



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Terreno Procedurale in P5.js

Progetto Computer Graphics and 3D

26 Novembre 2021

Serena Giachetti, Lisa Cresti



Indice

- **Introduzione**
 - Cosa si intende con terreno procedurale?
- **Tecnologie**
 - P5.js
- **Generazione Terreno**
 - Struttura
 - Perlin Noise
- **Aggiornamento Terreno**
- **Camera**
 - Creazione e movimento
- **Pannello di Controllo**
- **Altri oggetti nella scena**
 - Pattern di Voronoi



Cosa si intende con terreno procedurale?

È un metodo per la creazione algoritmica di dati che rappresentano l'altitudine del terreno generato. Questo processo permette di creare grandi quantità di dati senza rallentare il rendering dei risultati dando l'illusione di vedere un terreno infinito.





TERRENO

I dati computati simulano le altitudini di un terreno.

Questi nel nostro caso possono assumere valori positivi o negativi. Tutto quello che si trova al di sotto dello 0 sarà il fondale del lago e tutto quello che sta al di sopra disegnerà la forma delle pianure, colline e montagne.





PROCEDURALE

Il termine si riferisce al fatto che il terreno si rigenera con la computazione di una funzione. L'esplorazione è quindi in parte simulata visto che è il terreno che si rigenera con aspetti diversi a seconda della direzione del movimento. Il fatto che i valori ottenuti siano frutto di calcoli algoritmici ci permette facilmente di ricostruire le corrette strutture indipendentemente dalla direzione di esplorazione.



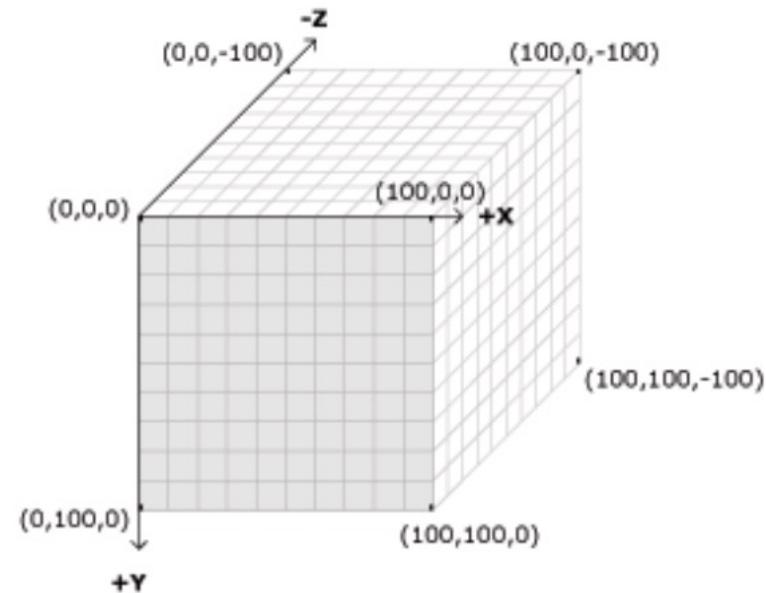


P5.js

È una libreria open-source di JavaScript.

Questa permette la creazione e il disegno di oggetti in 2D e 3D all'interno del canvas o addirittura in tutta la pagina del browser tramite oggetti HTML5.

Il sistema di riferimento utilizzato all'interno del canvas è questo:





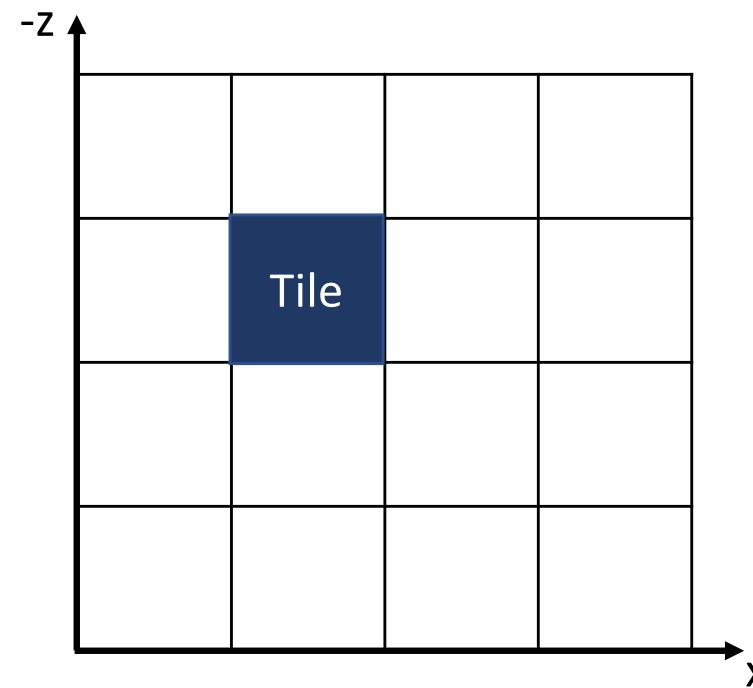
Struttura

La struttura del terreno si basa su una griglia 4x4 posizionata sul piano XZ.

Ogni cella della griglia (600x600) è composta da un oggetto Tile.

I tipi di tile si differenziano per la tipologia di heightMap utilizzata. Le categorie sono 4:

- O, mappa di origine
- A, flipZ
- B, flipX
- C, flipX e flipZ





Class Tile

Ogni tile al momento della creazione, contiene l'informazione:

- sulla sua posizione nella griglia bidimensionale (x, z)
- sui valori di altitudine: nello specifico viene passata una coppia composta dal tipo e dalla matrice delle altezze

Con la funzione drawMesh()
disegniamo nella scena il terreno
relativo alla tile, con la texture
desiderata, e sfruttando i
TRIANGLE_STRIP.

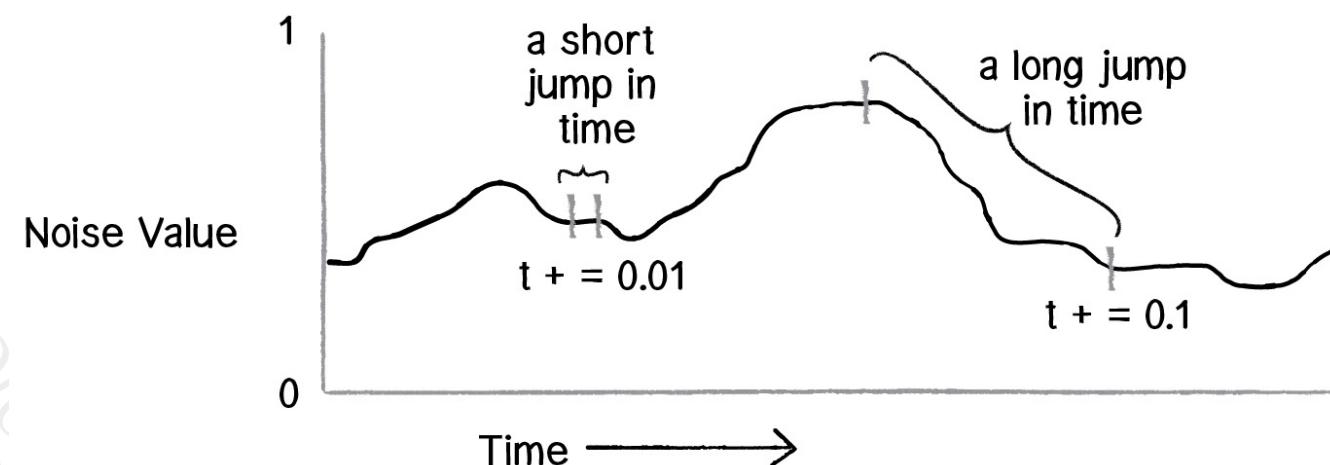
```
drawMesh(){
    texture(groundTex);
    var y1, y2;
    for (var z = 0; z < tileRows - 1; z++){
        beginShape(TRIANGLE_STRIP);
        // Build triangles strip
        //
        //   (x0, y01, z1)   (x1, y11, z1) ... (x(cols-1), y(cols-1)1, z1)
        //   *-----*-----*-----*
        //   | A   / | C   / | E   / | G   / |
        //   |   / |   / |   / |   / |
        //   | /   B | /   D | /   F | /   H |
        //   *-----*-----*-----*
        //   (x0, y00, z0)   (x1, y10, z0) ... (x(cols-1), y(cols-1)0, z0)
        //
        for(var x = 0; x < tileCols; x++){
            y1 = this.heightMap[z][x];
            y2 = this.heightMap[z+1][x];
            vertex(x*scl, -this.heightMap[z][x]*scl, -z*scl, map(y1, -mult, mult, 0, 2480), 1040/2);
            vertex(x*scl, -this.heightMap[z+1][x]*scl, -(z+1)*scl, map(y2, -mult, mult, 0, 2480), 1040/2);
        }
        endShape();
    }
}
```



Perlin Noise

È un tipo di rumore solitamente usato per generare texture procedurali tramite la creazione di numeri pseudo-randomici con valori fra loro vicini. Questo crea quindi curve smooth che danno un'apparenza molto naturale alla scena.

Il rumore di Perlin è una combinazione di sinusoidi che possono essere attenuate o accentuate andando a modificarne le frequenze e le ampiezze. I diversi rumori che si verranno a generare si chiamano ottave ed ognuno avrà un peso diverso sullo shape del terreno.





