Table of Contents

Threejs教學分享

此文件主要內容參考官方及網路教程，其餘記錄些使用過的例子，參考來源的資料有些相當完整，建議可以參考看看。

在各個類別的條目下會介紹用過的及常用的屬性和方法，但非完整，詳細請參考官方Documentation。

建議閱讀

https://developer.mozilla.org/zh-TW/docs/Web/Guide/HTML/Canvas_tutorial
http://workshop.chromeexperiments.com/
dat.GUI：http://workshop.chromeexperiments.com/examples/gui.

參考來源

http://threejs.org/docs/index.html#Reference/Textures/Texture

https://aerotwist.com/tutorials/

http://codex.wiki/post/185830-332

http://www.hewebgl.com/

Getting started

在此章節稍微介紹Three.js的概念，以及基本的實作方式。

在開始之前，我們需要有放置程式碼的地方。將下列的html檔案儲存在你的電腦之中、並複製一份three.min.js至js/的目錄之下，以瀏覽器開啓樣就行了。接下來只要把程式碼放在<script>標籤之中即可。

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8>
        <title>My first Three.js app</title>
        <style>
            body { margin: 0; }
            canvas { width: 100%; height: 100% }
        </style>
    </head>
    <body>a
        <script src="js/three.min.js"></script>
        <script>
            // Our Javascript will go here.
        </script>
    </body>
</html>
```

Creating the scene

以three.js來顯示任何東西，有三個必要的東西：

1. scene
2. camera
3. renderer

在使用上我們一開始在Script中打入的就是以下這幾行：

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight,
0.1, 1000 );
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

如果有碰過unity的人大概會有點概念，上面個別爲scene場景，camera攝影機，renderer渲染器，總之要加上這些你才能在你的輸出看到物件。

在建立整個創作的步驟通常是建立好以上三個物件，再將Object3D物件加入場景之中，並用不同的函式控制以呈現我們的創作。在以下的子章節中將以一些例子介紹這些概念。

參考資料

https://aerotwist.com/tutorials/getting-started-with-three-js/
http://threejs.org/docs/index.html#Manual/Introduction/Creating_a_scene

scene場景

*Scene*類別繼承自*Object3D*類別。

Scenes是用來放置你的物件的地方，你可以在Scenes中放置你的物件及光源，並藉由Three.js所提供得renderer(渲染器)呈現你的創作。

建構場景：

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

你可以使用 `scene.add()` 將物件加入到場景之中。

Camera攝影機

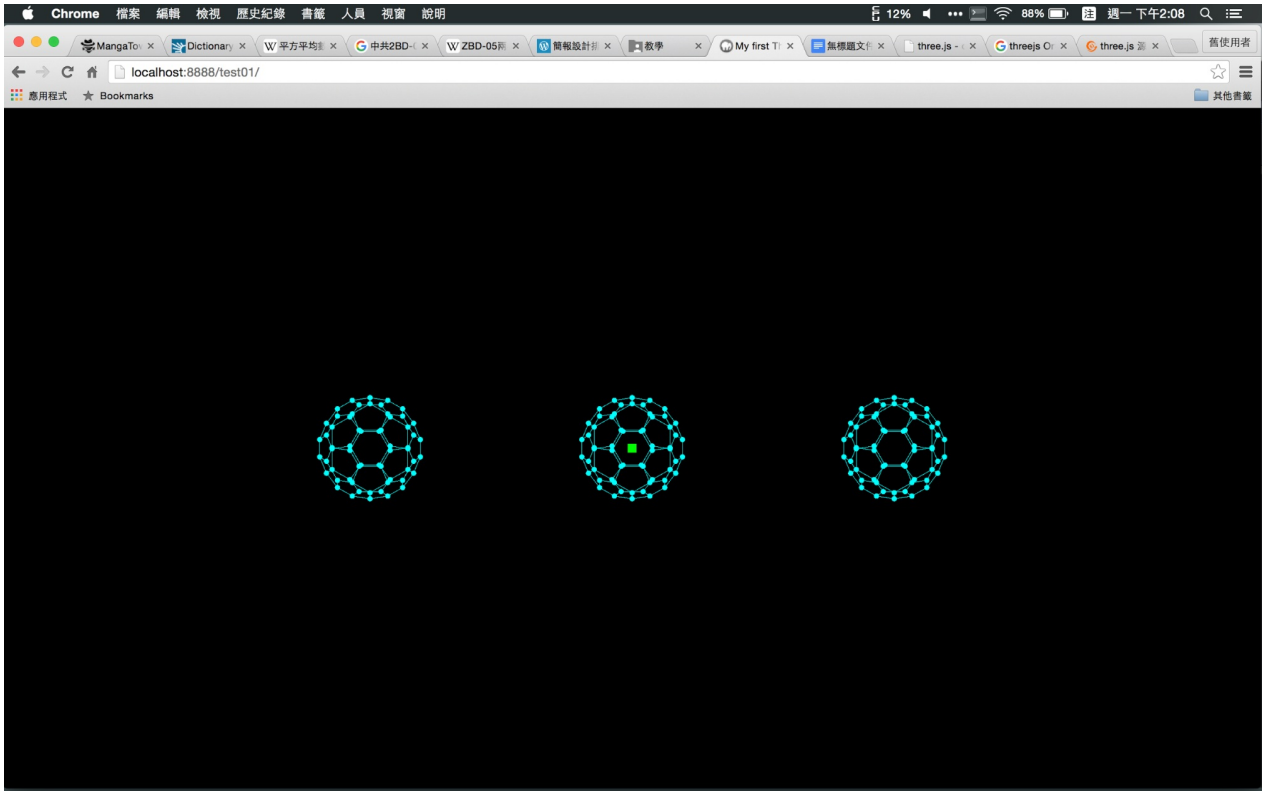
*Camera*類別繼承自*Object3D*類別。

在Camera的選擇會影響你所看到的物件，在這裡介紹兩種Camera：

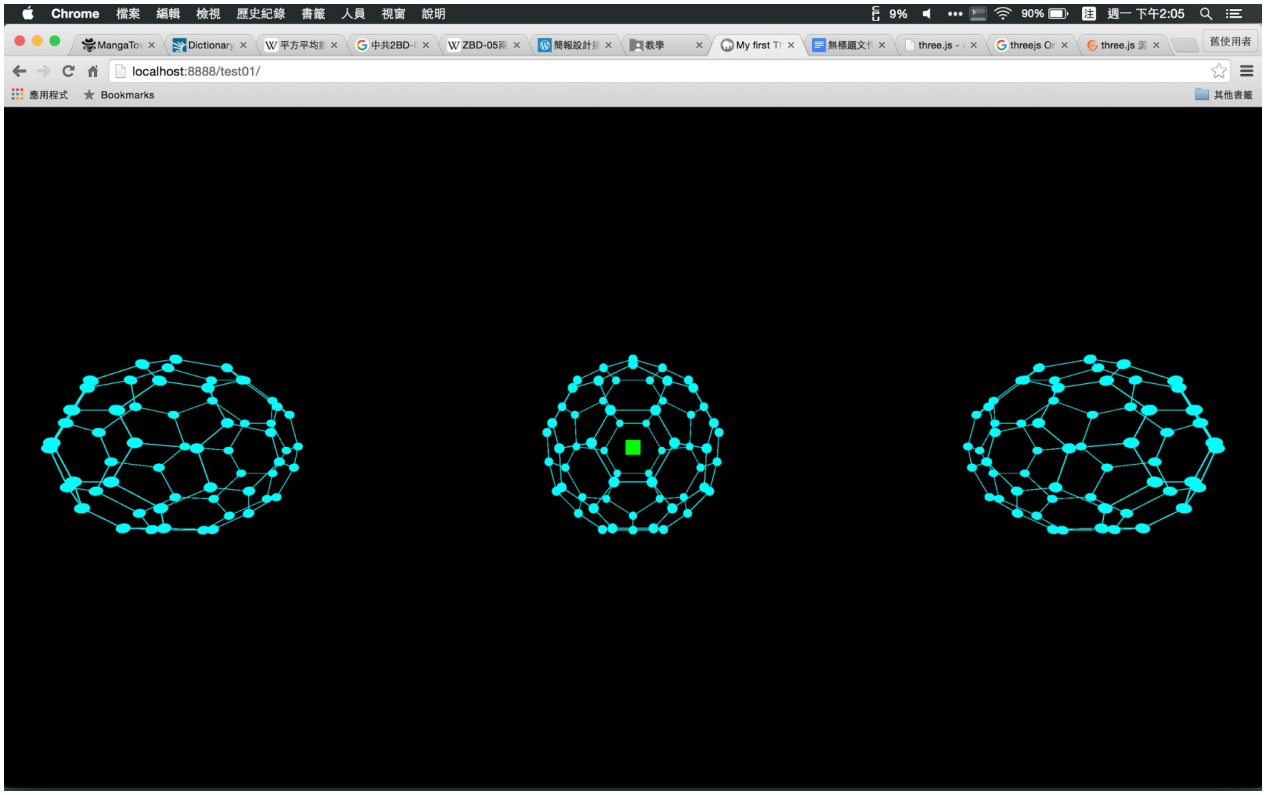
1. OrthographicCamera(left, right, top, bottom, near, far)
   left, right, top,bottom各爲坐標的位置，後面兩個似乎是類似的功能，我不太確定，總之我通常只條far，這個是用來控制深度，也就是說在比這個長度更遠的東西會是看不見的
2. PerspectiveCamera(fov, aspect, near, far)
   fov爲相機的可視角度
   aspect爲相機的可視長寬比
   near, far同上

而這兩種Camera有什麼不同呢？見下圖。

OrthographicCamera：



PerspectiveCamera：



其實還蠻明顯的吧，雖然參數有點差異，不過這樣就能表達兩者不同了，如果沒有要做出有眞實世界的感覺的話，我還是建議使用OrthographicCamera這個。

Renderer渲染器

彩現（英語：Render）在電腦繪圖中，是指：用軟體從模型生成圖像的過程。模型是用語言或者資料結構進行嚴格定義的三維物體或虛擬場景的描述，它包括幾何、視點、紋理、照明和陰影等信息。圖像是數字圖像或者位圖圖像。彩現用於描述：計算視頻編輯軟體中的效果，以生成最終視頻的輸出過程。

爲了呈現我們想要的效果，我們需要藉由渲染器將3D物件映射至2D平面，而在Three.js中有兩種渲染器可以選擇：

1. CanvasRenderer
2. WebGLRenderer

CanvasRenderer

Canvas renderer使用Canvas 2D Context API來顯示你的場景。 可以在一般場景時作爲WebGLRenderer後備的渲染器。

```
function webglAvailable() {
    try {
        var canvas = document.createElement( 'canvas' );
        return !!( window.WebGLRenderingContext && (
            canvas.getContext( 'webgl' ) ||
            canvas.getContext( 'experimental-webgl' ) )
        );
    } catch ( e ) {
        return false;
    }
}

if ( webglAvailable() ) {
    renderer = new THREE.WebGLRenderer();
} else {
    renderer = new THREE.CanvasRenderer();
}
```

Note: WebGLRenderer與CanvasRenderer皆是以<canvas>標籤嵌在HTML5網頁上的。在CanvasRenderer中的Canvas僅表示他是使用Canvas 2D而非WebGL。

WebGLRenderer

WebGLRenderer是以WebGL顯示你的場景，當然，前題是你的裝置支援WebGL。而這個渲染器的效能將優於CanvasRenderer。

渲染器的加入、使用方式

```
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );//設定大小
renderer.setClearColor( 0xff0000 );//設定畫布背景顏色
document.body.appendChild( renderer.domElement );//將畫布加入body
renderer.render( scene, camera );//將攝影機畫面渲染
```

建立好renderer物件後我們以 `setSize()` 來改變畫布的大小， 而上式中 `domElement` 代表的是你所渲染輸出的畫布，在物件建構時就會自動產生，在這裡我們把它加到body上， 最後以 `render()` 函式就可以藉由攝影機將場景展現出來。

參考資料

https://zh.wikipedia.org/wiki/%E5%BD%A9%E7%8F%BE
http://threejs.org/docs/index.html#Reference/Renderers/CanvasRenderer
http://threejs.org/docs/index.html#Reference/Renderers/WebGLRenderer

Object3D物件

`Object3D`是Three.js下的一個核心類別，許多Three.js提供的3D物件皆是繼承自`Object3D`，像是前述的scene、Camera也繼承自Object3D。

爲了在視窗中看見3D物件，我們一般利用Geometry(幾何體)和Material(材質)建構Mesh(網面)，再將之加入場景中。而在更多進階的應用中，我們不一定在Three.js中建立我們的幾何體，而是利用blender、3ds max、maya等建模軟體建構我們的物件。

在理解上可以想成:
Geometry+Meterial=Mesh -> Object3D

在Object3D中有幾個重要的屬性：position、rotation、scale，這些個別爲位置、旋轉、比例，在Three.js中幾乎所有的物件都需要操作到這些屬性及與之相關的方法。

> Object3D： http://threejs.org/docs/index.html#Reference/Core/Object3D

Geometry幾何體

描述幾何體的核心類別，包含所有描述3D模型必要的資料。

注意：爲了在視窗中能看見物件，要到Mesh的部分才有辦法提到，Geometry僅是描述3D模型的資料。

三角形平面：

```
var geometry = new THREE.Geometry();
geometry.vertices.push( new THREE.Vector3( 0, 0, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 0, 200, 0 ) );   // vertex 1
geometry.vertices.push( new THREE.Vector3( 300, 200, 0 ) );  // vertex 2
geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );  // make a triangle
```

上式中我們以 `Vector3` 建立三角形各端點並push進geometry中，以 `THREE.Face3()` 建立三維平面，這個函式的參數爲geometry中vertex的索引值，按照push的順序由0開始。

> Face3：
> http://threejs.org/docs/index.html#Reference/Core/Face3

Material材質

Material用來描述物件的外觀，在大部分情況下Material的定義上與渲染器無關(renderer-independent)，所以你並不需要爲了不同渲染器重新設計一個Material。

Three.js提供了不同種類的材質類別： LineBasicMaterial、 MeshBasicMaterial、 MeshLambertMaterial、 ShaderMaterial、 SpriteMaterial.....等，不同種類的類別。

以MeshBasicMaterial爲例：

