CNAM – Département Informatique

MUX101 – Multimédia et interaction humain-machine

B.Martin

ED6 – Techniques graphiques 2D

On souhaite réaliser une application de rendu procédural 2D en **Processing**, basée sur le principe des **Signed Distance Functions (SDF)**.
Une SDF associe à chaque point du plan une distance signée à une forme : négative à l'intérieur, positive à l'extérieur, nulle sur le contour.

L'objectif de cet exercice est de manipuler le rendu pixel par pixel, la géométrie implicite et l'interaction utilisateur.

L'objectif est de manipuler le rendu pixel par pixel et l'interaction utilisateur. Une classe *Tableau* est fournie. Elle gère : le rendu pixel par pixel une liste de formes SDF la composition de plusieurs formes (union) le mélange des couleurs associé, elle utilisée telle quelle comme base de travail.

1. **Implémentation des SDF :**
    - Implémenter une fonction pour tracer un cercle dans la classe SDFShape (voir https://iquilezles.org/articles/distfunctions2d/ pour inspiration)
    - Ajouter des contrôles utilisateur pour bouger, ajouter et supprimer les cercles
    - Ajouter un contrôle pour modifier la force du lissage entre les shapes
2. **Pour aller plus loin**
    - Ajouter un contrôle pour modifier le type d'union
    - Restructurer le code pour permettre à une forme d'être de différent type (cercle, triangle, …)
    - Ajouter des contrôles pour ajouter plusieurs type de shape.
3. **Consignes :**
    - Organiser le code sous forme de **classes**
    - Utiliser controlP5 pour l'interface utilisateur

Annexe :

```
enum UnionType {
  INTERSECTION_LISSE {
    float apply(float d1, float d2, float k) {
      float h = max(k - abs(d1 - d2), 0.0) / k;
      return max(d1, d2) + h*h*k*0.25;
    }
  }
  , SOUSTRACTION_LISSE {
    float apply(float d1, float d2, float k) {
      float h = max(k - abs(-d1 - d2), 0.0) / k;
      return max(d1, -d2) + h*h*k*0.25;
    }
  }
  ,
    UNION_LISSE {
    float apply(float d1, float d2, float k) {
      float h = max(k - abs(d1 - d2), 0.0) / k;
      return min(d1, d2) - h * h * k * 0.25;
    }
  };

  abstract float apply(float d1, float d2, float k);
}

public class Tableau {
  private final MainApp ctx; // le context

  public UnionType unionType = UnionType.UNION_LISSE;

  // Liste des formes à dessiner
  private final ArrayList<SDFShape> shapes = new ArrayList<>();

  public Tableau(MainApp ctx) {
    this.ctx = ctx;

    reset();
  }

  // Simple fonction reset
  // Bien regarder de quoi sont composer les objects.
  public void reset() {
    shapes.clear();
    shapes.add(new SDFShape(SDFType.CIRCLE, new float[]{-.1f, .1f, .15f}, color(50, 50, 255)));
    shapes.add(new SDFShape(SDFType.CIRCLE, new float[]{0f, -.1f, .15f}, color(255, 255, 255)));
    shapes.add(new SDFShape(SDFType.CIRCLE, new float[]{.1f, .1f, .15f}, color(255, 50, 50)));
  }

  // fonction draw à lancer dans le draw du ctx principale
  void draw() {
    ctx.loadPixels();

    int w = ctx.width;
    int h = ctx.height;
    if (ctx.pixels.length != w * h) return; // éviter les pb en cas de redimensionnement de la fenêtre

    // Si aucune shape, remplir avec du noir (ou transparent)
    if (shapes.isEmpty()) {
```

```
    for (int i = 0; i < ctx.pixels.length; i++) {
      ctx.pixels[i] = color(0); // noir
    }
    ctx.updatePixels();
    return;
  }

  for (int y = 0; y < h; y++) {
    for (int x = 0; x < w; x++) {
      int i = x + y * w;
      float px = (x - w/2f) / (float)h;
      float py = (y - h/2f) / (float)h;
      // Init avec la première shape
      float d = shapes.get(0).compute(px, py);
      color c = shapes.get(0).myColor;
      // Toutes les shapes avec color blending
      for (int s = 1; s < shapes.size(); s++) {
        SDFShape shape = shapes.get(s);
        float d2 = shape.compute(px, py);
        // Blend factor for smooth union
        float k = ctx.lissage;
        float t = constrain(0.5f + 0.5f * (d2 - d) / k, 0.0f, 1.0f);
        // Blend colors
        c = lerpColor(c, shape.myColor, 1.0f - t);
        // Apply union
        d = unionType.apply(d, d2, ctx.lissage);
      }
      // Shading
      float shade = map(d, -0.01f, 0.01f, 1, 0);
      shade = constrain(shade, 0, 1);
      ctx.pixels[i] = color(
        red(c) * shade,
        green(c) * shade,
        blue(c) * shade
        );
    }
  }
  ctx.updatePixels();
 }
}
```

J’ai choisi cet exercice parce qu’il est visuel et permet de découvrir (j’espère) d’autre façon d’afficher des formes via la programmation.

Il permet de découvrir le principe des SDF sans passer par des langages de shaders parfois compliqué, tout en manipulant un rendu procédural calculé pixel par pixel.