Lorenzo Ricci, 1757571

Extra Credit Facili

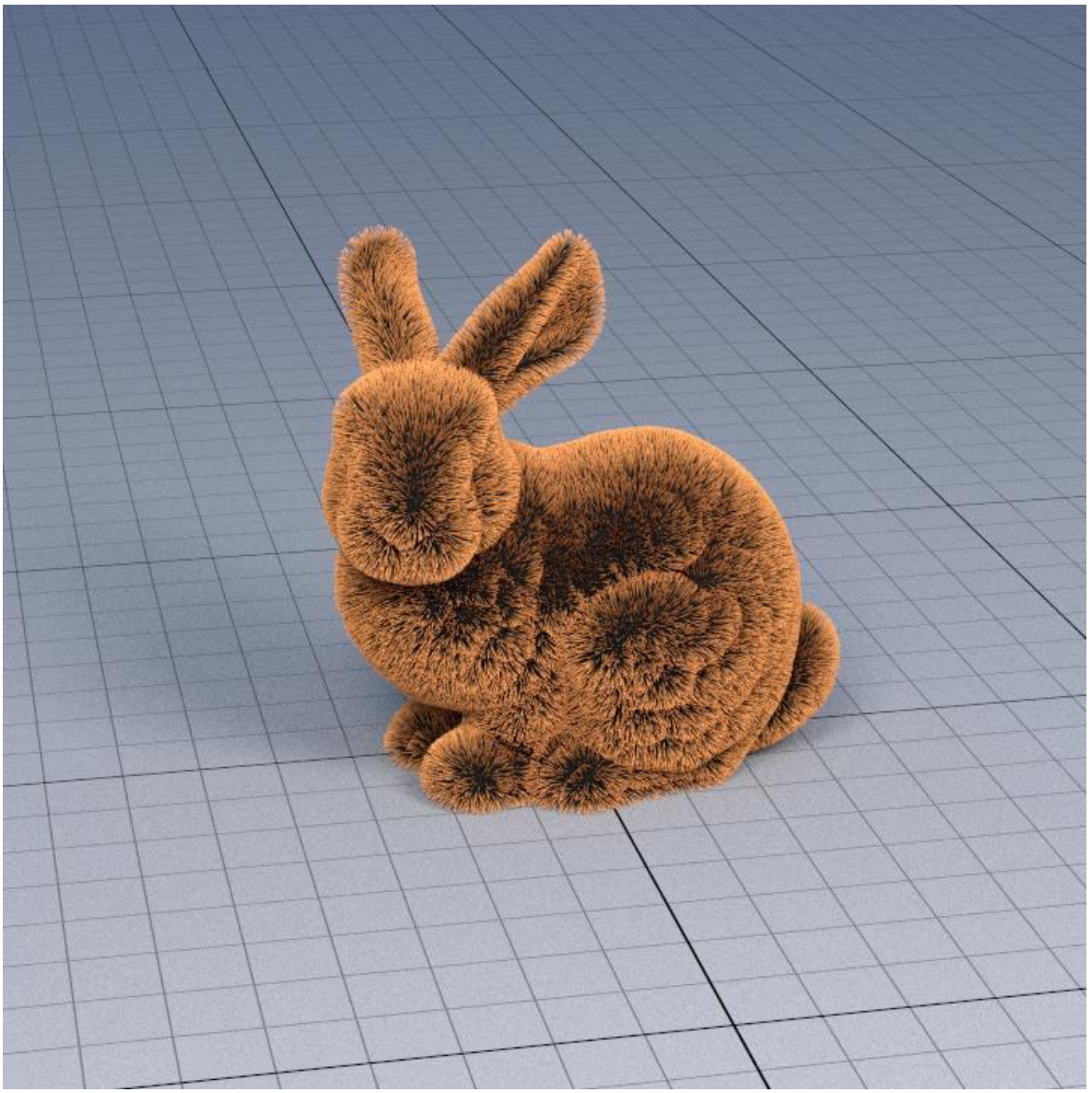
Density Control.

Il concetto di density control è stato applicato alla generazione del prato, effettuando alcune modifiche alle funzioni *make_grass* e *make_hair*. Per effettuare un controllo sulla densità sono stati seguiti questi step:

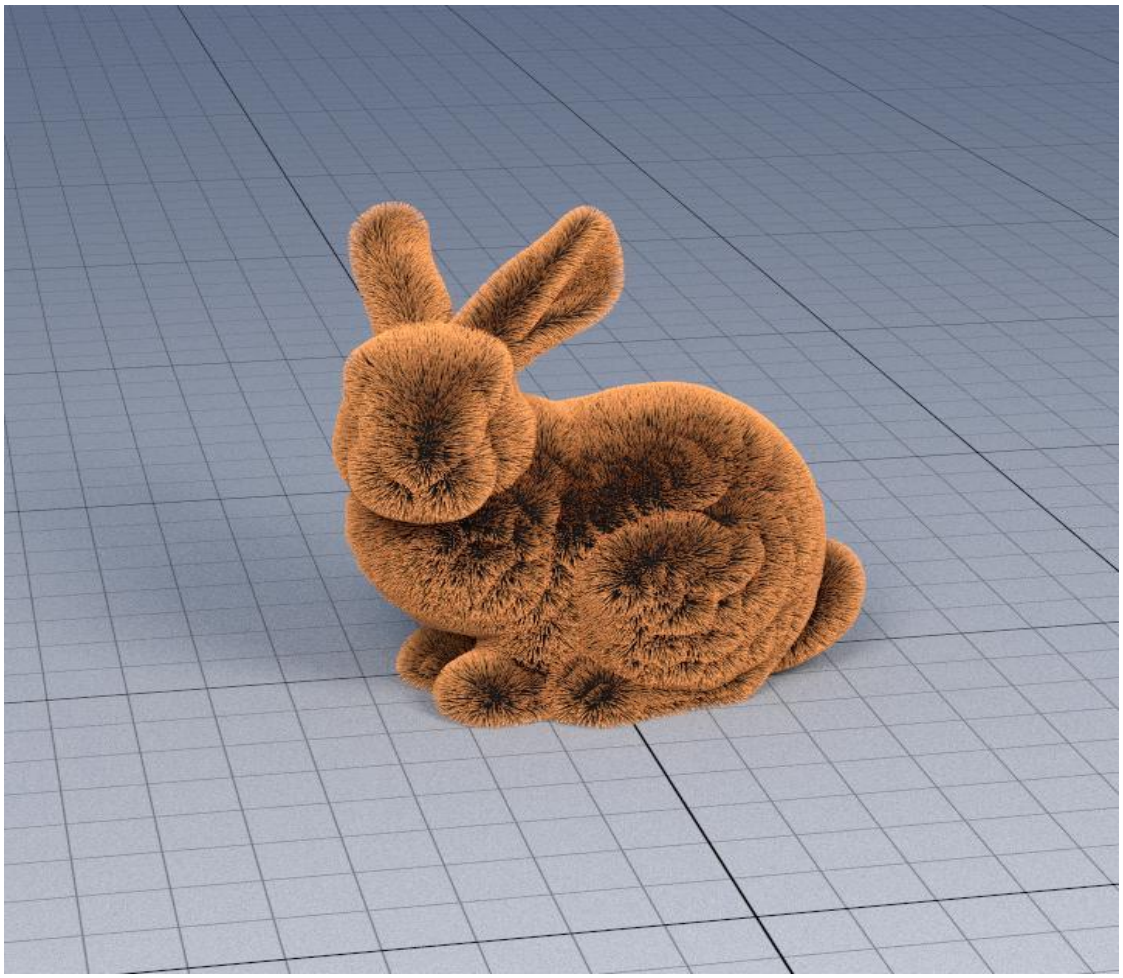
- È stato definito in *hair_params* e *grass_params* un valore float density modificabile da riga di comando attraverso l'opzione *--hairdens* e *--grassdens*
- Il valore density passato da riga di comando viene comparato con un valore random generato con *rand1f*. Pertanto, i valori potranno variare da 0 a 1, dove 0 comporterà una empty shape dato che non verrà generato alcun pelo/filo d'erba, mentre 1 comporterà la generazione del numero *num* totale
- Se il valore random è maggiore della densità allora viene saltata l'iterazione corrente con una istruzione *continue*, altrimenti viene generato il pelo/filo d'erba

Subito sotto riporto diverse immagini di comparazione per grass e hair2. Evito di riportare nel pdf tutte le istanze di hair per non allungare inutilmente questa sezione. Tutte le immagini generate (per ogni hair, con densità 0.2f e 0.7f) sono riportate comunque nella cartella "/out" consegnata.

Hair2.

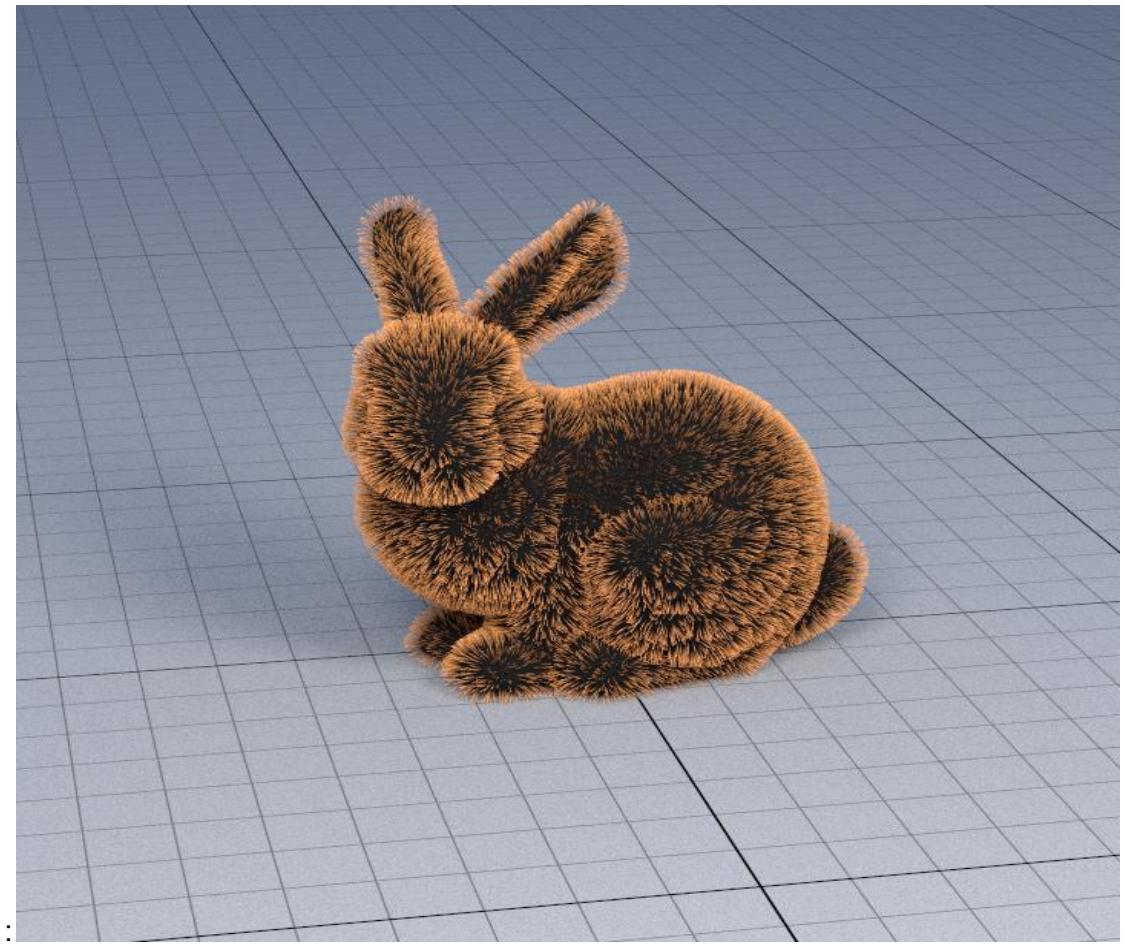


*Hair2 generato con density = 1.0f, ovvero con **num** peli di default; samples = 256, resolution = 720*



Hair2 generato con density = 0.7f; samples = 256, resolution = 720

Ovviamente data la “vicinanza” tra i valori di densità, la differenza c’è ma non è troppo visibile, soprattutto dato l’elevato numero di peli usato per generare questo *bunny*. Compariamo quindi con un valore molto basso:



