Images: elementi creativi

Lorenzo Piarulli 1884766

A tutte le immagini risultanti viene applicato il passaggio ad srgb, quindi risultano più chiare dell'originale

Weighted Voroni stippling and diagrams

Difficile

In questo paragrafo viene presentato l'algoritmo *Weighted Voronoi Stippling*. Quest'ultimo utilizza gli insiemi di Voronoi per manipolare l'immagine che verrà ricostruita utilizzando punti di nero assoluto. Con questo algoritmo ho generato due rappresentazioni diverse. Entrambe sfruttano la prima parte del codice tramite il quale vengono generati gli insiemi, successivamente si separano generando immagini completamente diverse.

Weighted Voronoi

Nella prima fase dell'algoritmo per prima cosa viene preparata un'immagine bianca delle stesse dimensioni dell'immagine d'origine, che viene desaturata. Subito dopo vengono inseriti dei punti in maniera casuale. Per ottenere una computazione più efficiente la probabilità che un punto venga generato su di un pixel molto scuro è decisamente maggiore rispetto ad uno più chiaro: in questo modo avremo dei punti che rispecchiano il soggetto dell'immagine già dopo la prima fase.

```
if(params.voronoi > 0){
    std::cout << "Voronoi algorithm start... (10 minutes)\n";

    auto resimg = make_image(img.width,img.height,true);
    for (auto i : range(resimg.width)){
      for (auto j : range(resimg.height)){
        resimg[{i,j}] = {1,1,1,1};
        auto c = xyz(img[{i,j}]);
        auto g = (c.x + c.y + c.z)/3;
        c = g + (c - g)*0;
        img[{i,j}] = {c.x, c.y, c.z, img[{i,j}].w};
      }
    }


    POINT vorPoints[20000] = {};
    int counter = 0;
    for (auto i : range(img.width)){
      for(auto j : range(img.height)){

        auto c = xyz(img[{i,j}]);
        auto max = c.x*100+c.y*100+c.z*100;
        auto conf = (int) rand() % (int) max;
```

```
        if (conf < 4 && max <150){
          if(counter >= 20000){
            break;
          }
          POINT p = {i, j,};
          vorPoints[counter] = p;
          counter++;

        }
      }
      if(counter >= 20000){
        break;
      }
    }
```

In questa fase l'algoritmo genera gli insiemi per ogni punto creato. Alla fine della computazione ogni punto avrà un insieme di pixel. Per generare l'immagine del tipo **Diagrams Coloring** la variabile *rounds* sarà posta uguale ad 1 mentre per quella di tipo **Stippling** sarà uguale ad un multiplo di 2 (fissato in questo caso a 8, maggiore è il multiplo più accurata è la rappresentazione). Questo perchè nella versione **Stippling** necessitiamo di più ripetizioni.

```
    int rounds = params.voronoi;

    for(auto time : range(rounds)){
      for (auto i : range(img.width)){
        for(auto j : range(img.height)){
          float min = 1000000;
          int minIndex;
          for(int k : range(counter)){

            auto xpow = std::pow((vorPoints[k].x-i),2);
            auto ypow = std::pow((vorPoints[k].y-j),2);
            auto dist = sqrt(xpow + ypow);
            if(dist < min){
              min = dist;
              minIndex = k;
            }
          }
          vorPoints[minIndex].pixels.push_back({i,j});
        }
      }
```

Dopo aver generato gli insiemi, calcoliamo il nuovo punto di ogni insieme effettuando una media pesata tra tutti i valori di *greyscale* di ogni pixel dell'insieme. Per ottenere maggiore accuratezza, nel caso fossimo interessati a generare la versione **Stippling**, l'algoritmo ripartirà dal calcolo degli insiemi di Voronoi per poi effettuare nuovamente la media pesata.

```
      for(int k : range(counter)){
        float sumx = 0;
        float sumy = 0;
        float sumw = 0;

        for(auto pixel : vorPoints[k].pixels){
          auto c = xyz(img[{pixel.x,pixel.y}]);
          float weigth = 1-(c.x + c.y + c.z)/3;

          sumx += pixel.x * weigth;
          sumy += pixel.y * weigth;
          sumw += weigth;
        }
        if(sumw == 0){
          sumw = 1;
        }
        sumx /= sumw;
        sumy /= sumw;

        vorPoints[k].x = sumx;
        vorPoints[k].y = sumy;
      }
    }
```

In queste ultime due fasi generiamo le due immagini diverse. Nella prima fase coloriamo, gli insiemi di Voronoi utlizzando tonalità casuali. Alla fine del processo avremo generato un'immagine in cui gli insiemi saranno visibili. **DIAGRAMS COLORING**

```
    if(params.voronoi == 1){
      for(int k : range(counter)){
        float rnx = ((double) rand()/RAND_MAX)/2.2;
        float rny = ((double) rand()/RAND_MAX)/2.2;
        float rnz = ((double) rand()/RAND_MAX)/2.2;
        for(vec2f pixel : vorPoints[k].pixels){
          auto c = xyz(img[{pixel.x, pixel.y}]);
          c = gain(c, 1 - 0.3);
          resimg[{pixel.x,pixel.y}] = {c.x-rnx, c.y-rny, c.z-rnz,
img[{pixel.x,pixel.y}].w};
        }
      }
    }
```

Nella seconda fase inseriamo i punti degli insiemi di Voronoi, generando un'immagine bianca dove le forme scure saranno rappresentate da gruppi di punti neri. **STIPPLING**

```
    if(params.voronoi > 1){
      for(int k : range(counter)){

        resimg[{vorPoints[k].x+1,vorPoints[k].y}] = {0,0,0,0};
        resimg[{vorPoints[k].x-1,vorPoints[k].y}] = {0,0,0,0};
        resimg[{vorPoints[k].x,vorPoints[k].y+1}] = {0,0,0,0};
        resimg[{vorPoints[k].x,vorPoints[k].y-1}] = {0,0,0,0};

        resimg[{vorPoints[k].x,vorPoints[k].y}] = {0,0,0,0};

        resimg[{vorPoints[k].x+1,vorPoints[k].y+1}] = {0,0,0,0};
        resimg[{vorPoints[k].x-1,vorPoints[k].y-1}] = {0,0,0,0};
        resimg[{vorPoints[k].x-1,vorPoints[k].y+1}] = {0,0,0,0};
        resimg[{vorPoints[k].x+1,vorPoints[k].y-1}] = {0,0,0,0};

      }
    }

    std::cout << "Voronoi image done.\n";

    return resimg;
  }
```

Vediamo le immagini a confronto: (1. Normale, 2. Diagrams coloring, 3. Stippling):

Weird filters: Experimental vs Splash

Questi ultimi due filtri seguono delle regole diverse dagli altri realizzati fin'ora, e manipolano l'immagine per ottenere una sua rappresentazione completamente creativa. In entrambi i filtri vengono generati degli sfondi, sopra i quali vengono impresse le parti dell'immagine che presentano tonalità più scure.

Experimental

Facile

Nel primo filtro lo sfondo viene generato colorando di nero ogni riga e colonna dei pixel aventi un valore di greyscale molto basso. Dove i pixel sono stati già colorati non viene effettuata alcuna colorazione ulteriore per ottenere un background più minimale.

```
if(params.experimental > 0){
    std::cout << "Mine experimental filter start...\n";

    auto resimg = img;
    for (auto i : range(resimg.width)){
      for (auto j : range(resimg.height)){
        resimg[{i,j}] = {1,1,1,1};
      }
    }

    for(auto i : range(img.width)){
      for(auto j : range(img.height)){
        auto c = xyz(img[{i,j}]);
        auto l = xyz(resimg[{i,j}]);
        if((c.x+c.y+c.z)/3 < 0.4 && (l.x+l.y+l.z)/3 > 0.5){
          for(auto u : range(img.width)){
            auto k = xyz(resimg[{u,j}]);
            resimg[{u,j}] = {k.x-(c.x+c.y+c.z)/9,k.y-(c.x+c.y+c.z)/9,k.z-
(c.x+c.y+c.z)/9,(c.x+c.y+c.z)/3};
          }
          for(auto v : range(img.height)){
            auto k = xyz(resimg[{i,v}]);
            resimg[{i,v}] = {k.x-(c.x+c.y+c.z)/9,k.y-(c.x+c.y+c.z)/9,k.z-
(c.x+c.y+c.z)/9,(c.x+c.y+c.z)/3};
          }
        }
      }
    }
```

Nella sezione seguente dell'algoritmo vengono tracciate le parti più scure in bianco e nero, invertendo la loro colorazione.

```
    for(auto i : range(img.width)){
      for(auto j : range(img.height)){
        auto c = xyz(img[{i,j}]);
        if((c.x+c.y+c.z)/3 < 0.6){
          resimg[{i,j}] = {1-c.x,1-c.y,1-c.z, img[{i,j}].w};
        }
      }
    }

    std::cout << "Experimental filter done.\n";
```

```
    return resimg;
  }
```

Splash filter

Medio

Nel secondo filtro lo sfondo generato è creato mediante una struttura dati apposita:

```
struct CIRCLE{
  int x;
  int y;
  float radius;
};
```

In questo caso la creazione dello sfondo sarà divisa in due parti: la prima genererà tre fasci obliqui di una tonalità di verde. La seconda genererà, utilizzando la struttura descritta in precedenza, dei cerchi viola che sfumeranno gradualmente man mano che ci si allontana dal loro centro. L'algoritmo genererà cerchi che non entrano in collisione, abbastanza distanti da poter coprire tutto lo sfondo.

```
if(params.splash > 0){
    std::cout << "Splash filter start...\n";

    auto resimg = img;
    for (auto i : range(resimg.width)){
      for (auto j : range(resimg.height)){
        resimg[{i,j}] = {0,0,0,0};
      }
    }

    for (auto i : range(resimg.width)){
      for (auto j : range(resimg.height)){
        if(i == j && i != 0){
          resimg[{i,j}] = {0,1,0.6,0};
          for(auto k : range(500)){
            if(j+k <= img.height){
              if(k<100 || (k>200 && k<300) || (k>400 && k<500) ){
                resimg[{i,j+k}] = {0,1,0.4,0};
              }
            }
          }
        }

      }
    }

    std::list<CIRCLE> circleList;

    for(auto k : range(25)){
```

```
      CIRCLE circle;
      bool isgood = false;

      while(!isgood){
        circle.x = (int) rand()% (int) img.width;
        circle.y = (int) rand()% (int) img.height;
        bool done = true;
        for(auto c : circleList){
          float dist = std::sqrt((c.x-circle.x)*(c.x-circle.x) + (c.y-
circle.y)*(c.y-circle.y));
          if(dist < c.radius + 150){
            isgood = false;
            done = false;
            break;
          }

        }
        if(done){
          isgood = true;
        }
      }
      circle.radius =  rand()%100+10;
      circleList.push_back(circle);
    }

    for(auto i : range(img.width)){
      for(auto j : range(img.height)){

        for(auto circle : circleList){
          float dist = std::sqrt((i-circle.x)*(i-circle.x) + (j-circle.y)*
(j-circle.y));


          if(dist > circle.radius+125){
            continue;
          } else if( dist <= circle.radius) {
            float k = circle.radius/2;
            if(dist < k){
              resimg[{i,j}] = {1,0,0.6,0};
            } else {
              float prob  = (float) rand()/RAND_MAX;
              if(prob < 0.3){
                resimg[{i,j}] = {1,0,0.6,0};
              }
            }
          } else {
            float prob  = (float) rand()/RAND_MAX;
            if(prob <= 0.03){
              resimg[{i,j}] = {1,0,0.6,0};
            }
          }
        }
```

```
    }
  }
```

Dopo aver generato lo sfondo l'algoritmo imprime le parti più scure invertendo la colorazione e applicando un filtro blu.

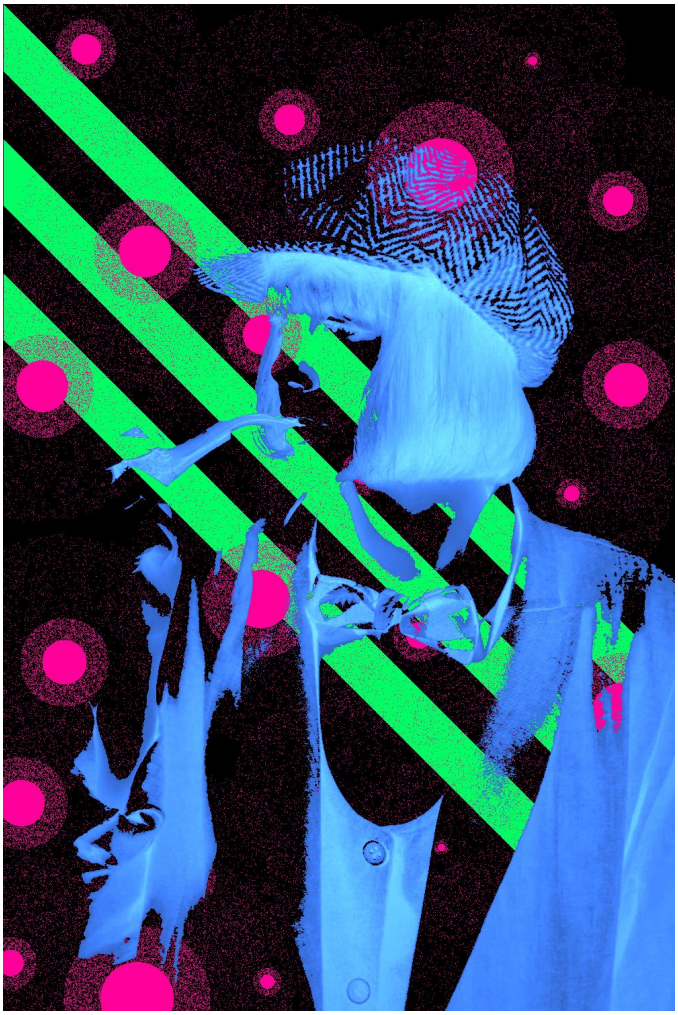
```
  for(auto i : range(img.width)){

    for(auto j : range(img.height)){

      auto c = xyz(img[{i,j}]);
      if((c.x+c.y+c.z)/3 < 0.6){
        resimg[{i,j}] = {(1-c.x)/2,(1-c.y),(1-c.z)*2, img[{i,j}].w};
      }
    }
  }

  std::cout << "Splash filter done.\n";
  return resimg;
}
```

Di seguito sono riportate le immagini ottenute applicando l'algoritmo: (1. Experimental, 2. Splash)



Average Blur vs Gaussian Blur

L'implementazione delle due tipologie di *blur* è molto simile. La differenza più importante è implicata dall'utilizzo di un kernel di ordine variabile nel Gaussian Blur. I risultati sono abbastanza differenti: si può notare che il Gaussian Blur effettua una sfocatura molto più accurata mantenendo maggiormente i dettagli dall'immagine originale.

Gaussian blur

Medio

In questa prima parte, si impostano le varie variabili e si crea il kernel utilizzando la funzione della distribuzione gaussiana.

```
if(params.gaussianblur > 0){
    std::cout << "Gaussian Blur start...\n";

    float sigma = 1.0;
    auto finalimg = img;


    int ord = params.gaussianblur;

    if(ord%2 == 0){
      ord++;
    }

    int ordquadro = ord*ord;
    int ordmid = ((ord-1)/2);

    float kernel[ord][ord];

    for(auto i : range(ord)){
      for(auto j : range(ord)){
        kernel[i][j] =  expf((-1*(((i-j)*(i-j))))/(2*(sigma*sigma)));
      }
    }
```

Qui si effettua il vero processo di sfocatura applicando la matrice di convoluzione, creata in precedenza, sulle sottomatrici che compongono l'immagine.

```
    for(auto i : range(img.width)){
      for(auto j : range(img.height)){
        if ( i<ordmid || i>=img.width-ordmid || j<ordmid || j>=img.height-
ordmid){
          finalimg[{i,j}] = img[{i,j}];
        } else {

          int ki = 0;
```

```
            float cx = 0;
            float cy = 0;
            float cz = 0;
            for (auto u = -ordmid; u <= ordmid; ++u){
              int kj = 0;
              for (auto v = -ordmid; v <= ordmid; ++v){
                auto c = xyz(img[{i+u,j+v}]);

                cx += c.x * kernel[ki][kj];
                cy += c.y * kernel[ki][kj];
                cz += c.z * kernel[ki][kj];
                kj++;
              }
              ki++;
            }
            finalimg[{i,j}] = {(cx/ordquadro)*ordmid, (cy/ordquadro)*ordmid,
(cz/ordquadro)*ordmid, img[{i,j}].w};
          }
        }
      }
      std::cout << "Gaussian blurred image done.\n";
      return finalimg;

    }
```

Average blur

Facile

In quest'ultimo non vi è la creazione di alcun kernel. Si effettua invece la media dei valori nelle sottomatrici dell'immagine.

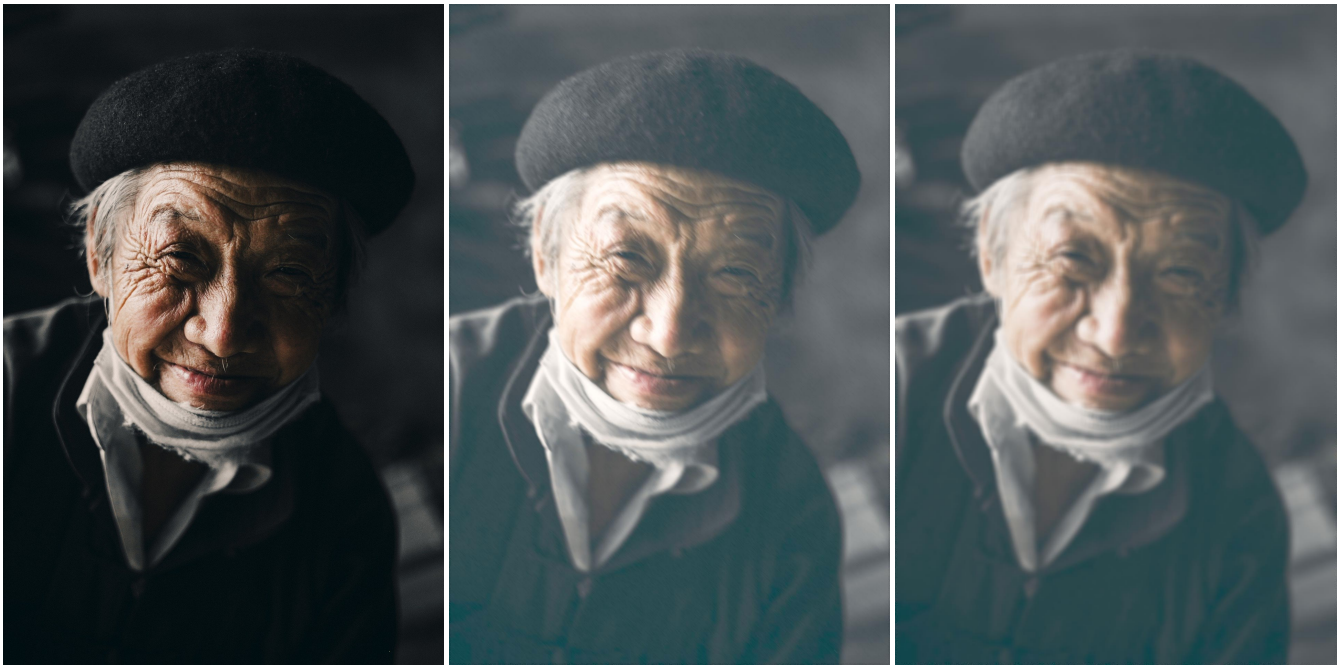
```
if(params.averageblur > 0){
    auto resimg = img;
    auto blur = params.averageblur;
    std::cout << "Average blur algorithm with "+std::to_string(blur)+"
starting..\n";
    for(auto i : range(img.width)){
      for(auto j : range(img.height)){
        if ( i<blur || i>=img.width-blur || j < blur || j>=img.height-blur)
{
          resimg[{i,j}] = img[{i,j}];
        } else {
          float sommax = 0;
          float sommay = 0;
          float sommaz = 0;
          for (auto u = -blur; u <= blur; ++u){
            for (auto v = -blur; v <= blur; ++v){
              auto c = xyz(img[{i+u,j+v}]);
              sommax += c.x;
              sommay += c.y;
```

```
                sommaz += c.z;
              }
            }
            int div = std::pow((blur*2)+1,2);
            resimg[{i,j}] = {sommax/div, sommay/div, sommaz/div,
img[{i,j}].w};
          }
        }
      }
      std::cout << "Average blurred image done.\n";

      return resimg;

    }
```

Vediamo le immagini a confronto: (1. Normale, 2. Gaussian Blur, 3. Average Blur)



Pop-art filter

Facile

Quest'algoritmo genera, a partire da un'immagine iniziale, un'altra immagine con stesse dimensioni e composta da quattro versioni dell'originale, ognuna con una patina di colore applicata.

```
if(params.popart > 0){
    std::cout << "Popart algorithm start...\n";

    auto finalimg = img;
    img = resize_image(img, img.width/2, img.height/2);

    for (auto i : range(img.width)){
      for (auto j : range(img.height)){
```

```
        set_pixel(finalimg,i,j,
{img[{i,j}].x+0.2,img[{i,j}].y,img[{i,j}].z});
        set_pixel(finalimg,i+img.width,j,
{img[{i,j}].x,img[{i,j}].y+0.2,img[{i,j}].z});
        set_pixel(finalimg,i,j+img.height,
{img[{i,j}].x,img[{i,j}].y,img[{i,j}].z+1});
        set_pixel(finalimg,i+img.width,j+img.height, {img[{i,j}].x+0.2,
img[{i,j}].y+0.2, img[{i,j}].z});

      }
    }

    std::cout << "Popart image done.\n";
    return finalimg;

  }
```

L'immagine risultante sarà la seguente: