając styl programowania współczesnej wersji OpenGL. Można również wykorzystać istniejące biblioteki, które przenoszą programowanie na poziom budowania sceny 3D z dostępnych obiektów. To drugie podejście jest prostsze, pozwala na szybkie uzyskanie interesujących efektów graficznych, choć wprowadza też pewne ograniczenia w renderowaniu.

W poniższym ćwiczeniu będziemy korzystać raczej z bibliotek „wysokopoziomowych". W następnych również, choć będziemy je też łączyć z programowaniem shaderów.

W WebGL obiektowe biblioteki wysokopoziomowe cieszą się dużą popularnością, podczas gdy OpenGL są zdecydowanie mniej powszechne.

Biblioteki wysokopoziomowe

Zapisywanie kodu grafiki wyłącznie za pomocą funkcji WebGL i shaderów jest generalnie dosyć uciążliwe. W praktyce często staramy się uciec od takiego rozwiązania i poszukujemy bibliotek, które pozwalają na programowanie bardziej intuicyjne i oderwane o technicznych, niskopoziomowych aspektów grafiki 3D.
WebGL oferuje przynajmniej kilka takich rozwiązań, które ułatwiają programowanie i radykalnie skracają kod. Z popularnych bibliotek możemy wymienić np.:

- Three.js (*https://github.com/mrdoob/three.js*) - z niej będziemy korzystać,
- BabylonJS (*http://babylonjs.com*), warta rozważenia konkurencja dla Three.js

Trochę z obowiązku wymieniam niektóre biblioteki/frameworki, które dobrze się zapowiadały, ale zostały zarzucone. Może jednak ktoś do nich wróci i je odświeży. Każda z tych bibliotek ma nieco inną filozofię budowania i renderowania scen, ale wszystkie starają się realizować wysokopoziomowy i przyjazny dla użytkownika interfejs programisty.

- GLGE (*http://www.glge.org*), projekt zarzucony w 2012 r.,
- SceneJS （*http://www.scenejs.org*), zarzucony w 2015 r., nawet strona już nie żyje,

- PhiloGL (*http://www.senchalabs.org/philogl*), zarzucony w 2012 r., a ciekawie się zapowiadał…
- SpiderGL (*http://spidergl.sourceforge.net*), zarzucony w 2010 r.

Dobry spis dwudziestu najpopularniejszych (najlepszych?) frameworków *Open Source*, jest na stronie https://medevel.com/16-webgl-opensource-frameworks/ Choć i ten spis nie jest już najnowszy bo pochodzi z roku 20219.

W dalszym ciągu będziemy się opierać na bibliotece three.js (*https://github.com/mrdoob/three.js*). Użytą w ćwiczeniu wersją biblioteki jest r116 – nie jest to wersja najnowsza.
Strona domowa *www.threejs.org* zawiera dokumentację biblioteki, liczne przykłady oraz specyfikację poszczególnych klas. 06.03.2022 aktualną wersją biblioteki jest r138. Raczej wyjątkowo w tym ćwiczeniu użyjemy starszej wersji biblioteki, bo łatwiej w niej uruchomić przykłady.

Biblioteka three.js

W bibliotece three.js scena jest budowana z obiektów. Sama scena jest również obiektem – kontenerem – w którym umieszczane są inne obiekty, zarówno te geometryczne jak i pozostałe charakteryzujące scenę.

Podstawowe obiekty, które tworzymy na podstawie wbudowanych w bibliotekę klas, to:

- **Renderery** (są przynajmniej dwa: Canvas renderer i WebGL renderer, ten ostatni z większymi możliwościami, ale też i potrzebami względem sprzętu – te jednak na ogół są dostępne nawet na przeciętnych komputerach)
- **Formy geometryczne**, gotowe, zapewnione przez bibliotekę (takie jak sześcian, sfera, itp.) oraz tworzone ręcznie jako siatka wielokątów.
- **Obiekty reprezentujące obserwatora sceny** (*camera*): jego położenie, punkt na który patrzy, kąt widzenie, etc.
- **Źródła światła**, które oświetlają scenę.
- **Materiały przypisane formom geometrycznym**, określające ich kolor i sposób odbijania światła.

Obiekty three.js mają swoje zmienne składowe, którym należy nadać określone wartości oraz swoje metody. Dobrą choć niekompletną dokumentacją opisującą między innymi strukturę obiektów jest ta na stronie *threejs.org*. Sugeruję korzystać z niej na bieżąco, w miarę potrzeb.

Przykłady do uruchomienia

Example 01

Example 01 rysuje trójkąt i kwadrat w kolorze białym. Bieżący przykład i kolejne w tym zestawie korzystają z biblioteki three.js, jednak figury tworzone są ręcznie, na podstawie wierzchołków – raczej nietypowy sposób w three.js.

W kodzie proszę zwrócić uwagę na:

- Stworzenie sceny.
  `scene = new THREE.Scene();`
  do sceny dołącza się następnie wszystkie obiekty: nasze obiekty geometryczne wraz materiałami, kamerę, źródła światła.
- Ustalenie położenia obserwatora – stworzenie kamery. Jakie parametry ma kamera, jakie współrzędne i gdzie jest skierowana? Jak dodaje się kamerę do sceny? Wygodnie posłużyć się dokumentacją ze strony threejs.org.
  W przykładzie mamy:
  ```
  camera = new THREE.PerspectiveCamera
           (45, canvasWidth/canvasHeight, 1, 100);
  camera.position.set(0, 0, 10);
  camera.lookAt(scene.position);
  scene.add(camera);
  ```
  Obserwator (camera) jest tworzony jako obiekt klasy `PerspectiveCamera` z parametrami określającymi kąt widzenia, proporcje kadru, bliski i daleki zakres widzenia. Po szczegóły proszę sięgnąć do dokumentacji.
- Jak buduje się geometrię trójkąta z wierzchołków?
  ```
  var triangleGeometry = new THREE.Geometry();
  triangleGeometry.vertices.push(new THREE.Vector3( 0.0, 1.0,
  0.0));
  triangleGeometry.vertices.push(new THREE.Vector3(-1.0, -1.0,
  0.0));
  triangleGeometry.vertices.push(new THREE.Vector3( 1.0, -1.0,
  0.0));
  triangleGeometry.faces.push(new THREE.Face3(0, 1, 2));
  ```
  W przykładzie posługujemy się obiektem z ogólnej klasy `Geometry`, w którym wypełniamy tablicę wierzchołków `(vertices)` współrzędnymi z klasy `Vector3`. W następnym kroku wypełniamy tablicę ścian (`faces`). `vertices` i `faces` są standardowymi polami w strukturze klasy `Geometry`.
- Jak tworzy się materiał, który określa m.in. kolor trójkąta?
  ```
  var triangleMaterial = new THREE.MeshBasicMaterial({
  color:0xFFFFFF,
  side:THREE.DoubleSide });
  ```
  Wykorzystujemy tu najprostszą klasę `MeshBasicMaterial` która nadaje obiektowi jednolity kolor (choć możliwości tej klasy są znacznie większe).
- Jak z geometrii i materiału tworzy się obiekt klasy Mesh, który można dodać do sceny?
  ```
  var triangleMesh = new THREE.Mesh(triangleGeometry,
  triangleMaterial);
  ```
- Jak można zmienić położenie gotowego obiektu na scenie?
  W przykładzie mamy:
  ```
  triangleMesh.position.set(-1.5, 0.0, 4.0);
  ```

Zmianę położenia obiektu można rozszerzyć o inne dostępne transformacje dla obiektów geometrycznych i kamery:

Atrybut	Opis
`position`	Określa położenie x,y,z obiektu, względem obiektu – rodzica. Zazwyczaj obiektem odniesienia jest `Scene`.
`rotation`	Określa obrót obiektu wokół każdej z osi (kąt podaje się w radianach).
`scale`	Określa skalowanie obiektu wzdłuż każdej z osi.
`translateX(amount)`	Przesuwa obiekt wzdłuż osi x o zadaną wielkość.
`translateY(amount)`	Przesuwa obiekt wzdłuż osi y o zadaną wielkość

`translateZ(amount)` Przesuwa obiekt wzdłuż osi z o zadaną wielkość

Położenie obiektu można ustawić na trzy sposoby.
Pierwszy:

```
cube.position.x=10;
cube.position.y=3;
cube.position.z=1;
```

Drugi: `cube.position.set(10,3,1);`

Trzeci `cube.postion=new THREE.Vector3(10,3,1);`

Podobnie można wykonać operacje `rotation` i `scale`. Nie działa jednak opcja `rotation.set` i `scale.set`

Example 02

Przykład 02 rysuje trójkąt i kwadrat z interpolowanymi kolorami. W przykładzie można odczytać w jaki sposób kolor jest przypisywany do każdego wierzchołka osobno. Niczego tu nie zmieniamy.

Example 03

Przykład 03 obraca oba obiekty. Nowością jest tu wprowadzenie prostej animacji. Odpowiedzialna jest za to nasza funkcja `animateScene()`, która jest z kolei wywoływana rekurencyjnie poprzez systemową funkcję `requestAnimationFrame(animateScene);` To wywołanie umożliwia automatyczną realizację animacji z prędkością 60 klatek na sekundę (jeżeli tylko komputer daje radę). Po przeanalizowaniu kodu możemy zmienić szybkość i oś obrotów.

Do zrobienia na podstawie przykładów 01, 02 i 03

Po przeanalizowaniu kodu

1. Zmień kolor trójkąta i kwadratu z białego na inny.
2. Przesuń obiekty do przodu, do tyłu – oceń co się zmienia. Podnieś kamerę i spójrz na oba obiekty z góry
3. Zbuduj z trzech przesuniętych trójkątów choinkę, a z kwadratu i trójkąta – domek. Można oczywiście rozbudować choinkę o pień, domek o komin, okno i drzwi, albo w ogóle złożyć coś innego. Można wykorzystać elementy kolejnych przykładów, np. choinka się obraca. Każdy poziom choinki można zbudować z prostopadłych trójkątów – wtedy podczas obrotów choinka zawsze będzie widoczna.
   Można też dołożyć najprostszy pojazd zbudowany z prostokąta na dwóch kołach. Zamiast budować koło ręcznie z trójkątów (co jest oczywiście pouczające) wygodniej wykorzystać obiekt z klasy `CircleGeometry`:
   ```
   var geometry = new THREE.CircleGeometry( 5, 32 );
   ```
   Dwa parametry w konstruktorze oznaczają odpowiednio promień i liczbę segmentów (trójkątów) z jakich zbudowane jest koło. Pojazd oczywiście warto zaanimować, tak żeby przejeżdżał przez scenę.
   Scenę w tym stylu, w formie pliku html (bez bibliotek) proszę przysłać jako zadanie laboratorium 1.

Example 04

Przykład 04 demonstruje pierwsze bryły 3d (budowane ręcznie) i ich obroty. Wykorzystujemy tu gotowe bryły (graphics primitives) z grupy Geometry. Proszę spojrzeć do dokumentacji na stronie threejs.org i zobaczyć jakie parametry są związane z poszczególnymi bryłami

Do zrobienia w przykładzie 04

Spróbuj zmodyfikować kształty brył (np. zamienić piramidkę w ścięty stożek) i kolory. Do poprzedniej sceny dołóż choinkę ze stożków. Czy wygląda lepiej?