Procedural Modeling: proceduralità creativa

Lorenzo Piarulli 1884766

In tutte le scene utilizzate è stato rimosso il pavimento per incrementare la velocità di computazione degli algoritmi.

Procedural Planet Generator

Difficile

Nel primo paragrafo di questo paper, viene presentato un piccolo progetto composto da due algoritmi: **planet_seed**, **planet_noise**. Gli algoritmi possono essere sintetizzati in cinque fasi:

1. Distribuzione di un numero randomico di punti sulla superficie di una sfera.
2. Calcolo del **Planet Noise** per ogni punto.
3. Definizione e modellazione delle aree ricoperte d'acqua e delle aree rappresentanti le vette.
4. Modifica dell'altezza di ogni punto secondo la normale alla superficie.
5. Applicazione dei colori.

Di seguito è riportato l'algoritmo principale:

```
void make_planet(shape_data& shape, const planet_params& params) {
  for (int i : range(shape.positions.size())) {
    vec3f lastPos = shape.positions[i];
    float sea_level = 0.22;
    float peak_level = 0;

    shape.positions[i] += shape.normals[i] *
                          planet_noise(shape.positions[i], params.scale, 4,
                              params.persistance, params.lacunarity,
sea_level,
                              peak_level) *
                          params.height;

    float dist = distance(shape.positions[i], lastPos) / params.height;

    if (dist <= sea_level + 0.02) {

      shape.colors.push_back(
          (interpolate_line(params.sea, params.ocean, dist)));

    } else if (dist > sea_level + 0.02 && dist < 0.45) {

      shape.colors.push_back(
          (interpolate_line(params.land, params.rock, dist)));

    } else if (dist >= 0.45 && dist < 0.7) {

      shape.colors.push_back(params.rock);
```

```
    } else {

      shape.colors.push_back(params.top);

    }
  }

  quads_normals(shape.normals, shape.quads, shape.positions);
}
```

Durante la fase di innalzamento dei punti, viene applicato il **Planet Noise**, quest'ultimo è la chiave fondamentale per la generazione del pianeta. Il **Planet Noise** è stato costruito basandosi sul più generale **Perlin Noise**. Quest'ultimo è composto da molti parametri tra i quali: *frequency, amplitude, persistance, lacunarity*. La modifica della *frequency* permette di ottenere catene montuose più o meno vaste e omogenee. In unione a questa, si ha l'*amplitude* con la quale si può modificare l'altezza di tali catene. Altre due proprietà caratterizzano il **Perlin Noise**, e sono la *persistance* e la *lacunarity*. La prima permette di ammorbidire il disturbo, aumentandola si incrementa la presenza di vette e la loro caratteristica acuminata. La seconda genera un ulteriore disturbo, con frequenza maggiore, che va ad incrementare il dettaglio. Le proprietà descritte in precedenza sono mantenute nel **Planet Noise**, per implementare gli effetti descritti.

```
float planet_noise(const vec3f& p, float scale, int octaves, float
persistance,
    float lacunarity, float sea_level, float peak_level) {
  float amplitude = 1;
  float frequency = 0.3;
  float noiseHeight = 0;

  for (int i = 0; i < octaves; i++) {
    vec3f newp = p * scale * frequency;

    float noiseValue = fabs(noise(newp));
    noiseHeight += noiseValue * amplitude;
    amplitude *= persistance;
    frequency *= lacunarity;
  }
```

In quest'ultima fase, si imposta il disturbo per le zone oceaniche e le zone a prevalenza montuosa.

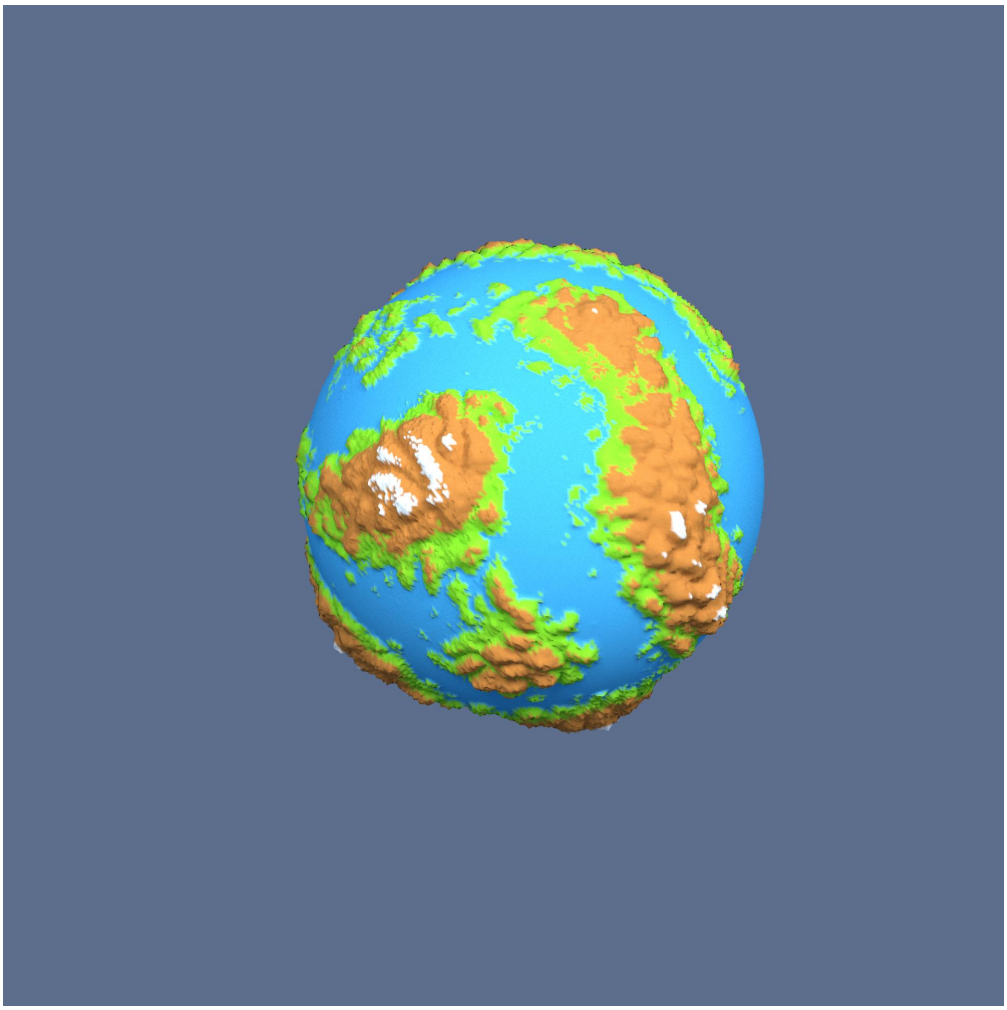
```
  // SEA OPTION
  if (sea_level != 0 && noiseHeight <= sea_level) {
    noiseHeight = sea_level;
  }

  // PEAK OPTION
  if (peak_level != 0 && noiseHeight >= peak_level) {
    noiseHeight = peak_level;
```

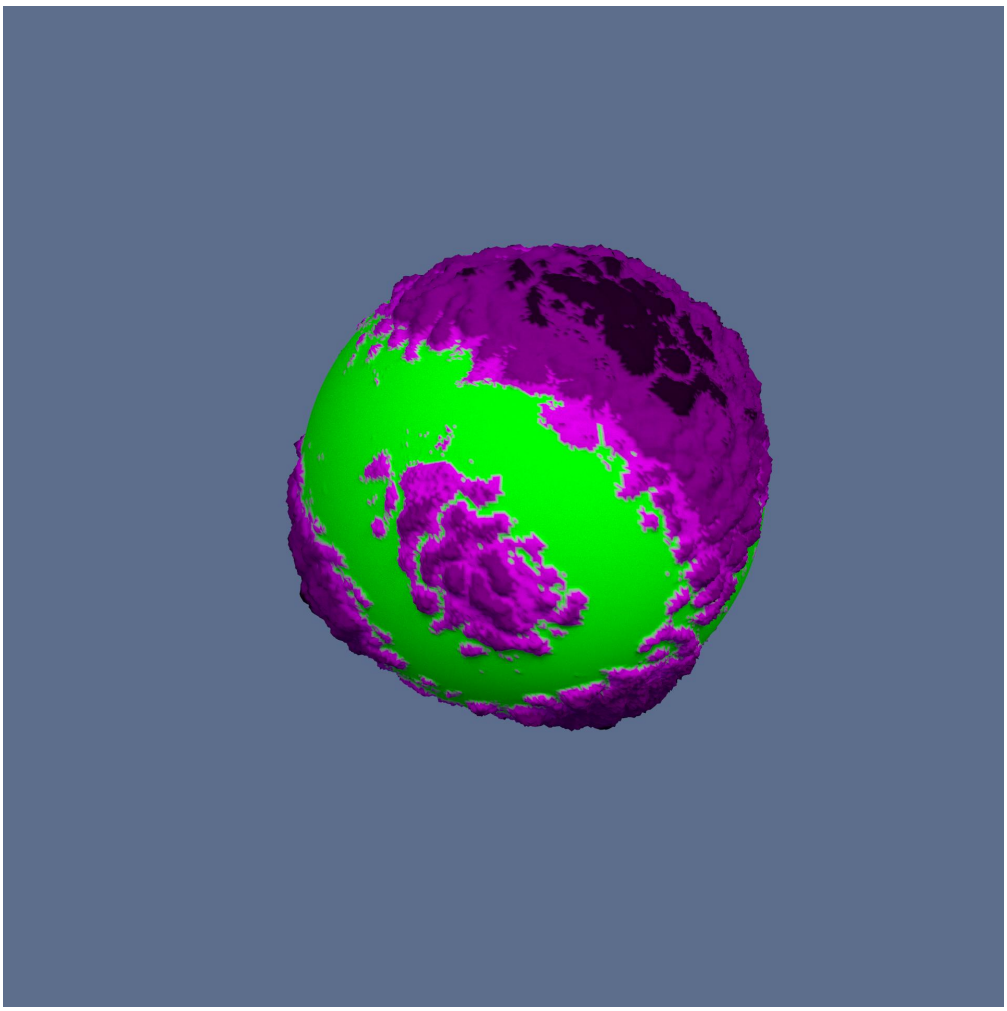
```
    }

    return noiseHeight;
  }
```

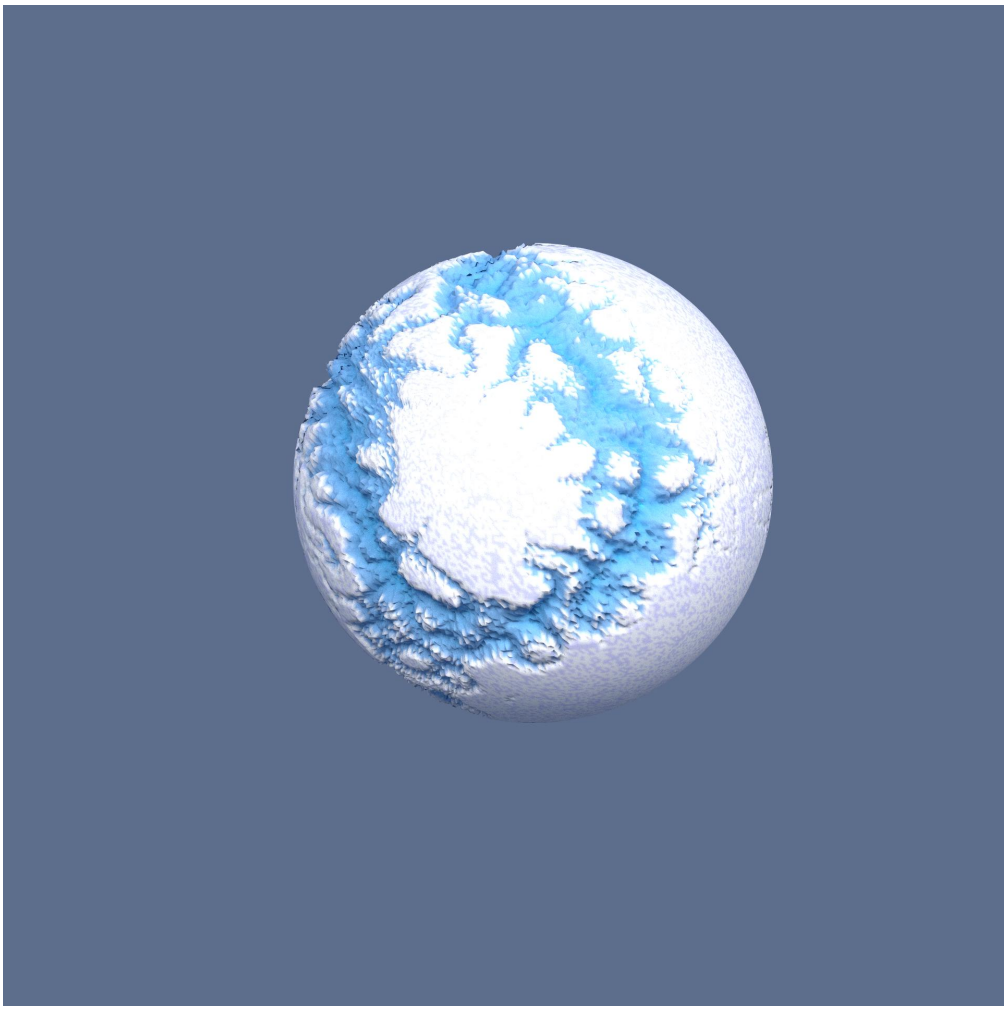
Alcune delle proprietà sono modificabili nei parametri descritti di seguito.

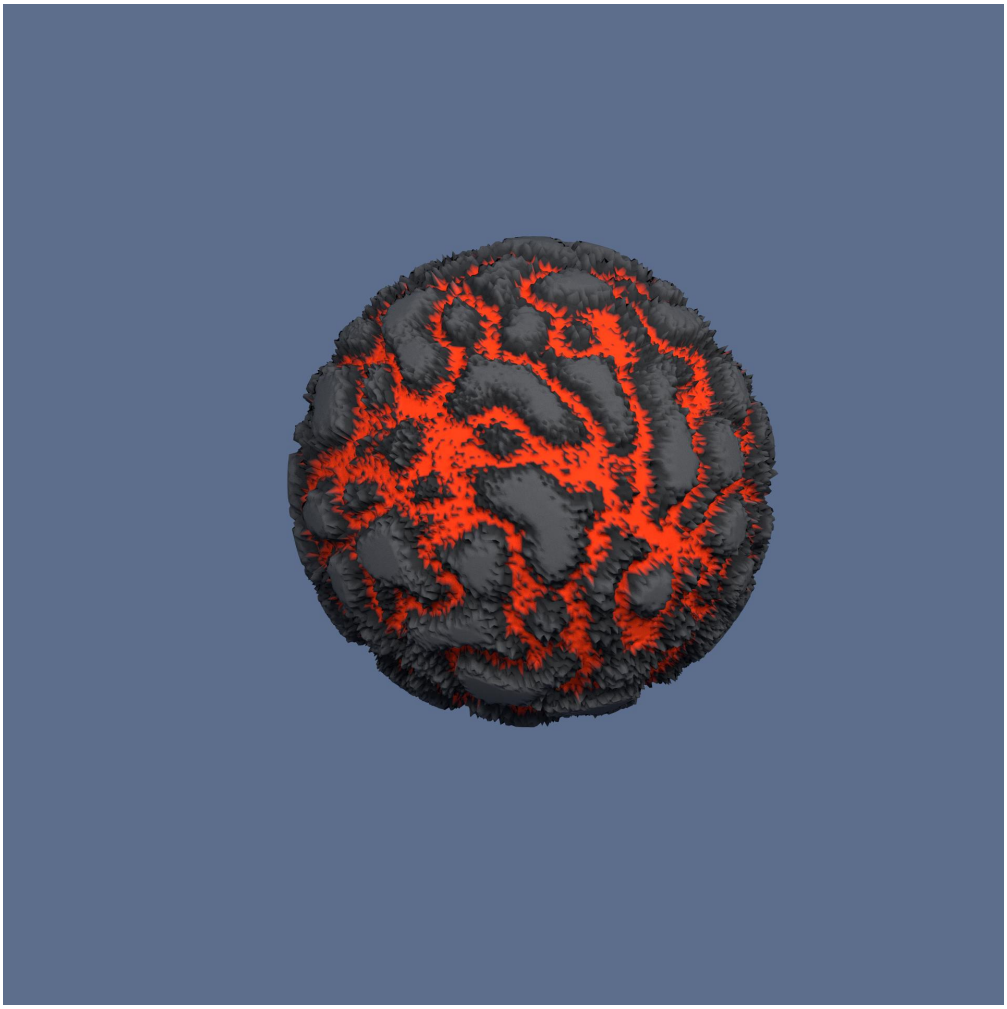
```
struct planet_params {
    float height  = 0.02f;
    float scale   = 50;
    int   octaves = 8;
    // Earth
    float persistance = 0.35;
    float lacunarity  = 7;
    vec4f ocean       = srgb_to_rgb(vec4f{64, 50, 255, 255} / 255);
    vec4f sea         = srgb_to_rgb(vec4f{64, 200, 224, 255} / 255);
    vec4f land        = srgb_to_rgb(vec4f{30, 255, 30, 255} / 255);
    vec4f rock        = srgb_to_rgb(vec4f{200, 133, 63, 255} / 255);
    vec4f top         = srgb_to_rgb(vec4f{240, 255, 255, 255} / 255);
};
```

Utilizzando tali algoritmi e applicando diversi valori per le diverse proprietà, sono stati ottenuti pianeti con caratteristiche decisamente diverse. Di seguito sono riportati i risultati più rilevanti. Inoltre è stata prodotta una gif renderizzando un pianeta con valori crescenti di *frequency*. Quest'ultima è allegata nella cartella con il nome di "Terraforming.gif".









Spiderweb vs Spikes

In questo paragrafo sono riportati due algoritmi che generano effetti completamente diversi, nonostante implementino la stessa metodologia. Entrambi distribuiscono dei nodi sopraelevati e creano connessioni applicando le proprie strategie.

Spiderweb

Medio

L'algoritmo presentato di seguito tenta di ricreare una struttura simile ad una ragnatela.

```
void make_spiderweb(shape_data& spiderweb, const shape_data& shape,
    const spiderweb_params& params) {
  shape_data shp = shape;

  vector<vec3f> nodes;
  int           n = 1024;
  sample_shape(shp.positions, shp.normals, shp.texcoords, shp, params.num);

  rng_state rng = make_rng(172784);

  for (int i : range(n)) {
    int   index      = rand1i(rng, shp.positions.size());
    vec3f elevatePos = shp.positions[index] +
                       params.lenght * shp.normals[index] * rand1f(rng) +
                       noise3(shp.positions[index]) * params.strength;

    nodes.push_back(elevatePos);
  }
```

Dopo aver aggiunto una distribuzione di punti ed innalzato alcuni di questi, che chiameremo nodi, viene effettuato il collegamento. Quest'ultimo viene applicato tra ogni nodo ed i suoi punti calcolati in un certo *range*.

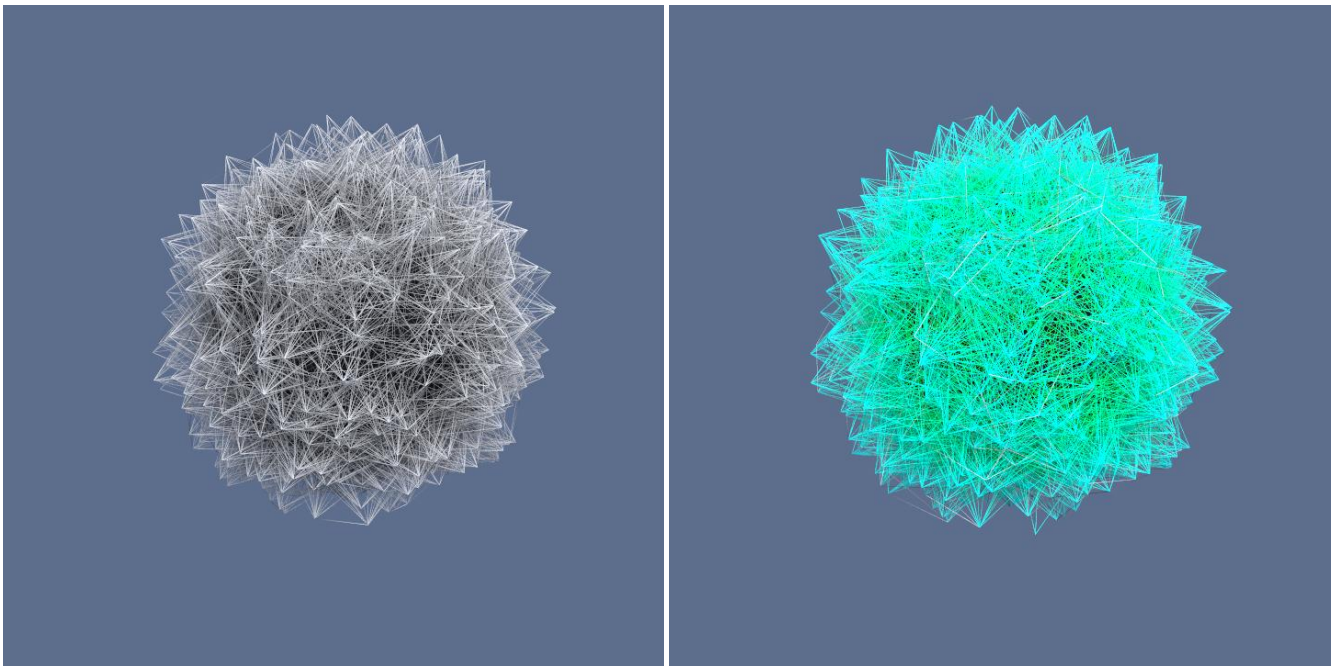
```
  for (int i : range(shp.positions.size())) {
    vector<vec3f> positions;
    vector<vec4f> colors;
    positions.push_back(shp.positions[i]);
    colors.push_back(params.bottom);
    vec3f normal = shp.normals[i];

    float dist = 100000;
    vec3f nextNode;
    vec3f nodeInrange[nodes.size()];

    int counter = 1;
    for (auto node : nodes) {
      if (distance(node, shp.positions[i]) < params.lenght * 2) {
        nodeInrange[counter] = node;
```

```
        counter++;
      }
    }

    if (counter != 0) {
      nextNode = nodeInrange[rand1i(rng, counter)];
      positions.push_back(nextNode);
    }

    colors.push_back(interpolate_line(params.bottom, params.top,
        distance(nextNode, positions[0]) / params.lenght));

    colors[1] = params.top;
    add_polyline(spiderweb, positions, colors);
  }
```

In quest'ultima fase vengono tracciate le connessioni.

```
for (auto nodei : nodes) {
  for (auto nodej : nodes) {
    if (distance(nodei, nodej) < params.lenght && nodei != nodej &&
        rand1f(rng) < 0.05) {
      vector<vec3f> connections;
      vector<vec4f> colors;

      connections.push_back(nodej);
      connections.push_back(nodei);
      float color = 255 * rand1f(rng);
      colors.push_back(srgb_to_rgb(vec4f{color, color, color, 255} / 255));
      add_polyline(spiderweb, connections, colors);
    }
  }
}

vector<vec3f> tang = lines_tangents(spiderweb.lines, spiderweb.positions);

for (int h : range(tang.size())) {
  auto  ta = tang[h];
  vec4f t  = vec4f{ta.x, ta.y, ta.z, 0.f};
  spiderweb.tangents.push_back(t);
}
}
```

Di seguito i risultati ottenuti, applicando diverse tonalità:



Spikes

Medio

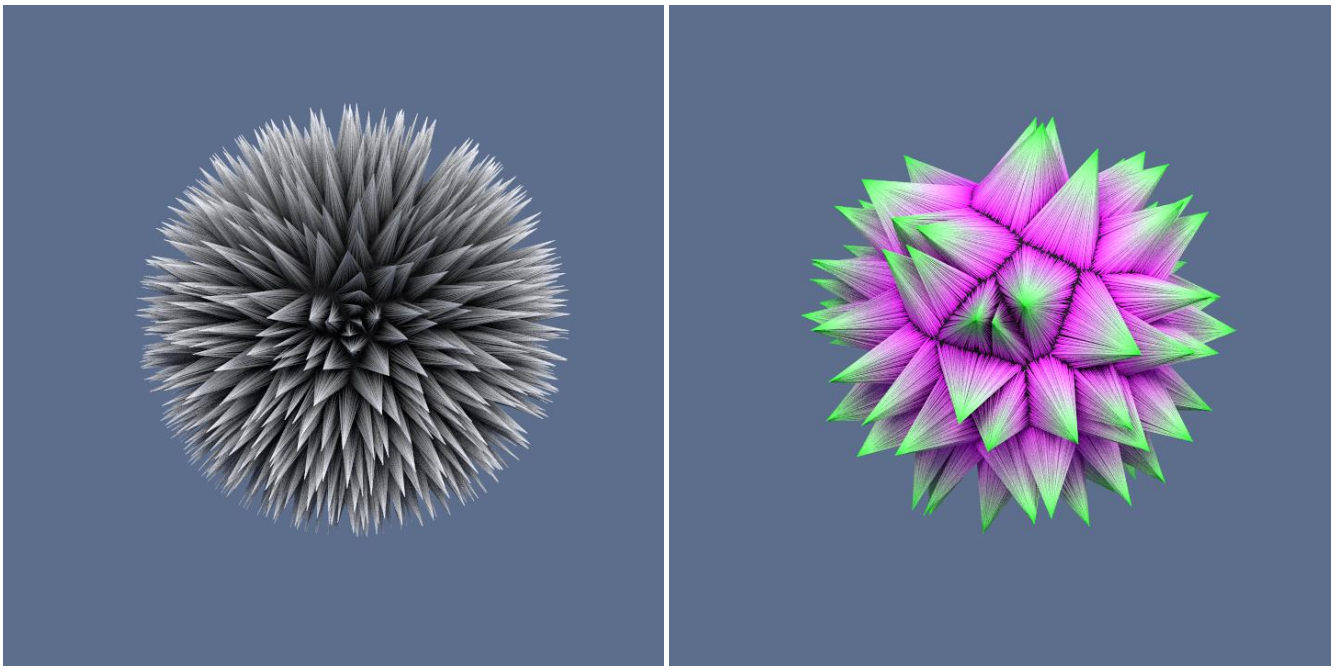
In questa sezione vediamo l'implementazione di un algoritmo, che genera delle punte formate da capelli. Le punte vengono costruite connettendo i nodi sopraelevati ai punti sulla sfera, contenuti nel loro range.

```
void make_spikes(
    shape_data& spikes, const shape_data& shape, const spikes_params&
params) {
  shape_data shp = shape;

  vector<vec3f> nodes;

  auto size = shp.positions.size();
  sample_shape(shp.positions, shp.normals, shp.texcoords, shp, params.num);

  rng_state rng = make_rng(172784);

  for (int i = size; i < shp.positions.size(); i++) {
    float rand = rand1f(rng);
    if (rand < 0.001) {
      int   index      = rand1i(rng, shp.positions.size());
      vec3f elevatePos = shp.positions[index] +
                         params.lenght * shp.normals[index];
      nodes.push_back(elevatePos);
    }
  }
```

In questa fase l'algoritmo invece di trovare i punti partendo dal nodo, come l'algoritmo precedente, itera direttamente sui punti e per ognuno di questi, trova il nodo più vicino.

```
for (int i : range(shp.positions.size())) {
  vector<vec3f> positions;
  vector<vec4f> colors;
  positions.push_back(shp.positions[i]);
  colors.push_back(params.bottom);
  vec3f normal = shp.normals[i];

  float dist = 100000;
  vec3f nextNode;
  vec3f nodeInrange[nodes.size()];

  for (auto node : nodes) {
    if (distance(node, shp.positions[i]) < dist) {
      dist     = distance(node, shp.positions[i]);
      nextNode = node;
    }
  }
  positions.push_back(nextNode);

  colors.push_back(interpolate_line(params.bottom, params.top,
      distance(nextNode, positions[0]) / params.lenght));

  colors[1] = params.top;
  add_polyline(spikes, positions, colors);
}

vector<vec3f> tang = lines_tangents(spikes.lines, spikes.positions);

for (int h : range(tang.size())) {
  auto  ta = tang[h];
  vec4f t  = vec4f{ta.x, ta.y, ta.z, 0.f};
  spikes.tangents.push_back(t);
}
}
```

Di seguito sono riportati alcuni risultati, ottenuti utilizzando colorazioni e distribuzioni con livelli di randomicità differenti.



Procedural Structures Algorithms

In questo paragrafo sono riporati tre algoritmi che generano strutture partendo dalla forma assegnata. Le struttura generate risultano essere indipendenti dalla forma da cui sono ispirate, poichè non vi è alcun punto di connessione con la forma originale.

Geometric structure

Medio

L'algoritmo descritto di seguito per generare la struttra evidenzata nei risultati segue le seguenti fasi:

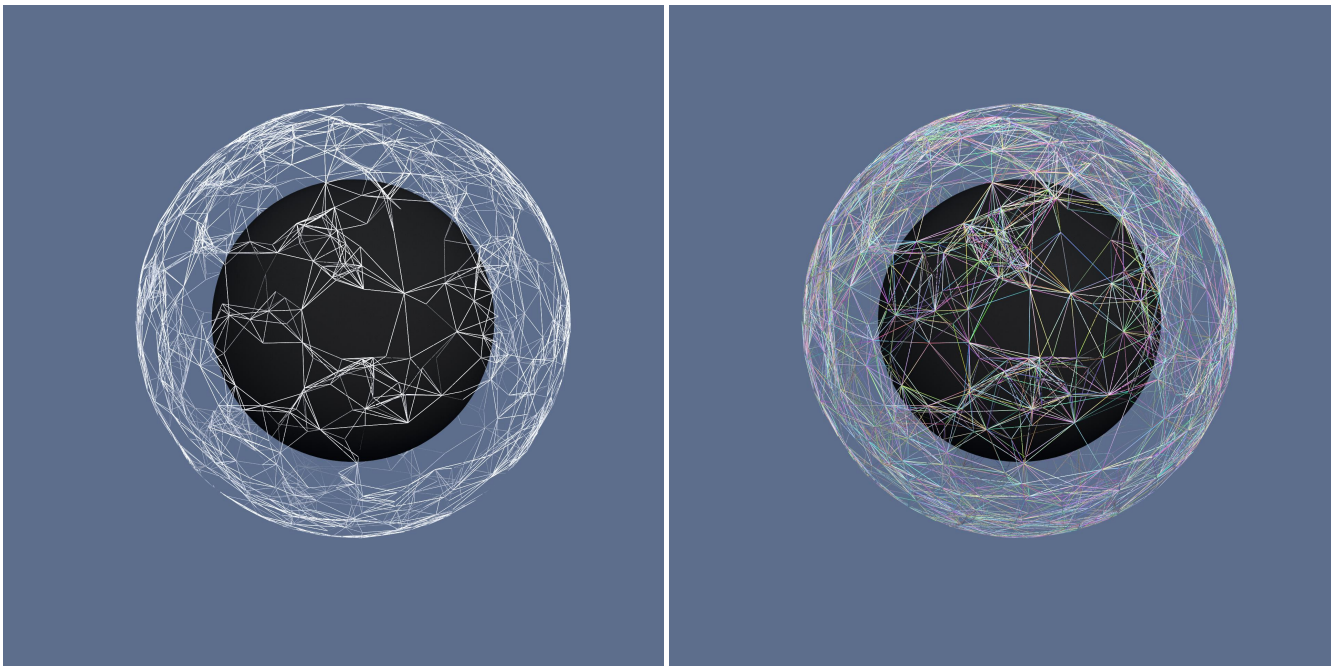
1. Distribuisce ed innalza, di un valore *params.lenght*, un numero *params.num* di punti.
2. Calcola per ogni punto innalzato (Nodo), i nodi che lo circondano in un *range* assegnato.
3. Connette tutti i nodi contenuti in tale range.

```
void make_structure(shape_data& structure, const shape_data& shape,
    const structure_params& params) {
  shape_data shp = shape;

  auto size = shp.positions.size();
  sample_shape(shp.positions, shp.normals, shp.texcoords, shp, params.num);

  rng_state rng = make_rng(172784);

  for (int i = size; i < shp.positions.size(); i++) {
    shp.positions[i] = shp.positions[i] +
                       params.lenght * normalize(shp.normals[i]);
  }

  for (int i = size; i < shp.positions.size(); i++) {
    float         dist = 100000;
    vector<vec3f> nodesInrange;
```

```
    vec3f         distnode;
    for (int h : range(20)) {
      float dist = 100000;
      for (int j = size; j < shp.positions.size(); j++) {
        if (distance(shp.positions[j], shp.positions[i]) < dist &&
            shp.positions[j] != shp.positions[i] &&
            distance(shp.positions[j], shp.positions[i]) < params.lenght) {
          bool stop = false;

          for (auto ranged : nodesInrange) {
            if (ranged == shp.positions[j]) {
              stop = true;
              break;
            }
          }

          if (stop == false) {
            dist     = distance(shp.positions[j], shp.positions[i]);
            distnode = shp.positions[j];
          }
        }
      }
      nodesInrange.push_back(distnode);
    }

    for (auto node : nodesInrange) {
      if (shp.positions[i] != node) {
        vector<vec3f> connections;
        vector<vec4f> colors;
        std::cout << std::to_string(node.x) + "\n";
        connections.push_back(shp.positions[i]);
        connections.push_back(node);
        float color = 255;

        // Black & White

        // colors.push_back(srgb_to_rgb(vec4f{color, color, color, 255} /
255));

        // Random colors
        colors.push_back(rgb_to_rgba(rand3f(rng)));

        auto dist = distance(node, shp.positions[i]);
        add_polyline(structure, connections, colors);
      }
    }
    std::cout << "\n\n\n";
  }

  vector<vec3f> tang = lines_tangents(structure.lines,
structure.positions);

  for (int h : range(tang.size())) {
    auto  ta = tang[h];
```

```
      vec4f t  = vec4f{ta.x, ta.y, ta.z, 0.f};
      structure.tangents.push_back(t);
    }
  }
```

Le strutture ottenute sono riportate di seguito. Tali risultati sono stati creati variando il numero di nodi per le connessioni e modificando la tipologia di colorazione.



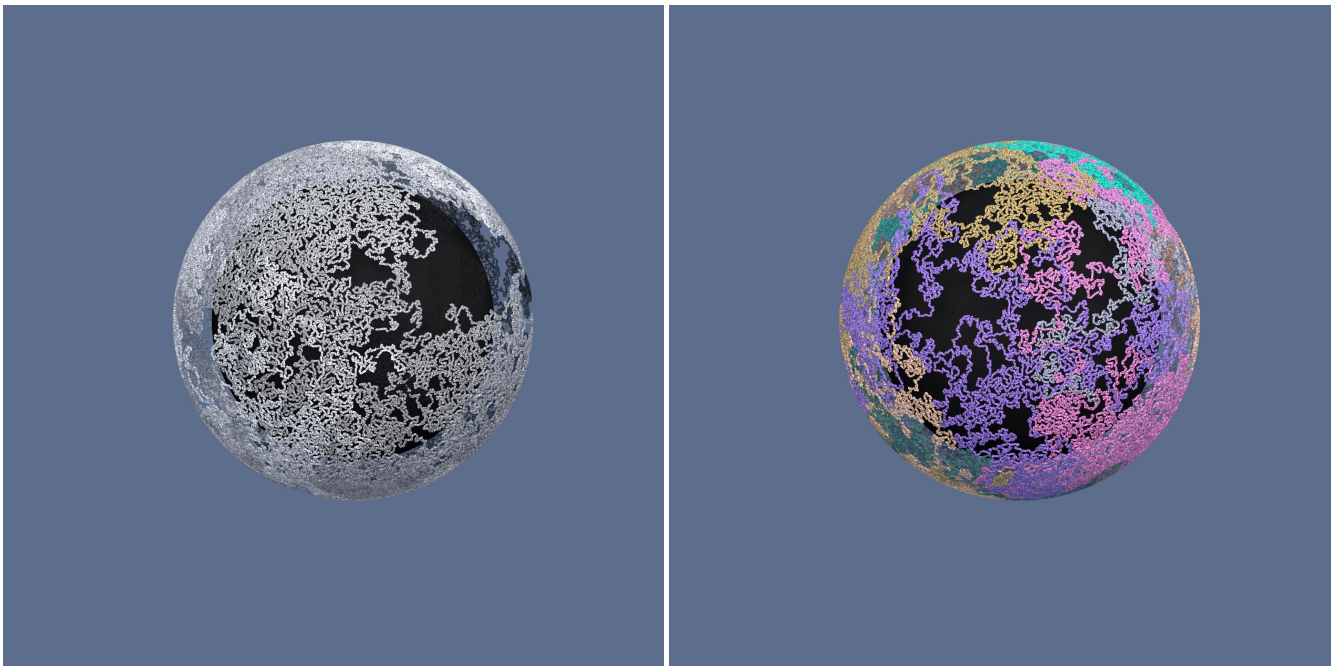
No-traversable path structure

Difficile

La funzione che viene affrontata in questo paragrafo, genera una struttura costituita da vari percorsi che per loro natura non si attraversano. Tali percorsi vengono costruiti collegando ogni punto al suo punto più vicino. Tutti i punti, vengono selezionati e marcati così che non vengano attraversati nuovamente. Se la figura dispone di un numero elevato di punti distributi, allora la proprietà di non attraversamento sarà soddisfatta.

```
void make_path(
    shape_data& path, const shape_data& shape, const path_params& params) {

  shape_data shp = shape;

  auto size = shp.positions.size();
  sample_shape(shp.positions, shp.normals, shp.texcoords, shp, params.num);

  rng_state rng = make_rng(172784);

  // Selection of first node (the root node)

  int  firstindex = rand1i(rng, shp.positions.size() - size) + size;
```

```
  bool marked[params.num];

  marked[firstindex - size] = true;

  for (int i = size; i < shp.positions.size(); i++) {
    shp.positions[i] = shp.positions[i] + params.lenght * shp.normals[i];
  }

  for (int round : range(7)) {
    vec3f root      = shp.positions[firstindex];
    int   rootIndex = firstindex - size;
    float col       = rand1f(rng);
    vec4f randcol   = rgb_to_rgba(vec3f{col, col, col});
    randcol         = rgb_to_rgba(rand3f(rng));

    for (int connections : range(12000)) {

      // Marking of the node used
      marked[rootIndex] = true;

      float minDist  = 10000;
      vec3f minPoint = root;
      int   minIndex = rootIndex;

      // Minimum node search
      for (int i = size; i < shp.positions.size(); i++) {
        if (distance(root, shp.positions[i]) < minDist && i != rootIndex &&
            marked[i - size] == false) {
          minDist  = distance(root, shp.positions[i]);
          minPoint = shp.positions[i];
          minIndex = i - size;
        }
      }

      // Minimum node connecting with root and update the root node with
the minimum node

      vector<vec3f> segment;
      vector<vec4f> color;

      segment.push_back(root);
      segment.push_back(minPoint);
      color.push_back(randcol);
      color.push_back(randcol / 2);

      add_polyline(path, segment, color, 0.0005f);
      root      = minPoint;
      rootIndex = minIndex;
    }
  }
}
```



Folding Fan connections structure

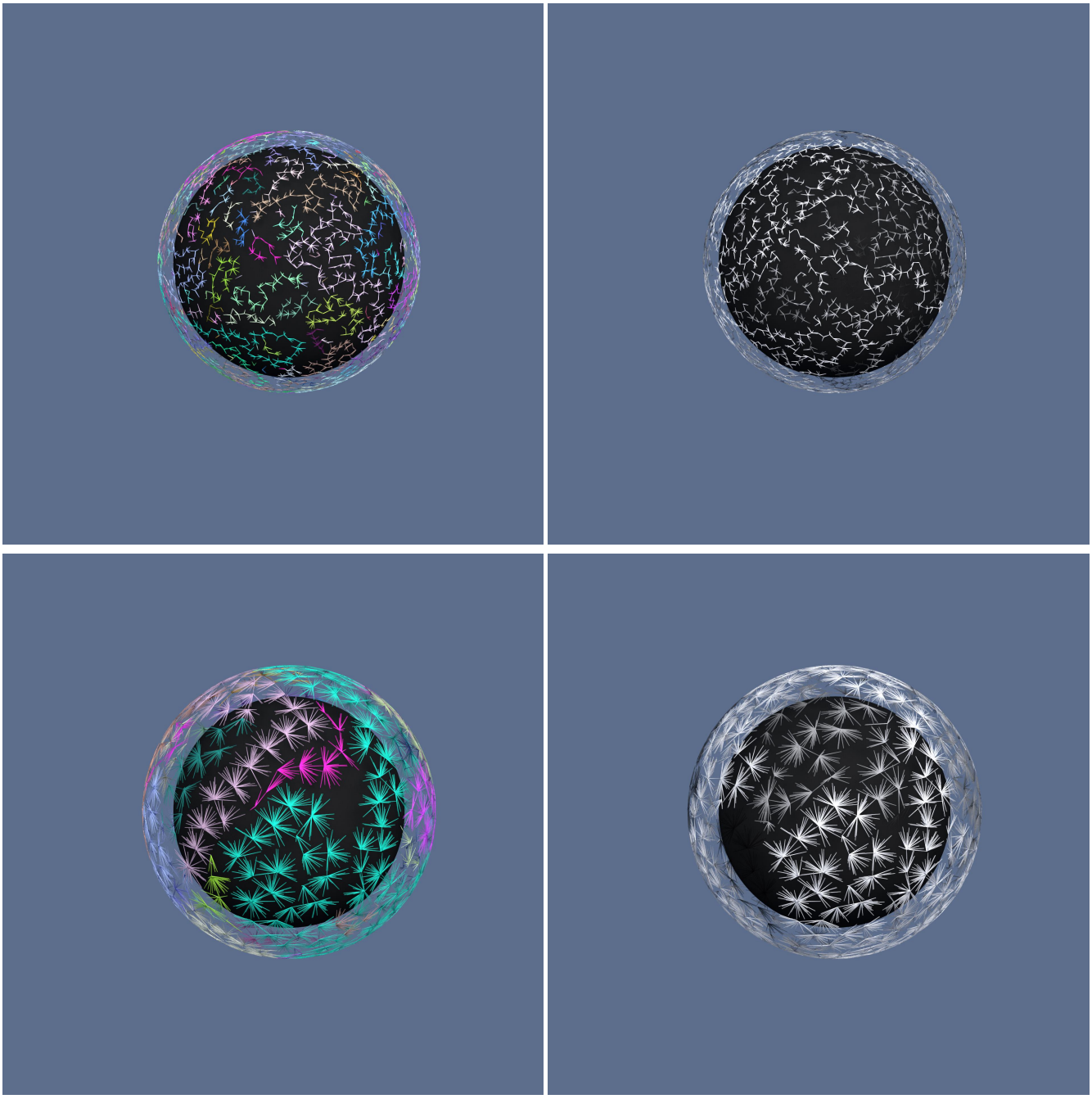
Difficile

L'algoritmo descritto di seguito segue le regole della sezione precedente: partendo da un nodo radice viene generato un percorso. La differenza sostanziale è presente nella scelta del nodo successivo e nel disegno dei nodi selezionati. Infatti, in questo caso, l'algoritmo raccoglie tutti i punti contenuti in un certo *range* e seleziona il più distante. Infine, vengono tracciate tutte le distanze tra i nodi raccolti e la nuova radice diventa il nodo selezionato. La struttura generata presenta più percorsi, dove ogni percorso è costruito da nodi che presentano una forma a ventaglio.

```
void make_connections(shape_data& connections, const shape_data& shape,
    const connections_params& params) {
  shape_data shp  = shape;
  auto       size = shp.positions.size();
  sample_shape(shp.positions, shp.normals, shp.texcoords, shp, params.num);

  rng_state rng = make_rng(172784);

  // SELECT FIRST RANDOM

  int marked[params.num] = {0};

  for (int i = size; i < shp.positions.size(); i++) {
    shp.positions[i] = shp.positions[i] + params.lenght * shp.normals[i];
  }

  for (int round : range(1000)) {
    int   firstindex          = rand1i(rng, shp.positions.size() - size) +
size;
    vec3f root                = shp.positions[firstindex];
    marked[firstindex - size] = 1;
```

```
      float col                 = rand1f(rng);

      // Black & White
      vec4f randcol = rgb_to_rgba(vec3f{col, col, col});

      // Random Color
      // randcol       = rgb_to_rgba(rand3f(rng));

      for (int conn : range(10000)) {
        vector<vec3f> rangedPoints;

        for (int i = size; i < shp.positions.size(); i++) {
          if (distance(root, shp.positions[i]) < 0.005f && marked[i - size]
== 0) {
            rangedPoints.push_back(shp.positions[i]);
            marked[i - size] = 1;
          }
        }

        float dist = 0;
        vec3f maxPoint;
        for (auto point : rangedPoints) {
          if (distance(root, point) > dist) {
            dist     = distance(root, point);
            maxPoint = point;
          }
        }

        for (auto point : rangedPoints) {
          vector<vec3f> segment;
          vector<vec4f> color;

          segment.push_back(root);
          segment.push_back(point);
          color.push_back(randcol);
          color.push_back(randcol / 2);

          add_polyline(connections, segment, color, 0.0002f);
        }

        root = maxPoint;
      }
    }
  }
```

I risultati riportati sono stato ottenuti applicando diversi parametri e diverse colorazioni:



Procedural Voronoi Implementations

Gli algoritmi presentati in quest'ultimo paragrafo sono stati creati utilizzando le regole dei **Diagrammi di Voronoi**. Infatti ognuno di questi presenta la prima porzione di codice identica. Tale parte di codice, si occupa della creazione e del bilanciamento degli insiemi di **Voronoi**, senza i quali non potremmo effettuare le diverse tipologie di modellazione. Il bilanciamento dei diagrammi è tanto accurato quante sono le iterazione ad esso dedicate. Maggiore è il bilanciamento maggiore è la simmetria e la precisione della geometria ottenuta.

Per costruire l'algoritmo è stata creata una struttura dati utile al fine di semplificare la computazione.

```
struct key {
  vec3f       position;
```

```
  vector<int> indexpoints;
  float       maxdistance;

  key(vec3f pos) {
    position    = pos;
    maxdistance = 0;
  }
};
```

Di seguito è riportata la porzione di codice utilizzata per ogni algoritmo, fondamentale per la generazione ed il bilanciamento dei **Diagrammi di Voronoi**.

```
void make_voronoi(shape_data& shape, const voronoi_params& params) {
  rng_state rng = make_rng(218394);

  // Random choice of keypoints
  vector<key> keypoints;
  for (int i : range(shape.positions.size())) {
    float rand = rand1f(rng);
    if (rand < 0.01) {
      keypoints.push_back(shape.positions[i]);
      std::cout << std::to_string(keypoints.size()) + "\n";
    }
  }

  // According to the value of rounds we're going to have more iterations
and more balancing

  for (int round : range(params.rounds)) {

    // Compute voroni diagrams
    for (int i : range(shape.positions.size())) {
      float mindist = 10000;
      int   keyindex;
      for (int k : range(keypoints.size())) {
        if (distance(shape.positions[i], keypoints[k].position) < mindist)
{
          mindist  = distance(shape.positions[i], keypoints[k].position);
          keyindex = k;
        }
      }
      keypoints[keyindex].indexpoints.push_back(i);
    }

    // Update key posiiton with average position

    for (auto k : range(keypoints.size())) {
      vec3f sumpos = {0, 0, 0};
      for (int index : keypoints[k].indexpoints) {
        sumpos += shape.positions[index];
      }
```

```
        keypoints[k].position = sumpos / keypoints[k].indexpoints.size();
      }

      if (round != params.rounds - 1) {
        for (auto k : range(keypoints.size())) {
          keypoints[k].indexpoints.clear();
        }
      }
    }

    // Compute the max distance in every set

    for (auto k : range(keypoints.size())) {
      float maxdist = 0;
      for (int index : keypoints[k].indexpoints) {
        if (distance(shape.positions[index], keypoints[k].position) >
  maxdist) {
          maxdist = distance(shape.positions[index], keypoints[k].position);
        }
      }
      keypoints[k].maxdistance = maxdist;
    }

    // Store the position before the modifications

    vector<vec3f> lastPositions;
    for (int i : range(shape.positions.size())) {
      lastPositions.push_back(shape.positions[i]);
    }

    // Generate procedural models: each type has its own implementation


    ...


    // Colors application
    for (int i : range(shape.positions.size())) {
      shape.colors.push_back(interpolate_line(params.bottom, params.top,
          distance(shape.positions[i], lastPositions[i]) / params.height));
    }
  }
```

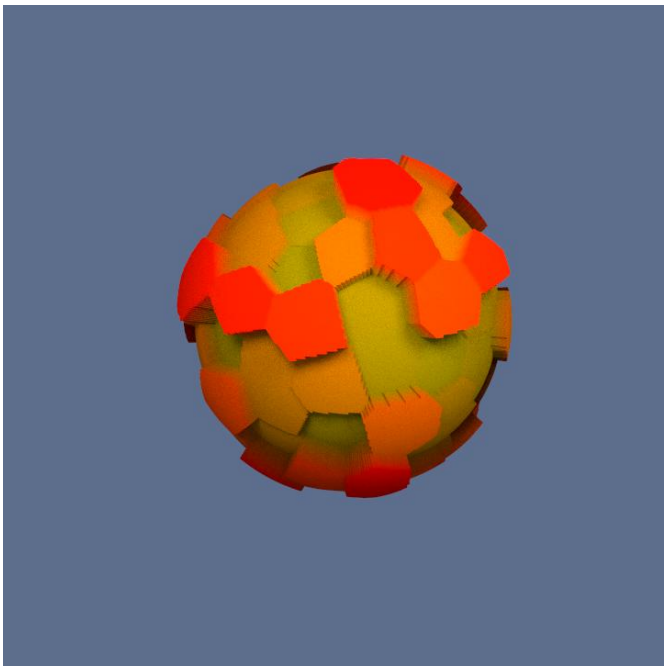
Dopo aver mappato e bilanciato sulla forma i **diagrammi di Voronoi**, l'algoritmo dispone di varie possibilità per sfruttare tale mappatura e generare forme interessanti.

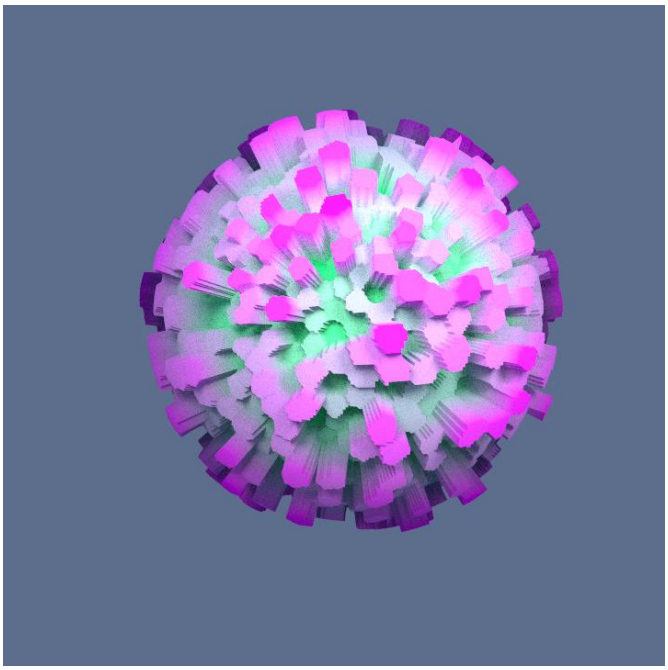
Random Geometric Peaks

Con la prima implementazione, si innalza ogni diagramma di un valore randomico. Così facendo, si formano delle piattaforme di diversa altezza. Come mostrato dai risultati, aumentando il numero di iterazioni, le piattaforme assumono delle forme regolari che tendono ad essere simili tra loro.

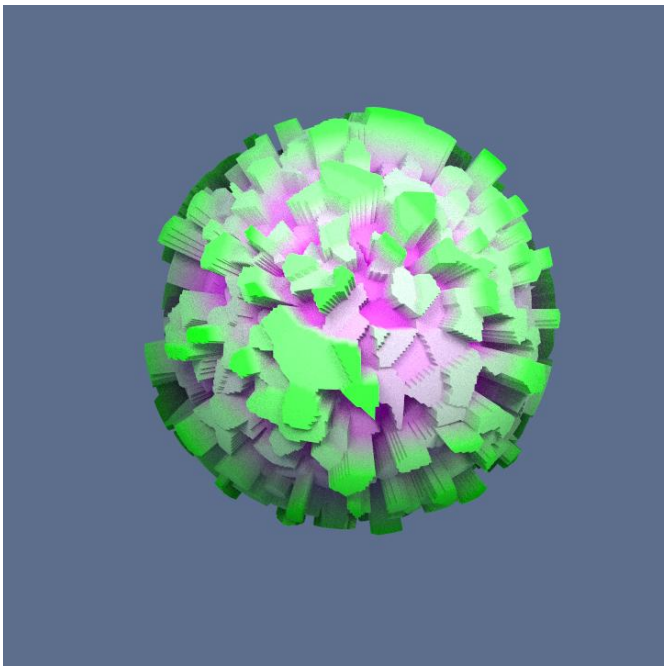
```
if (params.type == "RANDOM HIGH") {
    for (auto k : keypoints) {
      float noise = rand1f(rng);
      for (int index : k.indexpoints) {
        shape.positions[index] = shape.positions[index] +
                                 params.height * noise *
shape.normals[index];
      }
    }
}
```

32 iterazioni:





1 iterazione:



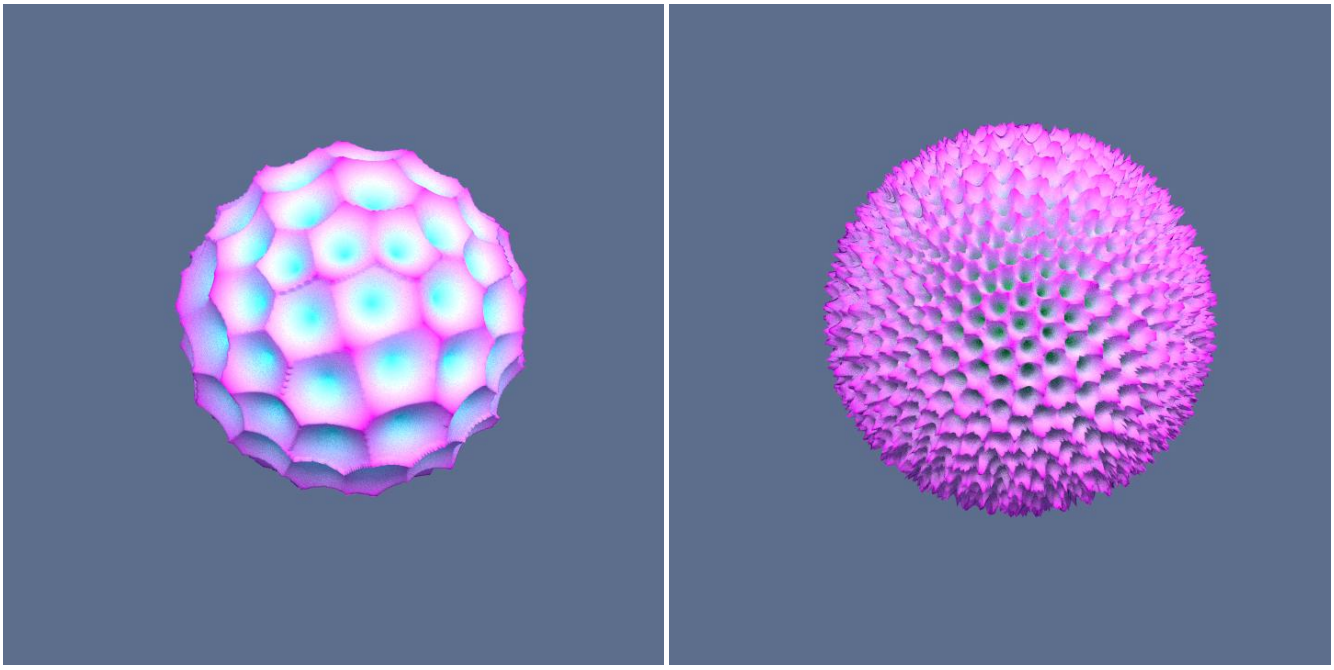
Procedural Nests

La seconda implementazione segue una metodologia lievemente più complessa. Per ogni diagramma, l'algoritmo itera sui punti raccolti e calcola la distanza dal keypoint di riferimento. La distanza calcolata viene utilizzata per innalzare il punto. Più il punto è vicino al keypoint meno viene innalzato. Applicando tale metodologia, si ottenengono delle cavità che tendono a formare delle vette sui bordi.