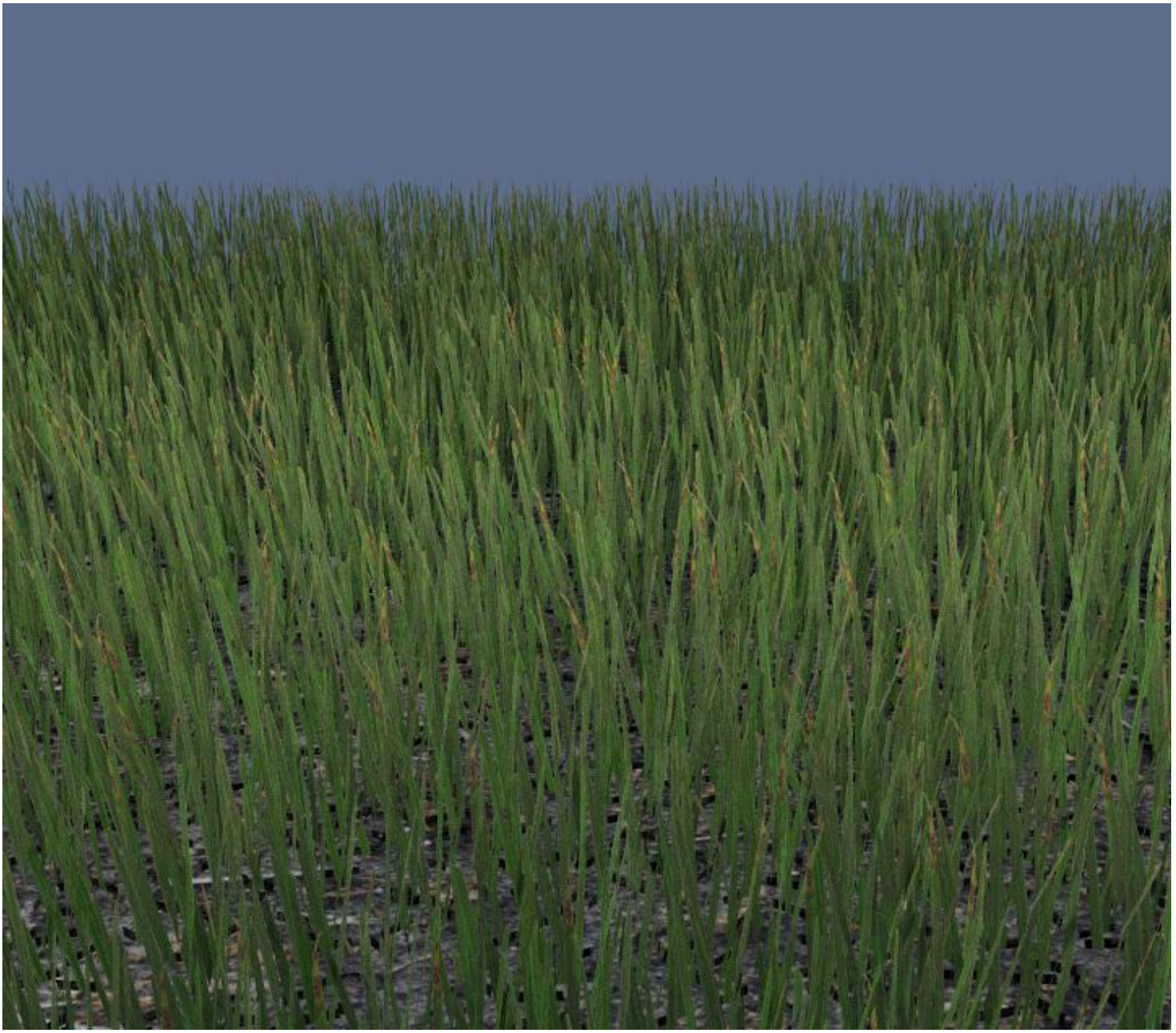
:

Hair2 generato con density = 0.2f; samples = 256, resolution = 720

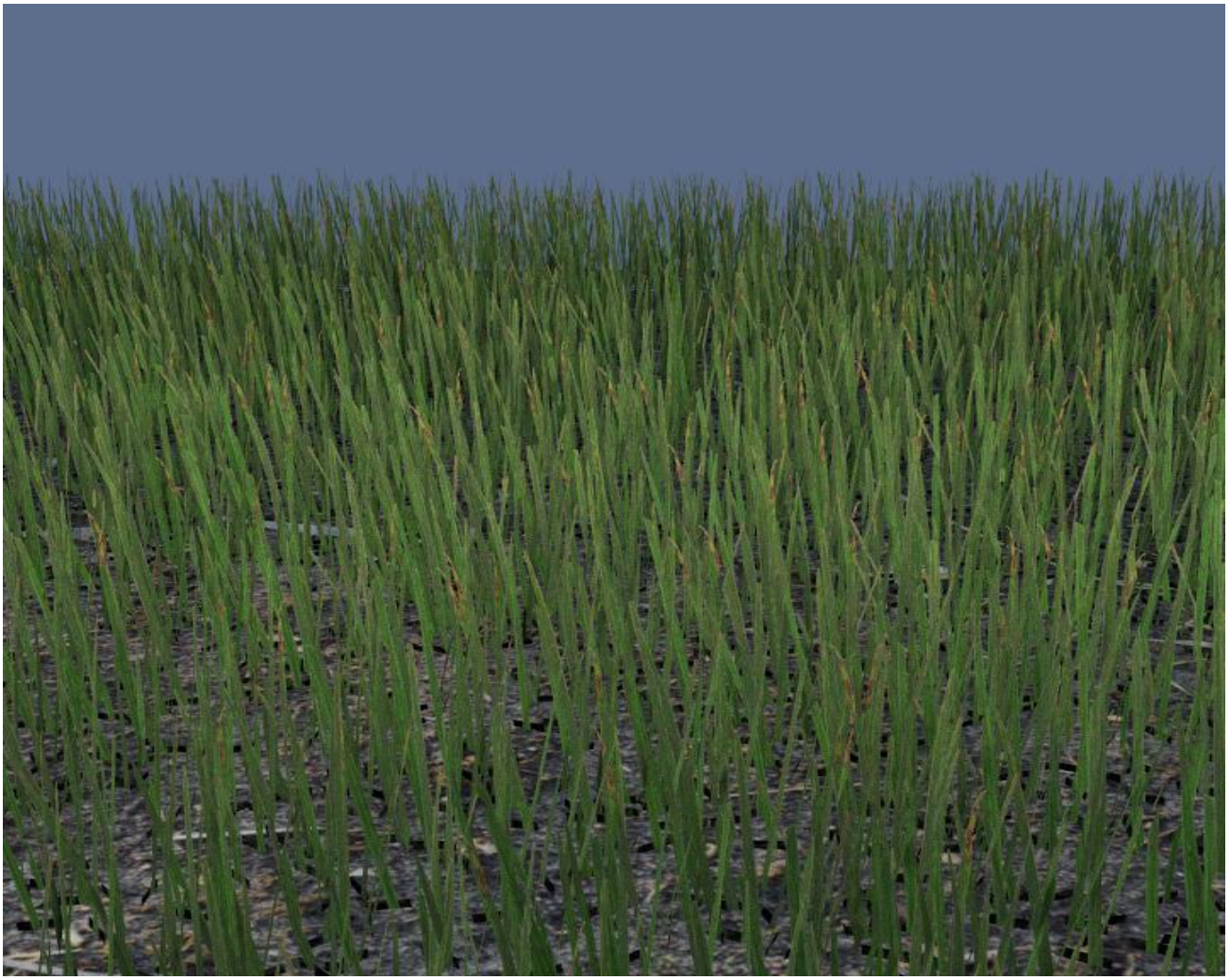
È da subito ben visibile la differenza tra la prima immagine con density 1.0f e la terza, con density 0.2f. È chiaro inoltre che per valori molto elevati di num, è necessario abbassare di molto il valore della densità per ottenere risultati palesemente visibili.

Grass.

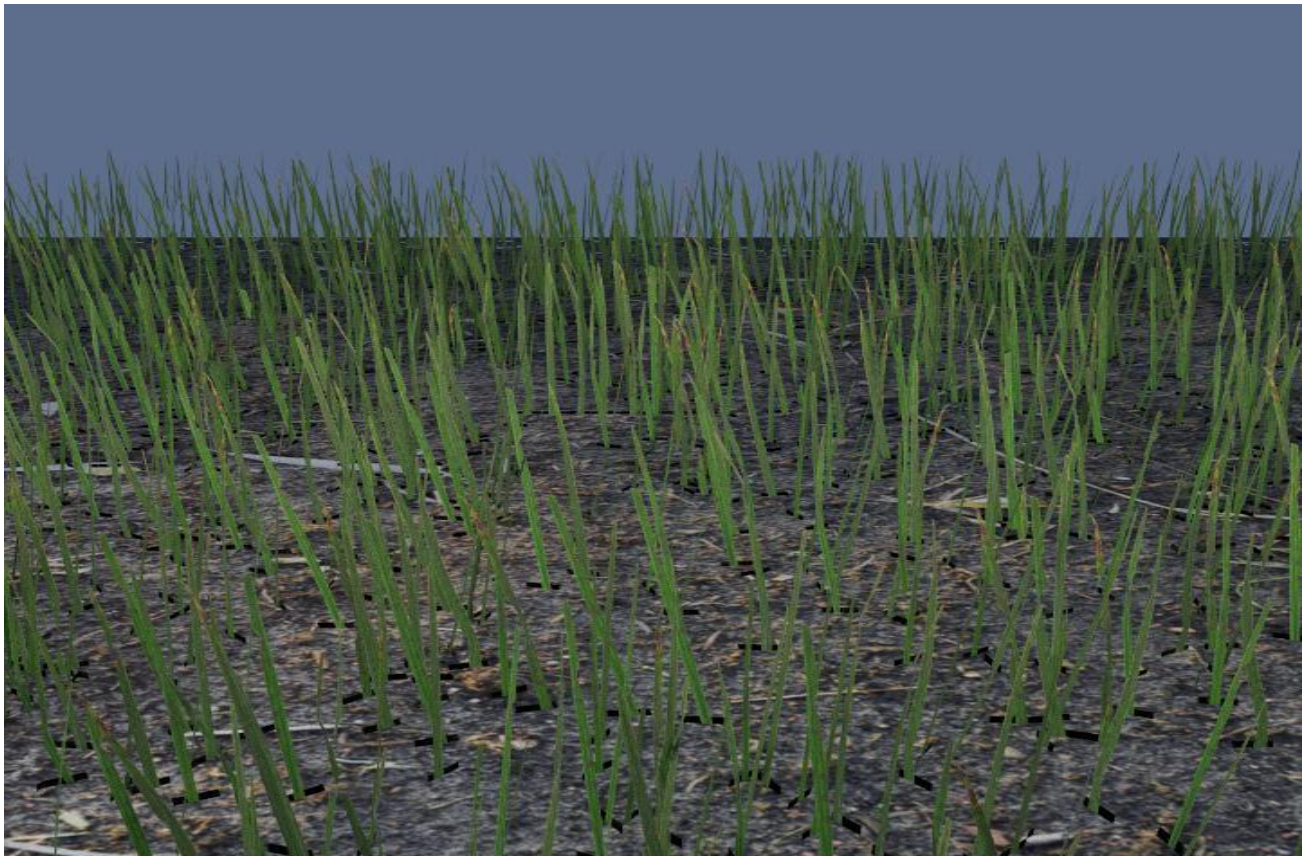
Vedremo ora nell'esempio di grass come una differenza anche minima nel valore density (1.0 - 0.7) sia già molto visibile e marcata, data la sostanziale differenza nella distribuzione e nell'area coperta dai fili d'erba. Come nel caso di hair, le immagini sono generate con valori density 1.0f, 0.7f e 0.2f.



Prato generato normalmente da make_grass con density = 1.0f, samples = 256; resolution = 720



Prato generato con density = 0.7f; samples = 256, resolution = 720



Prato generato con density = 0.2f; samples = 256, resolution = 720

CREDITS
Per questa sezione non è stata usata alcuna reference esterna. Il codice è stato scritto da me seguendo la definizione data nel readme. Le funzioni usate per il calcolo del CDF e per il rendering sono built-in di Yocto.

Procedural Noises - CellNoise & SmoothVoronoi.

Ho deciso di dare un'occhiata a Procedural Noises, in particolare alle versioni CellNoise e SmoothVoronoi. La generazione di questi due procedural noises è stata ottenuta modificando l'implementazione di *make_displacement*, seguendo l'articolo di Inigo Quilez linkato nel readme, credits a fine sezione. Ho usato diversi boolean per mantenere "malleabile" la funzione di *make_displacement* da linea di comando, in particolare è possibile scegliere queste opzioni:

- *--cellnoise* → applica cellnoise
- *--smoothvoronoi* → applica smoothvoronoi
    - Se nessuno dei due è specificato da linea di comando, la funzione applicherà *turbulence* noise come per i crediti base
- *--surface* → applica il noise selezionato sull'intera superficie, modificandone la forma. Se non viene specificato, il noise viene usato solo nel calcolo del colore, simulando l'applicazione di una texture su una shape
- *--tridimensional* → applica cellnoise o smoothvoronoi su tre dimensioni piuttosto che due. Se non viene specificato, l'implementazione del noise segue quella di Inigo Quilez dal link con un vec2f in input

Ovviamente a parte la trasformazione da vec2f a vec3f, entrambe le versioni delle funzioni cellnoise e smoothvoronoi applicano lo stesso algoritmo. Nel codice sono definite due funzioni in overload per ciascun noise, una con argomento vec3f e l'altra con argomento vec2f. Specifico nei bullet point a seguire alcune differenze con il codice riportato da Inigo, non specificherò invece nel dettaglio cosa fa l'algoritmo di per sé, dato che il resto dell'implementazione è esattamente come quella riportata nel sito.

- Per cellnoise, in entrambe le versioni in vec2f e vec3f, il fattore rng è generato attraverso il seed dato nella documentazione di Yocto. Il numero rand è quindi generato da un *make_rng* inizializzato con seed 172784
- Nella versione vec3f di entrambi i noise, per aggiungere la terza dimensione, ho aggiunto un terzo ciclo for (per entrambi i pass in cellnoise). Esegue lo stesso numero di iterazioni degli altri ed è posto prima degli altri due. Iteriamo prima su un valore k, poi su j e poi su i, ed istanziamo successivamente il vettore b come {i, j, k}. Riporto per completezza uno snippet per entrambi i pass di cellnoise (smoothvoronoi usa una versione uguale al primo screen):

```
for (float k = -1; k <= 1; k++)
    for (float j = -1; j <= 1; j++)
        for (float i = -1; i <= 1; i++) {
            vec3f b = vec3f{i, j, k};
```

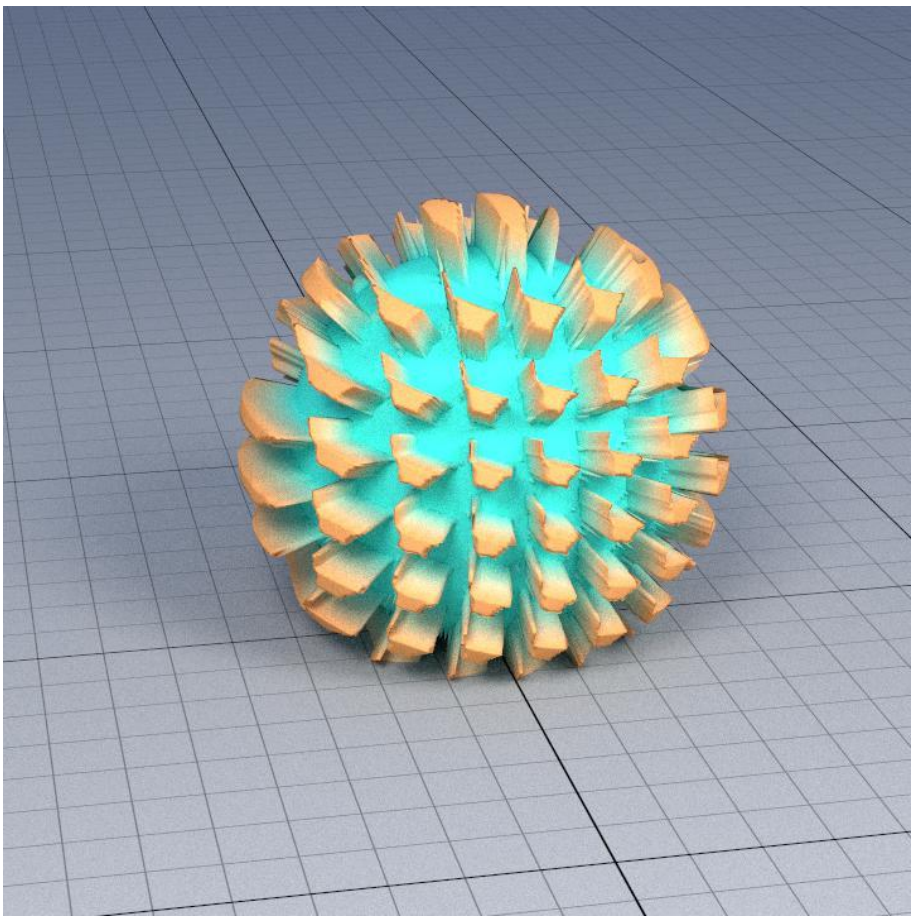
```
for(float k = -2; k <= 2; k++)
    for (float j = -2; j <= 2; j++)
        for (float i = -2; i <= 2; i++) {
            vec3f b = mb + vec3f{i, j, k};
```

- In generale, per entrambe le versioni vec2f e vec3f, ho usato sin dal principio i float piuttosto che utilizzare degli int come nell'implementazione di Inigo effettuando casts successivamente. Ho pensato che questo potesse evitare approssimazioni non volute dovute al casting da float ad integer (anche se magari di poco conto)

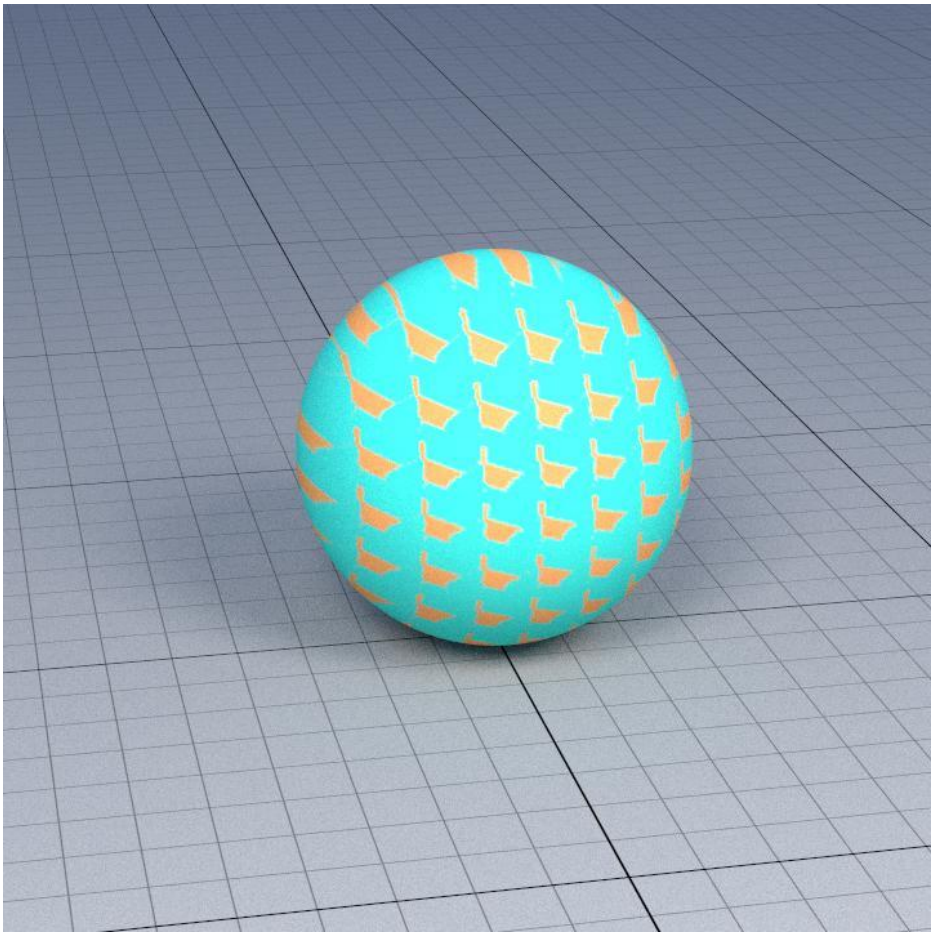
Riporto subito sotto i risultati del calcolo del cellnoise e dello smoothvoronoi, applicati sia su xy, sia su xyz. Per entrambi i casi riporto sia la versione surface dove viene applicata la modifica della shape, sia la versione "texture" dove non viene applicata alcuna modifica "fisica" alla shape, ma solo sul colore.

CELLNOISE

Vec2f: Surface & Texture.

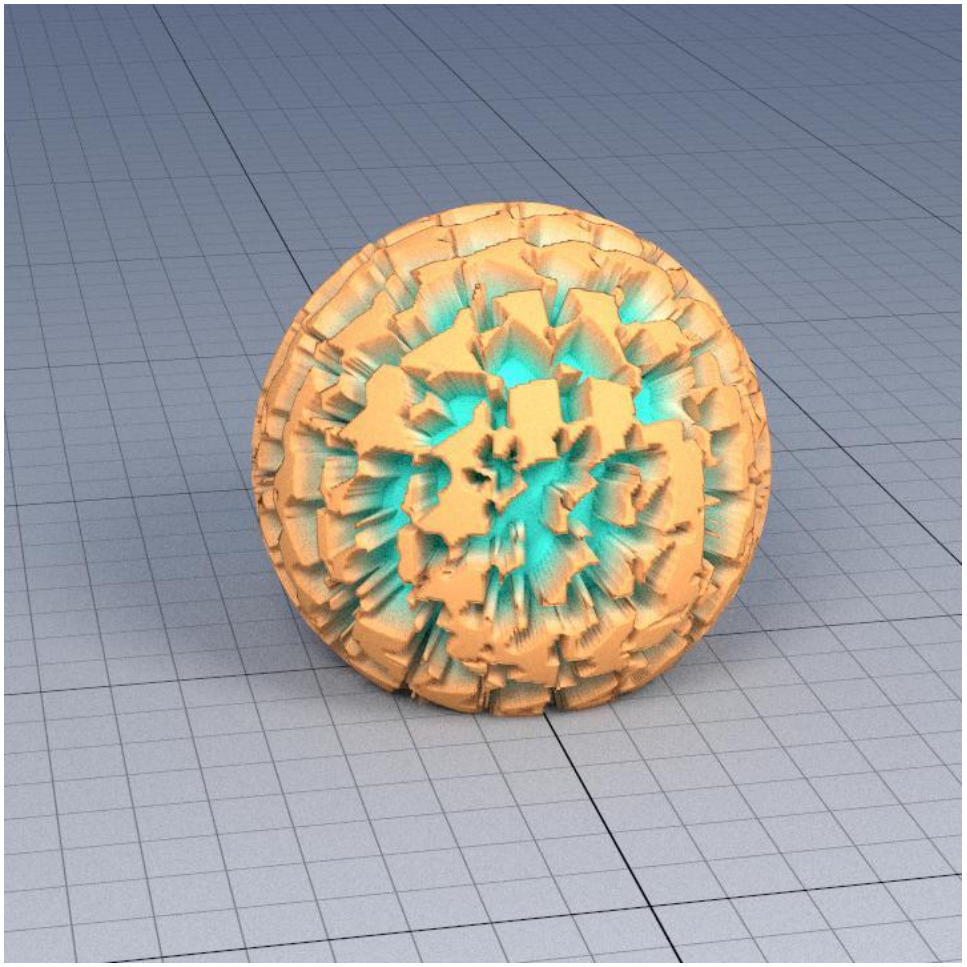


Sphere generata applicando cellnoise in due dimensioni, versione surface. samples = 256; resolution = 720

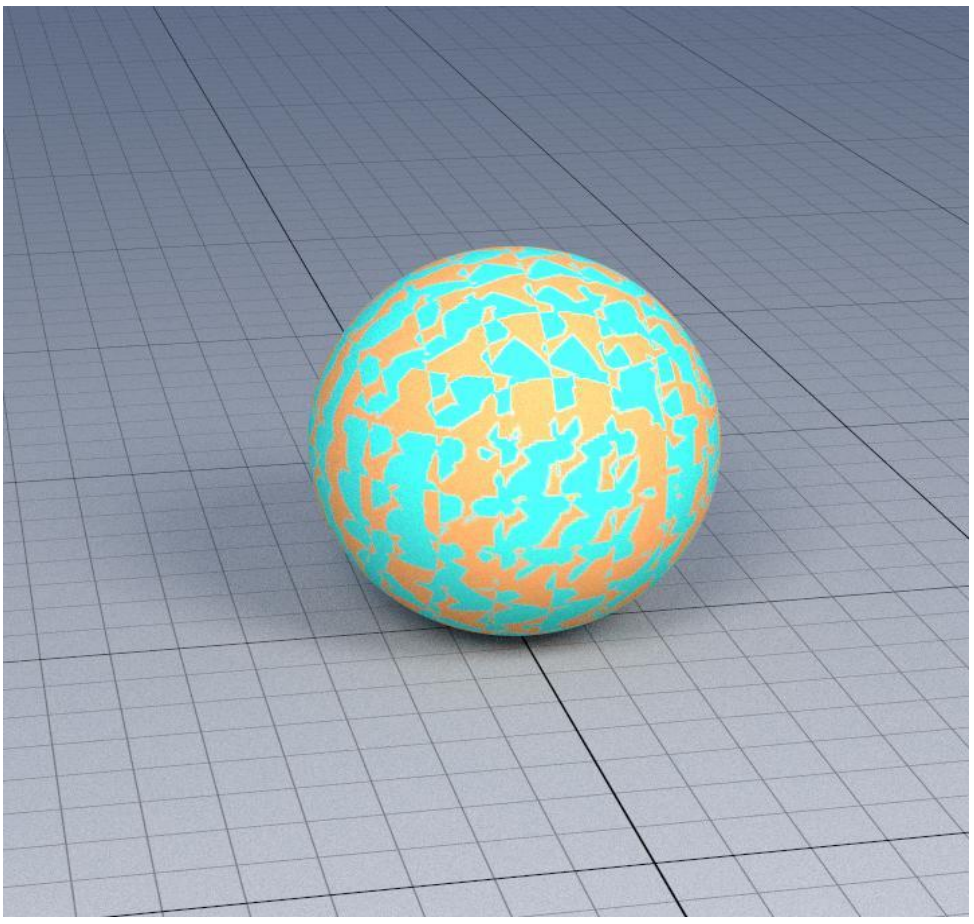


Sphere generata applicando cellnoise in due dimensioni, versione texture. samples = 256; resolution = 720

Vec3f: Surface & Texture.