Perlin Noise

GenerateAltitudeMap() è la funzione che genera la heightMap originale usando il Perlin Noise.

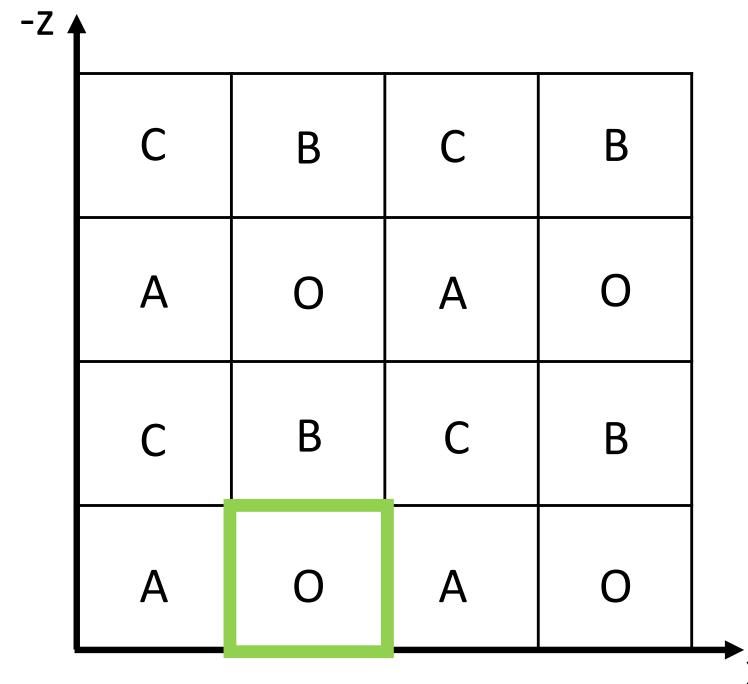
```
function generateAltitudeMap(tileCols, tileRows, inc, mult){
    var terrainMap = [];
    var xoff = 0;
    var zoff = 0;

    for(var z = 0; z < tileRows; z++){
        terrainMap.push([]);
        xoff = 0;
        for(var x = 0; x < tileCols; x++){
            n = (1 * noise(0.8 * xoff, 0.5 * zoff) + 0.08 * noise(9 * xoff, 9 * zoff));
            if (n < 0.15){
                n = 0.15;
            }
            terrainMap[z][x] = map(n, 0.15, (1 + 0.08), -mult, mult);
            xoff = xoff + inc;
        }
        zoff = zoff + inc;
    }

    return terrainMap;
}
```

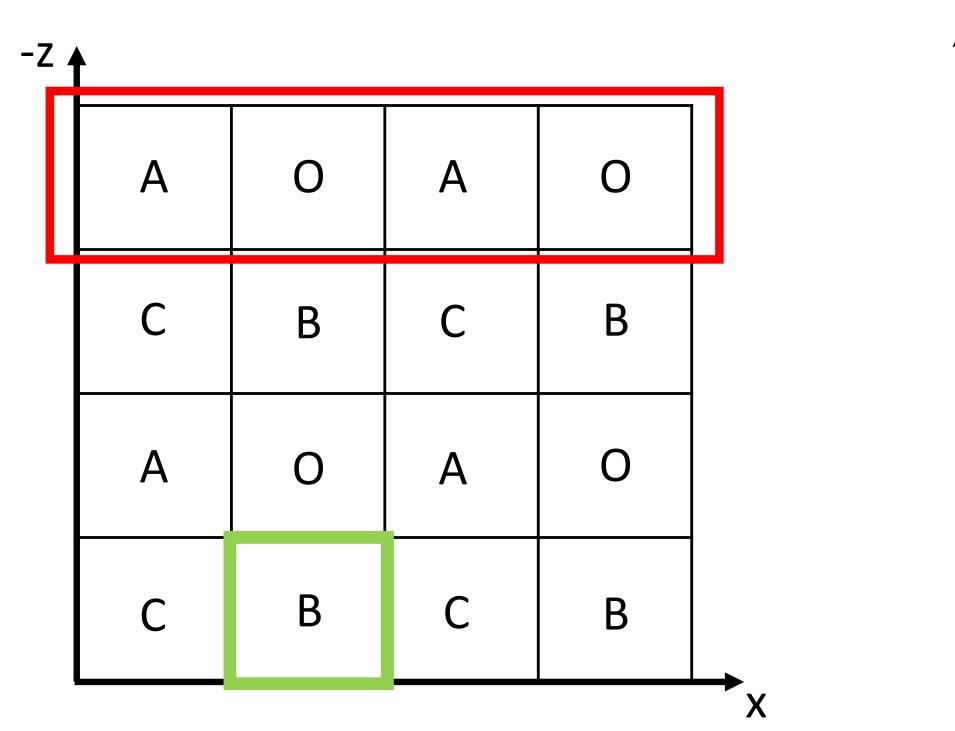


Aggiornamento della griglia



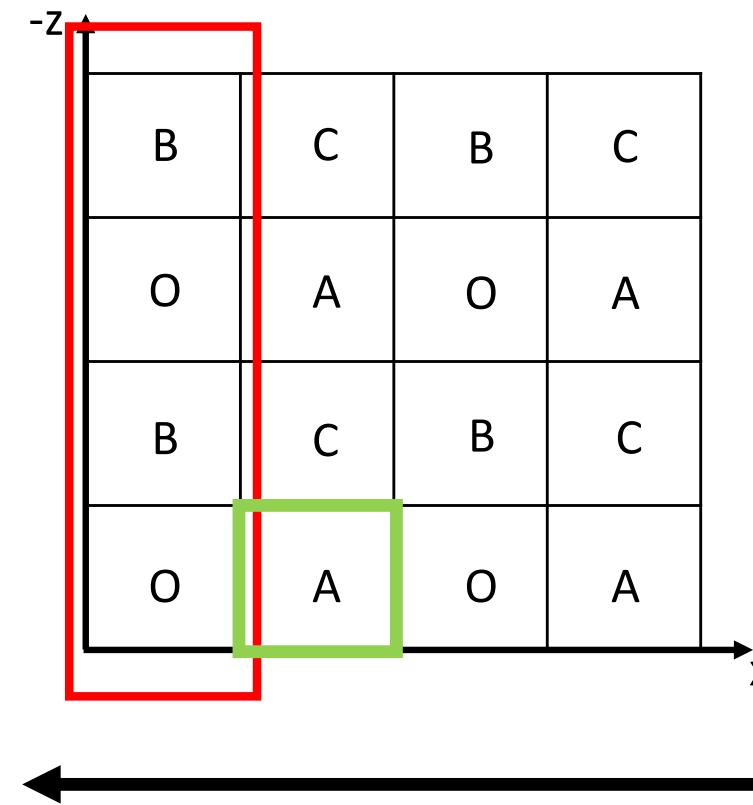


Aggiornamento della griglia





Aggiornamento della griglia





Aggiornamento della griglia

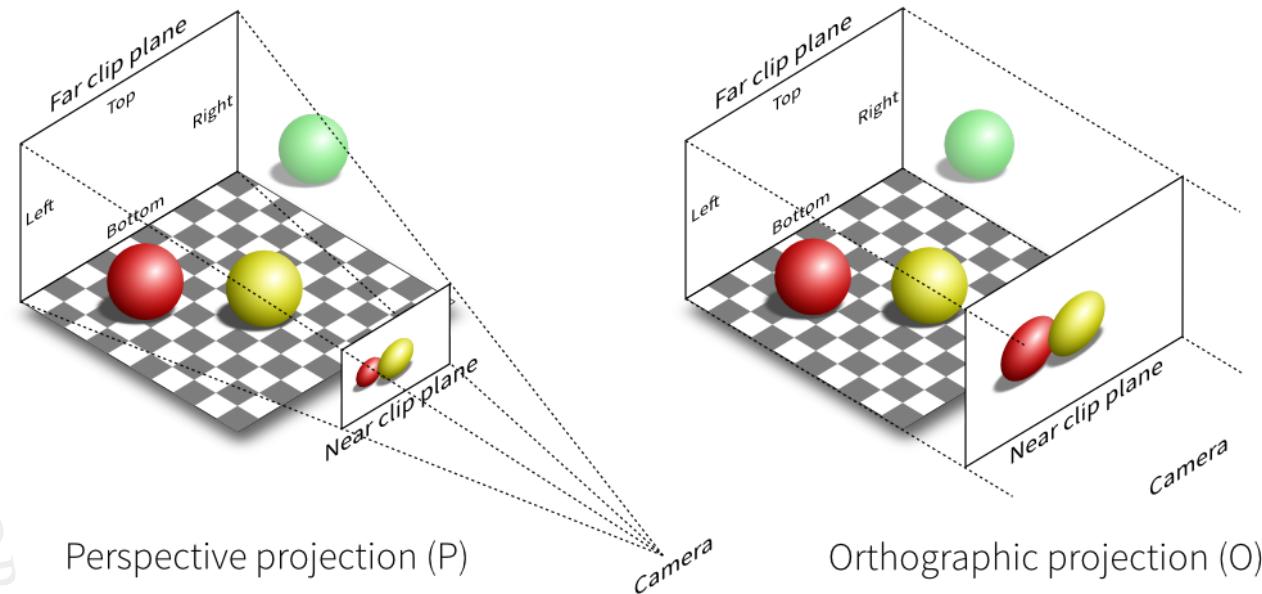
```
function updateTerrainGrid(){
    switch(direction) {
        case 'up':
            for (var i = 0; i < 4; i++){
                for (var j = 0; j < 4; j++){
                    if (j == 3){
                        terrainGrid[i][j].updateMap(terrainGrid[i][1].getInfo());
                    }
                    else {
                        terrainGrid[i][j].updateMap(terrainGrid[i][j+1].getInfo());
                    }
                }
            }
            break;
    }
}
```



Camera

Possiamo definire due tipologie di camera.

