Terreno procedurale

Computazione procedurale di una geometria mediante Three.js

Computer Graphics

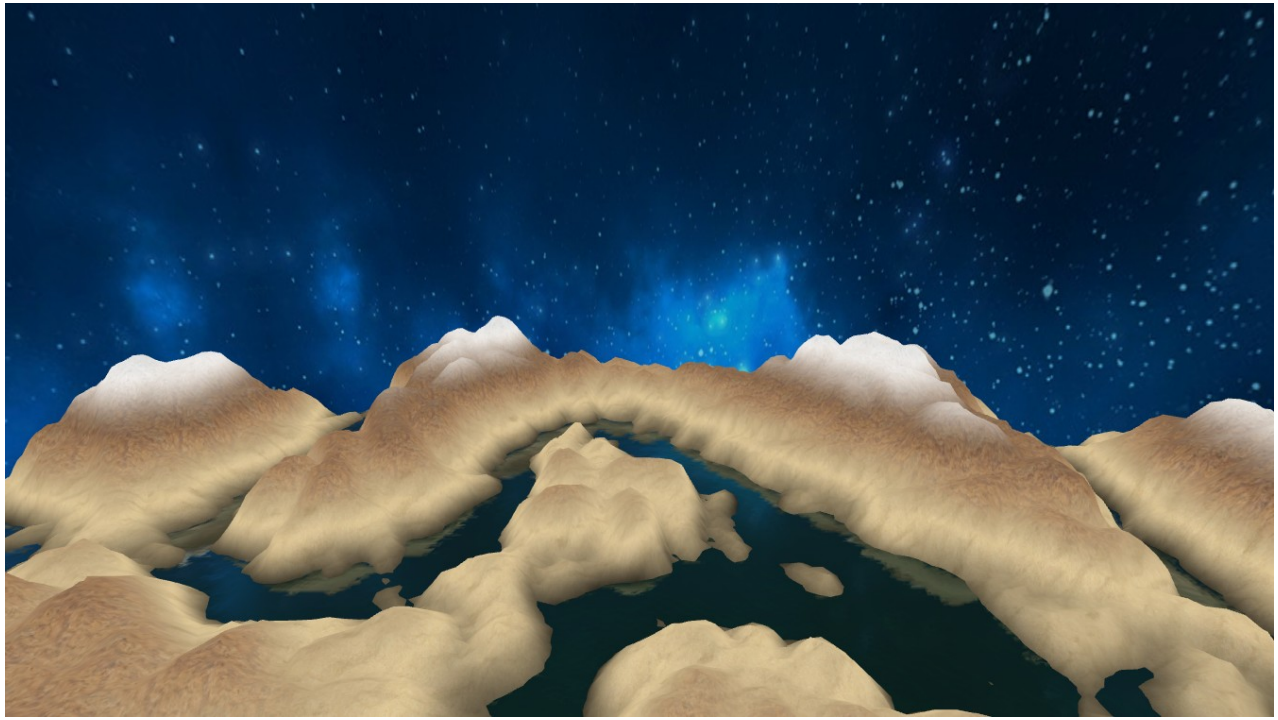
Di Guardo, Garritano, Marchetti

Firenze, 21 Giugno 2016

Procedurale: definizione

Il termine **procedurale** si riferisce a quel processo che calcola una particolare funzione.
I frattali sono un esempio di generazione procedurale.
I contenuti che vengono spesso creati proceduralmente sono le **texture**, le **mesh**, ma anche **suoni**.

Il nostro elaborato si è concentrato sullo sviluppo procedurale di una geometria che rappresentasse un paesaggio montano.



Tecnologia: Three.js

Three.js è una libreria **Javascript** cross-browser usata per creare e visualizzare animazioni 3D di computer graphic in un **browser** web.
Three.js usa **WebGl**, il quale è basato su **OpenGL** ES 2.0 e usa l'elemento canvas di **HTML5** e l'interfaccia Document Object Model (**DOM**).

Three.js è **open source**, fornito con licenza MIT. Il codice è disponibile su Github.

Il nostro lavoro

- Studio della **tecnologia** Three.js

  (scena, luci, camera, controlli, mesh geometrie e materiali, rendering)
- Divisione del lavoro
- Creazione **elementi base** per il rendering di una scena 3D
- Creazione **struttura portante** (piano suddiviso in sottosezioni e metodi per il calcolo procedurale)
- Modellazione della geometria attraverso una mappa delle altezze (**heightmap**)
- Implementazione della **texture** alla geometria mediante l'uso di uno shader.
- Aggiunta dell'**acqua** utilizzando un secondo shader
- Implementazione della **Skysphere**
- Implementazione del **Lens flair**

L'oggetto Block

L'oggetto base con cui è costruita la **griglia** del terreno è un `Object3D` composto da due **mesh**:

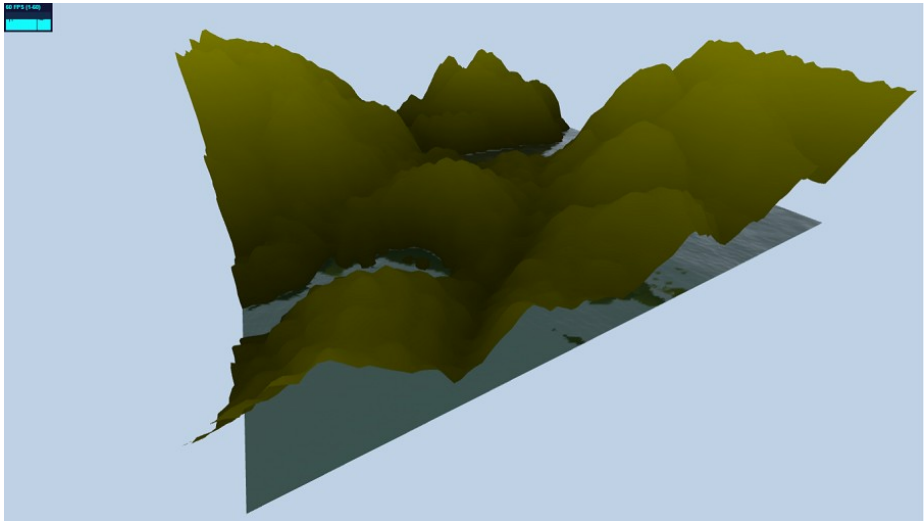
- Un piano in cui viene costruito il blocco di **terreno**
- Un piano per simulare l'**acqua** nel blocco

Per entrambi, abbiamo usato la geometria `PlaneBufferGeometry`, che permette un totale controllo degli attributi della `Geometry`.

Tale blocco è inserito all'interno di un oggetto Javascript, responsabile della **costruzione** e dell'**aggiornamento** delle mesh, nonché del **calcolo delle altezze** e dell'applicazione delle **textures**.

L'oggetto contiene gli attributi di base riguardo alla **posizione** e al **tipo** del blocco, nonché due metodi di utilità:

- `getMesh`() : per ottenere l' `Object3D` completo.
- `updateCoordinates`() : per aggiornare le sue coordinate



```
Block = function(x,y,type) {
    this.x = x; this.y = 0; this.z = y;
    this.type = type;
    var obj = new THREE.Object3D();
    var geometry = new
THREE.PlaneBufferGeometry( … );
    var material = new
THREE.MeshBasicMaterial( … );
    var mesh = new THREE.Mesh(geometry,
material);
    this.acqua = new THREE.Mesh( … );
    obj.add(mesh);
    obj.add(this.acqua);
    this.getMesh = function () { … };
    this.updateCoordinates = function () { … };
    [..]
}
```

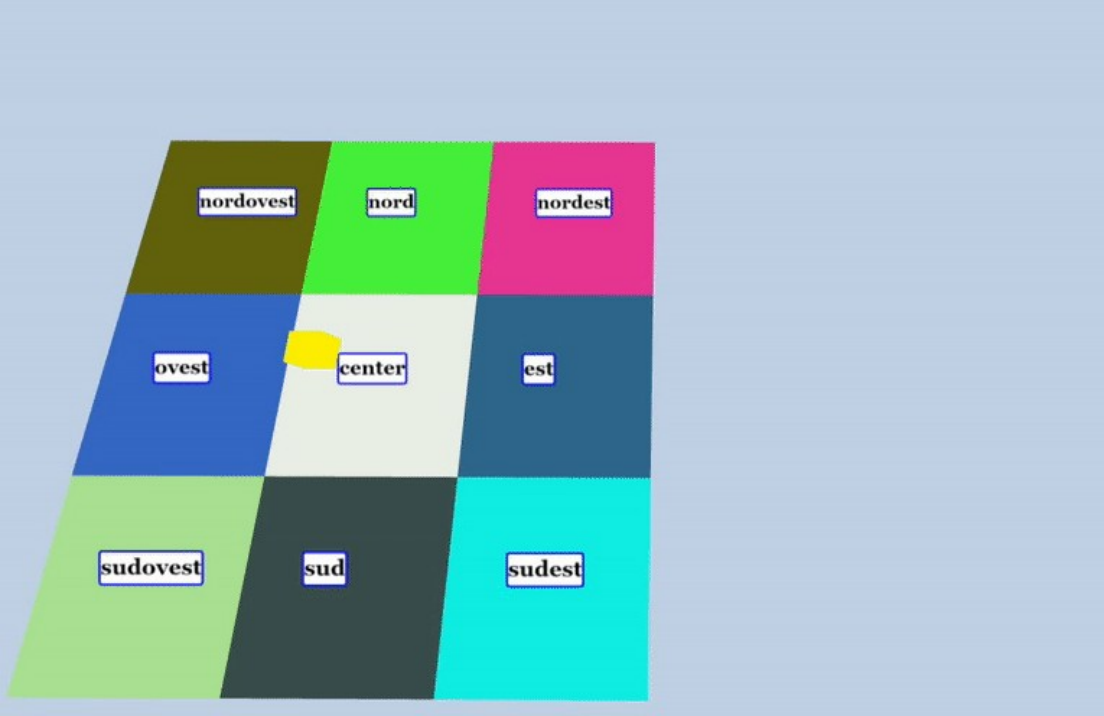
Costruzione della griglia procedurale

Per definire il terreno **procedurale**, l'idea è costruire una **griglia 3x3**, composta da **9 blocchi** di terreno, ognuno etichettati con una **label**.

In ogni momento la **camera** si trova nel blocco «**center**».

La generazione dei blocchi successivi avviene quando la camera si muove dal blocco centrale ad uno dei quattro blocchi: **nord**, **sud**, **ovest**, **est**.

Ogni volta che viene chiamata la funzione *render(), CheckCameraPosition()* controlla la **posizione** della camera e se necessario chiama *updatePlanes()* che **aggiorna** la griglia.

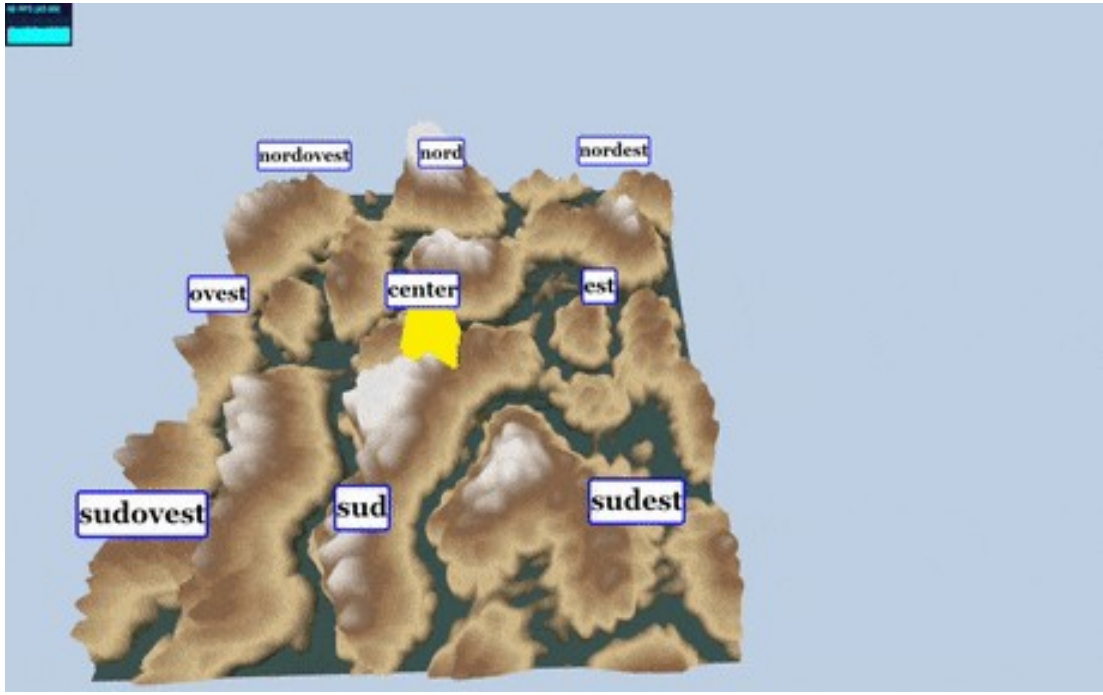


Aggiornamento della griglia procedurale

L'**aggiornamento** consiste in tre operazioni:

- Vengono **cancellati** i 3 blocchi più lontani «alle spalle» della camera
- Vengono **aggiornate** le coordinate e le labels dei restanti 6 piani
- Vengono **creati** ed etichettati 3 nuovi piani in base alla posizione e allo spostamento della camera

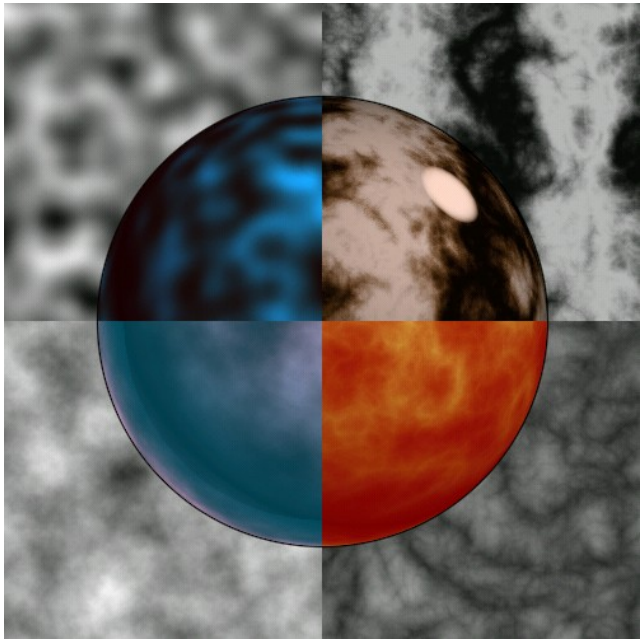
Tale meccanismo permette di creare un terreno **infinito**, memorizzando, in ogni momento, solo **9 blocchi**.



