Projet d'algorithme avancée

Triangulation d'un polygone convexe

Thomas CROIZET
Gurwan DELAUNAY

École Nationale Supérieure des Sciences Appliquées et de Technologie
Vendredi 6 Janvier 2023

Table des matières

1 Introduction

La triangulation de polygone est un problème très courant dans l'infographie. En effet, la triangulation de polygone est très utilisée dans le rendu 2D et 3D. Les triangles sont les primitives de base afin de créer des objets en 2D ou en 3D plus complexes. Cette technique permet de créer des jeux à mondes ouverts très immersifs ou des films avec des effets spéciaux toujours plus réalistes.

L'avantage d'utiliser des triangles pour former un polygone complexe en 2D est la suivante :

1. Un gain de place en mémoire

Néanmoins, en 3D nous avons deux avantages :

1. Un gain de place en mémoire
2. Un triangle est toujours une surface plane

En effet, une figure avec 4 côté n'est pas nécessairement plane. Pour rendre ce type de polygone, les calculs sont considérablement difficile et coûteux. L'avantage de trianguler les polygones, c'est qu'un triangle est toujours une figure plane. On peut donc, suivant une bonne triangulation, créer toute sorte de figure 3D. Il suffira d'incliner les différents triangles dans des plans différents pour créer un relief en 3D.

Tous les polygones dessinées à des fins d'explications seront des polygones réguliers. En effet, ceux-ci sont bien plus simple à tracer et évitent de nombreux problèmes.

2 Questions préliminaires

2.1 Question 1

La figure 1 nous montre le nombre d'arc différent dans un carré.

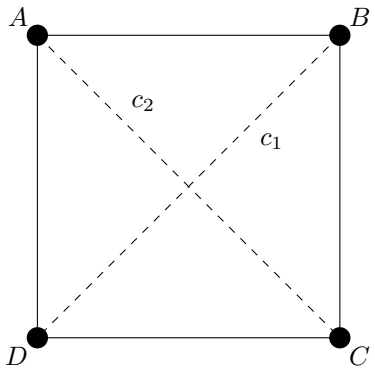

FIGURE 1 – Nombre de cordes différentes dans un carré

D'après la figure 1, nous avons 2 cordes différentes. La figure 2 montre le nombre de cordes différentes dans un pentagone.

D'après la figure 2 nous avons 5 cordes différentes. On en déduit la formule pour trouver le nombre de cordes différentes qui est :

$$C = \frac{n \times (n-3)}{2}$$

Le $(n-3)$ provient du fait que nous supprimons un triangle. En effet, un triangle ne peut être trianguler, on retire ses cordes qui sont invalides. On peut vérifier que cette formule est bonne pour le carré et le pentagone :

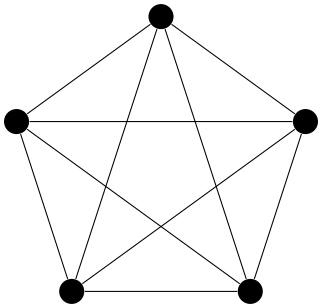

FIGURE 2 – Nombre de cordes différentes dans un pentagone

$$\text{Carré} = \frac{4 \times 1}{2} = 2$$
$$\text{Pentagone} = \frac{5 \times 2}{2} = 5$$

On retrouve bien les résultats que nous avons observés sur les figures

2.2 Question 2

Pour $n = 3$, le polygone est un triangle et il n'y a qu'une seule triangulation possible, qui ne contient pas de cordes. Pour $n = 4$, si on prend un carré (ou un rectangle), d'après la figure 3 nous avons 2 triangulations possible.

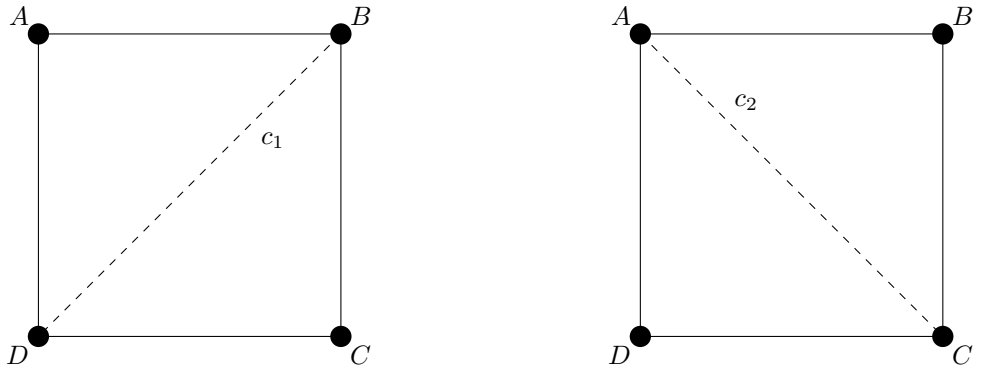

FIGURE 3 – Triangulation possible pour un carré

Supposons maintenant que pour tout polygone à m sommets avec toutes les triangulations ont le même nombre de cordes. Considérons un polygone à n sommets avec $n > m$.

On peut choisir un sommet quelconque du polygone et dessiner une corde reliant ce sommet à un sommet qui n'est pas adjaçent. On obtient deux polygones à $\frac{n}{2}$ sommets. Ceci divise le polygone en deux polygones plus petits, qui ont chacun moins de sommets que le polygone original. En appliquant notre hypothèse de récurrence, ces polygones ont le même nombre de cordes pour toutes leurs triangulations.

Ensuite, on peut construire une autre triangulation à partir des petits polygones que l'on a créer avec l'ajout de la corde. Comme chaque polygone plus petit a le même nombre de triangulations, et que la corde ajoutée ne crée pas de triangle supplémentaire, chaque triangulation possible du polygone original est obtenue de cette manière. Ainsi, toutes les triangulations du polygone original contiennent la même corde, et il s'ensuit que toutes les triangulations ont le même nombre de cordes.

Par conséquent, on peut conclure que toutes les triangulations d'un polygone à n sommets ont le même nombre de cordes.

3 Essais successifs

3.1 Question 1

Dans tous nos codes, nous supposerons l'existence d'une structure Polygon avec comme champs :

— n : le nombre de côté du polygone
— summits : le tableau des sommets du polygone. Un sommet est un couple (x, y) de coordonnées
— arcs : le tableau des cordes du polygone. Une corde est un couple (i, j) d'indice de sommet du polygone

De plus, nous supposerons quelques autres fonctions en lien avec le projet définie dans la section 8.1 :

— getAllArcs(polygon) : une fonction retournant toutes les cordes possible d'un polygone
— distance(p1, p2) : une fonction retournant la distance euclidienne entre 2 points

La fonction dans le code est nommée : arcIsValid.

```
# On suppose une fonction : inArray(item, array) retournant true si item est dans array, sinon false

# Vérifie si une corde est valide pour le polygone. C'est-à-dire qu'elle ne forme pas le polygone, n'existe pas déjà et n'intersecte pas une autre corde.
Function arcIsValid(polygon, i, j):
    # Si les sommets sont identiques
    if i == j:
        return false

    # Si l'arc compose la figure (i.e est entre deux sommets consécutifs)
    if (i - 1) mod polygon.n == j or (j - 1) mod polygon.n == i:
        return false

    # Si l'arc est un arc déjà existant
    if inArray((i, j), polygon.arcs) or inArray((i, j), polygon.arcs):
        return false

    # Vérifie les intersections
    for k, l in polygon.arcs:
        # Si un des sommets compose les deux arcs on continue l'itération
        if i == k or j == k or i == l or j == l:
            continue

        if doIntersect(polygon.summits[i], polygon.summits[j], polygon.summits[k], polygon.summits[l]):
            return false

    return true

# Cette fonction nous permet de vérifier que deux lignes s'intersectent entre elles en utilisant la règle de Cramer pour résoudre un système linéaire à deux équations.
Function doIntersect(p1, q1, p2, q2):
    s1_x = q1[0] - p1[0]
    s1_y = q1[1] - p1[1]
    s2_x = q2[0] - p2[0]
    s2_y = q2[1] - p2[1]

    s = (-s1_y * (p1[0] - p2[0]) + s1_x * (p1[1] - p2[1])) / (-s2_x * s1_y + s1_x * s2_y)
    t = ( s2_x * (p1[1] - p2[1]) - s2_y * (p1[0] - p2[0])) / (-s2_x * s1_y + s1_x * s2_y)
```

```
37
38      if s >= 0 and s <= 1 and t >= 0 and t <= 1:
39          return true
40
41      return false
```

3.2 Question 2

On considère un algorithme par essais successifs basé sur la stratégie suivante : à l'étape i, on trace l'une des cordes valides issues de Si ou on ne trace rien.

En supposant cet algorithme sans la validation de cordes, nous pouvons calculer plusieurs fois la même triangulation en partant d'un sommet différent compris dans le même triangle. Cependant une fois qu'une triangulation est effectuée, il n'est pas possible de la refaire en partant de l'hypothèse que nous utilisons dans cet algorithme la fonction "validecorde". En effet, si une corde de la triangulation a déjà été tracée par l'algorithme, elle ne sera pas retracée. En commençant par le premier sommet et en traçant la triangle, l'algorithme va empêcher un nouveau calcul du même triangle.

Cette méthode ne permet donc pas d'obtenir toutes les triangulations car la fonction "validecorde" va l'en empêcher. Cet algorithme se retrouve donc inefficace

3.3 Question 3

On veut maintenant pouvoir construire toute triangulation en une seule fois.

3.3.1 Programme

En utilisant notre structure de donnée Polygon définie en 3.1, on peut utiliser comme stratégie :

1. Si le polygone est un triangle, dans ce cas on renvoi les cordes reliant les sommets entre eux
2. On explore toutes les cordes possibles
3. Si la corde est valide on l'ajoute aux cordes du polygone alors :
   (a) On divise le polygone en 2 sous-polygone gauche et droit
   (b) Puis nous ré-appliquons la méthode triangulation sur ces deux sous-polygones.
   (c) On retourne un tableau des cordes tracés via la fonction des deux sous-polygones
4. Sinon, on retourne un tableau vide car le polygone est déjà triangulé

```
1  # On suppose une fonction : append(array, item) ajoutant item à la fin de array
2  # On suppose une fonction : length(array) retournant la longueur d'un tableau
3
4  # Triangule le polygone
5  Function triangulation_essais_successifs(polygon, polygonAllArcs, polygonArcsCount,
   ↪  polygonSummits):
6      # Tant qu'on a pas tracé tous les arcs
7      if summitIndex < polygon.arcsMax:
8          triangulation_essais_successifs(polygon, polygonAllArcs, polygonArcsCount,
           ↪  summitIndex + 1)
9
10     if arcIsValid(polygon, polygonAllArcs[summitIndex][0],
       ↪  polygonAllArcs[summitIndex][1]):
11         # On ajoute l'arc à la triangulation
12         append(polygon.arcs, polygonAllArcs[summitIndex])
13
14         if polygonArcsCount == polygon.n - 3:
```

```
            # On retourne les arcs de la triangulation
            return polygon.arcs

        if summitIndex < polygon.arcsMax:
            # On a tracé une corde qui forme le polygone
            triangulation_essais_successifs(polygon, polygonAllArcs,
            ↪  length(polygon.arcs) + 1, summitIndex + 1)

    return polygon.arcs
```

3.3.2 Complexité

Pour chaque appel de cette fonction, nous engendrons au maximum :

$$\begin{cases} T(m) & = 2^m \\ m & = \frac{n \times (n-3)}{2} \text{ M est le nombre maximum de cordes traçable} \end{cases}$$

Ceci provient du fait que nous allons tester toutes les combinaisons de corde possible à chaque appel. Néanmoins, l'algorithme ne calculant pas toutes les combinaisons, cette complexité est majoré. Avec la formule donnée pour M on a comme complexité :

$$\Theta(2^m) = \Theta(2^{\frac{n \times (n-3)}{2}}) = \Theta(2^{n^2})$$

3.3.3 Jeux d'essai

Afin de prouver l'efficacité et le caractère exact de cet algorithme par essais successifs, nous avons effectué plusieurs tests avec des polygones de différentes tailles.

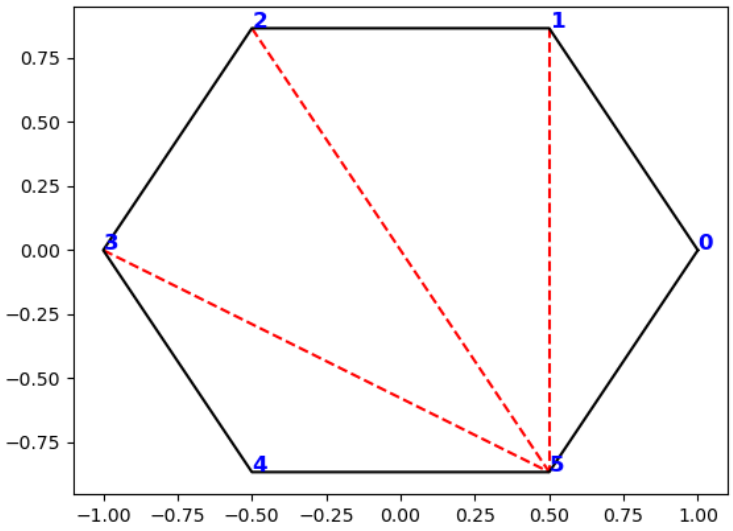

FIGURE 4 – Résultat de l'algorithme utilisant les essais successifs sur un hexagone régulier

On peut remarquer dans le cadre de cet algorithme par essais successifs, que la triangulation se fait à partir d'un unique sommet et on peut légitimement douter que cette technique permette une triangulation minimale. Les distances euclidiennes totales des arcs de triangulation sont respectivement :

— Pour la figure 4 : 5,46

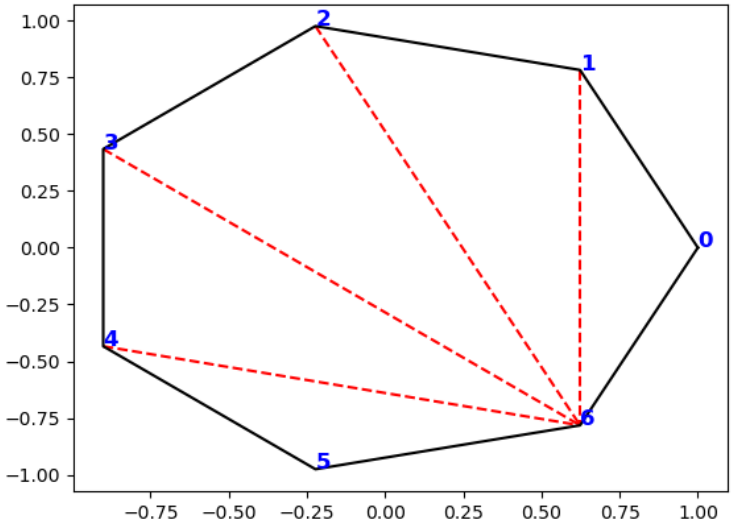

FIGURE 5 – Résultat de l'algorithme utilisant les essais successifs sur un heptagone régulier

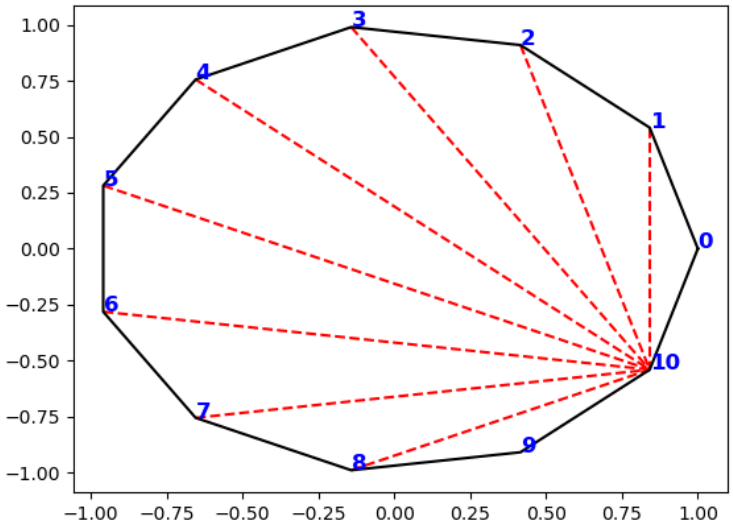

FIGURE 6 – Résultat de l'algorithme utilisant les essais successifs sur un polygone hendécagone régulier (polygone à 11 côtés)

— Pour la figure 5 : 7,02
— Pour la figure 6 : 12,78

4 Programmation dynamique

4.1 Question 1

Nous cherchons à trianguler des polygones. Nous voulons que cette triangulation soit optimale : que la longueur des cordes soit minimale. Pour trouver le taille optimale de la triangulation, nous savons que la formule de récurrence sera de la forme : $T_{i,t} = \min(f)$.

Pour chaque calcul, nous devons déterminer si la corde est valide dans le polygone que l'on considère et sa longueur. Nous remplirons un tableau $n \times n$ où n est le nombre de sommet avec le longueur de chaque corde traçable à partir du sommet. La longueur d'une corde correspond à la distance entre deux sommets. Nous obtenons la formule suivante :

$$\forall i, j \in \{1, ..., n\}, \text{arcLength}[i, j] = \text{distance}(s_i, s_j)$$

s_i et s_j sont les sommets du polygone à l'indice i et j. Réalisons le tableau pour la figure 7.

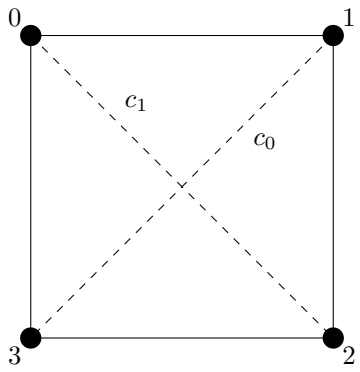

FIGURE 7 – Application de l'algorithme de programmation dynamique sur un carré

Le tableau 1 montre le tableau résultant.

	0	1	2	3
0	-1	-1	distance$(0, 2)$	-1
1	-1	-1	-1	distance$(1, 3)$
2	distance$(2, 0)$	-1	-1	-1
3	-1	distance$(3, 1)$	-1	-1

TABLE 1 – Tableau résultant de l'algorithme de programmation dynamique sur un carré

A la lecture de ce tableau :

1. Toutes les cordes formant le polygone ont -1 comme longueur
2. Sur les deux cordes traçables, nous voyons la même corde 2 fois (on voit celle formé par le couple (i, j) et (j, i), or celles-ci sont strictement identiques)

Pour la mise en équation du problème, on prend le sous-problème de taille de t débutant au sommet s_i. Pour un triangle, nous avons la formule suivante :

$$\forall i < t, \forall t \in \{1, 2, 3\}, T_{i,t} = 0$$

Pour les autres cas, avec les informations de l'énonce nous avons comme formule :

$$\begin{cases} T_{i,t} &= \text{arcLength}[s_i, s_{i+t-2}] + \text{arcLength}[s_i, s_{i+t-1}] + \text{arcLength}[s_i, s_{i+t-2}] + T_{i,t-1} \\ T_{i,t} &= \text{arcLength}[s_i, s_{i+t-1}] + \text{arcLength}[s_{i+1}, s_{i+t-1}] + \text{arcLength}[s_i, s_{i+1}] + T_{i+1,t-1} \\ T_{i,t} &= \text{arcLength}[s_i, s_{i+k}] + \text{arcLength}[s_{i+k}, s_{i+t-1}] + T_{i,k+1} + T_{i+k,t-k} \end{cases}$$

La figure 8 montre l'algorithme de programmation dynamique appliquée sur un polygone quelconque.

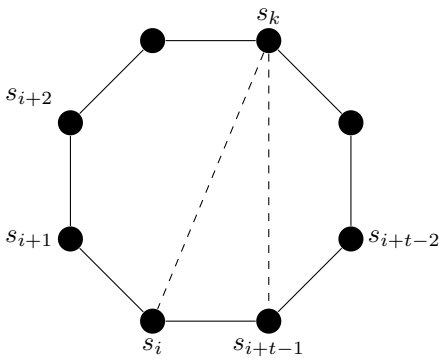

FIGURE 8 – Application de l'algorithme de programmation dynamique sur un polygone quelconque

Nous voyons que pour calculer la triangulation du polygone, nous avons besoin des résultats suivants :

— $T_{i,k+1}$
— $T_{i+k,t-k}$

En faisant varier les indices modulo le nombre de sommets pour éviter un indice trop grand, on prend k sur l'intervalle : $[i+1, i+t-2]$. Deux cas se dégagent :

1. $k = 1$: On est sur une arrête du polygone et donc $T_{i,2} = -1$ et $\text{arcLength}[s_i, s_{i+1}] = -1$
2. $k = 2$: On est sur une arrête du polygone et donc $T_{i+t-2,2} = -1$ et $\text{arcLength}[s_i, s_{i+t-2}] = -1$

Nous trouvons donc la formule de récurrence suivante :

$$\begin{cases} T_{i,t} &= \min\limits_{\forall k \in [2,t-3]} \{\text{arcLength}[s_i, s_{(i+k) \ mod) \ n}] + \text{arcLength}[s_{(i+k)) \ mod) \ n}, s_{(i+t-1)) \ mod) \ n}] + T_{i,k+1} + T_{i+k,t-k}\} \\ T_{i,t} &= \text{arcLength}[s_{i+1}, s_{i+t-1}] + T_{i+1,t-1}, \ k = 1 \\ T_{i,t} &= \text{arcLength}[s_i, s_{i+t-2}] + T_{i,t-2+1}, \ k = 2 \end{cases}$$

4.2 Question 2

Nous allons implémenter l'algorithme décrit dans la question 1 et dans l'énoncé. Nous allons construire notre tableau des longueurs des cordes puis notre tableau lié à la méthode de la programmation dynamique : T. Nous appliquons ensuite la formule de récurrence que nous avons trouvé dans la question 1 et si la corde est valide nous enregistrons le résultat.

```
# On suppose une fonction : append(array, item) ajoutant item à la fin de array
# On suppose une fonction : length(array) retournant la longueur d'un tableau

Function triangulation_dynamique(polygon):
    arcLength = []
    for i start 0 to polygon.n:
        tmp = []
        for j start 0 to polygon.n:
            append(tmp, -1)
```

```
10          append(arcLength, tmp)
11
12      allArcs = getAllArcs(polygon)
13
14      for arc in allArcs:
15          i,j = arc
16          arcLength[i][j] = distance(polygon.summits[i], polygon.summits[j])
17          arcLength[j][i] = arcLength[i][j]
18
19      T = []
20      for i start 0 to polygon.n:
21          tmp = []
22          for j start 0 to polygon.n:
23              append(tmp, 0)
24          append(T, tmp)
25
26      for i start 4 to polygon.n + 1:
27         for j start 0 to polygon.n:
28              for k start 1 to i - 1 :
29                 opti = inf
30                 min = arcLength[j][(j + k) % polygon.n] + arcLength[(j + k) %
                    ↪  polygon.n][(j + i - 1) % polygon.n] + T[j][k + 1] + T[(j + k) %
                    ↪  polygon.n][i - k]
31
32                 if min < opti:
33                     opti = min
34                     T[j][i - 1] = k
35
36         for j start 0 to polygon.n:
37              k = T[j][i - 1]
38              t = (j + k) % polygon.n
39
40              if arcIsValid(polygon, j, t):
41                  append(polygon.arcs, (j, t))
42
43
44      return polygon.arcs
```

4.2.1 Jeux d'essai

Afin de prouver l'efficacité et le caractère exact de cet algorithme par programmation dynamique nous avons effectué plusieurs tests avec des polygones de différentes tailles.

Les résultats obtenus par programmation dynamique sont significativement différents de ceux obtenus par essais successifs. La triangulation des polygones construit maintenant un grand triangle principal et plusieurs petits faisant le tour du polygone. Cette technique semble plus optimale. Les distances euclidiennes totales des arcs de triangulation sont respectivement :

— Pour la figure 9 : 5,19
— Pour la figure 10 : 6,64
— Pour la figure 11 : 10,55

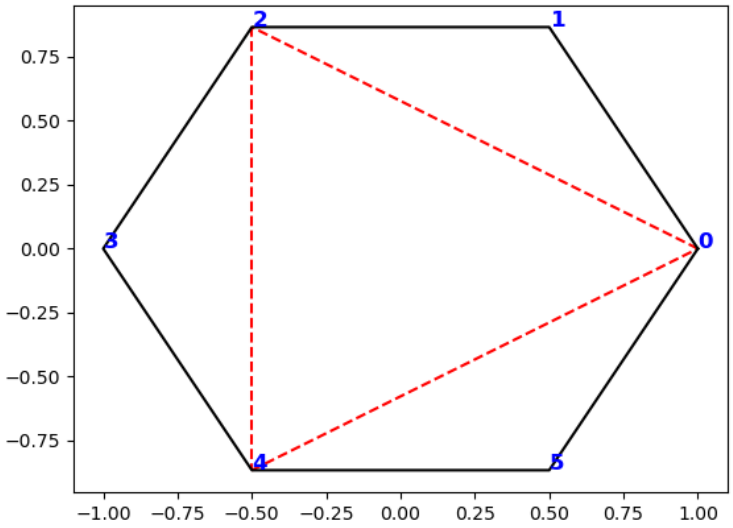

FIGURE 9 – Résultat de l'algorithme utilisant la programmation dynamique sur un hexagone régulier

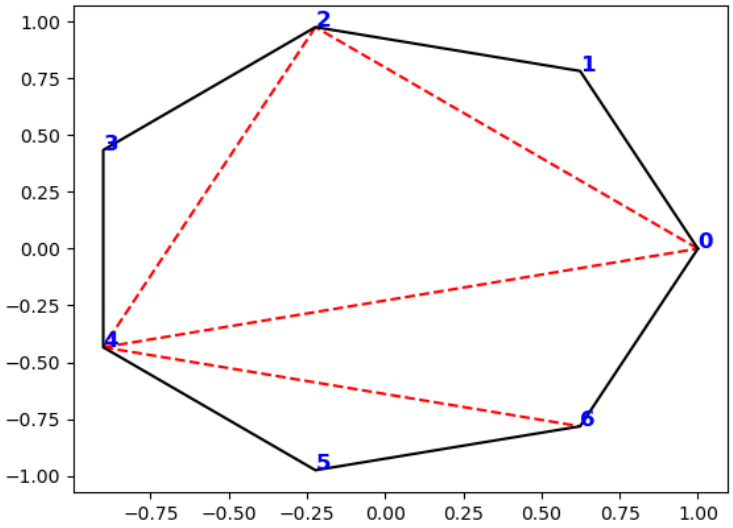

FIGURE 10 – Résultat de l'algorithme utilisant la programmation dynamique sur un heptagone régulier

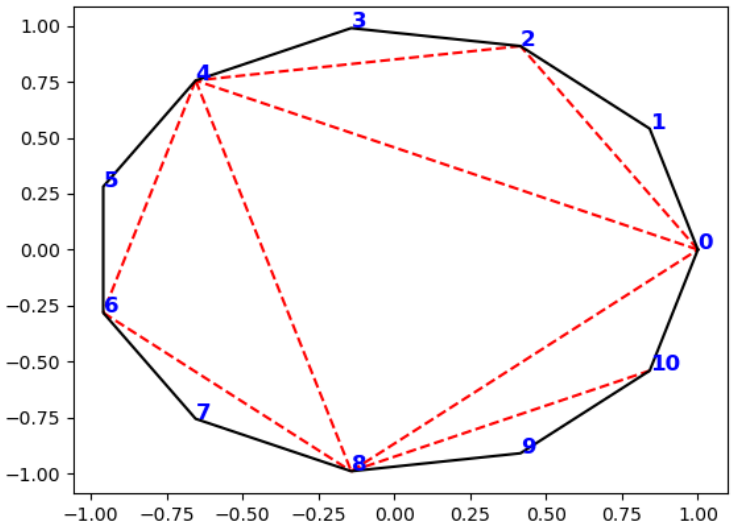

FIGURE 11 – Résultat de l'algorithme utilisant la programmation dynamique sur un polygone hendécagone régulier (polygone à 11 côtés)

4.3 Question 3

4.3.1 Complexité spatiale

Tout algorithme de programmation dynamique réalise du calcul tabulaire. Ici nous avons un tableau à 2 dimensions dont chacune des dimensions est de $n =$ length(polygon.summits). La complexité spatiale est donc en $\Theta(n^2)$.

4.3.2 Complexité temporelle

Par une simple analyse, notre algorithme possède 3 boucles imbriquées. La complexité temporelle semble être en $\Theta(n^3)$.

4.4 Question 4

L'intérêt d'avoir un sommet commun pour décomposer ce problème en 2 sous-problèmes est de nous assurer que nous couvrirons l'ensemble du polygone car le sommet de départ sert à décomposer le problème en 2 sous-problèmes. Si nous tirerons une corde du polygone au hasard nous devrons nous assurer :

1. Que les deux cordes ne s'intersectent pas
2. Que les deux cordes relient deux sommets consécutifs (qu'elle ne servent pas à tracer le polygone)

Pour vérifier ces conditions, nous devrons appeler à chaque fois la fonction : arcIsValid.

La figure 12 montre une exécution possible de cet algorithme sur un polygon quelconque.

D'après la figure 12 nous observons que nous avons 3 sous-polygones au lieu de 2. Ces polygones deviennent les nouveaux sous-problèmes. Nous pourrons modifier notre algorithme précédent pour prendre en compte les 3 sous-problèmes au lieu de 2 en réalisant quelque chose comme ceci :

$$T_{i,t} = \min\{T_{i,i+k} + T_{j,j+l} + T_{\text{nouveau\_polygone}} + \text{coût(cordes\_tracées)}\}$$

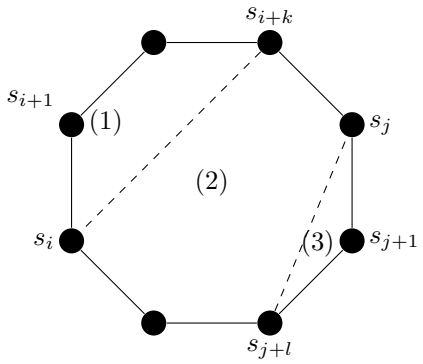

FIGURE 12 – Application de l'algorithme de programmation dynamique en tirant 2 cordes aléatoires sur un polygone quelconque

Néanmoins, comme nous considérons les sommets successivement dans l'ancien algorithme, un de ces polygones n'a pas tous ses côtés consécutifs. Lors de l'appel pour $T_{\text{nouveau\_polygone}}$ nous devrons le considérer comme un nouveau polygone à part entière et non un sous-polygone. C'est-à-dire que nous avons divisé le problème en 2 sous-problèmes et 1 nouveau problème.

5 Algorithme glouton

5.1 Question 1

L'implémentation par algorithme glouton va nous permettre de trianguler certains polygones compatibles avec cette méthode. L'idée est de créer des triangles dans le polygone en cherchant l'arc extérieur le plus court.

```
# On suppose une fonction : append(array, item) ajoutant item à la fin de array
# On suppose une fonction : length(array) retournant la longueur d'un tableau

Function triangulation_glouton(polygon):
    allArcs = getAllArcs(polygon)
    allArcsLen = length(allArcs)

    while allArcsLen > 0:
        shortest = null

        for arc in allArcs:
            if arcIsValid(polygon, arc[0], arc[1]):
                if shortest == null or distance(polygon.summits[arc[0]],
                ↪  polygon.summits[arc[1]]) < distance(polygon.summits[shortest[0]],
                ↪  polygon.summits[shortest[1]]):
                    shortest = arc

        if shortest == null:
            break

        append(polygon.arcs, shortest)

    return polygon.arcs
```

5.1.1 Jeux d'essai

Afin de prouver l'efficacité et le caractère exact de cet algorithme par algorithme glouton, nous avons effectué plusieurs tests avec des polygones de différentes tailles.

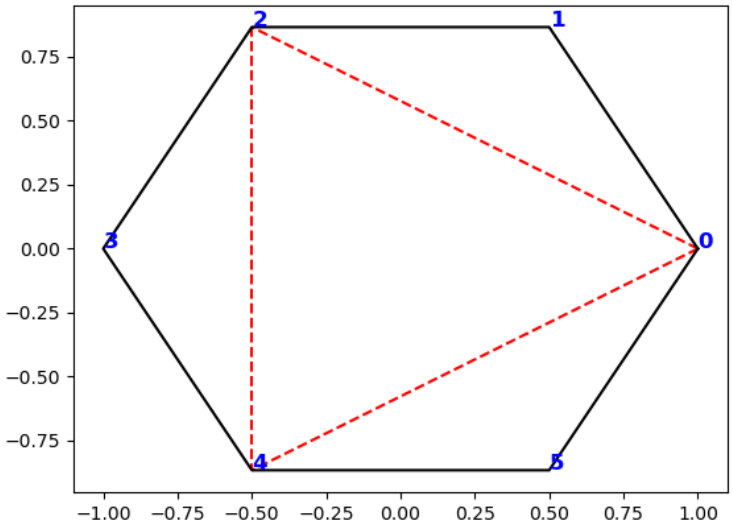

FIGURE 13 – Résultat de l'algorithme utilisant un algorithme glouton sur un hexagone régulier

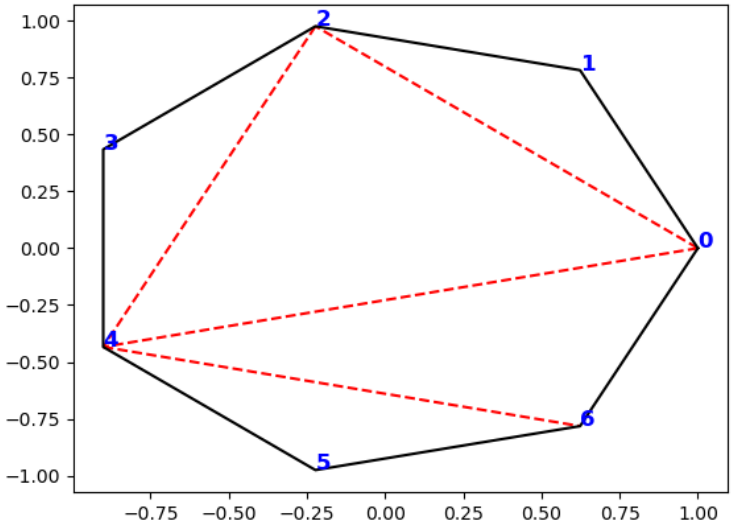

FIGURE 14 – Résultat de l'algorithme utilisant un algorithme glouton sur un heptagone régulier

L'algorithme glouton triangule les polygones d'une façon qui peut nous rappeler la programmation dynamique. Pour l'hexagone et l'heptagone, les triangulations sont identiques entre les deux programmes. Dans ces cas là, il serait préférable d'utiliser l'algorithme glouton qui est plus efficace et rapide que celui par programmation dynamique.

Pour l'hendécagone, cela semble différent et la triangulation est construite de manière parallèle entre la partie droite et gauche du polygone. On peut penser que cette triangulation n'est pas minimale et que dans ce cas, il semblerait plus judicieux d'utiliser la programmation dynamique afin d'obtenir un résultat optimal.

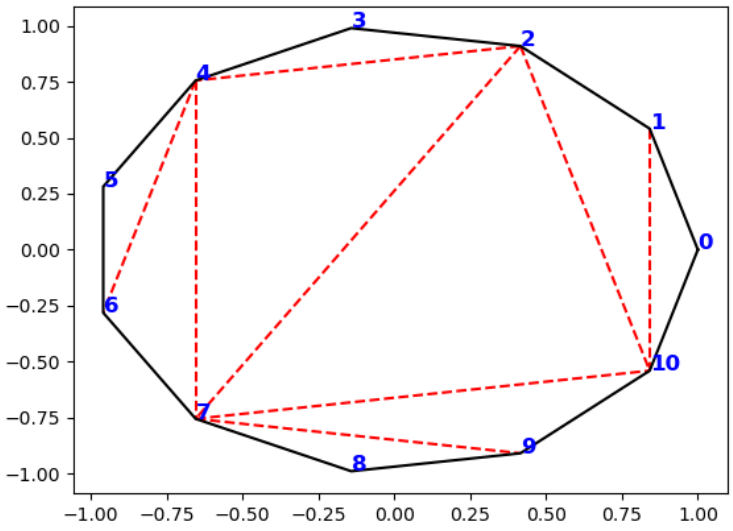

FIGURE 15 – Résultat de l'algorithme utilisant un algorithme glouton sur un polygone hendécagone régulier (polygone à 11 côtés)

— Pour la figure 13 : 5,19
— Pour la figure 14 : 6,64
— Pour la figure 15 : 10,83

5.2 Question 2

En recherchant la corde la plus courte pour réaliser la triangulation, nous observons vite que cette méthode n'est pas optimale pour de nombreux polygones. Néanmoins, elle reste optimale pour une sous-famille de polygones : les polygones réguliers.

En effet, dans un polygone régulier, tous les côtés ont la même longueur, donc ils ont tous des côtés extérieur minimal. L'algorithme est donc optimale pour cette famille de polygone.

De plus, cet algorithme est optimal pour une sous-famille de polygone : ceux ayant des côtés extérieur de longueur croissantes. Pour cette famille de polygone, l'algorithme trouvera le côté le plus court nommée i. En itérant, le côté extérieur $i+1$ sera le plus court, et ainsi de suite. La triangulation se fait de manière optimale car il a juste à trouver le côté le plus court une fois puis itérer naturellement sur les côtés extérieur suivant.

6 Recommandation argumentée

Afin de résoudre ce problème, nous pouvons avoir à utiliser différents algorithmes pensés de manières différentes les uns des autres. Lorsque le polygone à trianguler est compatible avec l'algorithme glouton alors ce programme sera le plus rapide et le plus efficace pour résoudre le problème.

Cependant il arrive que le polygone ne soit pas compatible de manière optimale avec cet algorithme et nous pourrions donc avoir à utiliser un autre type de programme comme l'algorithme par programmation dynamique. En effet, les algorithmes par programmation dynamique ou par essais successifs (backtracking) sont plus précis et permettent donc de résoudre une plus large palette de polygone de manière optimale.

7 Conclusion

Pour conclure, la triangulation de polygone est un problème complexe. Nous avons actuellement 3 implémentations différentes, fournissant des résultats différents avec des complexités différentes. Cette technique étant massivement utilisées en infographie, il est nécessaire de trouver une technique toujours optimale et rapide.

On peut rapprocher nos techniques avec l'algorithme de la Triangulation de Delaunay permettant de trianguler un polygone dans n'importe quelle dimension de efficacement. Néanmoins, il existe plusieurs techniques pour le faire, la meilleur possédant une complexité temporelle en $\Theta(n \log(n))$.

Pour en revenir à nos algorithmes, voici les différentes complexités que nous avons dans le meilleur des cas :

1. Algorithme par essais successifs :
   — Complexité temporelle : $\Theta(2^{n^2})$
   — Complexité spatiale : $\Theta(1)$
2. Algorithme par programmation dynamique :
   — Complexité temporelle : $\Theta(n^3)$
   — Complexité spatiale : $\Theta(n^2)$
3. Algorithme glouton :
   — Complexité temporelle : minimum $\Theta(n)$, moyenne $\Theta(n^2)$ par une simple analyse
   — Complexité spatiale : $\Theta(1)$

L'algorithme par essais successifs est bien trop lent pour être utilisé par sa complexité temporelle. Il nous reste l'algorithme par programmation dynamique et l'algorithme glouton. L'algorithme par programmation dynamique fournira toujours un résultat optimal. L'algorithme glouton peut, mais que pour un nombre réduit de polygone que nous ne connaissons pas.

Néanmoins, l'algorithme peut nous fournir un résultat du premier coup : c'est-à-dire en $\Theta(n)$. De plus cette solution est forcément optimale. La complexité spatiale de l'algorithme par programmation dynamique l'handicape pour des grands polygones.

Pour conclure, si nous cherchons une solution optimale, l'algorithme par programmation dynamique semble être le choix à prendre. Néanmoins, dans des cas pratique, l'algorithme glouton semble être à privilégier car il peut fournir un résultat bien plus rapidement que l'algorithme par programmation dynamique tout en étant assez proche. Il faudrait pour cela s'intéresser à sa complexité en moyenne et dans le pire des cas ce qui est hors du cadre de ce projet.

8 Annexe

8.1 Fonctions auxiliaires

```
# On suppose l'existance d'une fonction : append(array, item) ajourtant à la fin du tableau item
Function getAllArcs(polygon):
    arcs = []

    for i start 0 to polygon.n:
        for j start i + 2  to polygon.n:
            append(arcs, (i, j))

    return arcs

# On suppose l'existance d'une fonction : sqrt(x) retournant la racine carée de x
# On suppose l'exustance d'une fonction : pow(x, n) retournant la puissance x à la puissance n
```

```
Function distance(p1, p2):
    return sqrt(pow(p1[0] - p2[0], 2) + pow(p1[1] - p2[1], 2))
```