Podstawy Three.js

April 23, 2024

1 Introduction

Celem jest wprowadzenie do podstaw biblioteki Three.js

Oto strona projektu Ricardo Cabello (autora biblioteki)
`https://mrdoob.com/`

Oto strona biblioteki Three.js `http://threejs.org`

Three.js używa pojęć, które już znasz, takich jak obiekty geometryczne, transformacje, światła, materiały, tekstury i kamery. Ale ma też dodatkowe funkcje, które bazują na mocy i elastyczności WegGL.

Możesz pobrać three.js i przeczytać dokumentację na głównej stronie internetowej `http://threejs.org`. Pobieranie jest dość duże, ponieważ zawiera wiele przykładów i plików pomocy.

Podstawowe funkcje three.js są zdefiniowane w jednym dużym pliku JavaScript o nazwie **„three.js”**, który można znaleźć w katalogu kompilacji w pliku do pobrania. Istnieje również mniejsza wersja „minified”, **three.min.js**, która zawiera te same definicje w formacie, który nie jest być czytelny dla człowieka.

Aby użyć three.js na stronie internetowej, należy dołączyć jeden z dwóch skryptów w elemencie <script> na stronie. Na przykład, zakładając, że three.min.js znajduje się w tym samym folderze, co strona internetowa, wtedy element skryptu będzie:

```
<script src="three.min.js"></script>
```

Oprócz tego pobieranie core three.js zawiera katalog zawierający wiele przykładów i różne pliki obsługi, które są używane w przykładach. Chociaż będę trzymał się głównie core, skorzystam z kilku dodatków.

2 Scena, Renderer, kamera

Three.js działa z elementem HTML `<canvas>`, tym samym, co w przypadku grafiki 2D. W wielu przeglądarkach internetowych oprócz interfejsu API grafiki 2D, kanwa obsługuje również rysowanie w 3D przy użyciu WebGL, który jest mniej więcej taki sam, jak może być z 2D API. WebGL nie jest dostępny w niektórych przeglądarkach obsługujących `<canvas>`. Dotyczy to na przykład przeglądarki Internet Explorer 9 i 10. Ale WebGL jest zaimplementowany w

Internet Explorerze 11, a także w najnowszych wersjach Chrome, Safari, Firefox i Edge. Działa również w przeglądarkach na wielu urządzeniach mobilnych.

Three.js to interfejs API zorientowanego obiektowo grafu sceny. Podstawową procedurą jest zbudowanie grafu sceny z obiektów three.js, a następnie renderowanie obrazu sceny, którą reprezentuje. Animację można zaimplementować, modyfikując właściwości grafu sceny między ramkami.

Biblioteka three.js składa się z dużej liczby klas. Trzy najbardziej podstawowe to THREE.Scene, THREE.Camera i THREE.WebGLRenderer. Program three.js będzie potrzebował co najmniej jednego obiektu każdego typu. Obiekty te są często przechowywane w zmiennych globalnych

```
var scene, renderer, camera;
```

Zauważ, że prawie wszystkie trzy klasy i stałe, które będziemy używać, są właściwościami obiektu o nazwie `THREE`, a ich nazwy zaczynają się od **„THREE”**. Czasem będę odnosił się do klas bez użycia tego prefiksu i zwykle nie jest używany w dokumentacji three.js, ale prefiks musi zawsze być zawarty w rzeczywistym kodzie programu.

Obiekt `Scene` jest uchwytem dla wszystkich obiektów tworzących świat 3D, w tym świateł, obiektów graficznych i prawdopodobnie kamer. Działa jako węzeł główny dla grafu sceny.

`Camera` to specjalny rodzaj obiektu, który reprezentuje punkt widzenia, z którego można zrobić obraz świata 3D. Reprezentuje kombinację transformacji oglądania i projekcji.

`WebGLRenderer` to obiekt, który może utworzyć obraz z grafu sceny.

`Scene` jest najprostszym z trzech obiektów. Scenę można utworzyć jako obiekt typu THREE. Scena za pomocą konstruktora bez parametrów:

```
scene = new THREE.Scene();
```

Funkcja `scene.add (element)` może być używana do dodawania kamer, świateł i obiektów graficznych do sceny. Prawdopodobnie jest to jedyna funkcja sceniczna, którą trzeba wywołać. Funkcja `scene.remove (element)`, która usuwa element ze sceny, jest również czasami przydatna.

Istnieją dwa rodzaje kamer: jedna z projekcją ortograficzną, druga z projekcją perspektywiczną. Są reprezentowane przez klasy `THREE.OrthographicCamera` i `THREE.PerspectiveCamera`, które są podklasami `THREE.Camera`. Konstruktorzy określają projekcję, używając parametrów znanych z OpenGL:

```
camera = new THREE.OrthographicCamera (left, right, top, bottom, near, far);
```

lub

```
camera = new THREE.PerspectiveCamera (fieldOfViewAngle, aspect, near, far);
```

Parametry kamery ortograficznej określają granice x, y i z bryły widoku, we współrzędnych oka - to znaczy w układzie współrzędnych, w którym kamera znajduje się na $(0, 0, 0)$ patrząc w kierunku ujemnym osi Z, z osią Y skierowaną w górę widoku. Parametry `near` i `far` dają limity z pod względem odległości od

kamery. W przypadku rzutu ortograficznego `near` może być ujemny, umieszczając „bliską" płaszczyznę obcinania z tyłu kamery. Parametry są takie same jak dla funkcji OpenGL `glOrtho()`, z wyjątkiem odwrócenia kolejności dwóch parametrów określających górną i dolną płaszczyznę obcinania.

Kamery perspektywiczne są bardziej powszechne. Parametry kamery perspektywicznej pochodzą z funkcji `gluPerspective ()` w bibliotece OpenGL GLU. Pierwszy parametr określa pionowy zasięg objętości widoku, podany jako kąt mierzony w stopniach. Aspektem jest stosunek między poziomym i pionowym zasięgiem; powinien być zwykle ustawiony na szerokość płótna podzieloną przez jego wysokość. I `near` i `far` dają limity z na bryłę widoku jako odległości od kamery. W przypadku projekcji perspektywicznej oba muszą być dodatnie, z mniejszym `near` niż `far`. Typowy kod do tworzenia kamery perspektywicznej to:

```
camera = new THREE.PerspectiveCamera (45, canvas.width / canvas.height, 1, 100);
```

gdzie `canvas` przechowuje odwołanie do elementu `<canvas>`, w którym obraz będzie renderowany. Wartości `near` i `far` oznaczają, że na obrazie znajdują się tylko rzeczy od 1 do 100 jednostek przed kamerą. Pamiętaj, że użycie niepotrzebnie dużej wartości dla `far` lub niepotrzebnie małej wartości dla `near` może zakłócać dokładność testu głębokości.

Kamera, podobnie jak inne obiekty, może być dodana do sceny, ale nie musi być częścią grafu sceny, który ma być użyty. Możesz dodać ją do grafu sceny, jeśli chcesz, aby była ona rodzicem lub dzieckiem innego obiektu na grafu. W każdym razie na ogół chcesz zastosować transformację modelowania do kamery, aby ustawić jej pozycję i orientację w przestrzeni 3D. Omówię to później, kiedy bardziej ogólnie będę mówił o transformacjach.

(Instalacja three.js zawiera kilka przykładów alternatywnych klas renderujących, które mogą renderować do różnych celów (targets). Na przykład istnieje `CanvasRenderer`, który transluje grafikę 3D na interfejs API Canvas 2D, który został opisany wcześniej. Inne renderery mogą renderować grafikę 3D używając SVG, a nawet CSS, jednak te alternatywne renderery nie obsługują wielu funkcji renderera **WebGL**.Tu użyjemy tylko renderera WebGL.)

Mechanizm renderujący, który renderuje za pomocą WebGL, jest instancją klasy `THREE.WebGLRenderer`. Jego konstruktor ma jeden parametr, który jest obiektem JavaScript zawierającym ustawienia, które wpływają na mechanizm renderujący. Ustawienia, które najprawdopodobniej określisz, to `canvas`, który mówi rendererowi, gdzie ma rysować, i `antialias`, który prosi renderera, by użył antyaliasingu, jeśli to możliwe:

```
renderer = new THREE.WebGLRenderer( {
                        canvas: theCanvas,
                        antialias: true
                    } );
```

Tutaj `theCanvas` byłby odwołaniem do elementu `<canvas>`, w którym mechanizm renderujący wyświetla obrazy, które tworzy. (Zauważ, że technika posiadania obiektu JavaScript jako parametru jest używana w wielu funkcjach three.js.

Umożliwia obsługę dużej liczby opcji bez konieczności długiej listy parametrów, które muszą być określone w określonej kolejności. Zamiast tego wystarczy określić opcje, dla których chcesz podać wartości inne niż domyślne, i możesz określić te opcje według nazwy, w dowolnej kolejności.)