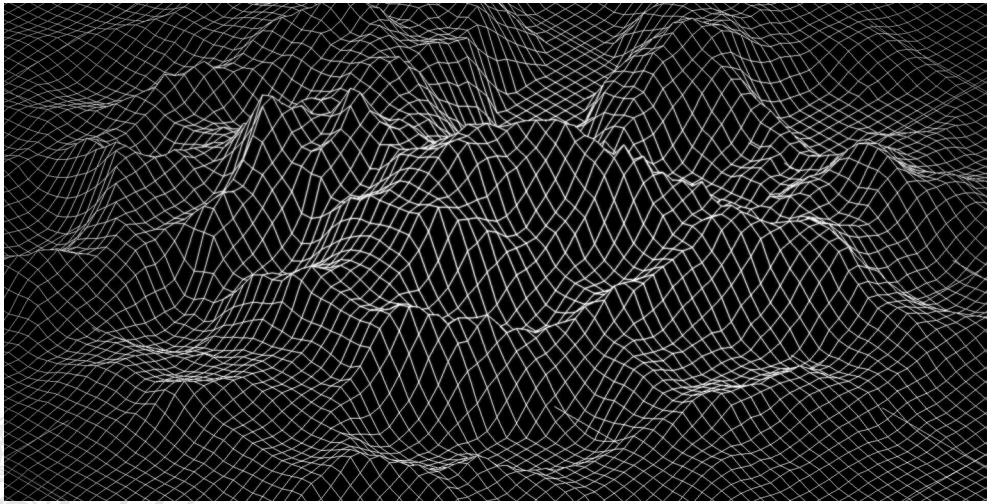
Inoltre, grazie al meccanismo di creazione del **rumore**, quando viene creato un blocco nella stessa posizione in cui ne era stato cancellato un altro, tale blocco risulterà **visivamente identico** al precedente.

Perlin Noise

Perlin Noise è un metodo per calcolare una forma di **rumore** coerente. Si può utilizzare per diversi scopi, come la creazione della turbolenza dell'acqua o del fuoco, la superficie di un metallo, la forma delle nuvole, la texture di una roccia, ecc.



La geometria dell'oggetto Block è stata modellata fissando le altezze di **ogni vertice**, calcolate con il **Perlin Noise**.

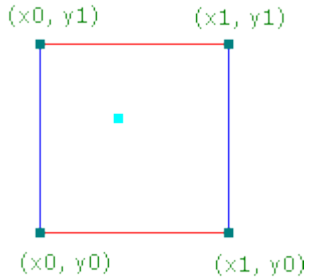


Perlin Noise: Teoria

Nel nostro caso è stato utilizzato un **noise 2D**. Supponiamo di avere una funzione del tipo

`noise2d(x, y) = z` con x,y e z numeri reali (floating-point).

Definiamo una funzione noise su una **griglia regolare**, i cui punti sono definiti per ogni numero intero.
Vediamo quindi che il punto (x,y) è circondato **da 4 punti** che chiamiamo (x0, y0), (x0, y1), (x1, y0), e (x1, y1).



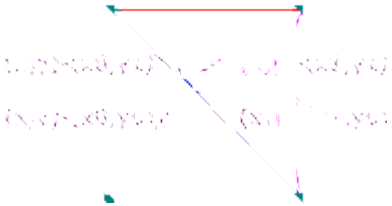
Per ogni punto della griglia regolare calcoliamo in modo **pseudocasuale** un vettore.
Per pseudocasuale si intende che ogni volta che viene calcolato "g" su un punto (x0,y0), si ha comunque lo stesso risultato. Ogni direzione ha la **stessa probabilità** di essere scelta.

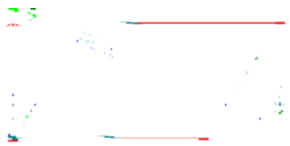
Perlin Noise: Teoria

Per ogni punto della griglia si calcola un vettore che va in **direzione** `(x,y)`

Si calcolano dei **pesi** per ogni punto della griglia regolare

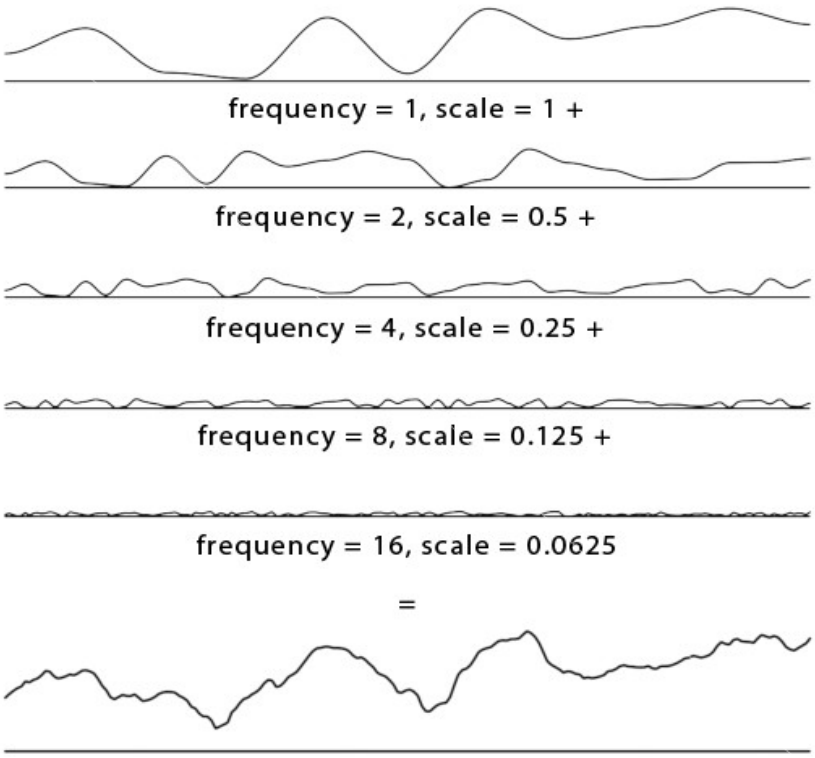
```
s = g(x0,y0) · ((x,y) - (x0,y0))
t = g(x1,y0) · ((x,y) - (x1,y0))
u = g(x0,y1) · ((x,y) - (x0,y1))
v = g(x1,y1) · ((x,y) - (x1,y1))
```

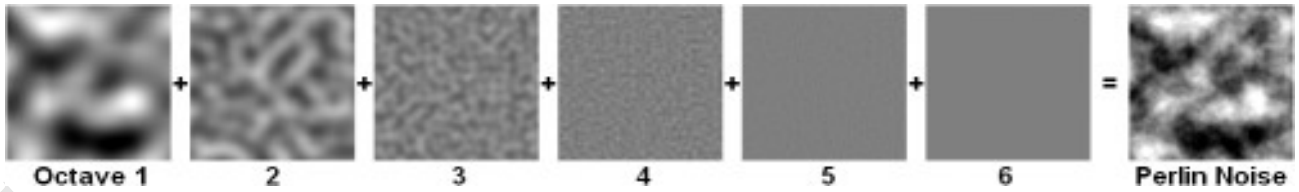




Si interpolano questi pesi per calcolare il valore **z** della funzione `noise2d(x,y) = z`

Perlin Noise: Pratica





Perlin Noise: Pratica

```
[...]
perlin = new ImprovedNoise();
var quality = 5; //serve per gestire le altezze in relazione alle frequenze
var wideX = 15; //dilatazione rispetto all'asse X
var wideY = 15; //dilatazione rispetto all'asse Y
var seaLevel = 30; //altezza del livello del mare
var height = 2; //moltiplicatore delle altezze di ogni vertice del terreno
var initX = worldWidth*squareIdX;
var initY = worldDepth*squareIdY;
var period = worldWidth;
/*  Si sommano 5 risultati del Perlin, una per ogni frequenza.
 Ogni vertice è visto come un una coordinata (x,y) di un quadrato.
 Per dare continuità al terreno è necessario che l'ultima fila di punti di un quadrato e la prima del quadrato
vicino abbiano gli  stessi valori.
 Per questo viene sommato un fattore correttivo per ogni x e y.
 Il fattore correttivo è ugual al lato del quadrato, diviso il numero dei punti
*/
for (var x = initX; x < (initX)+period; x++) {
    for (var y = initY; y < (initY)+period; y++) {
                verticesMatrix[x%(initX)][y%(initY)] =0;
               for (var j = 0; j<5; j++)
                       verticesMatrix[x%(initX)][y%(initY)] += Math.abs(perlin.noise((wideX*x+wideX*(((x)%(initX)
                                   *(1/(period-1))))) / Math.pow(quality,j), (wideY*y+wideY*(((y)%(initY)*
                                   (1/(period-1))))) / Math.pow(quality,j), z));


        verticesMatrix[x%(initX)][y%(initY)] *=height; //moltiplica l'altezza del vertice per il fattore height
        verticesMatrix[x%(initX)][y%(initY)] +=-(j*seaLevel);
    }
}
```

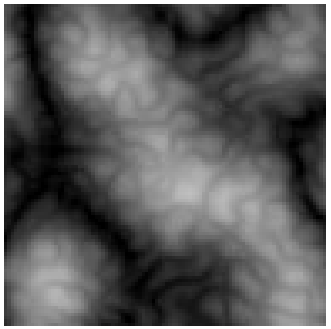
Applicazione Texture

OBIETTIVO: Texture diverse a seconda dell'altezza del terreno



Tre texture utilizzate: file jpg che rappresentano **sabbia**, **roccia** e **neve**

STRATEGIA: Creare una **heightmap** per ogni blocco generato e passare la mappa allo **shader**



HEIGHTMAP: ogni pixel dell'immagine rappresenta una altezza quantificata dal livello di grigio

Ogni altezza è ricavata dalla **matrice** delle altezze creata dall'algoritmo Perlin.

Livello di grigio **0** = altezza **minima.**
Livello di grigio **255** = altezza **massima.**

Valore dell'altezza normalizzata fra 0 e 255

Shader

Passo fondamentale: creare il **materiale** per la texture che verrà applicato alla **geometria**.

- In three.js una mesh si "renderizza" combinando la geometria a un materiale

```
var mesh = new THREE.Mesh( geometry, customMaterial);
```

In Three.js, **ShaderMaterial** permette di modellare il materiale utilizzando gli shaders.

```
var customMaterial = new THREE.ShaderMaterial(
    {
        uniforms:customUniforms,
        vertexShader:   document.getElementById( 'vertexShader'   ).textContent,
        fragmentShader: document.getElementById( 'fragmentShader' ).textContent,

            }   );
```

Uniform

```
customUniforms = { //crea le variabili uniform da passare allo shader
    heightmap: { value: heightmap },
    sandyTexture:   { type: "t", value: sandyTexture },
    rockyTexture:   { type: "t", value: rockyTexture },
    snowyTexture:   { type: "t", value: snowyTexture }
};
```

Vertex Shader

Vertice <-> value heightmap tramite coordinate uv

```
uniform sampler2D heightmap;
varying float value;
varying vec2 vUV;


void main()
{
      vUV = uv;
      vec4 height = texture2D( heightmap, uv );

      value = height.r;
      vec3 newPosition = position + value;

      gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
}
```

value: valore dell'altezza (0<value<1)

Mappatura UV (Texture Mapping):
tramite le coordinate **uv** fare corrispondere i **pixel** della texture heightmap
con i vertici della geometria

- vec4 **texture2D**(sampler2D sampler, vec2 coord)
  Restituisce il valore del **colore** della texture
  nella specifica **coordinate uv**

- **modelViewMatrix**: matrice dell'oggetto rispetto la camera
- **projectionMatrix:**matrice di proiezione della camera

Fragment Shader

```
uniform sampler2D sandyTexture;
uniform sampler2D rockyTexture;
uniform sampler2D snowyTexture;

varying vec2 vUV;
varying float value;

void main()
{
    vec4 sandy = (smoothstep(0.01, 0.20, value) - smoothstep(0.19, 0.40, value)) *
texture2D( sandyTexture, vUV * 10.0 );

    vec4 rocky = (smoothstep(0.21, 0.40, value) - smoothstep(0.30, 0.70, value)) *
texture2D( rockyTexture, vUV * 20.0 );

    vec4 snowy = smoothstep(0.40, 0.65, value) * texture2D( snowyTexture, vUV *
10.0 );
    gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0) + sandy  + rocky + snowy;

}
```

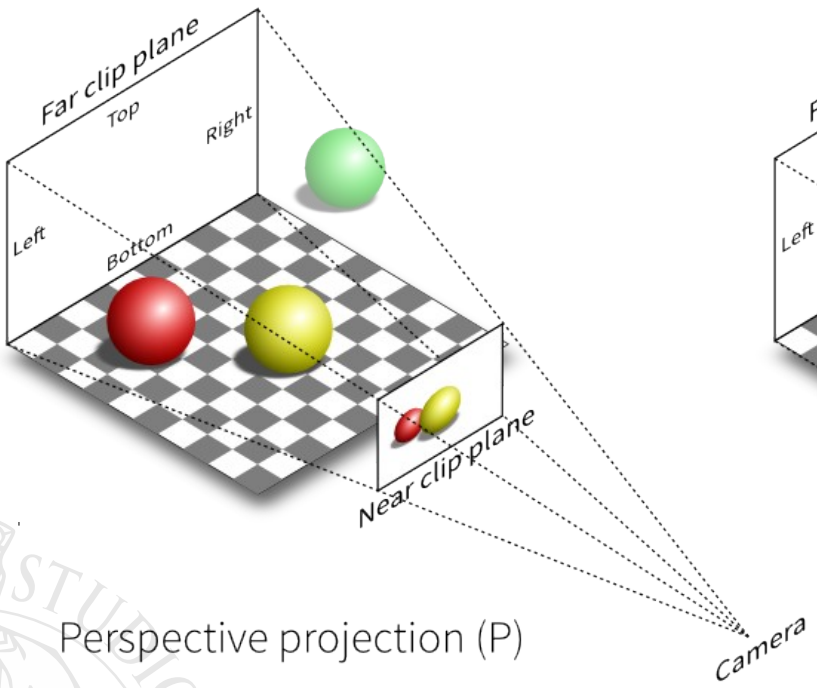
- float **smoothstep**(float edge0, float edge1, float x)
  Se x<edge0 restituisce 0, se >edge1 restituisce 1
  Se edge0<x<edge1 esegue una **interpolazione** usando il polinomio Hermite

- Ogni pixel è assemblato tramite le tre componenti delle texture (**sandy**, **rocky**, **snowy**).
- Ogni componente è pesata utilizzando la funzione di interpolazione *smoothstep* tramite il valore dell'altezza

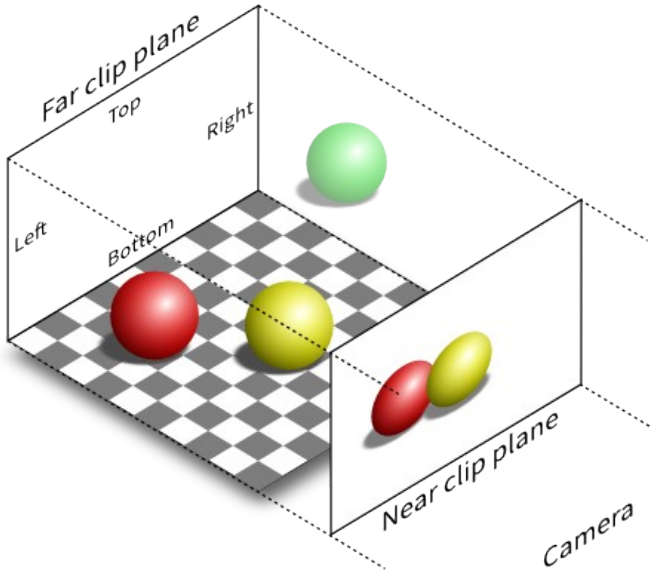
Camera

- Camera scelta**: *prospettica***
  - Prospettiva del **mondo reale**: visione dell'occhio umano
  - Gli oggetti distanti appaiono più piccoli rispetto a quelli più vicini

- *Prospettica Vs Ortografica:*
- la camera ortografica presenta gli oggetti alla **stessa dimensione**
  - indipendentemente dalla loro **distanza**



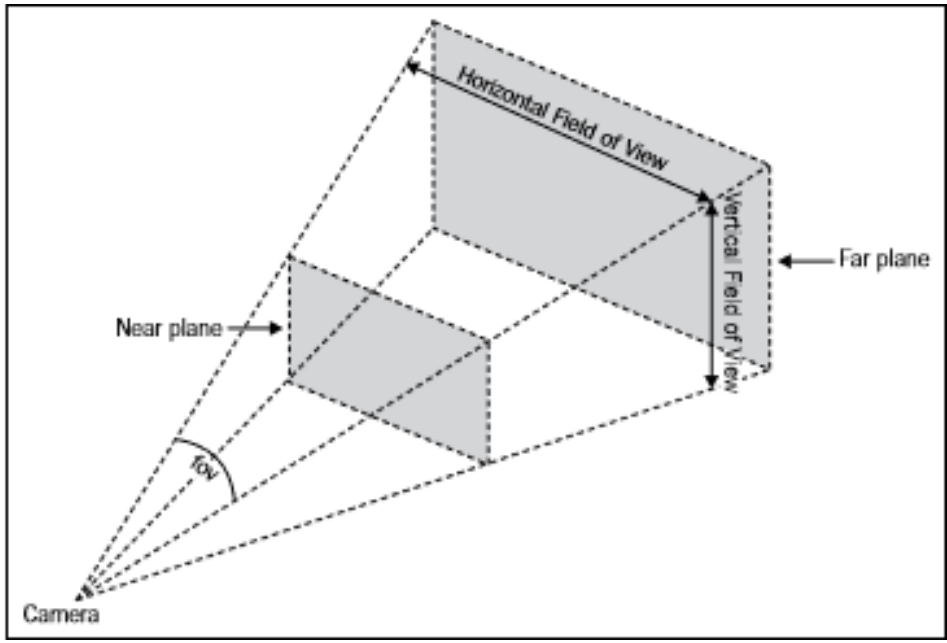
Perspective projection (P)



Orthographic projection (O)

Implementazione Camera

```
THREE.PerspectiveCamera(70, window.innerWidth /
 window.innerHeight, 0.5, 5000);
```



Controllo Camera:

- Libreria utilizzata Three.js: **FirstPersoncontrols.**
- Stile: visuale in **prima persona**.
- Controllo: tastiera-movimento, visuale-mouse.

```
controls = new THREE.FirstPersonControls( camera );
controls.movementSpeed = 600;
controls.lookSpeed = 0.05;
```

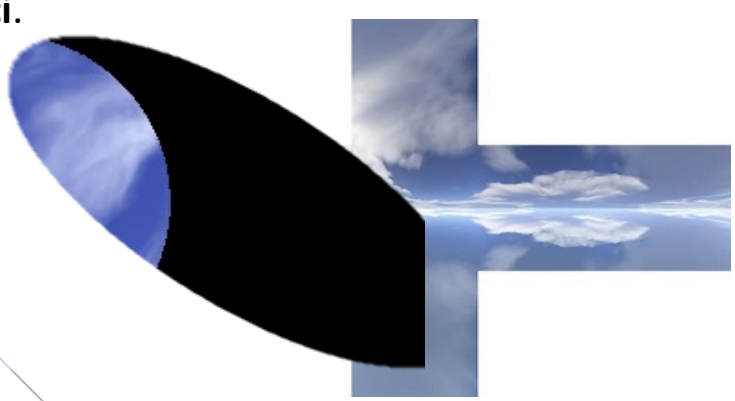
La SkySphere

La **SkySphere** è una **sfera** che circonda la griglia, che, unita ad una apposita **texture**, simula il cielo e le nuvole.

In ogni momento, la **camera** si trova esattamente al centro della sfera, così questa si vede sempre alla **stessa distanza**.

Inoltre, ogni volta che viene chiamata la funzione di rendering *render*(), la sfera **ruota** leggermente intorno gli assi x e y, per simulare il movimento delle nuvole.

Inoltre, premendo «spazio», si passa alla modalità **notte**, in cui cambia la **texture** della sfera e diminuisce l'intensità delle **luci**.



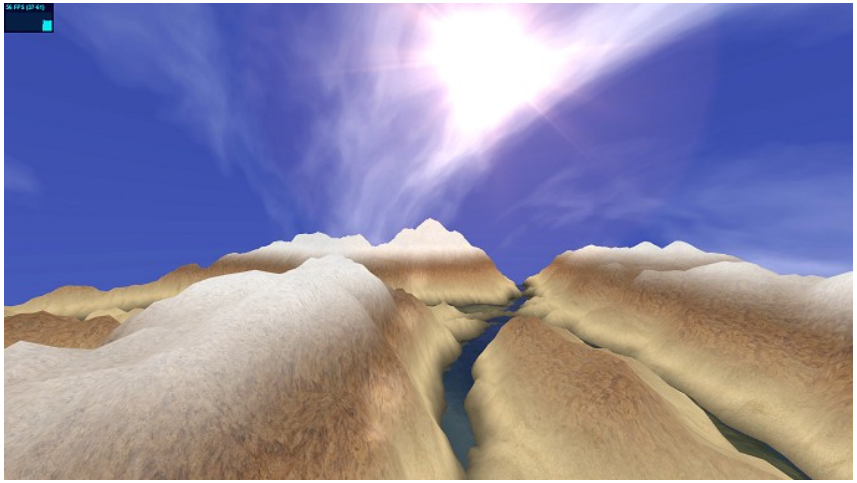
Rispetto ad una **skybox** (cubica), la skysfera fornisce un aspetto **più realistico** alla scena, però è adatta solo per scene di piccole dimensioni perchè richiede un numero di poligoni superiori rispetto ad un cubo.

La Luce

Nella scena sono presenti **due luci**, agenti su differenti tipi di materiale:

- Una luce **direzionale** che agisce principalmente **sull'acqua**.
- Una luce **puntuale** che agisce globalmente **su ogni mesh** della scena.

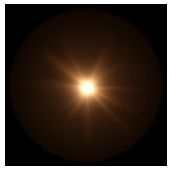
La luce direzionale garantisce gli effetti di **riflesso** e **trasparenza** dell'acqua



La luce puntuale cambia intensità in base alla **distanza** della camera dalla fonte stessa

Lens Flare

La Lens Flare simula gli effetti del **Sole** nella scena.



ThreeJS fornisce l'oggeto LensFlare che si basa su una texture:

```
lensFlare = new THREE.LensFlare( textureFlare0, 800, 0, THREE.AdditiveBlending, flareColor );
```

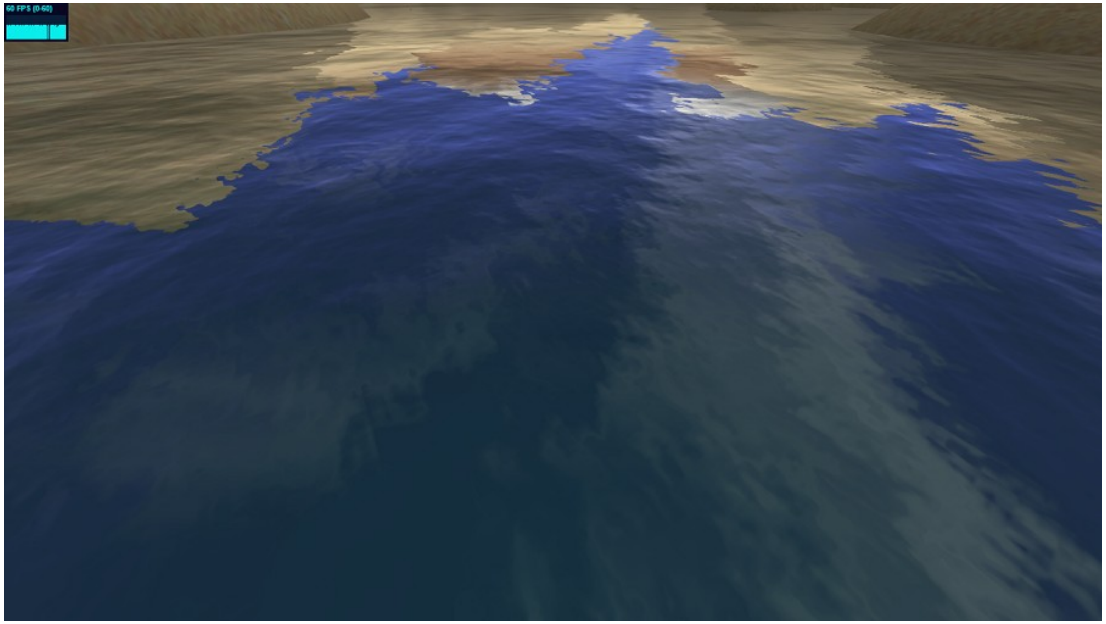
È posto in corrispondenza della luce **puntuale**, scelta perché emette luce in ogni direzione.

Si crea così l'effetto di **riflesso** tipico di quando una videocamera inquadra il Sole.

In modalità *notte* **l'intensità** le **dimensioni** della luce della lens flare sono ridotte

Water Shader

L'acqua è formata da un **piano**, posto al livello 0 (y=0) di ogni **blocco** di terreno, quando viene creato.



È generata e controllata da uno **shader**, importato da ThreeJS, che ne definisce le proprietà di **trasparenza** e di **riflesso** del mondo che la circonda.

Tali proprietà dipendono dalla **luce direzionale**, la cui posizione è presa come punto di riferimento.

Inoltre è presente la classe THREE.Water che gestisce la creazione e il **movimento** dell'acqua.

```
THREE.Water = function (renderer, camera, scene, options) { … }
```