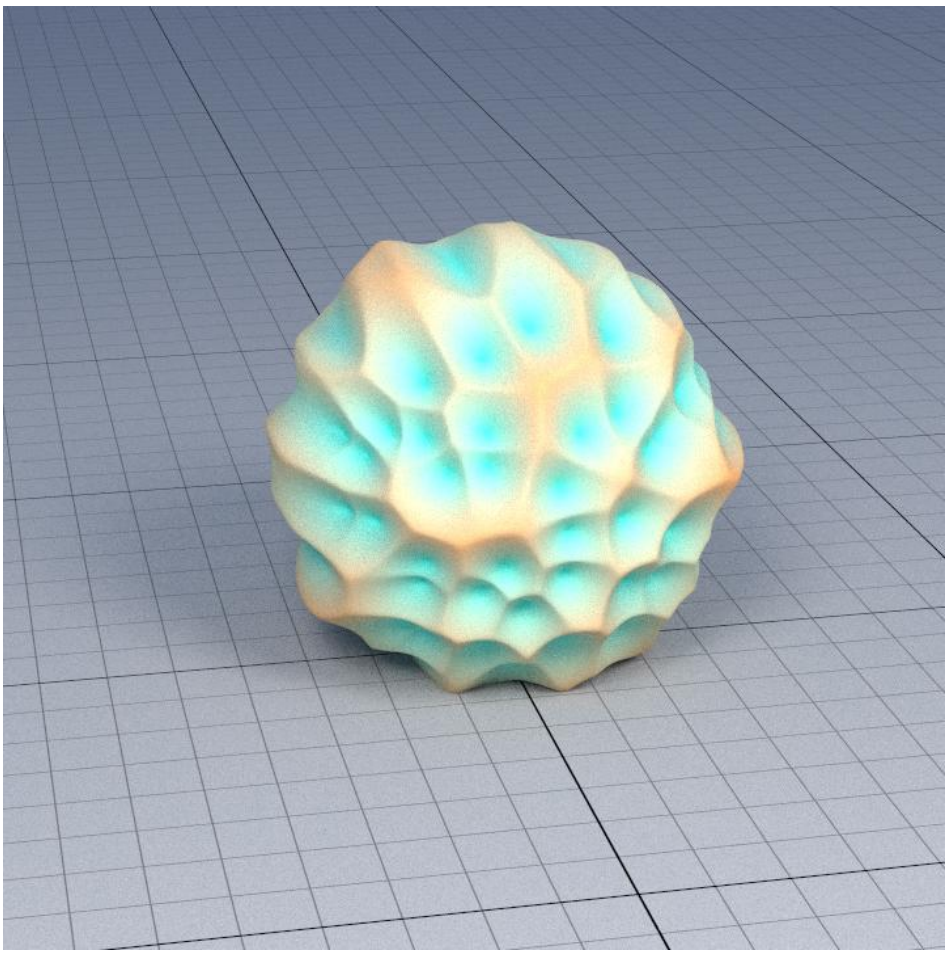
Sphere generata applicando cellnoise in tre dimensioni, versione surface. samples = 256; resolution = 720



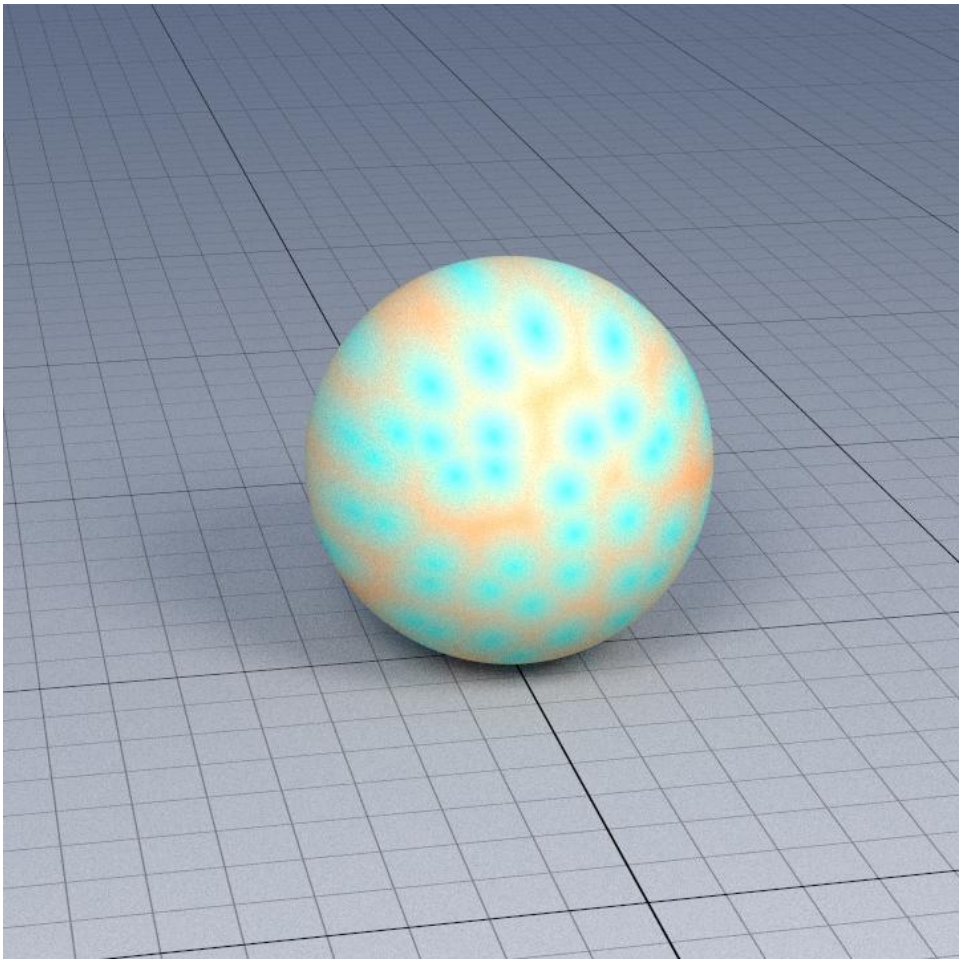
Sphere generata applicando cellnoise in tre dimensioni, versione texture. samples = 256; resolution = 720

SMOOTHVORONOI

Vec2f: Surface & Texture.