Najważniejszą rzeczą, którą chcesz zrobić z rendererem, jest renderowanie obrazu. Do tego potrzebna jest także scena i kamera. Aby wyświetlić obraz danej `sceny` z punktu widzenia danej `kamery`, użyj

```
renderer.render(scena, kamera);
```

To jest naprawdę centralne polecenie w aplikacji three.js.

(Powinienem zauważyć, że większość przykładów, które widziałem, nie dostarcza obrazu do renderera; zamiast tego umożliwiają renderowanie do jego utworzenia. Canwas można następnie uzyskać z renderera i dodać do strony. Ponadto, canvas zazwyczaj wypełnia całe okno przeglądarki.

3 THREE.Object3D

Graf sceny three.js składa się z obiektów typu `THREE.Object3D` (w tym obiektów należących do podklas tej klasy). Kamery, światła i widoczne obiekty są reprezentowane przez podklasy `Object3D`. W rzeczywistości sama `THREE.Scene` jest również podklasą Object3D.

Każdy `Obiekt3D` zawiera listę obiektów podrzędnych, które są również typu `Object3D`. Listy podrzędne definiują strukturę grafu sceny. Jeśli węzeł i obiekt są typu `Object3D`, to metoda `node.add(obiekt)` dodaje obiekt do listy elementów podrzędnych węzła. Metoda `node.remove(object)` może być użyta do usunięcia obiektu z listy.

Graf sceny three.js musi w rzeczywistości być drzewem. Oznacza to, że każdy węzeł na grafu ma unikalny węzeł macierzysty, z wyjątkiem węzła głównego, który nie ma rodzica. `Obiekt3D`, `obj`, ma właściwość `obj.parent`, która wskazuje na rodzica obiektu na grafu sceny, jeśli taki istnieje. Nigdy nie należy ustawiać tej właściwości bezpośrednio. Jest ustawiany automatycznie, gdy węzeł zostanie dodany do listy podrzędnej innego węzła. Jeśli `obj` ma już rodzica, gdy jest dodawany jako potomek węzła, to `obj` jest najpierw usuwany z listy podrzędnej jego bieżącego rodzica, zanim zostanie dodany do listy podrzędnej węzła.

Elementy potomne obiektu `Object3D, obj`, są przechowywane w właściwości o nazwie `obj.children`, która jest zwykłą tablicą JavaScript. Jednak należy zawsze dodawać i usuwać elementy potomne `obj` za pomocą metod `obj.add ()` i `obj.remove ()`.

Aby ułatwić duplikowanie części struktury grafu sceny, `Object3D` definiuje metodę `clone()`. Ta metoda kopiuje węzeł, w tym rekurencyjne kopiowanie potomków tego węzła. Ułatwia to dołączenie wielu kopii tej samej struktury na grafu sceny:

```
var node = THREE.Object3D();
      .
```

```
      .  // Add children to node.
      .
scene.add(node);
var nodeCopy1 = node.clone();
      .
      .  // Modify nodeCopy1, maybe apply a transformation.
      .
scene.add(nodeCopy1)
var nodeCopy2 = node.clone();
      .
      .  // Modify nodeCopy2, maybe apply a transformation.
      .
scene.add(nodeCopy2);
```

`Objekt3D, obj`, ma powiązaną transformację, która jest podana przez właściwości `obj.scale, obj.rotation` i `obj.position`. Właściwości te reprezentują transformację modelowania, która ma być zastosowana do obiektu i jego dzieci podczas renderowania obiektu. Obiekt jest najpierw skalowany, a następnie obracany, a następnie przeniesiony zgodnie z wartościami tych właściwości. (Transformacje są w rzeczywistości bardziej skomplikowane niż to, ale na razie utrzymamy prostotę i powrócimy do tego tematu później).

Wartości `obj.scale` i `obj.position` są obiektami typu `THREE.Vector3`. `Vector3` reprezentuje wektor lub punkt w trzech wymiarach. (Istnieją podobne klasy `THREE.Vector2` i `THREE.Vector4` dla wektorów w wymiarach 2 i 4.) Obiekt `Vector3` może być skonstruowany z trzech liczb, które dają współrzędne wektora:

```
var v = new THREE.Vector3( 17, -3.14159, 42 );
```

Ten obiekt ma właściwości `v.x, v.y` i `v.z` reprezentujące współrzędne. Właściwości można ustawić indywidualnie; na przykład: `v.x = 10`. Można je także ustawić jednocześnie, używając metody `v.set (x, y, z)`. Klasa `Vector3` ma również wiele metod implementujących operacje wektorowe, takie jak dodawanie, produkt kropkowany i produkt krzyżowy.

W przypadku `Object3D` właściwości `obj.scale.x`, `obj.scale.y` i `obj.scale.z` podają wielkość skalowania obiektu w kierunkach `x, y i z`. Domyślne wartości to oczywiście 1. Wywołanie

```
obj.scale.set (2,2,2);
```

oznacza, że obiekt zostanie poddany jednorodnemu współczynnikowi skalowania 2, gdy jest renderowany. Polecenie

```
obj.scale.y = 0.5;
```

zmniejszy go do połowy rozmiaru tylko w kierunku y (zakładając, że `obj.scale.x` i `obj.scale.z` nadal mają domyślne wartości).

Podobnie właściwości `obj.position.x`, `obj.position.y` i `obj.position.z` podają wartości przeniesień, które zostaną zastosowane do obiektu w kierunkach x, y i z, gdy jest on renderowany. Na przykład, ponieważ kamera jest `Obiektem3D`, ustawienie

```
camera.position.z = 20;
```

oznacza, że kamera zostanie przeniesiona z domyślnej pozycji w punkcie początkowym do punktu (0,0,20) na dodatniej osi Z. Ta transformacja modelowania w kamerze staje się transformacją oglądania, gdy kamera jest używana do renderowania sceny.

Obiekt `obj.rotation` ma właściwości `obj.rotation.x`, `obj.rotation.y` i `obj.rotation.z`, które reprezentują obroty wokół osi x, y i z. Kąty są mierzone w radianach. Obiekt jest obracany najpierw wokół osi X, następnie wokół osi Y, a następnie wokół osi Z. (Można zmienić tę kolejność.) Wartość `obj.rotation` nie jest wektorem. Zamiast tego należy do podobnego typu, `THREE.Euler`, a kąty obrotu są nazywane kątami Eulera.

4 Obiekt, geometria, materiał

Widoczny obiekt w three.js składa się z punktów, linii lub trójkątów. Pojedynczy obiekt odpowiada prymitywu OpenGL, takim jak `GL_POINTS`, `GL_LINES` lub `GL_TRIANGLES`. Istnieje pięć klas reprezentujących te możliwości: `THREE.Points` dla punktów, `THREE.Mesh` dla trójkątów i trzy klasy dla linii: `THREE.Line`, która używa prymitywu `GL_LINE_STRIP`; `THREE.LineSegments`, które używają prymitywu `GL_LINES`; i `THREE.LineLoop`, który używa prymitywu `GL_LINE_LOOP`.

Widoczny obiekt składa się z pewnej geometrii i materiału, który określa wygląd tej geometrii. W three.js geometria i materiał widocznego obiektu są same w sobie reprezentowane przez klasy JavaScript `THREE.Geometry` i `THREE.Material`.

Obiekt typu `THREE.Geometry` ma właściwość o nazwie `vertices`, która jest tablicą `Vector3`. Podczas ręcznego tworzenia geometrii możemy po prostu wcisnąć wektory do tej tablicy. Załóżmy na przykład, że chcemy reprezentować chmurę 1000 losowych punktów wewnątrz kuli o promieniu jeden w środku na początku układu współrzędnych:

```
var points = new THREE.Geometry();
while ( points.vertices.length < 1000 ) {
    var x = 2*Math.random() - 1;  // (between -1 and 1)
    var y = 2*Math.random() - 1;
    var z = 2*Math.random() - 1;
    if ( x*x + y*y + z*z < 1 ) { // use vector only if length is less than 1
        var pt = new THREE.Vector( x, y, z );
        points.vertices.push(pt);
    }
}
```

Aby ta chmura punktów stała się widocznym obiektem, potrzebujemy również materiału. Dla obiektu typu `THREE.Points` możemy użyć materiału typu `THREE.PointsMaterial`, który jest podklasą materiału. Materiał może określać kolor i rozmiar punktów, wśród innych właściwości:

```
var pointMaterial = new THREE.PointsMaterial( {
              color: "yellow",
              size: 2,
              sizeAttenuation: false;
          } );
```

Możliwe jest również przypisanie wartości do właściwości materiału po utworzeniu obiektu. Na przykład,

```
var pointMaterial = new THREE.PointsMaterial();
pointMaterial.color = new THREE.Color("yellow");
pointMaterial.size = 2;
pointMaterial.sizeAttenuation = false;
```

Po uzyskaniu geometrii i materiału możemy użyć ich do utworzenia widocznego obiektu typu `THREE.Points` i dodać go do sceny:

```
var sphereOfPoints = new THREE.Points( points, pointMaterial );
scene.add( sphereOfPoints );
```

Istnieje kilka sposobów konstruowania obiektu `THREE.Color`.

```
var c1 = new THREE.Color("skyblue");
var c2 = new THREE.Color(1,1,0);  // yellow
var c3 = new THREE.Color(0x98fb98);  // pale green
```

5 Światła

W porównaniu do geometrii i materiałów, światła są łatwe! Three.js ma kilka klas do reprezentowania świateł. Klasy to podklasy `THREE.Object3D`. Do sceny można dodać obiekt, a następnie oświetlić obiekty w scenie. Przyjrzymy się światłom kierunkowym, światłom punktowym, światłom otoczenia i reflektorom.

Światło kierunkowe

```
var light = new THREE.DirectionalLight(); // default white light
light.position.set( 0, 0, 1 );
scene.add(light);
```

Światło punktowe

```
new THREE.PointLight( color, intensity, cutoff )
```

Światło otoczenia

```
new THREE.AmbientLight( color )
```

Reflektor (spotlight)

```
new THREE.SpotLight( color, intensity, cutoff, coneAngle, exponent )
```

Literatura

Uwaga!

Tworzenie aplikacji z wykorzystaniem three.js

`https://docplayer.pl/20001297-Pisanie-aplikacji-z-wykorzystaniem-three-js.html`

W języku angielskim

- książka interakcyjna: Three.js: Basics `http://math.hws.edu/graphicsbook/c5/s1.html` (rozdział 5.1)

6 Zadanie

Celem jest konstruowanie złożonego modelu za pomocą three.js - animowanej karuzeli (podstawa karuzeli jest wielokątem odpowiednio z konfiguracją zadania) i co najmniej jednego innego wybranego modelu (patrz Fig. 1). Pliki do pobrania znajdują się poniżej. Głównym plikiem jest `lab9.html`. Podfolder zasobów `resources` zawiera dwa pliki JavaScript używane przez program oraz model konia, którego używamy w karuzeli. Zawiera również kilka plików graficznych, które można wykorzystać jako tekstury.

Przykłady tworzenia modelu złożonego Three.js:
`http://math.hws.edu/graphicsbook/source/threejs/diskworld-1.html`
oraz
`http://math.hws.edu/eck/cs424/f17/lab9/windmill/windmill.html`

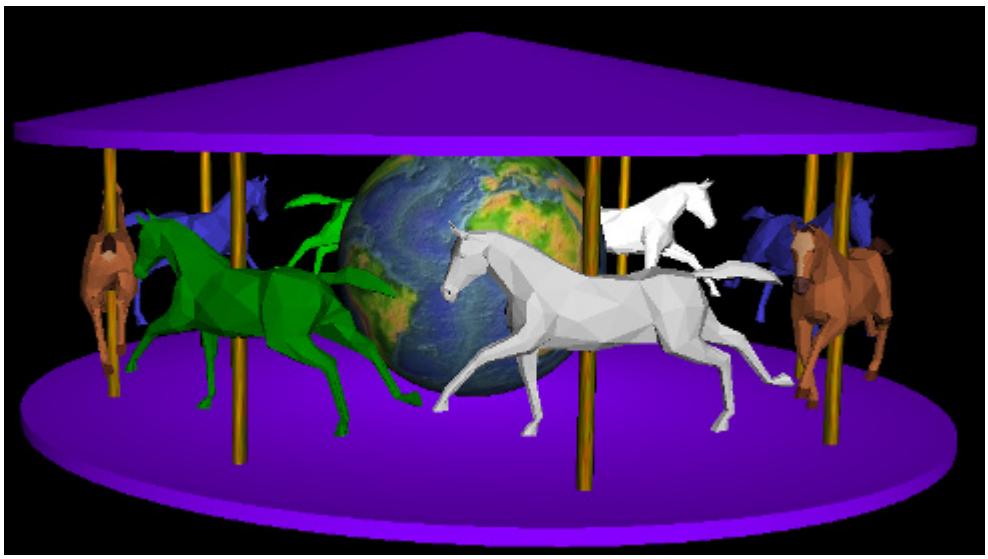

Figure 1: Model karuzeli