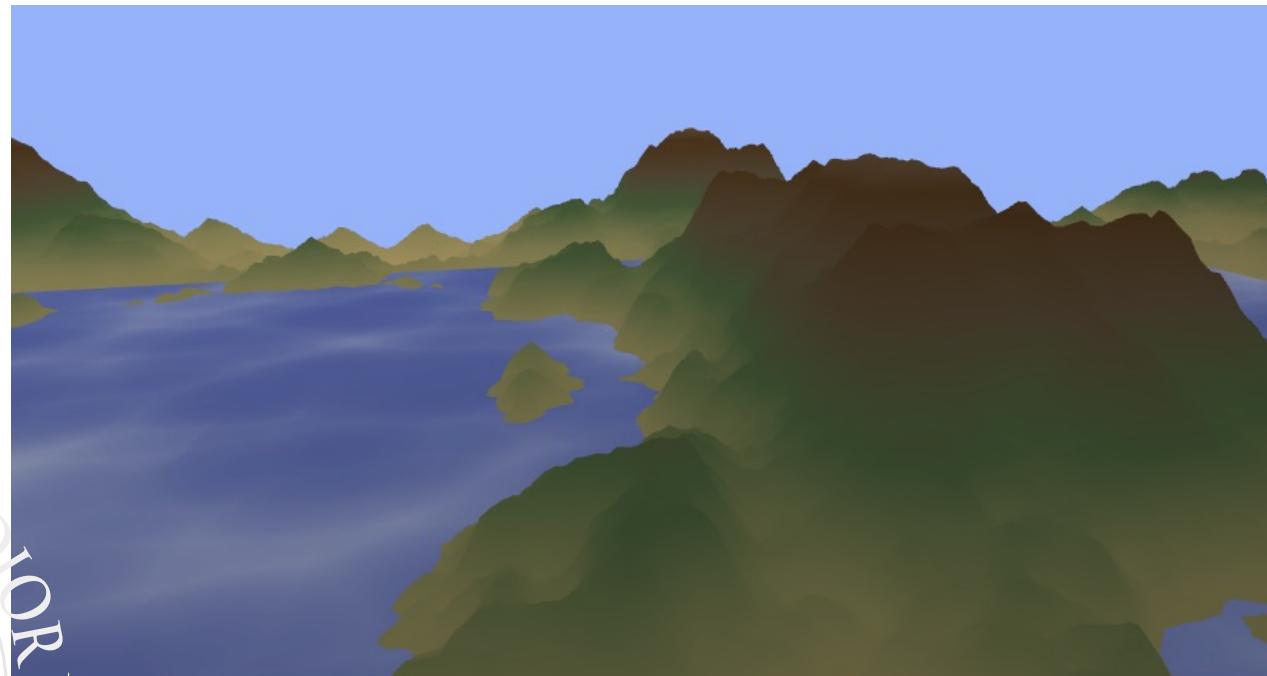
La camera prospettica è il modo in cui vediamo il mondo reale. Le cose hanno profondità e possiamo capire la distanza. La camera ortografica, invece, rimuove la prospettiva e gli oggetti vengono disegnati senza distorsione prospettica.





Camera

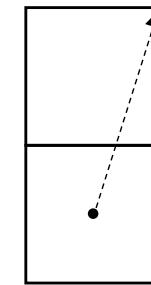
Dato che il nostro obiettivo era creare una camera che simulasse la vista umana abbiamo utilizzato una camera prospettica. In questo modo gli oggetti appaiono di dimensioni diverse in base alla loro distanza. A parità di dimensione un oggetto più vicino è più grande di un oggetto lontano.





Creazione camera

Abbiamo posizionato la camera al centro della tile e specificato le coordinate a cui punta la camera in modo che fosse leggermente più a destra.



Essendo una camera prospettica abbiamo definito anche i parametri del frustum della camera.

In particolare quello relativo al campo visivo verticale (dal basso all'alto) in gradi, dell'aspect ratio e le posizioni del near e del far plane.

```
camera = createCamera();
camera.setPosition(eyeX, eyeY, eyeZ);
camera.lookAt(atX, atY, atZ);
camera.perspective(60, width/height, 0.1, 1999);
```



Movimento Camera

Ogni volta che si ridisegna viene chiamata la funzione calculateDirection().

Questa funzione controlla la direzione e la velocità selezionate nel pannello di controllo e calcola gli spostamenti lungo X e Z per la camera e per gli altri oggetti dalla scena.

```
function calculateDirection(){
    var speed = pane.exportPreset().Speed/10;
    switch(pane.exportPreset().Direction) {
        case 'forward':
            xDisp = 0;
            if (flyingZ + speed > tileSize/2){
                flyingZ = flyingZ + speed - tileSize;
                zDisp = speed + tileSize;
                direction = 'up';
            }
            else {
                flyingZ = flyingZ + speed;
                zDisp = speed;
            }
            break;
        ...
    }
}
```



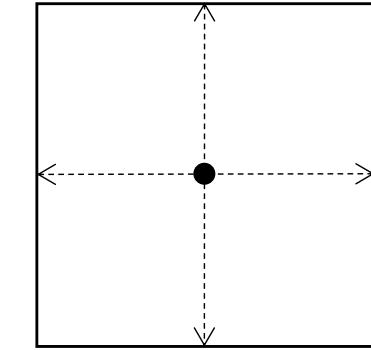


Movimento Camera

Gli spostamenti di camera sono accumulati nelle variabili

$$flyingX, flyingZ \in [-300, 300]$$

che vengono aggiunte alle coordinate della camera e della lookAt.



Si controlla, inoltre, la height map della tile attuale nella nuova posizione ($eyeX + flyingX$, $eyeZ - flyingZ$) e si aggiorna di conseguenza la posizione sull'asse verticale della camera. In questo modo ci troviamo sempre 10 pixel sopra il terreno.

Quando $flyingX$ o $flyingZ$ escono fuori dal range si procede all'aggiornamento della tile attuale come descritto precedentemente.





Pannello di Controllo

Per la creazione di un pannello di controllo abbiamo utilizzato la libreria Tweakpane.

Attraverso di esso possiamo modificare il colore dello sfondo, la direzione di spostamento (Stop, Forward, Back, Left, Right) e la velocità (range 0-100).





Altri oggetti nella scena

Gli altri oggetti nella scena:

- Sole
- Mongolfiera
- Nuvole
- Laghi

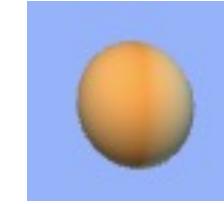




Sole

Costituito da: sfera fissa colorata con una texture.

Posizione: aggiornamento lungo X e Z solidale allo spostamento della camera per dare l'impressione che non si muova. Inoltre abbiamo posizionato una luce puntuale in corrispondenza del sole e luce ambientale diffusa nella scena.



Mongolfiera

Costituita da: oggetto 3D .obj colorato di rosso.

Posizione: aggiornamento inverso allo spostamento della camera lungo X e Z, e spostamento costante lungo Y invertendo la direzione al raggiungimento di un'altezza massima o minima.

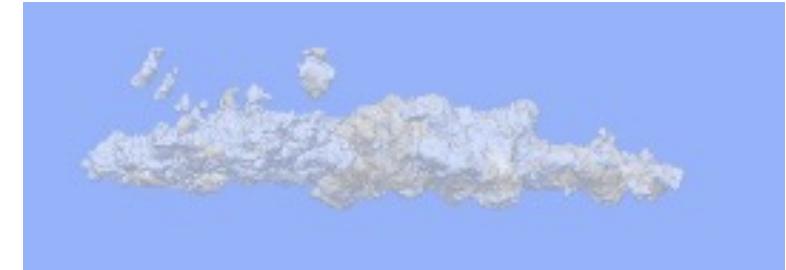




Nuvole

Costituite da: 5 oggetti 3D .obj colorati di grigio.

Posizione: random lungo X, aggiornamento inverso allo spostamento della camera lungo X e Z. Se escono fuori dal Far o dal Near plane della camera, vengono riposizionate con diversa rotazione e posizione, dal lato opposto a quello di uscita.





Laghi

Per creare i laghi abbiamo intersecato alla mesh del terreno un piano XZ posizionato al livello Y = 0.

Per creare un effetto più realistico di quello ottenuto applicando un colore abbiamo utilizzato uno shader per creare un pattern di Voronoi.

P5.js permette, dopo aver importato lo shader (.vert, .frag) nel preload, di passare i valori uniformi necessari usando l'apposita funzione.





.vert

Il vertex shader è molto semplice e attribuisce alla variabile `gl_Position` la posizione del vertice con l'aggiunta della quarta componente omogenea.

```
// our vertex data
attribute vec3 aPosition;

void main() {
    // copy the position data into a vec4, using 1.0 as the w component
    vec4 positionVec4 = vec4(aPosition, 1.0);

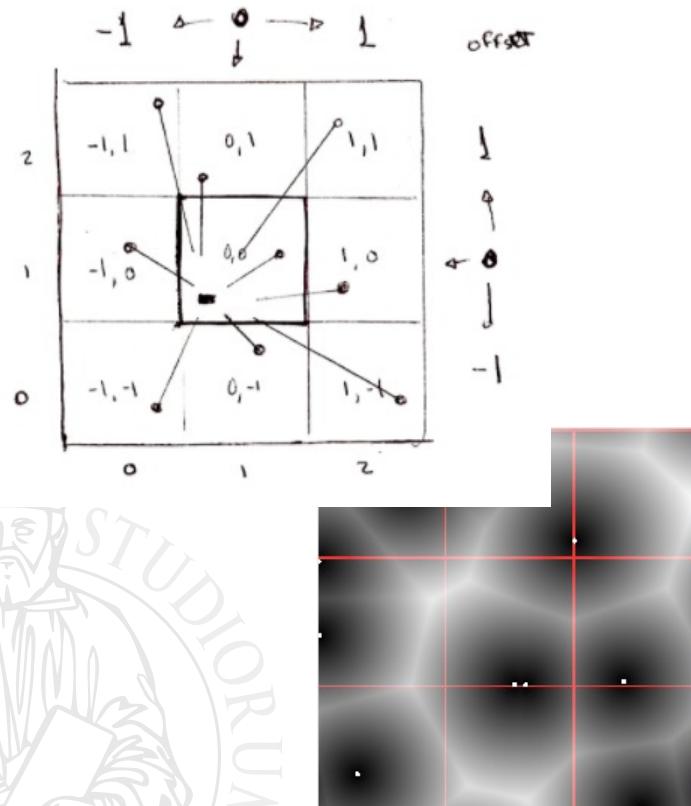
    // send the vertex information on to the fragment shader
    gl_Position = positionVec4;
}
```



.frag

Il fragment shader permette visualizzare il pixel con la sua posizione, il colore ed eventuali altre informazioni, sullo schermo.

Abbiamo scelto di determinare il colore dei vertici usando l'algoritmo di Voronoi, che lo assegna in base alla posizione del punto più vicino.



In questo algoritmo il dominio viene partizionato in una griglia, viene determinata la cella in cui si trova il vertice corrente x , viene scansionato l'insieme di 3×3 celle centrate in esso, viene generato un punto pseudo-casuale in ciascuna di queste 9 celle, e viene registrata la distanza da x al punto più vicino.



.frag – voronoi()

Una volta estratte le parti intere e frazionarie del vertice e quindi identificata la cella in cui stiamo lavorando, ci interessa solo cosa succede attorno a questa cella.

```
float voronoi(vec2 p) {  
    vec2 n = floor(p);  
    vec2 f = fract(p);  
    float md = 5.0;  
    vec2 m = vec2(0.0);  
    for (int i = -1;i<=1;i++) {  
        for (int j = -1;j<=1;j++) {  
            vec2 g = vec2(i, j);  
            vec2 o = hash2(n+g);  
            o = 0.5+0.5*sin(iTime+5.038*o);  
            vec2 r = g + o - f;  
            float d = dot(r, r);  
            if (d<md) {  
                md = d;  
                m = n+g+o;  
            }  
        }  
    }  
    return md;  
}
```

Più in dettaglio:

Dato un vertice. Itera sulle celle adiacenti e:

1. Prende la posizione della cella nella griglia
2. Calcola l'offset del vertice rispetto alla griglia (animandolo in base al tempo)
3. Calcola la distanza tra la cella + offset (vertice associato al sito di voronoi) e vertice dato
4. Seleziona la distanza minore tra quelle calcolate



.frag – main()

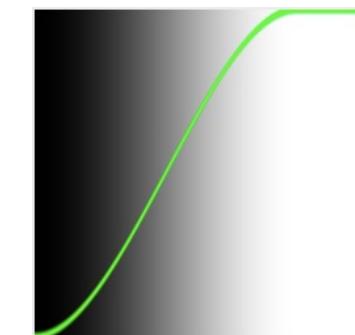
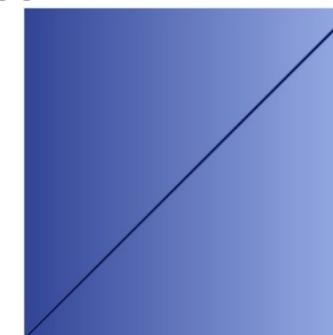
Per determinare la variabile `gl_FragColor` abbiamo utilizzato la funzione `mix()` che esegue un'interpolazione lineare pesata con il valore ritornato dalla funzione `smoothstep()` tra i due colori scelti ($a \times (1 - c) + b \times c$).

La funzione `smoothstep()` esegue l'interpolazione con il polinomio di Hermite se il valore ritornato dalla funzione di voronoi (ovvero la distanza minore) è compresa tra 0 e 0.8, altrimenti ritorna 0/1 se è minore di 0 o maggiore di 0.8.

```
uniform vec3 iResolution; // viewport resolution (in pixels)
uniform float iTime; // shader playback time (in seconds)

void main()
{
    vec2 uv = gl_FragCoord.xy / iResolution.xy;

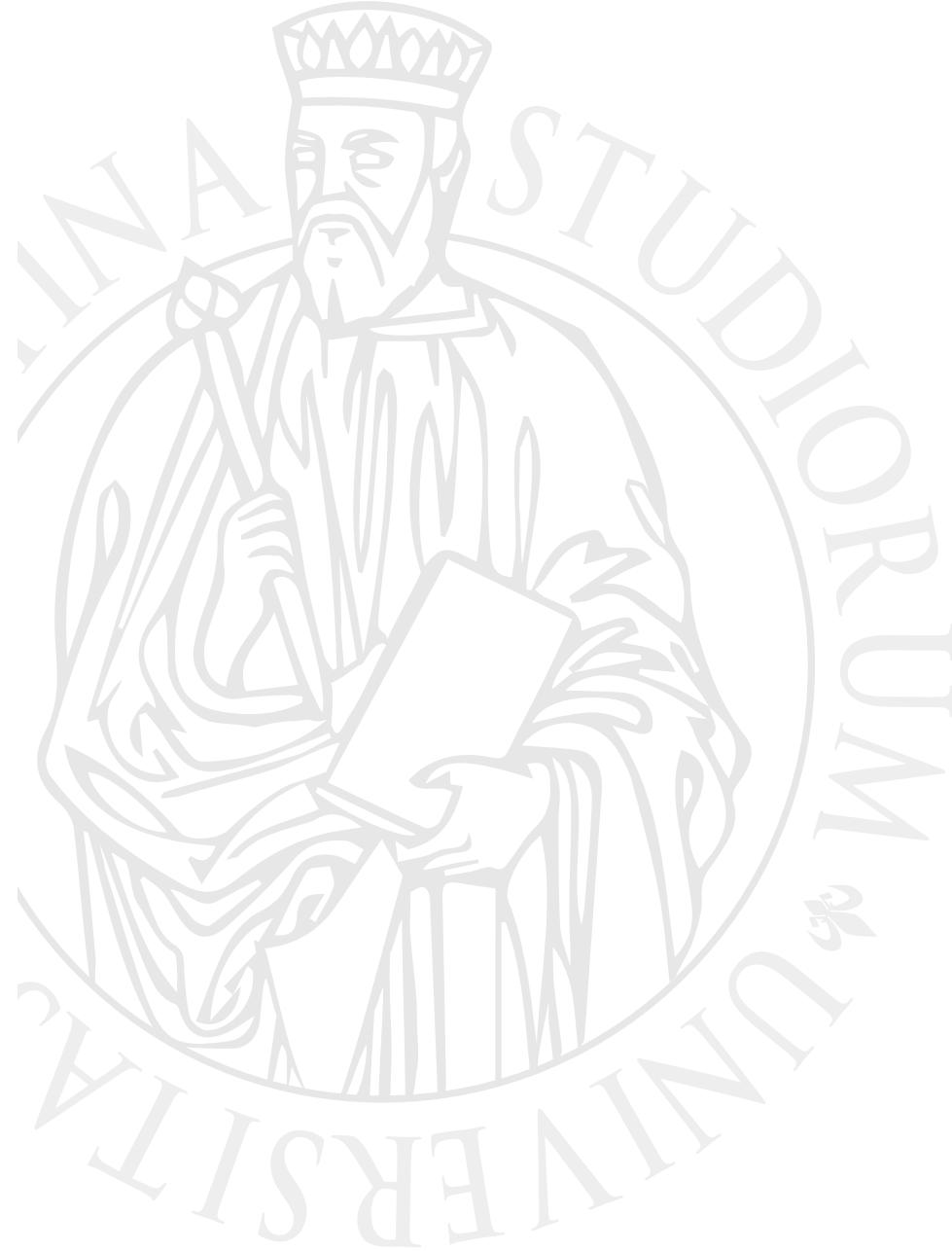
    vec4 a = vec4(0.196, 0.270, 0.588, 0.9); //50, 69, 150 blu
    vec4 b = vec4(0.596, 0.682, 0.901, 0.5); // 152, 174, 230 azzurrino
    gl_FragColor = vec4(mix(a, b, smoothstep(0.0, 0.8, voronoi(uv*3.0)))));
}
```







UNIVERSITÀ
DEGLI STUDI
FIRENZE



Grazie
per l'attenzione

Serena Giachetti,
Lisa Cresti