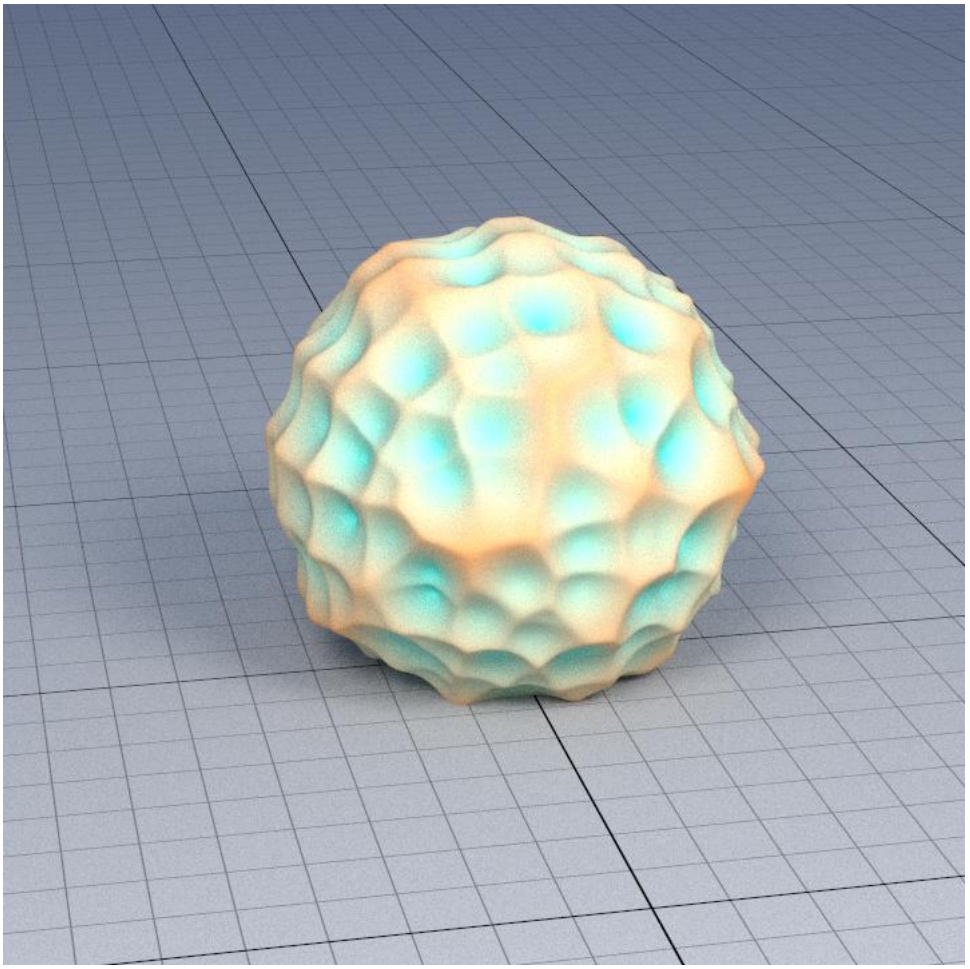


Sphere generata applicando smoothvoronoi in due dimensioni, versione surface. samples = 256; resolution = 720

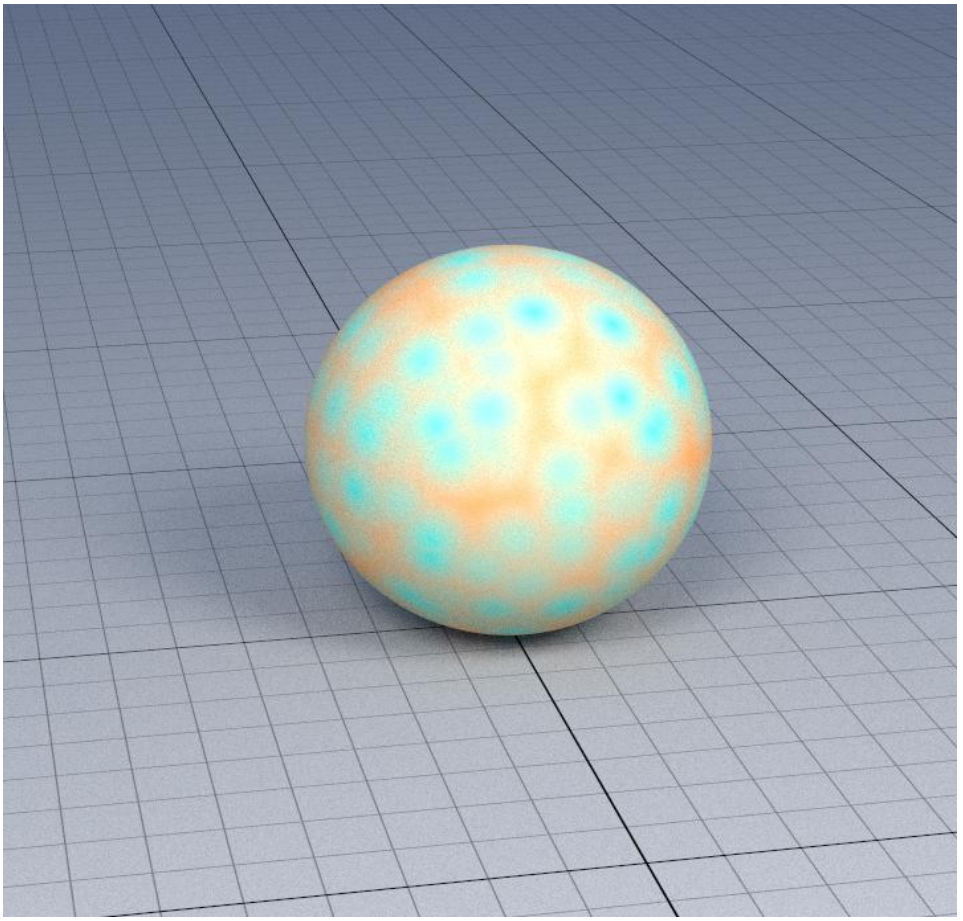


Sphere generata applicando smoothvoronoi in due dimensioni, versione texture. samples = 256; resolution = 720

Vec3f: Surface & Texture



Sphere generata applicando smoothvoronoi in tre dimensioni, versione surface. samples = 256; resolution = 720



Sphere generata applicando smoothvoronoi in tre dimensioni, versione texture. samples = 256; resolution = 720

CREDITS

Per questa sezione sono stati utilizzati come reference le seguenti pagine web di Inigo Quilez:

- Cellnoise → https://www.iquilezles.org/www/articles/voronoilines/voronoilines.htm.
- Smoothvoronoi → https://www.iquilezles.org/www/articles/smoothvoronoi/smoothvoronoi.htm

Per l'implementazione della funzione "hash2f" usata per smoothvoronoi è stata usata la seguente pagina shadertoy come riferimento, sempre di Inigo Quilez: https://www.shadertoy.com/view/ldB3zc.

Le piccole modifiche elencate sopra sono state effettuate da me. Il rendering è effettuato con le funzioni built-in di Yocto.