```
var material = new THREE.MeshBasicMaterial( { color: 0xff0000, side: THREE.DoubleSide
} );
```

上式建立了一個顏色爲：0xff0000的雙面材質。
side是繼承自Material的一個屬性，在選擇上我們有THREE.FrontSide(預設)、THREE.BackSide、THREE.DoubleSide這三種。

> Material: http://threejs.org/docs/index.html#Reference/Materials/Material

Mesh網面

*Mesh*類別繼承自*Object3D*類別。

我們知道兩點能夠定義直線，而三個點能夠定義一個三角形平面，多個平面則可定義出不同種類的多邊形，以多個平面建構物件的方式就稱爲多邊形建模，建立出的物件我們稱之爲Mesh(網面)或是網格。

以下以三角形平面爲例：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8>
        <title>My first Three.js app</title>
        <style>
            body { margin: 0; }
            canvas { width: 100%; height: 100% }
        </style>
    </head>
    <body>
        <script src="build/three.min.js"></script>
        <script>
            var scene = new THREE.Scene();
            var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.i
nnerHeight, 0.1, 1000 );
            camera.position.z = 500;
            var renderer = new THREE.WebGLRenderer();
            renderer.setSize( window.innerWidth, window.innerHeight );
            document.body.appendChild( renderer.domElement );

            var material = new THREE.MeshBasicMaterial( { color: 0xff0000, side: THREE
.DoubleSide } );
            var geometry = new THREE.Geometry();
            geometry.vertices.push( new THREE.Vector3( 0, 0, 0 ) );  // vertex 0
            geometry.vertices.push( new THREE.Vector3( 0, 200, 0 ) );   // vertex 1
            geometry.vertices.push( new THREE.Vector3( 300, 200, 0 ) );  // vertex 2
            geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );  // make a triangle
            var mesh = new THREE.Mesh( geometry, material );  // create an object
            scene.add( mesh ); // add object to the scene to make it visible

            renderer.render( scene, camera );
        </script>
    </body>
</html>
```

其中：

```
var material = new THREE.MeshBasicMaterial( { color: 0xff0000, side: THREE.DoubleSide
} );
var geometry = new THREE.Geometry();
geometry.vertices.push( new THREE.Vector3( 0, 0, 0 ) );  // vertex 0
geometry.vertices.push( new THREE.Vector3( 0, 200, 0 ) );   // vertex 1
geometry.vertices.push( new THREE.Vector3( 300, 200, 0 ) );  // vertex 2
geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );  // make a triangle
var mesh = new THREE.Mesh( geometry, material );  // create an object
scene.add( mesh ); // add object to the scene to make it visible
```

在上式中我們個別建立好材質和幾合體，以Mesh的建構式就能輕鬆建立好物件，而最後只要利用 `scene.add()` 就能將建立好的物件加入到場景之中。

Light光源

Light類別繼承自Object3D類別。

這次我們以球體和點光源作爲例子：

會用到的兩個類別：
PointLight(hex, intensity, distance, decay)
SphereGeometry(radius, widthSegments, heightSegments, phiStart, phiLength, thetaStart, thetaLength)

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8>
        <title>My first Three.js app</title>
        <style>
            body { margin: 0; }
            canvas { width: 100%; height: 100% }
        </style>
    </head>
    <body>
        <script src="build/three.min.js"></script>
        <script>
            var scene = new THREE.Scene();
            var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.i
nnerHeight, 0.1, 1000 );
            camera.position.z = 500;
            var renderer = new THREE.WebGLRenderer();
            renderer.setSize( window.innerWidth, window.innerHeight );
            document.body.appendChild( renderer.domElement );

            var ball =new THREE.Mesh(new THREE.SphereGeometry( 70, 16, 8 ), new THREE.
MeshPhongMaterial( { color: 0x555555, specular: 0x111111, shininess: 50 }  ));
            ball.position.set(0,0,0);
            scene.add(ball);

            var light1 = new THREE.PointLight( 0xffaa00, 1, 500 );
            light1.add( new THREE.Mesh( new THREE.SphereGeometry( 5, 16, 8 ), new THRE
E.MeshBasicMaterial( { color: 0xffaa00 } ) ) );
            light1.position.set(70,70,100);
            var light2 = new THREE.PointLight( 0x80ff80, 1, 500 );
            light2.add( new THREE.Mesh( new THREE.SphereGeometry( 5, 16, 8 ), new THRE
E.MeshBasicMaterial( { color: 0x80ff80 } ) ) );
            light2.position.set(-70,-70,100);
            scene.add( light1 );
            scene.add( light2 );

            renderer.render( scene, camera );
        </script>
    </body>
</html>
```

建立於點中央，半徑爲70的球體：

```
var ball =new THREE.Mesh(new THREE.SphereGeometry( 70, 16, 8 ), new THREE.MeshPhongMat
erial( { color: 0x555555, specular: 0x111111, shininess: 50 }  ));
ball.position.set(0,0,0);
scene.add(ball);
```

建立點光源並用一顆球體標記：

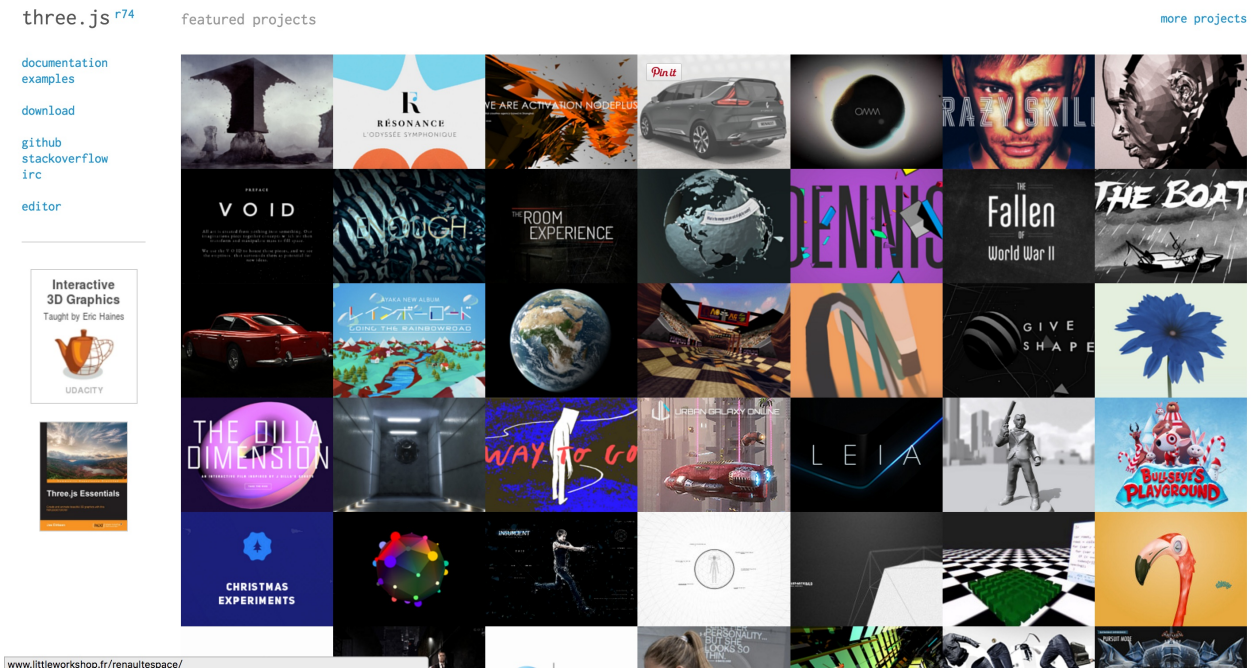
```
var light1 = new THREE.PointLight( 0xffaa00, 1, 500 );
light1.add( new THREE.Mesh( new THREE.SphereGeometry( 5, 16, 8 ), new THREE.MeshBasicM
aterial( { color: 0xffaa00 } ) ) );
light1.position.set(70,70,100);
var light2 = new THREE.PointLight( 0x80ff80, 1, 500 );
light2.add( new THREE.Mesh( new THREE.SphereGeometry( 5, 16, 8 ), new THREE.MeshBasicM
aterial( { color: 0x80ff80 } ) ) );
light2.position.set(-70,-70,100);
scene.add( light1 );
scene.add( light2 );
```

Three.js基礎入門

在本章節將討論一些實作時所需要的基本觀念，以及介紹一些常會用到的類別，並略提到相關的屬性質及方法，而完整的類別內容則建議參考官方Documentation。

實用的工具 - Editor

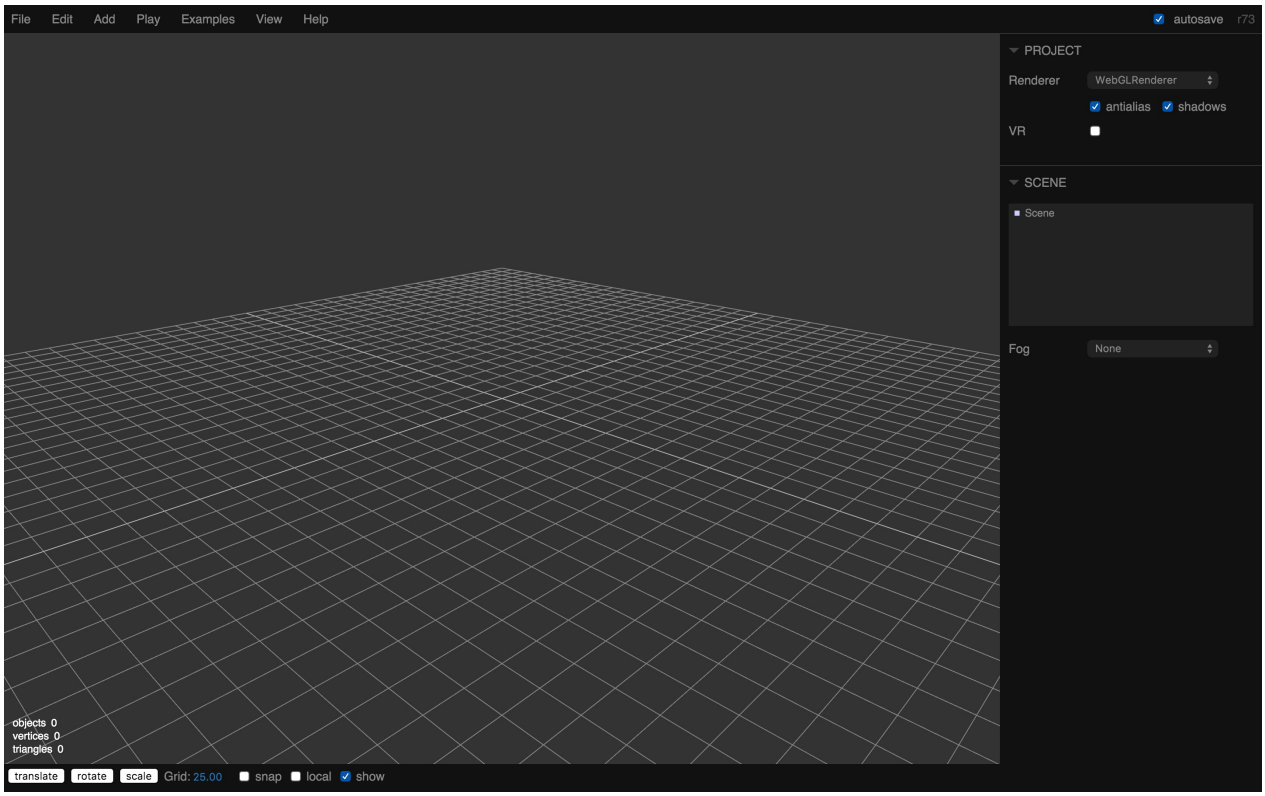
Three.js官方不只釋出了Three.js的相關套件，亦提供了一個方便的編輯工具，讓使用者可以在圖型的界面上完成自己的創作。



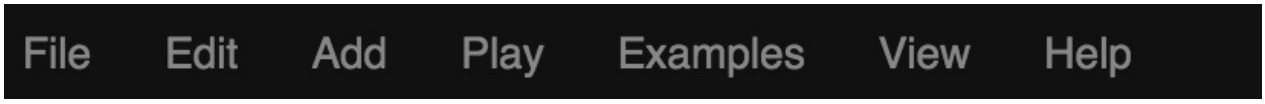
左方的editor連結，可以讓使用者進行線上編輯，或者你可以到官方的GitHub上下載整個套件在localhost上進行編輯。

Note:目前我看到釋出的版本是r74，我個人建議下載前一版r73，因爲我在使用上的時候，自動完成有點問題，但也有可能是我自己這邊的問題。

讓我們先來看看一開始進來Editor的初始畫面：



選單列的功能：



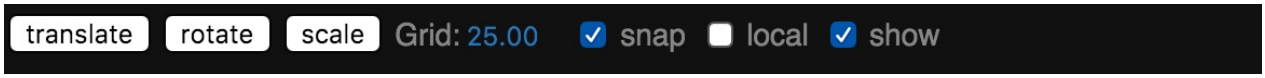
File選單可以新增一個新的場景、輸出選到的物件、及輸出最後的成果。

Edit選單可以複製物件

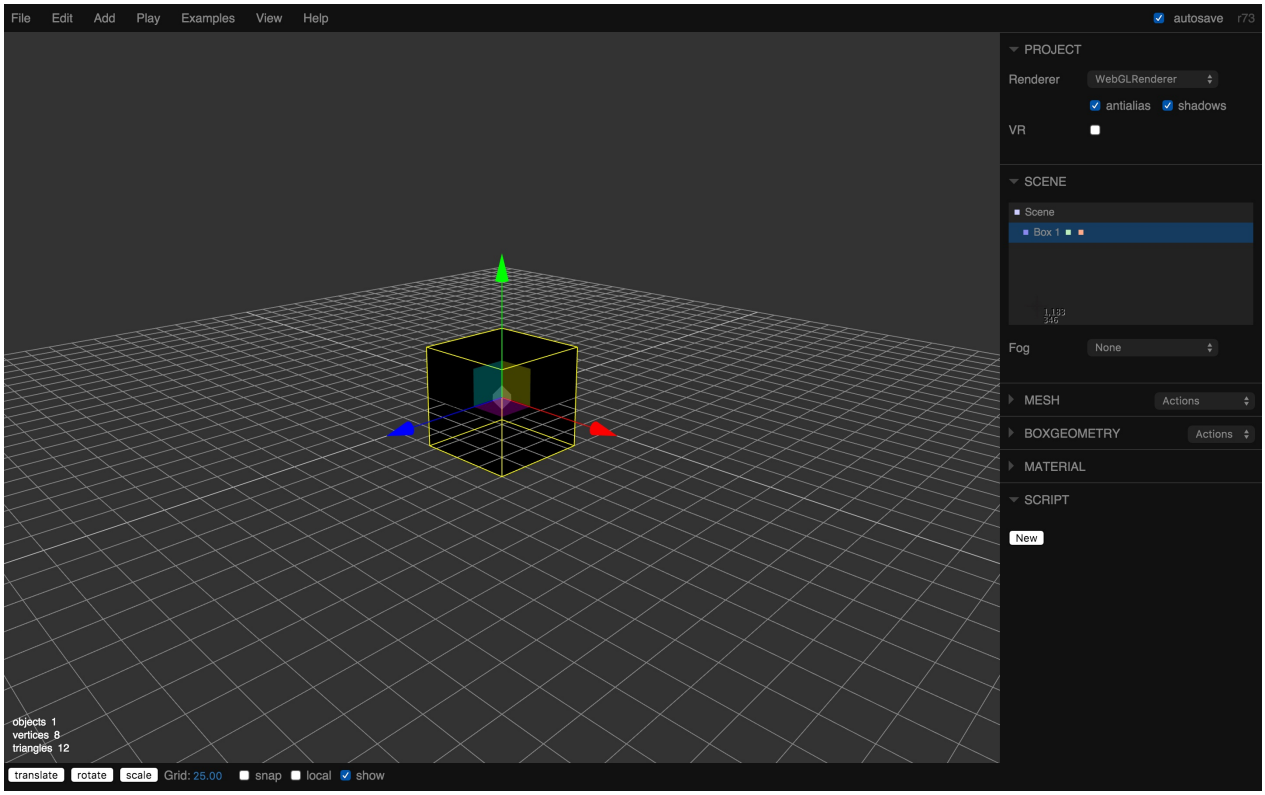
Add選單可以在場景中加入官方所提供的各種不同基本物件，如正方體、球體、圓柱等等。

View選單中可以選擇白色或是黑色的介面、以及全螢幕。

如果你想要看到場景在攝影機下的效果，你可以按下Play，你就能看見運行的狀態了。

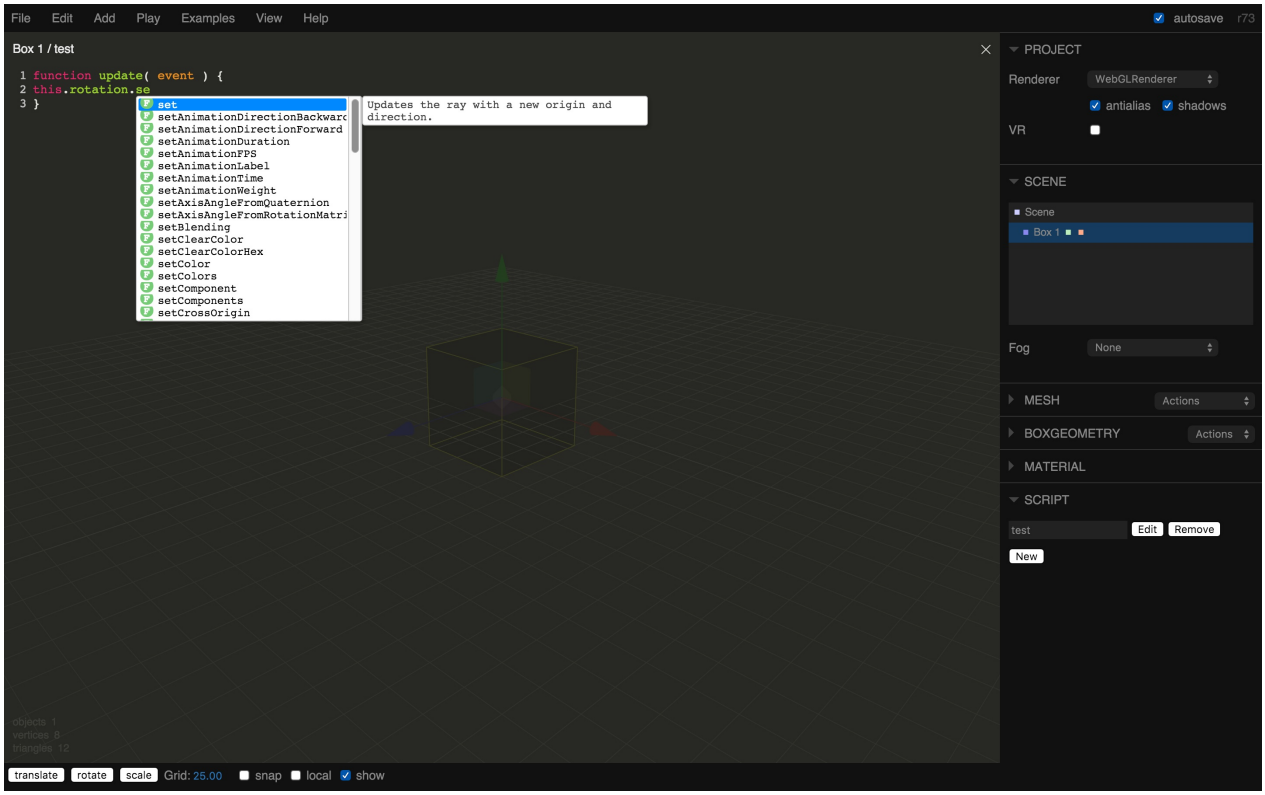


在下方的部分有三個按鈕translate、rotate、scale，分別可以讓你在畫面上進行移動、選轉、縮放的功能。而右方的選項分別爲snap、local、show，點選snap可以在移動時只在格點上進行移動；點選local能夠讓你選擇世界座標或是區域座標；show則是選擇是否顯時格點。



右方的區域中，能夠看見物件的相關屬性，如geometry和material，而在下方的script中你可以新增腳本，方便你控制物件的行爲，只要新增後按下edit即可編輯。

操作畫面：



Editor的使用的確使得創作更加容易，不過在之後的章節中還是可能不藉由Editor進行教學，不過大致的觀念是相通的，而Editor的用法也有一些不同，還需要大家自行進行一些摸索。

dat.GUI

dat.GUI的大致源碼我並沒有認眞看，只知道他似乎是使用了一個controllers.factory來判斷你所加入的Controller是哪一類型，anyway，這裡就整理下使用方式。

基本上使用相當簡單，你只要把物件丟給 `add()` 就能夠判斷並控制你的數值，在這裡以json格式爲例，建構 `dat.GUI()` 後使用 `add()` 他就能判定你所使用的Controller或Folder了。

```
var test = {
    //想要丟給gui的參數與初始值
    text : 'text',
    speed : 0.8,
    boolean : false,
    slider1: 2,
    slider2: 0,
    slider3: 5,
    slider4: 5,
    menu:'menu',
    color0 : "#ffae23", // CSS string
    color1 : [ 0, 128, 255 ], // RGB array
    color2 : [ 0, 128, 255, 0.3 ], // RGB with alpha
    color3 : { h: 350, s: 0.9, v: 0.3 }, // Hue, saturation, value
};
gui = new dat.GUI();
gui.add(test, 'text');//新增text
var slider = gui.addFolder('slider');//新增Folder
    slider.add(test, 'slider1').step(5); // 增加量
    slider.add(test, 'slider2', -5, 5); // 上線和下限
    slider.add(test, 'slider4').min(0).step(0.25); // Mix and match
    slider.add(test, 'slider4').min(0).max(10).step(0.25); // Mix and match
gui.add(test, 'menu', [ 'test1', 'test2', 'test3' ] );//下拉選單
gui.add(test, 'boolean');//布林值
var color = gui.addFolder('color');
    color.addColor(test, 'color0');
    color.addColor(test, 'color1');
    color.addColor(test, 'color2');
    color.addColor(test, 'color3');
color.open();//Folder初始是打開的
slider.close(); //Folder初始是關閉的
```

如何更變數值？

以json爲例，使用 `onChange()` ：

```
var geometry = new THREE.BoxGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial( {color:0xffffff} );
cube = new THREE.Mesh(geometry,material);
cube.rotation.z+=10;
cube.rotation.x+=10;
scene.add(cube);

var test = {
    //想要丟給gui的參數與初始值
    cube : 'cube',
    scale_x: 1,
    scale_y: 1,
    scale_z: 1,
};
gui = new dat.GUI();
gui.add(test, 'cube');//新增text
var scale = gui.addFolder('scale');
var s_x = scale.add(test,'scale_x').min(0).max(5).step(0.1);
var s_y = scale.add(test,'scale_y').min(0).max(5).step(0.1);
var s_z = scale.add(test,'scale_z').min(0).max(5).step(0.1);
s_x.onChange(function(x){ cube.scale.x = x;});
s_y.onChange(function(y){ cube.scale.y = y;});
s_z.onChange(function(z){ cube.scale.z = z;});
```

以物件爲例：

```
var geometry = new THREE.BoxGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial( {color:0xffffff} );
cube = new THREE.Mesh(geometry,material);
cube.rotation.z+=10;
cube.rotation.x+=10;
cube.name = "cube1";
scene.add(cube);

gui = new dat.GUI();
gui.add(cube, 'name');//新增text
var scale = gui.addFolder('scale');
scale.add(cube.scale,'x').min(0).max(5).step(0.1);
scale.add(cube.scale,'y').min(0).max(5).step(0.1);
scale.add(cube.scale,'z').min(0).max(5).step(0.1);
```

參考資料：

https://www.youtube.com/watch?v=8xFYBjXKYgE&feature=iv&src_vid=PcsJAy-VRts&annotation_id=annotation_2837750455
http://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage

Textures(紋理)

用於物體表面的紋理，或用於反射貼圖(reflection map)與折射貼圖(refraction map)。

3D世界中紋理的紋理由圖片所組成，也能說是貼圖，將貼圖以一定的規則映射至幾何體上，賦予幾何體紋理，一般是三角形。

建構函式如下：

```
THREE.Texture( image, mapping, wrapS, wrapT, magFilter, minFilter, format, type, aniso
tropy )
```

各參數的意思是：

1. Image：這是一個圖片類型，可以使用ImageUtils來加載，如下代碼

   ```
   var texture = THREE.ImageUtils.loadTexture(url);
   ```

   ，或是使用canvas：

   ```
   var texture = new THREE.Texture( canvas);
   ```

2. Mapping：是一個 `THREE.UVMapping()` 類型，它表示的是紋理坐標。下一節，我們將說說紋理坐標。

3. wrapS：表示x軸的紋理的包覆方式，亦即當紋理的寬度小於需要貼圖的平面的寬度的時候，平面剩下的部分應該以何種方式進行貼圖。

4. wrapT：表示y軸的紋理包覆方式。

5. magFilter和minFilter表示過濾的方式，這是OpenGL的基本概念。

6. format：表示加載的圖片的格式，這個參數可以取值THREE.RGBAFormat，RGBFormat等。THREE.RGBAFormat表示每個像素點要使用四個分量表示，分別是紅、綠、藍、透明來表示。RGBFormat則不使用透明，也就是說紋理不會有透明的效果。

7. type： The default is `THREE.UnsignedByteType`. Other valid types (as WebGL allows) are THREE.ByteType, `THREE.ShortType`, `THREE.UnsignedShortType`, `THREE.IntType`, `THREE.UnsignedIntType`, `THREE.FloatType`, `THREE.UnsignedShort4444Type`, `THREE.UnsignedShort5551Type`, and `THREE.UnsignedShort565Type`.

8. anisotropy：各向異性過濾。使用各向異性過濾能夠使紋理的效果更好，但是會消耗更多的內存、CPU、GPU時間。

紋理坐標

參考資料：

http://threejs.org/docs/index.html#Reference/Textures/Texture
http://www.hewebgl.com/article/getarticle/68

Geometry

在Getting Start時我們就已經稍微提到Geometry的相關概念了，雖然我們可以用vertices和faces建構出我們的幾何體，但我們有時候不需用這麼做。在Three.js之中就已經有提供一些基本的幾何體，如正方體、圓形、圓柱.....等，大部分內容，官方的文件一看就能夠明瞭。

1. BoxGeometry
2. CircleGeometry
3. CylinderGeometry
4. DodecahedronGeometry
5. **ExtrudeGeometry - Creates extruded geometry from a path shape**
6. IcosahedronGeometry
7. **LatheGeometry - Class for generating meshes with axial symmetry. Possible uses include donuts, pipes, vases etc. The lathe rotate around the Y axis.**
8. OctahedronGeometry
9. **ParametricGeometry - Generate geometry representing a parametric surface.**
10. PlaneGeometry
11. **PolyhedronGeometry - A polyhedron is a solid in three dimensions with flat faces. This class will take an array of vertices, project them onto a sphere, and then divide them up to the desired level of detail. This class is used by DodecahedronGeometry, IcosahedronGeometry, OctahedronGeometry, and TetrahedronGeometry to generate their respective geometries.**
12. RingGeometry
13. **ShapeGeometry - Creates a one-sided polygonal geometry from one or more path shapes. Similar to ExtrudeGeometry**
14. SphereGeometry
15. TetrahedronGeometry
16. TextGeometry
17. TorusGeometry
18. TorusKnotGeometry
19. **TubeGeometry - Creates a tube that extrudes along a 3d curve**

ExtrudeGeometry

LatheGeometry

ParametricGeometry

PolyhedronGeometry

ShapeGeometry

TubeGeometry

Material

Material是描述物體外觀的基礎類別，許多不同的材質都繼承自這個類別，而在Three.js中有幾種Material可以選擇：

1. LineBasicMaterial - 用來建立線架構幾何體。
2. LineDashedMaterial - 用來建立線架構幾何體， with dashed lines.
3. MeshBasicMaterial - 用來建立簡單陰影的物件，如平面、線架構。
4. MeshDepthMaterial - 以深淺作爲繪製的依據，離攝影機愈遠則越白，反之越黑。
5. MultiMaterial - A Material to define multiple materials for the same geometry. The geometry decides which material is used for which faces by the faces materialindex. The materialindex corresponds with the index of the material in the materials array.
6. MeshLambertMaterial - 用來建立非光澤(Lambertian)表面的材質，每一像素計算一次。
7. MeshNormalMaterial - A material that maps the normal vectors to RGB colors.
8. MeshPhongMaterial - 用來建立有光澤的表面，每一像素計算一次。
9. PointsMaterial - 粒子系統的預設材質。
10. RawShaderMaterial - This class works just like ShaderMaterial, except that definitions of built-in uniforms and attributes are not automatically prepended to the GLSL shader code.
11. ShaderMaterial
12. SpriteMaterial - Sprite的材質。

Material Constants

給定參數能夠改變材質的性質、功能：

Side 對應參數 `side`

THREE.FrontSide
THREE.BackSide
THREE.DoubleSide

Shading 對應參數 `Shading`

THREE.FlatShading
THREE.SmoothShading

Colors 對應參數 `vertexColor`

THREE.NoColors
THREE.FaceColors
THREE.VertexColors

Blending Mode 對應參數 `blending`

THREE.NoBlending - 不混合
THREE.NormalBlending - 普通混合
THREE.AdditiveBlending - 相加
THREE.SubtractiveBlending - 相減
THREE.MultiplyBlending - 相乘
THREE.CustomBlending - 自定義

參數

大部分的材質都有共同的參數，不過還是要看情況，有些不同：

繼承自**Material:**

`transparent` - 材質是否透明，預設爲false。
`opacity` - 透明度，由0.0到1.0，預設爲1。
`visible` - 預設爲true。
`needsUpdate` - Specifies that the material needs to be updated at the WebGL level. Set it to true if you made changes that need to be reflected in WebGL. This property is automatically set to true when instancing a new material.

其他：

`color` - 十六進制，幾何體的顏色，預設爲0xffffff。
`map` - 紋理，預設爲null。
`wireframe` - 是否以線架構渲染，布林值。
`wireframeLinewidth` - 線架構厚度，預設爲1。
`specular` - 材質的反射顏色,Setting this the same color as the diffuse value (times some intensity) makes the material more metallic-looking; setting this to some gray makes the material look more plastic. Default is dark gray.
`shininess` - specular反射的程度，預設30。
`emissive` - 材質的放射光顏色，essentially a solid color unaffected by other lighting.預設黑色。
`emissiveIntensity` - 放射光的強度. Modulates the emissive color. 預設爲1。

修編中

Loader

Matrix transformations

在three.js之中我們以矩陣(matrices)達成變換(transformations)：

1. 移動
2. 旋轉
3. 縮放

所有Object3D之instance 都有個矩陣儲存物件的位置(position), 旋轉(rotation), 及比例(scale)。

Convenience properties and matrixAutoUpdate

以下有兩種方式更新物件的transformations：

1. 修改物件的位置(position)，四元數(quaternion)，及比例(scale)屬性，再讓three.js重新計算物件的矩陣。

   ```
   object.position.copy(start_position);
   object.quaternion.copy(quaternion);
   ```

   在預設的情況下 `matrixAutoUpdate` 的屬性是設定爲 `true` 的，這樣矩陣會自動進行重複的計算。若物件是靜態的或你期望手動來控制計算的發生，你可以將此一屬性設爲 `false` ：

   ```
   object.matrixAutoUpdate = false;
   ```

   並在更改任何屬性後手動更新矩陣：

   ```
   object.updateMatrix();
   ```

   2.直接修改物件的矩陣。在Matrix4類別中提供了多種方法來達到此目的：

   ```
   object.matrix.setRotationFromQuaternion(quaternion);
   object.matrix.setPosition(start_position);
   object.matrixAutoUpdate = false;
   ```

   在這個情況之下 `matrixAutoUpdate` 屬性必須設定爲 `false` ，且你必須確定不會呼叫到 `updateMatris()` 。因爲 `updateMatris()` 會依照前述提到過的屬性來計算矩陣。

Object and world matrices

一個物件的矩陣所記錄的變換(transformation)是相對於父物件的，若是要得到此物件對應於世界座標的變換，你必須使用此物件的 `Object3D.matrixWorld` 。 當父物件或子物件之變換改變時，你可以呼叫 `updateMatrixWorld()` 來更新子物件的 `matrixWorld` 屬性。

Rotation and Quaternion

three.js提供了兩種方式來表示3d的旋轉：歐拉角(Euler angles)以及四元數(Quaternions)，並也提供了在兩者間轉換的方法。而歐拉角有個主要的問題稱作"gimbal lock"在某些結構下會缺少了一個自由度。因此，物件的旋轉將儲存在物件的四元數中。可參考gimbal lock

在前述的程式庫中包涵 `useQuaternion` 這一屬性，當設置爲false時，物件的矩陣將以歐拉角來計算。在實踐中這是不建議的，你反而該以 `setRotationFromEuler` 方法來去更新四元數。

參考資料

http://threejs.org/docs/#Manual/Introduction/Creating_a_scene

四元數：
http://blog.roodo.com/sayaku/archives/19544672.html
https://ccjou.wordpress.com/2014/04/21/%E5%9B%9B%E5%85%83%E6%95%B8/
https://ccjou.wordpress.com/2014/04/23/%E5%9B%9B%E5%85%83%E6%95%B8%E8%88%87%E4%B8%89%E7%B6%AD%E7%A9%BA%E9%96%93%E6%97%8B%E8%BD%89/
https://www.ptt.cc/bbs/Flash/M.1282360275.A.6B7.html

旋轉：
https://ccjou.wordpress.com/2014/04/29/%E4%B8%89%E7%B6%AD%E7%A9%BA%E9%96%93%E7%9A%84%E6%97%8B%E8%BD%89%E7%9F%A9%E9%99%A3/
http://s.epb.idv.tw/han-shi-ku/unity/005unity3dxuanzhuanzhongdeshuxuezhishijigegehanshushuomingquaternion

WebGLRenderer

Properties

domElement

`domElement` 是你所渲染輸出的畫布，在物件建構時就會自動產生，你僅需要將它加進頁面即可。

Method

setSize (width, height, updateStyle)

`setSize()` 用來改變畫布的大小，並且重新設定檢視區域以符合大小，而updateStyle在我看到的例子通常都省略掉了，原文是說"Setting updateStyle to true adds explicit pixel units to the output canvas style."

render (scene, camera, renderTarget, forceClear)

`render()` 可以藉由攝影機將場景展現出來。
The render is done to the renderTarget (if specified) or to the canvas as usual.
If forceClear is true, the depth, stencil and color buffers will be cleared before rendering even if the renderer's autoClear property is false.
Even with forceClear set to true you can prevent certain buffers being cleared by setting either the .autoClearColor, .autoClearStencil or .autoClearDepth properties to false.

官方

http://threejs.org/docs/index.html#Reference/Renderers/WebGLRenderer

應用

未分類、相關知識

Shader

WebGL並未提供一個固定渲染管道(Fixed Function Pipeline)。而他所提供的是可編程管道(Programable Function Pipeline)，更爲強大，卻也複雜。可編程管道的意思是程式設計師需要先找到這些點才能夠渲染至螢幕上。Shaders就是這種管道的一部分，在這裡我們有兩種個以選擇：

1. Vertex shaders
2. Fragment shaders

我想你會同意這兩者的名稱並沒什麼特殊的意義，你僅需要知道它們只在你的GPU上運行。高級的GPU會盡可能優化shader所使用的到的函數，讓我們能夠更好的使用。

關於Fixed function pipeline與Programable Function Pipeline參考：

https:\/\/cg2010studio.com\/2011\/06\/29\/shader\/

https://www.ptt.cc/bbs/GameDesign/M.1425531225.A.23D.html

VERTEX SHADERS

舉例來說，一個球體爲不同的vertex所構成，而會有個vertex shader會依序的給予到每個vertex並且能夠更動他們。

無論vertex shader做了什麼，他有個必要的任務：在某個點上設置 gl_Position ，一個四維的浮點數向量，它是vertex最後在螢幕上的位置。這是個很有序的過程，畢竟我們是在討論3維的位置或是2維的螢幕。幸好，在我們使用Three.js之類的套件時，我們有更簡單的方式去設定gl_Position。

FRAGMENT SHADERS

在我們有了這些vertices構成的物件，且將他們投影到我們的2D螢幕上之時，那我們的顏色呢？或是紋理及光影？這些就是fragment shader在做的事情。

有點像vertex shader在做的事情一般，fragment shader也有個必須做到的任務：他需要設定或消除 gl_FragColor 變數，同樣爲一個四維浮點數向量，爲fragment最終的顏色。那麼，什麼是fragment呢？讓我們考慮三個vertices所構成的一個三角形吧，在這三角形上所有的pixel都需要被繪製出。fragment是由那三個vertices所提供之繪製三角形上所有pixel的資料。

Because of this the fragments receive interpolated values from their constituent vertices. If one vertex is coloured red, and its neighbour is blue we would see the colour values interpolate from red, through purple, to blue.

SHADER VARIABLES

在我們討論這些變數前，你三種可以宣告：Uniforms、Attributes、Varyings。 你可以這樣想：

Uniforms are sent to both vertex shaders and fragment shaders and contain values that stay the same across the entire frame being rendered. A good example of this might be a light's position.

Uniforms是發送給vertex shaders及fragment shaders，且在render時他的值在整個frame之中都相同。一例爲light的位置。

Attributes are values that are applied to individual vertices. Attributes are only available to the vertex shader. This could be something like each vertex having a distinct colour. Attributes have a one-to-one relationship with vertices.

Attributes只使用在獨立的vertices上，且只有vertex shader會用到。這可能是在不同點個別有不同顏色時會用到，也就是說Attributes與點有一對一的關係。

Varyings are variables declared in the vertex shader that we want to share with the fragment shader. To do this we make sure we declare a varying variable of the same type and name in both the vertex shader and the fragment shader. A classic use of this would be a vertex's normal since this can be used in the lighting calculations.

Varyings 是我們在vertex shadder中宣告與fragment shader分享的變數。在做這個時，我們要確定我們在vertex shader和fragment shader上以相同型別及名稱宣告varying variable。在一般的使用上這可能是一個vertex的normal，讓他可以在lighting的計算上使用。

現在我們已經說明了這兩種shaders與能夠使用的變數型別了，現在就讓我們來看看shader最簡單的例子:

Here, then, is the Hello World of vertex shaders:

```
/**
 * Multiply each vertex by the
 * model-view matrix and the
 * projection matrix (both provided
 * by Three.js) to get a final
 * vertex position
 */
void main() {
  gl_Position = projectionMatrix *modelViewMatrix *vec4(position,1.0);
}
```

and here's the same for the fragment shader:

```
/**
 * Set the colour to a lovely pink.
 * Note that the color is a 4D Float
 * Vector, R,G,B and A and each part
 * runs from 0.0 to 1.0
 */
void main() {
  gl_FragColor = vec4(1.0,  // R
                      0.0,  // G
                      1.0,  // B
                      1.0); // A
}
```

That's really all there is to it. If you were to use that you would see an 'unlit' pink shape on your screen. Not too complicated though, right?

In the vertex shader we are sent a couple of uniforms by Three.js. These two uniforms are 4D matrices, called the Model-View Matrix and the Projection Matrix. You don't desperately need to know exactly how these work, although it's always best to understand how things do what they do if you can. The short version is that they are how the 3D position of the vertex is actually projected to the final 2D position on the screen.

I've actually left them out of the snippet above because Three.js adds them to the top of your shader code itself so you don't need to worry about doing it. Truth be told it actually adds a lot more than that, such as light data, vertex colours and vertex normals. If you were doing this without Three.js you would have to create and set all those uniforms and attributes yourself. True story.

USING A MESHSHADERMATERIAL

OK, so we have a shader set up, but how do we use it with Three.js? It turns out that it's terribly easy. It's rather like this:

```
/**
 * Assume we have jQuery to hand
 * and pull out from the DOM the
 * two snippets of text for
 * each of our shaders
 */
var vShader = $('vertexshader');
var fShader = $('fragmentshader');
var shaderMaterial =
  new THREE.ShaderMaterial({
    vertexShader:   vShader.text(),
    fragmentShader: fShader.text()
  });
```

From there Three.js will compile and run your shaders attached to the mesh to which you give that material. It doesn't get much easier than that really. Well it probably does, but we're talking about 3D running in your browser so I figure you expect a certain amount of complexity.

We can actually add two more properties to our MeshShaderMaterial: uniforms and attributes. They can both take vectors, integers or floats but as I mentioned before uniforms are the same for the whole frame, i.e. for all vertices, so they tend to be single values. Attributes, however, are per- vertex variables, so they are expected to be an array. There should be a one- to-one relationship between the number of values in the attributes array and the number of vertices in the mesh.

CONCLUSION I'll stop there for now as we've actually covered a rather large amount, and yet in many ways we've only just scratched the surface. In the next guide I'm going provide a more advanced shader to which I will be passing through some attributes and uniforms as well as doing a bit of fake lighting.

I've wrapped up the source code in this lab article so you have it as a reference. If you've enjoyed this let me know via Twitter - it makes for a happy Paul.

資料來源：

https:\/\/aerotwist.com\/tutorials\/an-introduction-to-shaders-part-1\/