

Algoritmo di Marching Squares

Mesh procedurali e l'algoritmo di Marching Squares

Raffo Luca



Astrazione

Questa ricerca è realizzata con lo scopo di analizzare gli algoritmi di generazione procedurale in ambito videoludico, nello specifico nell'applicazione dell'algoritmo di marching squares al fine di generare mesh procedurali tramite una demo altamente configurabile, comparando alla fine le prestazioni tra CPU e GPU.

Dedicato a chiunque ne abbia bisogno.

Indice dei contenuti

1. Introduzione
 - 1.1. Procedural Content Generation
 - 1.2. Cos'è una mesh poligonale
 - 1.3. Vertici e triangoli
 - 1.4. Mesh in Unity
2. Marching Squares
 - 2.1. Scopo
 - 2.2. Algoritmo
 - 2.3. Interpolazione dei vertici
 - 2.4. UV mapping
3. Ottimizzazione da Unity
4. Progetto Demo
 - 4.1. Scopo della demo
 - 4.2. Settaggio e Menu
 - 4.3. Debug
5. Conclusioni

Sitografia e References

1. Introduzione

Negli ultimi decenni abbiamo assistito ad una svolta tecnologica non indifferente e ad un incremento del calcolo computazionale che ha permesso ai videogiochi di essere sempre più graficamente appaganti e complessi.

La richiesta sempre maggiore di contenuti, data da una più alta disponibilità di risorse hardware, ha aperto le porte a diverse tecniche di generazione di contenuti non manuali.

In un videogioco, quando si parla di contenuti, ci si riferisce principalmente a tutto ciò che può avere una base creativa, tra cui i primitivi: textures, mesh, suoni, animazioni e testi.

Essi possono essere combinati per realizzare personaggi, livelli, storia... papere?

L'arte può essere generata... in qualche modo.

1.1. Procedural Content Generation

La **Procedural Content Generation** è un qualsiasi metodo che permette la creazione dei contenuti in maniera algoritmica, utilizzando un set di regole predefinite. Una volta stabilite l'algoritmo (le regole) e quali dati accetta, è possibile creare uno spazio di possibilità definito dalla combinazione dei diversi dati.



Figura 1: Uno screenshot del videogioco Minecraft, dove tutta la mappa di gioco viene generata proceduralmente.

1.1.1 - Vantaggi e Svantaggi della Generazione Procedurale

La generazione procedurale può portare varietà all'interno di un gioco, ma ci sono dei punti da prendere in considerazione.

Sicuramente la generazione procedurale ha dei vantaggi:

- Permette di **risparmiare tempo e soldi** agli artisti creando una grande mole di contenuti in poco tempo.
- Permette di **risparmiare spazio sul disco rigido** occupato dal gioco (Si prenda per esempio, No Man's Sky attualmente pesa circa 10GB, alle origini anche appena 1GB), in quanto il contenuto non è immagazzinato nella memoria fisica ma creato a runtime e scritto in RAM.
- Permette di creare runtime dei **contenuti unici e quasi irripetibili**.
- **Aumenta la rigiocabilità** del gioco, in quanto un giocatore non vedrà sempre le stesse cose all'interno del gameplay.

Però non è tutto rose e fiori. La parte più difficile sta proprio nella creazione e scrittura dell'algoritmo. I computer non sono creativi, semplicemente fanno quello che diciamo loro. Questo porta a diversi **svantaggi**:

- Puoi perdere **il parziale controllo diretto sul gioco**. Questo cambia molte cose se si tratta di un elemento di gameplay e non solo grafico.
- Può essere molto **dispendioso a livello di hardware**.
- Se fatto male **può risultare ripetitivo** o creare qualcosa di non usabile.

C'è da dire una cosa, però, su questi svantaggi.

L'hardware ormai non è quasi più un problema, almeno per quanto riguarda i dispositivi fissi. Anche gli smartphone moderni ormai supportano tecnologie GPU sempre più avanzate e sempre più dispositivi riescono a reggere complessi algoritmi di generazione procedurale se scritti bene.

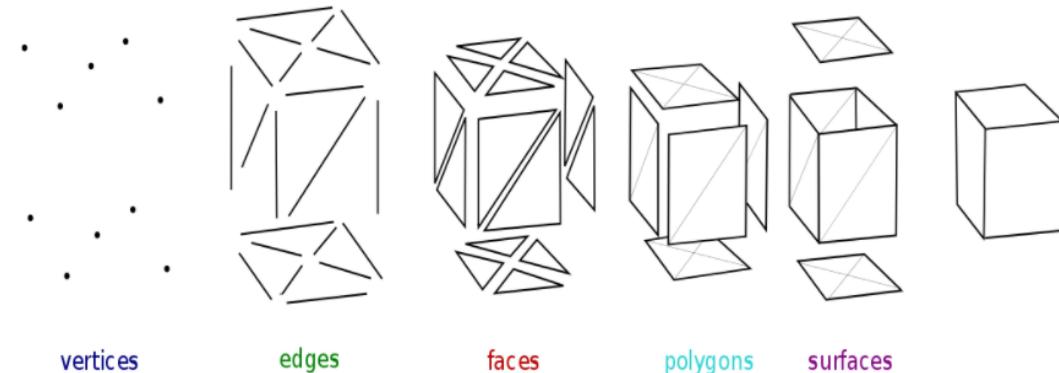
Inoltre, il lavoro sull'algoritmo deve essere fatto bene a prescindere perché si sostituisce un lavoro umano, di qualità superiore, con quello di una macchina.

Se realizzata bene, la generazione di contenuto procedurale può dare molto ad un videogioco, senza tutti gli svantaggi elencati.

1.2. Mesh

Una mesh è una maglia di vertici, spigoli e facce che definiscono come un oggetto debba essere renderizzato in uno spazio 3D.

Ogni vertice di una mesh contribuisce alla creazione dei triangoli e quad che compongono le facce.



1.3. Quad, Triangoli e Vertici

La geometria di una mesh è semplice, ovvero sono composte da tanti piccoli triangoli interconnessi.

Ogni vertice possiede diverse informazioni, oltre alla posizione:

- Colore
- UV
- Normali
- Light map
- etc....

Ogni triangolo è composto da 3 vertici ed è la forma base utilizzata nella grafica 3D.

Questo per vari motivi logici e geometrici:

- 1) E' il **numero minimo di punti** per poter creare una forma.
- 2) E' **facilmente renderizzabile** perché crea sempre una superficie piatta e senza curve.

I **Quad** sono composti da 4 vertici, è permettono di ottimizzare la geometria di una mesh, riducendo il numero di vertici e triangoli. Essi però vanno usati con cautela perché possono creare delle forme curve, risultando in problemi di qualità della mesh.

2 - Marching Squares

L'algoritmo di [marching square](#), tradotto "Quadrati marcianti", è un algoritmo per il rendering di [linee di contorno](#) (o "di livello") a partire da un campo scalare o una nuvola di punti 2D.

E' una variante 2D dell'algoritmo di Marching Cubes.

2.1. Scopo

Lo scopo dell'algoritmo è principalmente la **triangolazione** di una Mesh in maniera efficiente a partire da un campo scalare 2D.

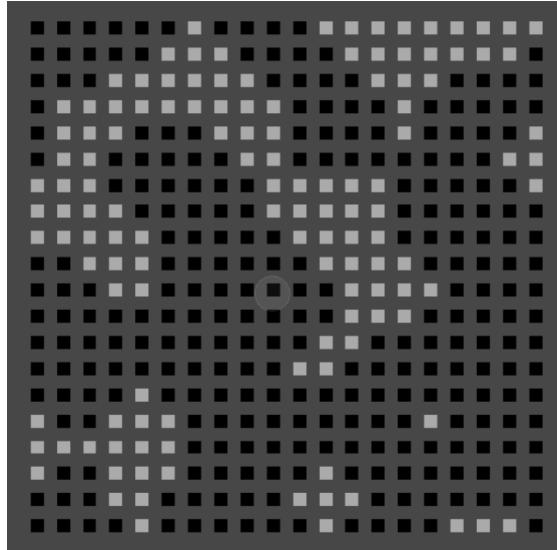
2.2. Algoritmo

2.2.1. Stabilire i punti

Ogni punto, a prescindere dall'algoritmo di generazione della griglia, dovrà conservare un numero che si chiama **Iso-valore**.

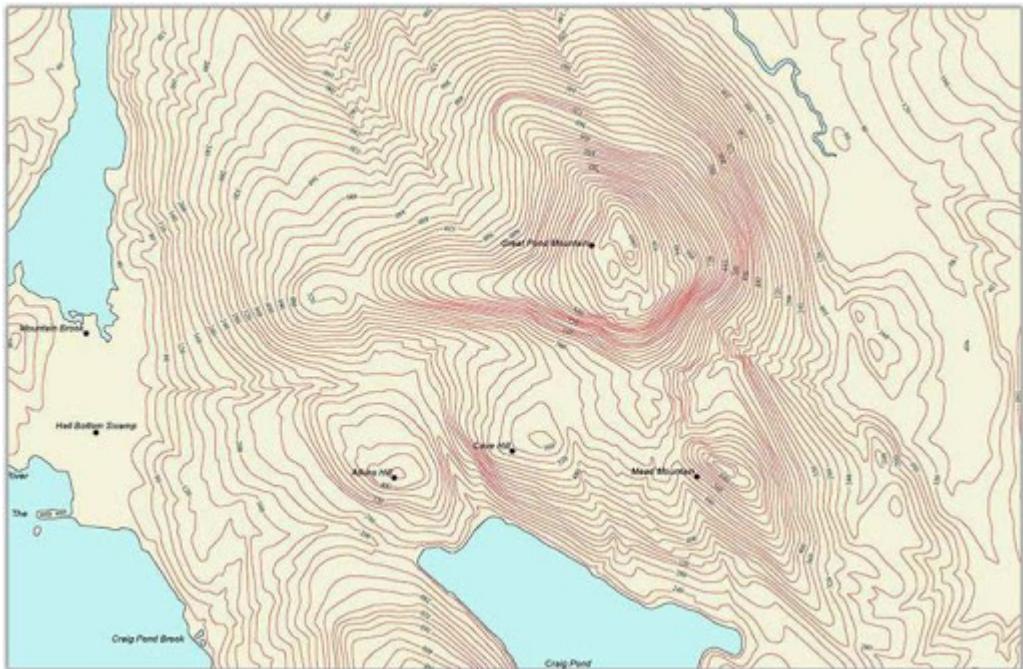
L'algoritmo utilizza questo valore per capire quali punti sono considerati dall'algoritmo.

Il valore di un punto può essere stabilito in 2 diversi modi:



- 1) Mediante un algoritmo di generazione di valori, come un **Random.Range()** oppure un algoritmo che generi dei valori omogenei come il Perlin Noise.
L'attuale implementazione genera un campo scalare di valori da un Perlin Noise.

- 2) Letto da un'immagine, in questo caso elaborata, in modo da mappare e creare mappe di densità. Esempio:



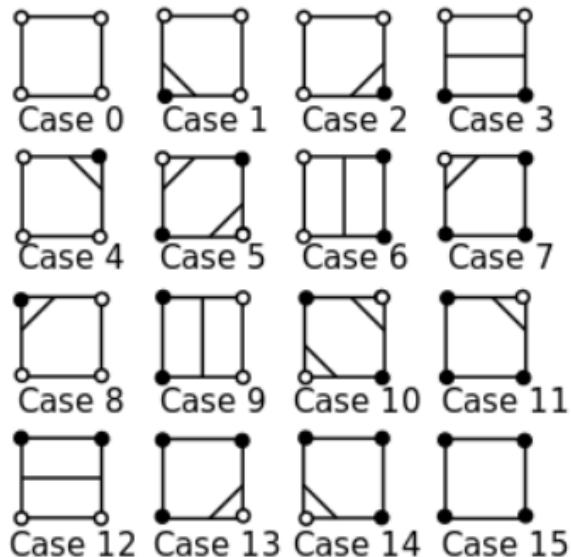
2.2.2. Triangolazione - Teoria

L'algoritmo scorre con una cella virtuale ogni cella della griglia, intersecando sempre 4 vertici.

Leggendo l'**iso-valore** dal vertice viene stabilito tramite una soglia, che chiameremo **iso-livello** (O threshold), quali vertici devono essere considerati dall'algoritmo.

Ogni configurazione determina quali triangoli devono essere creati, ritornando eventualmente una lista di punti.

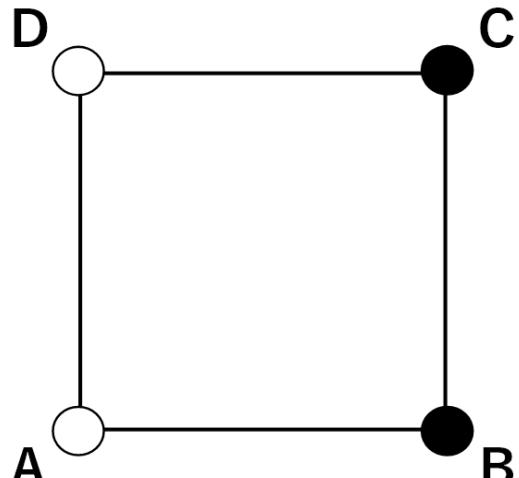
E' possibile capire, conoscendo il numero del caso, quali triangoli creare tramite una tabella di triangolazione.



Esempio di accesso all'indice di triangolazione:

Il cubetto si ferma in una configurazione 1001 (**A****B****C****D**), dove A e D sono intersecati, in tal caso avremo la configurazione numero 9 (8 + 1, in binario 1000 + 0001):

$$1000 = 2^3 = 8$$



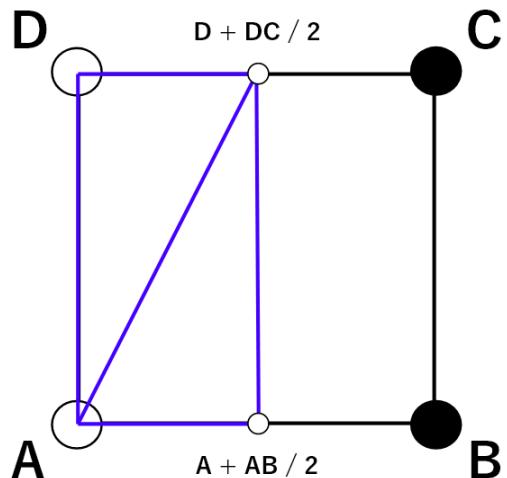
$$0001 = 2^0 = 1$$

La configurazione 9 ci dice di creare due triangoli:

$$\{ A, D, D + DC/2 \},$$

$$\{ A, D + DC/2, A + AB / 2 \}.$$

$$1000 = 2^3 = 8$$



$$0001 = 2^0 = 1$$

2.2.3. Triangolazione - Implementazione

Creando due liste, una per i vertici e una per i triangoli:

```
List<int> triangles;
List<Vector3> vertices;
```

E' possibile usare l'algoritmo di marching squares su una griglia di punti, scorrendo 4 punti alla volta. Per ogni quattro punti è possibile stabilire una configurazione per l'algoritmo in modo che sappia quale triangoli generare e in quale posizione.

Si può stabilire la configurazione utilizzando questa operazione binaria.

```
// Triangulation table
int cellType = 0; // Cell type may vary from 0 to 15

if (a.isoValue > isoLevel) {
    cellType |= 1;
}
if (b.isoValue > isoLevel) {
    cellType |= 2;
}
if (c.isoValue > isoLevel) {
    cellType |= 4;
}
if (d.isoValue > isoLevel) {
    cellType |= 8;
}
```

Il risultato sarà un numero da 0 a 15; Questo è il risultato della combinazione dei punti che superano la soglia di **iso-valore** stabilito dall'algoritmo di generazione del campo scalare di punti.

Una volta stabilito il codice della configurazione, si procede alla creazione del triangolo.

PS: Right, Left, Top e Bottom si riferiscono alle coordinate corrispondenti ai punti di mezzo mostrati nelle immagini nella parte della triangolazione.

```

//      top
//      C-----D
//  left |       | right
//      |       |
//      A-----B
//      bottom

switch (cellType)
{
    case 0:
        return;
    case 1:
        AddTriangle(a.position, left, bottom);
        break;
    case 2:
        AddTriangle(b.position, bottom, right);
        break;
    case 4:
        AddTriangle(c.position, top, left);
        break;
    ....
}

private void AddTriangle(Vector3 a, Vector3 b, Vector3 c)
{
    // Get the current point of the array without losing the index.
    int vertexIndex = vertices.Count;
    vertices.Add(a);
    vertices.Add(b);
    vertices.Add(c);

    triangles.Add(vertexIndex);
    triangles.Add(vertexIndex + 1);
    triangles.Add(vertexIndex + 2);
}

```

Una volta triangolate tutte le celle, è possibile disegnare la mesh, passando le strutture dati ai componenti MeshFilter.

```

// Looping all the voxels of the grid
for (int y = 0; y < cells; y++, voxelIndex++) {
    for (int x = 0; x < cells; x++, voxelIndex++) {
        TriangulateVoxel(
            voxels[voxelIndex],
            voxels[voxelIndex + 1],
            voxels[voxelIndex + chunkResolution],
            voxels[voxelIndex + chunkResolution + 1]);
    }
}

// Apply vertices and triangles to the mesh on mesh filter
meshFilter.sharedMesh.SetVertices(vertices);
meshFilter.sharedMesh.SetTriangles(triangles, 0);

```

Il risultato ottenuto sarà il seguente:

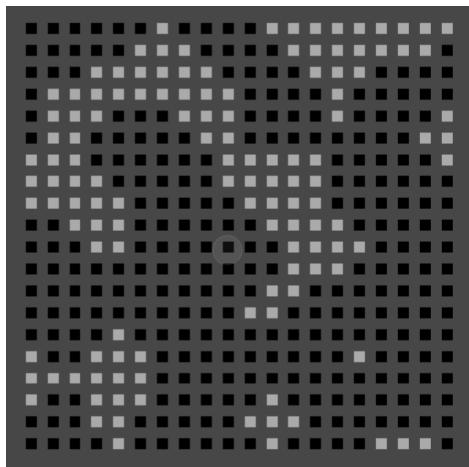


Figura 2: Una griglia di punti scalari ottenuta tramite algoritmo di Perlin Noise.



Figura 3: Mesh generata dalla Triangolazione

2.3. Interpolazione dei vertici e edge smoothing

Il risultato ottenuto dall'algoritmo è ottimizzabile. Aumentando la risoluzione della griglia dei punti, in modo tale da aumentare la risoluzione percepita, otteniamo un effetto problematico.

I punti sono vincolati alla griglia, perciò darà l'effetto cubettoso finché non si raggiunge un numero punti pari al numero di pixel dello schermo.

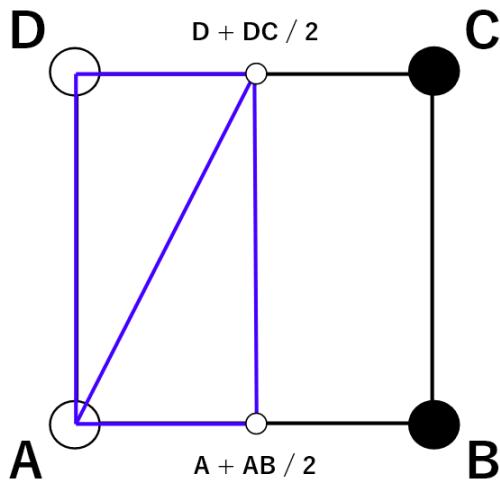
Esiste fortunatamente un modo per approssimare la posizione di un vertice in base al suo **iso-valore**, in modo da dare un peso diverso ad ogni punto.

L'interpolazione dei vertici può aiutarci a risolvere il problema: muovendo i vertici sui bordi in base agli iso-valori di ogni singolo punto, è possibile rendere la forma più armoniosa e dare peso al valore del singolo punto.

```
// Instead of bottom you lerp between A and B to get the position.  
// Instead of right you lerp between D and B, etc  
//  
//      top  
//      C-----D  
// left  |      |  right  
//      |      |  
//      A-----B  
//      bottom  
  
float t_top = (noiseGenerator.isoLevel - c.value) / (d.value - c.value);  
Vector2 top = Vector2.Lerp(c.position, d.position, t_top);  
  
float t_right = (noiseGenerator.isoLevel - d.value) / (b.value - d.value);  
Vector2 right = Vector2.Lerp(d.position, b.position, t_right);  
  
float t_bottom = (noiseGenerator.isoLevel - b.value) / (a.value - b.value);  
Vector2 bottom = Vector2.Lerp(b.position, a.position, t_bottom);  
  
float t_left = (noiseGenerator.isoLevel - a.value) / (c.value - a.value);  
Vector2 left = Vector2.Lerp(a.position, c.position, t_left);
```

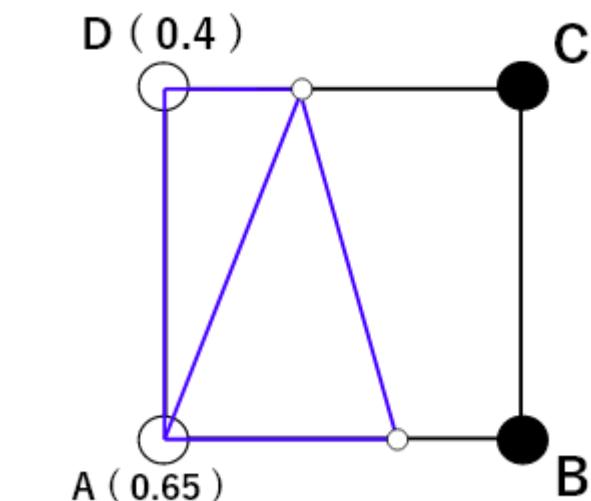
In questo modo, al posto di calcolare i punti sui bordi come nell'esempio precedente:

$$1000 = 2^3 = 8$$



$$0001 = 2^0 = 1$$

$$\text{Lerp}(D, C, \text{iso} - c.\text{val} / d.\text{val} - c.\text{val})$$



$$\text{Lerp}(A, B, \text{iso} - b.\text{val} / a.\text{val} - b.\text{val})$$

Adesso i punti sui lati sono calcolati in modo da prendere in considerazione l'iso-valore dei punti adiacenti, interpolando il vertice sul bordo.

Ottenendo come risultato qualcosa di più armonioso:



2.4. UV Mapping

Il nome UV è un equivalente di X e Y. La mappa usa i vertici inseriti all'interno dell'array dei vertici in modo da appiattire i triangoli e applicare una texture sopra di essi in maniera corretta e omogenea.

La mappa UV ha bisogno dei vertici del triangolo e delle loro coordinate spaziali.

Quando viene aggiunto un triangolo, ne viene calcolata anche l'UV:

```
private void AddTriangle(Vector3 a, Vector3 b, Vector3 c) {
    int vertexIndex = vertices.Count;
    vertices.Add(a);
    vertices.Add(b);
    vertices.Add(c);

    // Add uvs
    uvs.Add(Vector2.right * vertices[vertexIndex].x + Vector2.up *
vertices[vertexIndex].y);

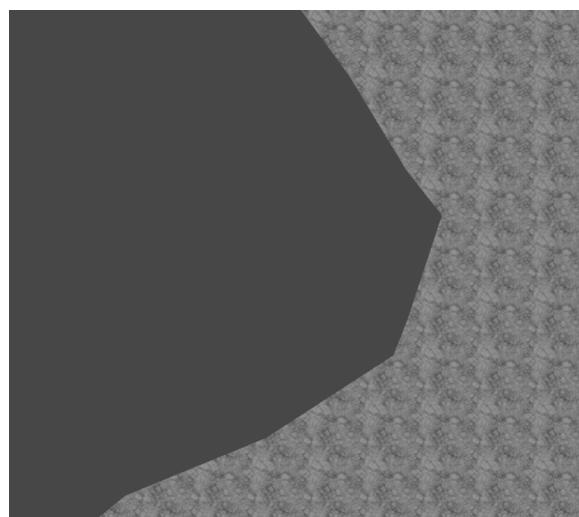
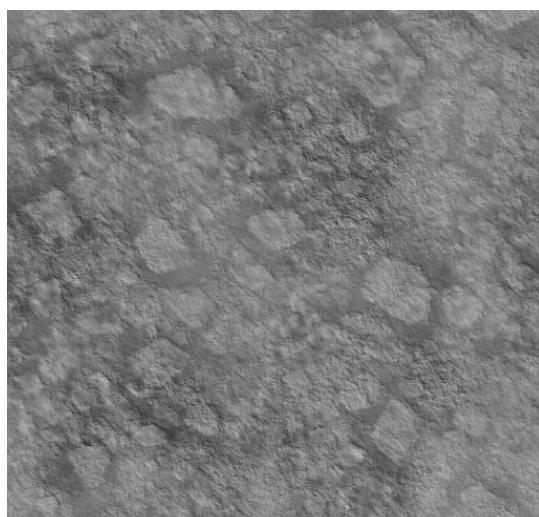
    uvs.Add(Vector2.right * vertices[vertexIndex + 1].x + Vector2.up *
vertices[vertexIndex + 1].y);

    uvs.Add(Vector2.right * vertices[vertexIndex + 2].x + Vector2.up *
vertices[vertexIndex + 2].y);

    triangles.Add(vertexIndex);
    triangles.Add(vertexIndex + 1);
    triangles.Add(vertexIndex + 2);
}

...

mesh.SetVertices(vertices);
mesh.SetUVs(0, uvs);
mesh.SetTriangles(triangles, 0);
```



3. Ottimizzazione da Unity

Di seguito, una serie di ottimizzazioni utili per Unity se si vuole disegnare una mesh procedurale.

3.1. Mesh MarkDynamic

E' possibile richiamare sull'oggetto mesh, prima di inserire i vertici al suo interno con `mesh.SetVertices()`, l'istruzione `mesh.MarkDynamic()`. In questo modo la mesh crea internamente un buffer dinamico per i vertici in modo che possano essere gestiti in maniera più efficiente. La documentazione di Unity è un'po scarsa a riguardo:

<https://docs.unity3d.com/ScriptReference/Mesh.MarkDynamic.html>

3.2. IndexFormat.UInt32

Dal momento in cui gli indici dei buffer sono a 16 bit di default, una mesh supporta fino a 65535 vertici. Per aumentare il numero di vertici massimo, bisogna cambiare il tipo all'indice, da `UInt16` a `UInt32`, in modo che la mesh possa supportare un numero di vertici fino a 4 miliardi.

<https://docs.unity3d.com/ScriptReference/Mesh-indexFormat.html>

3.3. Graphics DrawMesh e MeshFilter/MeshRenderer

Il sistema a componenti di Unity appesantisce non poco la gestione delle mesh e della loro visualizzazione. Per una massima efficienza è necessario passare direttamente alle funzioni che operano sotto `mesh.filter`. Una chiamata al metodo statico `Graphics.DrawMesh()`. ci permette di annullare tutto l'overhead del sistema a componenti di Unity e dare direttamente la mesh da disegnare in pasto alla GPU.

<https://docs.unity3d.com/ScriptReference/Graphics.DrawMesh.html>

3.4. Compute Shader

A scopo di ricerca, è stato realizzato un Compute Shader per tutto il carico di lavoro dovuto alla triangolazione dei vertici e il calcolo delle UVs da parte dell'algoritmo.

Un compute shader è un tipo di shader dedicato quasi unicamente alla computazione di informazioni arbitrarie, dove tutto il calcolo viene effettuato in parallelo sulla GPU. Raramente viene utilizzato per disegnare, in quanto non è stato progettato per quello scopo.

4. Progetto Demo



E' possibile scaricare la demo dal mio profilo github, [Marching Squares](#), e scaricare l'ultima release. Dentro lo zip ci sarà l'eseguibile.

4.1. Scopo

Lo scopo della ricerca è la creazione di una demo altamente configurabile e testabile dell'algoritmo di Marching Squares. E' inoltre stato realizzato per comparare le prestazioni dell'algoritmo sia sulla CPU che sulla GPU.

Comandi e interfaccia

L'interfaccia si presenta con due macrodivisioni. A sinistra si trova la griglia sulla quale è applicato l'algoritmo di marching squares e a destra si trova la parte di UI che ci permette di comandare l'algoritmo, insieme a una barra di scorrimento per muoversi tra i diversi menu.

E' inoltre possibile spostare l'offset di generazione del noise, usato dall'algoritmo, utilizzando **wasd** o **il dpad**.

4. Conclusioni

Con questa ricerca è stato dimostrato come è possibile creare mesh procedurali, utilizzando l'algoritmo di marching squares.

Durante lo sviluppo del progetto si è spesso incappati nella necessità di rendere ancora più efficiente l'algoritmo, come l'implementazione dell'interpolazione dei vertici, l'istanziamento della mesh via GPU e tantissime altre micro-ottimizzazioni.

E' stata inoltre realizzata la controparte compute shader per comparare l'efficacia tra CPU e GPU.

Sitografia e References

- [Noise Function - https://www.cs.utexas.edu](https://www.cs.utexas.edu)
- [Metaballs and Marching Square](#)
- [Dual Contouring, Marching Cubes and Marching Squares](#)
- [Polygonising a scalar field](#)
- [Sebastian Lague](#)
- [Procedural Content Generation in Games](#)
- [Quad Mesh Optimization](#)