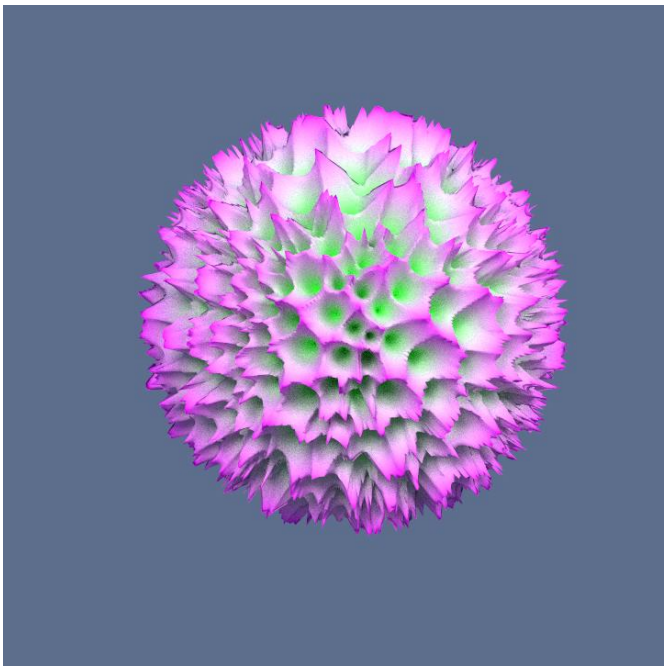
```
} else if (params.type == "HOLES") {
    for (auto k : keypoints) {
      for (int index : k.indexpoints) {
        float noise = distance(shape.positions[index], k.position) /
                      k.maxdistance;

        shape.positions[index] = shape.positions[index] +
                                 params.height * noise *
shape.normals[index];
      }
    }
    quads_normals(shape.normals, shape.quads, shape.positions);
}
```

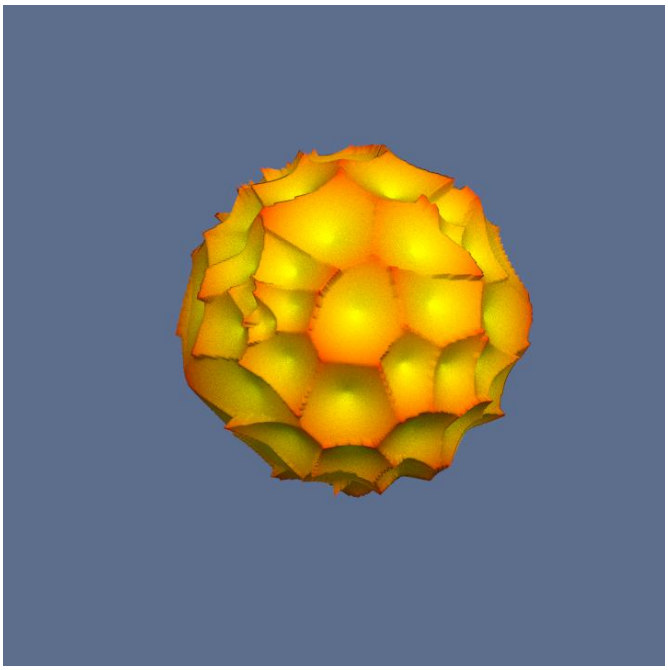
Di seguito sono riportati i risultati ottenuti, applicando tonalità differenti.

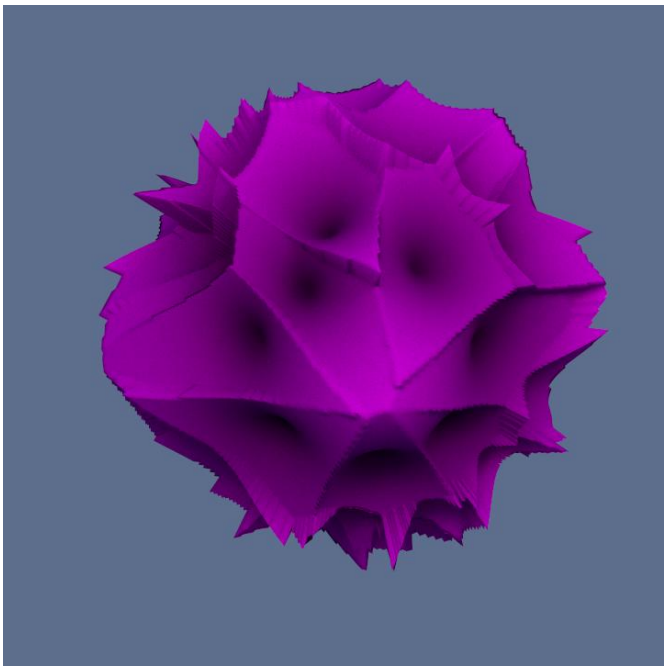
32 iterazioni:



1 iterazione:







Procedural Berries

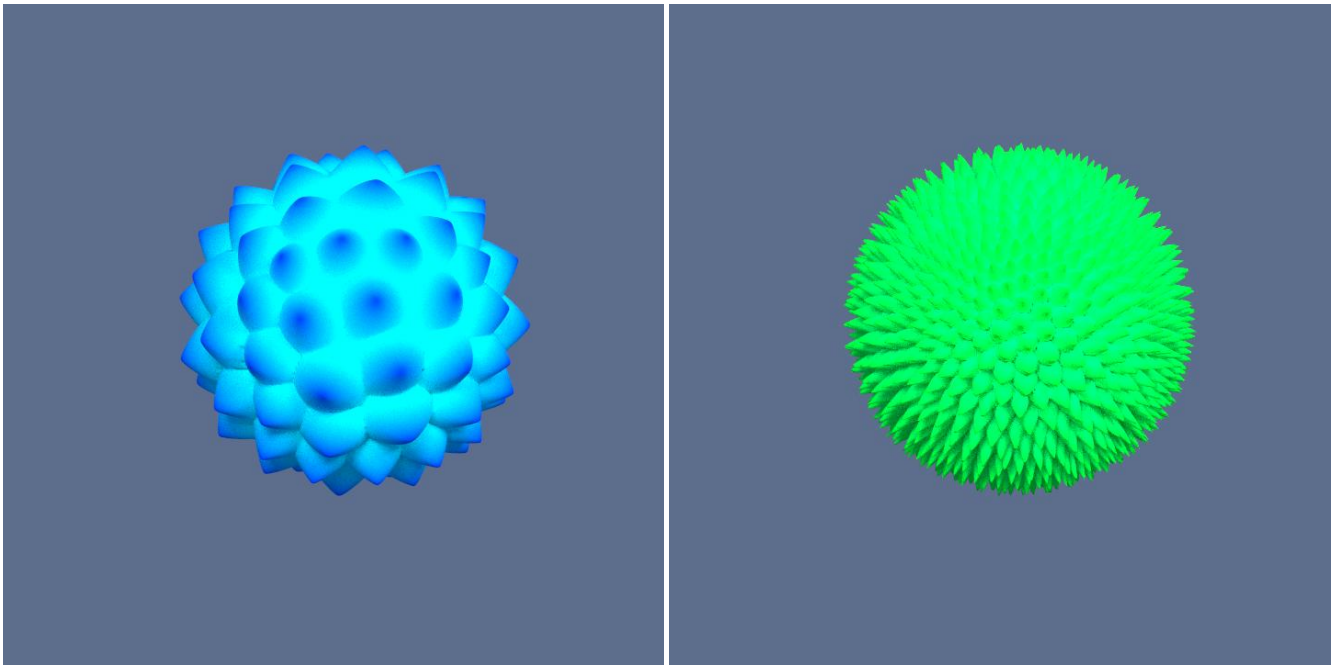
L'ultima implementazione è figlia delle precedente: viene effettuato lo stesso ragionamento invertendo il valore della distanza.

```
} else if (params.type == "BALLS") {
    for (auto k : keypoints) {
      for (int index : k.indexpoints) {
        float noise = distance(shape.positions[index], k.position) /
                      k.maxdistance;

        shape.positions[index] = shape.positions[index] +
                                 params.height * (1 - noise) *
                                     shape.normals[index];
```

```
        }
      }
      quads_normals(shape.normals, shape.quads, shape.positions);
    }
```

Le immagini sono state ottenute variando il numero di iterazioni e la tipologia delle colorazioni.

32 iterazioni:



1 iterazione: