


Relazione Computer Graphics

Michele Saraceno - 1905065

 Progetto scelto: **Sample Elimination**

Sono stati realizzati degli algoritmi che producono un set di punti distanti tra loro in modo omogeneo a partire da punti inseriti in modo casuale.

Gli algoritmi sono stati realizzati in **python** utilizzando le librerie: *numpy*, *opencv*, *math*, *heapq*, *scipy* e *matplotlib*.

I codici sono tutti commentati (*in inglese*).

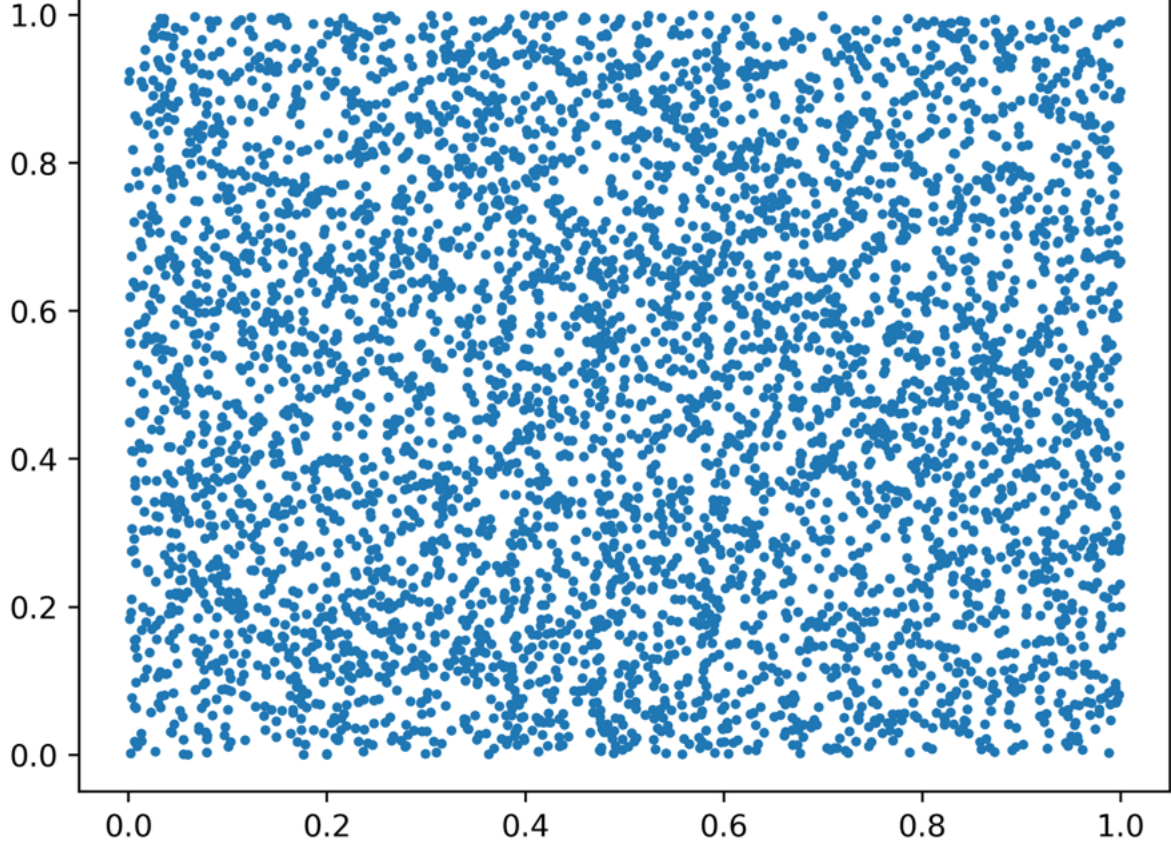
Random sampling

file: `random_sampling.py`

Questo algoritmo posiziona punti su una superficie 2D in modo casuale. Serve principalmente per mostrare le differenze con gli altri algoritmi.

```
n_points = 5000
x = np.random.rand(n_points)
y = np.random.rand(n_points)
```

Generazione di 5000 punti casuali



La distribuzione dei 5000 punti è completamente casuale, infatti sono presenti zone con densità di punti diverse.

Weighted elimination

file: `2D_weighted_elimination.py`

In questo algoritmo vengono utilizzati dei parametri e delle formule

r_M è il raggio massimo di ricerca per i punti vicini

$$r_M = \sqrt{\frac{1 * 2}{2\sqrt{3} * outputSize}}$$

r_m è il raggio minimo

$$\beta = 0.65$$
$$\theta = 1.5$$
$$r_m = r_M * \left(1 - \left(\frac{inputSize}{outputSize}\right)^\beta\right) * \theta$$

il peso di un punto p_1 è la somma dei pesi del punto p_1 rispetto ad ogni punto p_2 intorno ad esso entro $2 * r_M$

$$\alpha = 8$$
$$weight[p_1] = weight[p_1] + \left(1 - \frac{distance(p_1, p_2)}{2 * r_M}\right)^\alpha$$
$$distance(p_1, p_2) = \sqrt{(p_1[x] - p_2[x])^2 + (p_1[y] - p_2[y])^2}$$

Come per il random sampling, vengono generati dei punti casuali.

```
points = np.random.rand(input_size,2)
```

Dopo averlo fatto, passa alla *fase di eliminazione dei punti*; questa fase si divide in:

- inserimento dei punti all'interno di un *albero a due dimensioni (2-d tree)*

```
tree = spatial.cKDTree(points)
```

- assegnamento dei *pesi* ad ogni punto, in base alle distanze dagli altri punti entro un determinato raggio

```
weights = np.zeros(len(points))
for i in range(len(points)):
    indexes = tree.query_ball_point(points[i], 2*r_max)
    for idx in indexes:
        if i != idx:
            distance = calculateDistance(points[i],points[idx],r_max,r_min)
            weights[i] += (1-(distance/(2*r_max)))**alpha
```

- viene creato un *MAX-heap* in cui vengono inseriti i pesi dei punti

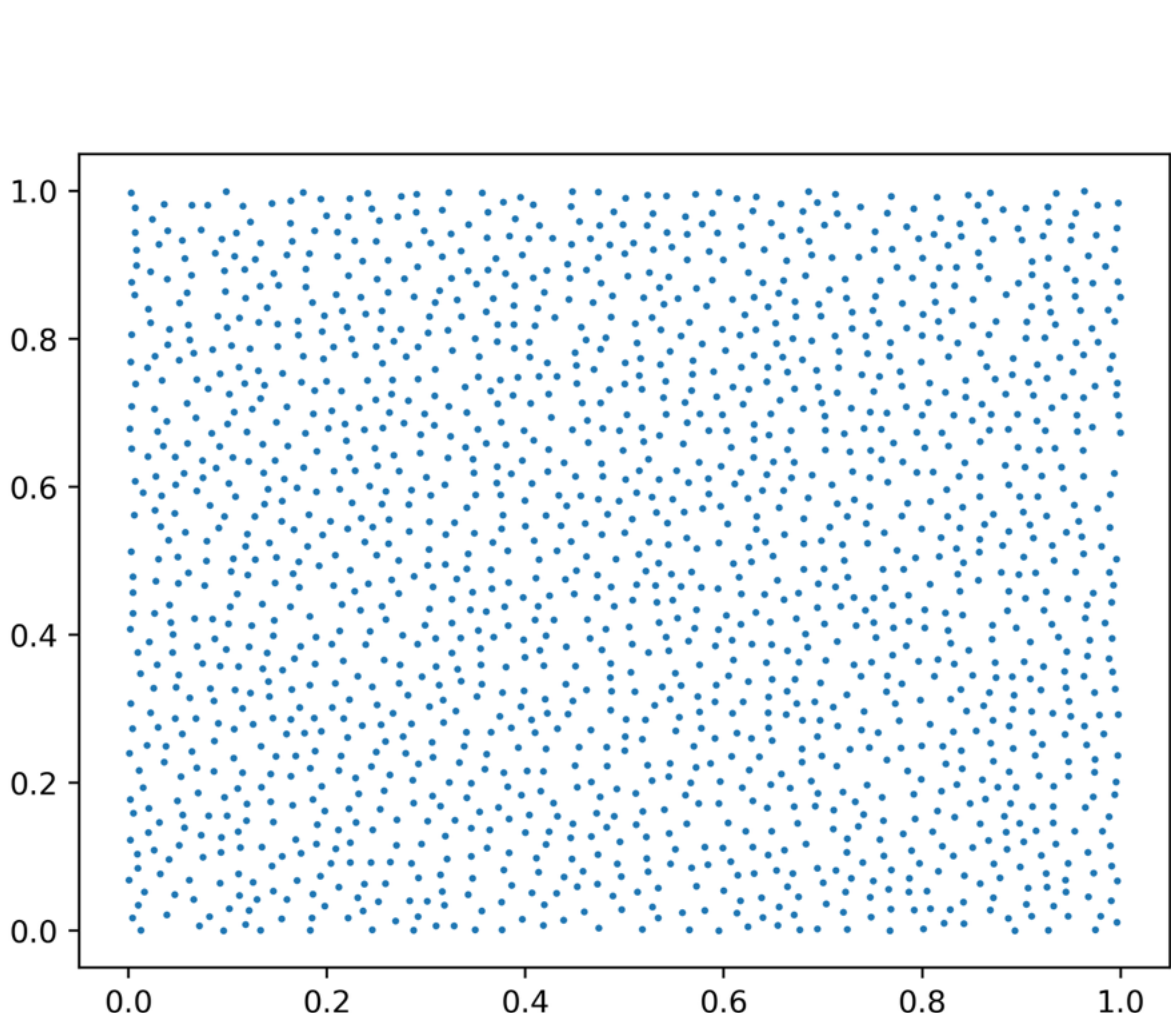
```
heap = generateHeap(weights)
```

```
def generateHeap(weights: np.ndarray) -> list:
    heap = []
    heapq.heapify(heap)
    for i in range(len(weights)):
        if not math.isnan(weights[i]):
            heapq.heappush(heap, (-1*weights[i],i))
    return heap
```

- fin quando la dimensione dell'*heap* è maggiore alla dimensione desiderata del *sub-set*, viene effettuato il *pop* del primo elemento dell'*heap* e il relativo punto viene eliminato dalla lista (per eliminazione si intende che il suo peso e le sue coordinate assumono il valore *np.nan*), decrementando il valore dei pesi dei punti vicini a quello eliminato

```
while len(heap) > output_size:
    pop = heapq.heappop(heap)
    indexes = tree.query_ball_point(points[pop[1]], 2*r_max)
    for idx in indexes:
        if idx != pop[1]:
            distance = calculateDistance(points[pop[1]],points[idx],r_max,r_min)
            app_weight = weights[idx]
            weights[idx] -= (1-(distance/(2*r_max)))**alpha
            if not math.isnan(weights[idx]):
                heap.remove((-1*app_weight,idx))
                heap.append((-1*weights[idx],idx))
    points[pop[1],0] = np.nan
    points[pop[1],1] = np.nan
    weights[pop[1]] = np.nan
    heapq.heapify(heap)
```

La dimensione ottimale del *sub-set* è circa da $\frac{1}{3}$ a $\frac{1}{5}$ della dimensione del *set* di partenza.



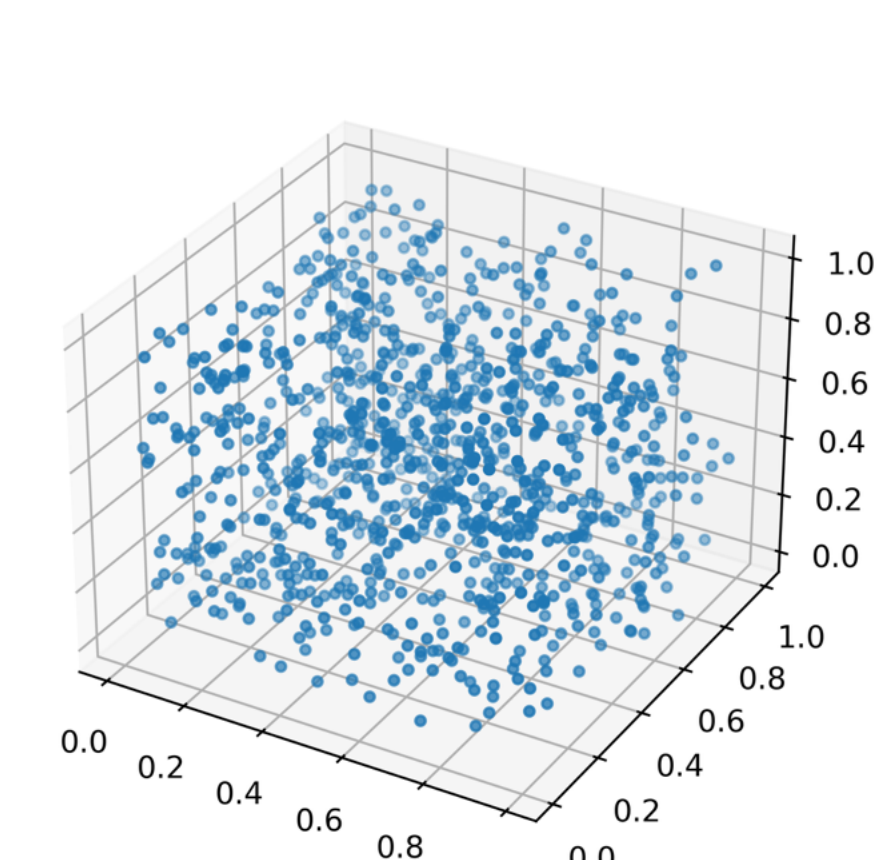
Dai 5000 punti è stato generato un *sub-set* di 2000 punti (1/3 della dimensione iniziale) distribuiti in modo omogeneo sulla superficie.

Lo stesso algoritmo può essere utilizzato anche per spazi 3D, cambiando le formule per calcolare la distanza tra due punti e il raggio massimo adattandole alle tre dimensioni e sostituendo il 2-d tree con un 3-d tree.

$$distance(p_1, p_2) = \sqrt{(p_1[x] - p_2[x])^2 + (p_1[y] - p_2[y])^2 + (p_1[z] - p_2[z])^2}$$

$$r_M = \sqrt[3]{\frac{1^3}{4\sqrt{2} * outputSize}}$$

file: `3D_weighted_elimination.py`



Dai 5000 punti è stato generato un *sub-set* di 1666 punti (1/5 della dimensione iniziale) distribuiti in modo omogeneo nello spazio. Dall'immagine non è facilmente intuibile quindi si consiglia di visualizzarlo dalla finestra di matplotlib che appare all'esecuzione del codice.

Adaptive elimination

file: `adaptive_elimination.py`

L'*adaptive elimination* utilizza come base il codice della *weighted elimination* cambiando il calcolo del peso dei punti. Data un'immagine in input, essa viene trasformata in scala di grigi (utilizzando la libreria *opencv*) in modo tale da ottenere un valore di intensità per ogni pixel che influirà il peso del punto con posizione relativa a quella del pixel. Nelle parti più scure dell'immagine, l'intensità è più bassa e quindi il peso sarà minore mentre sarà maggiore nelle parti più chiare. Così facendo potremo ottenere una densità di punti maggiore nelle zone corrispondenti alle parti scure dell'immagine e una densità minore nelle zone corrispondenti a quelle più chiare.

```
def transformDistanceDensity(distance:float,point:np.ndarray,gray_img:np.ndarray,d_max:float) -> float:
    if math.isnan(point[0]):
        return d_max
    y = int(point[0] * gray_img.shape[0])
    x = int(point[1] * gray_img.shape[1])
    density = gray_img[y,x]
    distance = distance * (3 - 2*(density/255))
    return min(distance,d_max)
```

I seguenti esempi sono tutti da 4000 punti, partendo dai 12000 iniziali:

