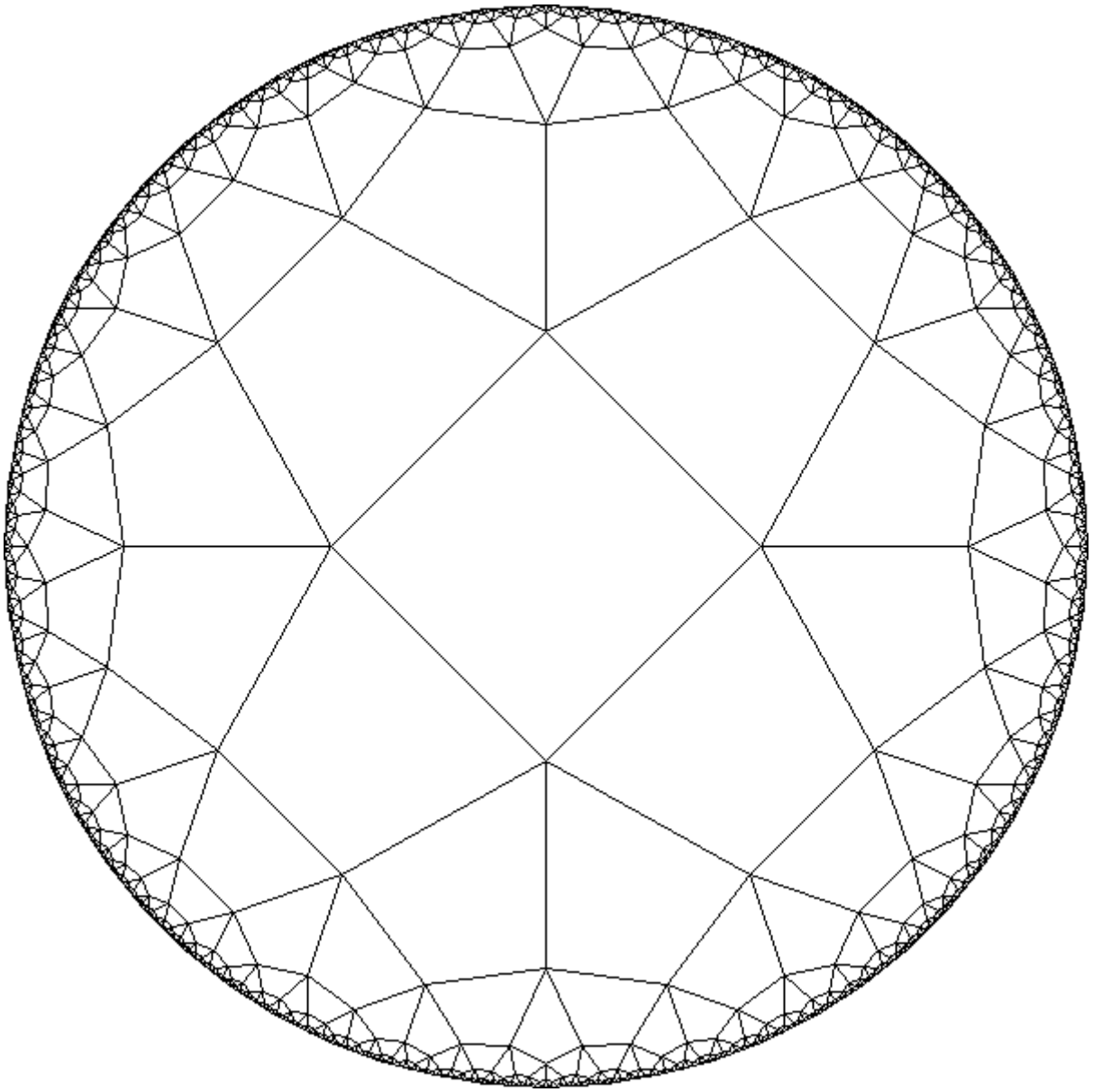
Snake Hyperbolique

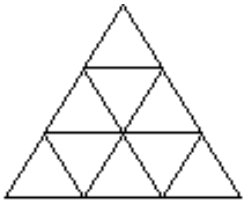
Terrisse Lysandre – Terminale 5



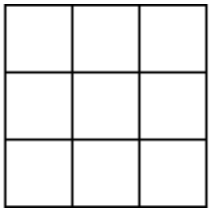
Sommaire

Introduction

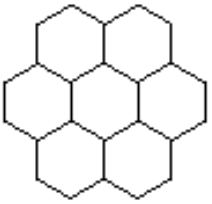
La majorité des jeux vidéos se jouent sur des pavages. En **géométrie euclidienne**, si on veut **paver le plan en utilisant qu'un seul polygone régulier**, on a trois possibilités : le pavage triangulaire, le pavage carré, et le pavage hexagonal. **Chaque pavage peut être représenté selon le couple (m, n)**, où m représente le nombre de côtés par polygone, et n représente le nombre de polygones par coin.



Pavage triangulaire
(3, 6)

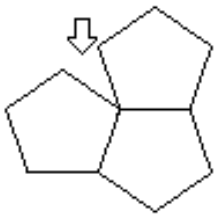


Pavage carré
(4, 4)



Pavage hexagonal
(6, 3)

Cette limitation de trois pavages fait qu'on ne peut pas paver le plan avec certains polygones, comme le pentagone régulier. Cela implique aussi que dans un pavage régulier, il n'y a que 3, 4, ou 6 cases adjacentes, ce qui **réduit le nombre de directions possibles** dans les jeux où l'on ne peut se déplacer que d'une case à une autre, comme Snake.



Pavage pentagonal impossible

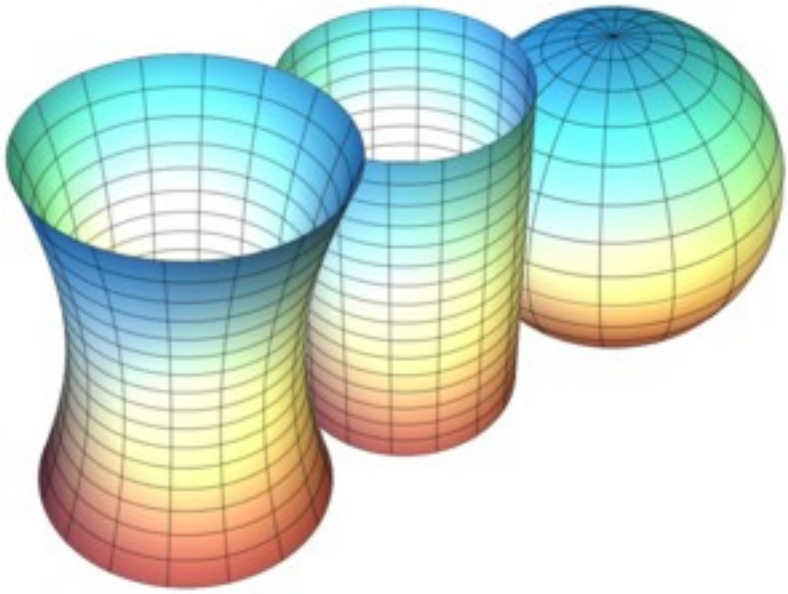
Et si cette limitation n'était pas incontournable ? Si on pouvait paver le plan avec d'autres polygones réguliers ? Pour cela, il faut sortir de la géométrie euclidienne. En 300 avant notre ère, **Euclide proposa cinq vérités géométriques** sur lesquelles baser n'importe quelle autre vérité géométrique. Les quatre premiers axiomes sont très simples et ressemblent à des définitions :

- Un segment peut être tracé en reliant deux points distincts
- Un segment peut être prolongé indéfiniment
- Un cercle peut être tracé en prenant un segment et une de ses extrémités comme rayon et centre
- Tous les angles droits peuvent se superposer

Cependant, le cinquième axiome ne ressemble pas à une définition. Cet axiome dit qu'à partir d'une droite et d'un point en dehors ce celle-ci, il n'existe qu'une seule droite parallèle à la première passant par ce point.

Le cinquième axiome d'Euclide

Les mathématiciens ont pendant longtemps essayé de déduire cet axiome à partir des quatre autres, mais depuis **on a découvert des géométries où les quatre premiers axiomes sont vérifiés mais où le cinquième est faux** : la géométrie sphérique, et la géométrie hyperbolique.



La géométrie hyperbolique, la géométrie euclidienne, et la géométrie sphérique

En **géométrie sphérique**, toutes les droites finissent par s'intersecter, il n'y a donc pas de droites parallèles. Cela laisse un peu plus d'options pour paver le plan sphérique : trois triangles par coin, quatre triangles par coin, cinq triangles par coin, trois carrés par coin, et trois pentagones par coin.

(3, 3)

(3, 4)

(3, 5)

(4, 3)

(5, 3)

Quand on veut représenter un plan sphérique sur un écran, donc sur un plan euclidien, on a plusieurs choix possibles. On peut faire une représentation en deux dimensions d'une sphère en trois dimensions (comme ci-dessus), mais il y aura toujours la moitié de la sphère cachée. C'est pour cela que les géographes préfèrent utiliser des cartes, et qu'ils ont créés des projections de la sphère pendant des millénaires. La plus célèbre est la projection de Mercator, mais il y en a aussi d'autres comme la projection conique et la projection azimutale.

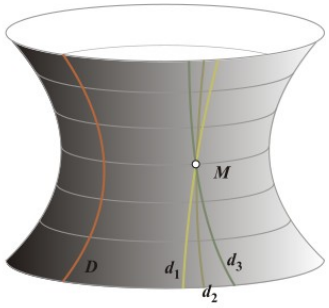
Projection de Mercator

Projection conique

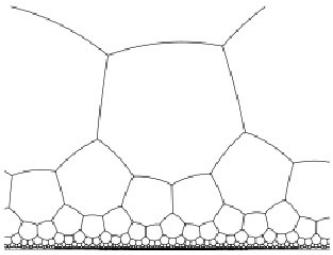
Projection azimutale

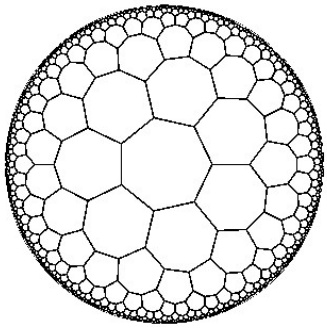
Mais la géométrie la plus étonnante est la **géométrie hyperbolique**. En géométrie hyperbolique, il y a une infinité de droites parallèles à toute droite. Ainsi, n'importe quel pavage (m, n) est possible tant que **1/m + 1/n < 1/2**. Cela veut dire que non seulement on peut paver le plan hyperbolique avec n'importe quel polygone régulier, mais il existe une infinité de pavages en n'utilisant que ce polygone.

Comme pour la géométrie sphérique, il y a plusieurs moyens de représenter un plan hyperbolique sur un plan euclidien. Les plus utilisés sont la représentation en trois dimensions, le demi-plan de Poincaré, et le disque de Poincaré.



La représentation en 3D


Le demi-plan de Poincaré


Le disque de Poincaré

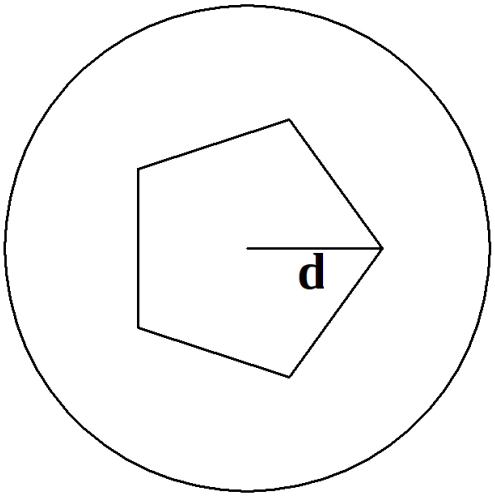
Mon but pour ce projet est de **programmer en Python un jeu Snake dans le disque de Poincaré.** Pour le rendu graphique, j'utiliserai Turtle, car Turtle m'a semblé plus rapide que Tkinter pour tracer des droites. À chaque fois que l'on passera d'une case à une autre, la case d'arrivée se recentrera. Si l'on va vers une case déjà remplie par le Snake, ou si on traverse le bord du pavage (puisque le pavage est fini et ne boucle pas sur lui même), alors on aura perdu. Si on va sur une case contenant un fruit, le Snake grandit d'une case et un nouveau fruit sera placé. Le jeu est gagné lorsque aucune case n'est libre. **On pourra modifier les options du pavage dans un menu en appuyant sur Échap**. Pour se déplacer, on devra **pointer la souris vers la case voulue**.

Ce projet peut être divisé en **quatre parties** qui peuvent être représentées comme ceci :

- Dessiner n'importe quel pavage hyperbolique avec le premier polygone au centre.
- Bouger les polygones.
- Passer d'une case à l'autre
- Implémenter le Snake

Première partie : Le pavage centré

Dans un plan euclidien, si un polygone régulier peut paver le plan, alors il pourra toujours le faire même si on change sa taille. Cependant, **dans un plan hyperbolique, la taille des polygones compte**. Dans un pavage (m, n), la distance entre le centre du disque et un coin du polygone central peut être calculée comme ceci :

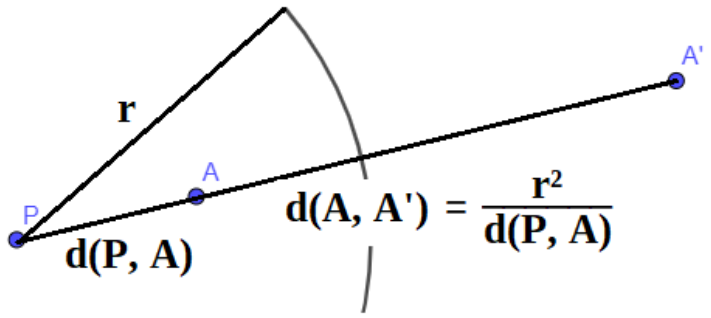


$$d = \sqrt{\frac{\tan\left(\frac{\pi}{2} - \frac{\pi}{n}\right) - \tan\left(\frac{\pi}{m}\right)}{\tan\left(\frac{\pi}{2} - \frac{\pi}{n}\right) + \tan\left(\frac{\pi}{m}\right)}}$$

(**Normalement, dans un disque de Poincaré, les segments des polygones doivent être des arcs de cercle.** Mais faire cela réduit considérablement la vitesse du programme pour peu de différence, et il n'existe pas de fonction en Python permettant de remplir un polygone constitué d'arcs de cercle. C'est pour cela que **j'ai décidé de tracer les segments comme des lignes droites**.)

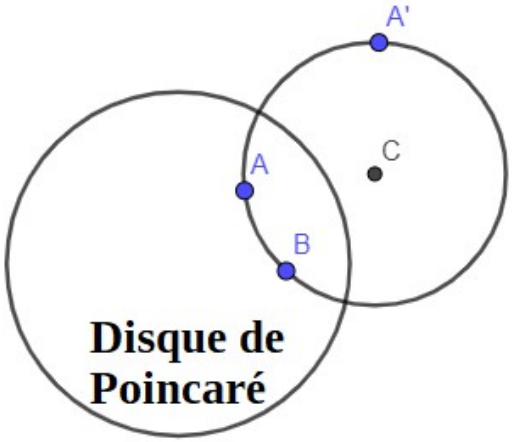
Cette formule nous permet ainsi de créer une fonction getCenterPolygon(angle), angle étant l'angle initial de la tortue. Cela permet d'avoir un polygone plus ou moins penché. À partir de cette fonction, on peut créer un **premier polygone** égal à getCenterPolygon(0).

On voudrait maintenant **construire les voisins du premier polygone.** Une simple symétrie axiale ne suffirait pas, car on a déjà vu qu'on ne pouvait pas paver le plan avec des pentagones de cette manière. À la place, il faut faire une **inversion** par rapport à une droite hyperbolique. Pour calculer l'inverse A' du point A par rapport à une droite hyperbolique, il faut connaître le centre P et le rayon r de la droite hyperbolique :



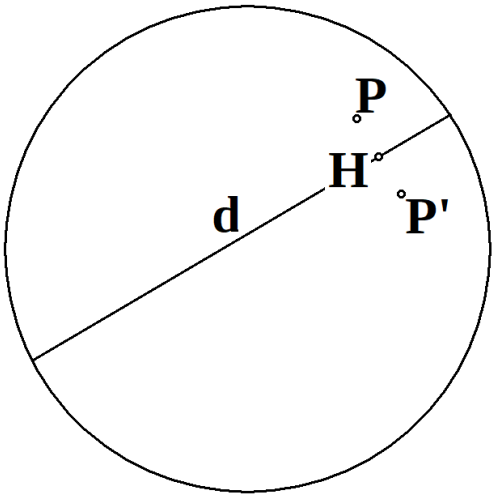
Inversion A' de A par rapport à la droite hyperbolique de centre P et de rayon r

On veut inverser le premier polygone par rapport à chacun de ses côtés. **Il nous faut donc la droite hyperbolique de chacun de ses côtés.** Comment obtenir une droite hyperbolique passant par les deux extrémités A et B d'un côté ? Pour cela, il faut calculer le centre d'un cercle passant par trois points de cette droite, mais on a que deux points de cette droite. Pour obtenir un troisième point de cette droite, on va utiliser une propriété intéressante des droites hyperboliques : L'inversion d'un point d'une droite hyperbolique h par rapport au disque de Poincaré est compris dans h :



Comment trouver le centre C d'une droite hyperbolique passant par A et B. Il suffit d'inverser A et de trouver le cercle passant par A, A', et B.

Cependant, **il y a des cas où il n'y a pas de centre**. Lorsque A, B, et le centre du disque de Poincaré sont alignés, la droite hyperbolique d n'est qu'une simple droite euclidienne. **Dans ce cas, P' est la symétrie de P par rapport à d**. Pour calculer P', on calcule le projeté orthogonal H de P sur d, et on calcule la distance PH. La distance PP' est le double de la distance PH :

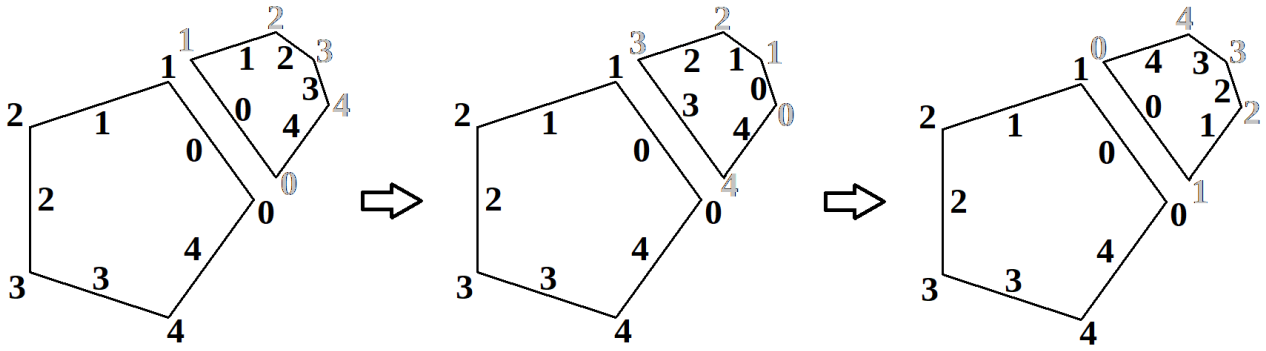


Comment trouver la symétrie P' de P par rapport à d

Maintenant, on peut inverser chaque point de getCenterPolygon(0) par rapport à ces côtés. Mais **il y a un petit détail qu'il faut corriger.** Inversons par exemple le premier polygone par rapport à son côté 0. Si on place les points dans l'ordre [0', 1', 2', 3', 4'], alors on peut voir que l'ordre de placement des points (représenté en gris ci-dessous) se fait dans le sens de rotation opposé à ceux du premier polygone. En effet, **avant l'inversion, les points sont dans le sens trigonométrique, mais après, ils sont dans le sens horaire.** Cela veut dire que les faces changeraient de sens de rotation à chaque mouvement, ce qui rendrait compliqué la représentation des polygones par le chemin de faces à emprunter.

En inversant l'ordre de placement, donc en utilisant le tableau [4', 3', 2', 1', 0'], on constate que la face 0 du nouveau polygone n'est pas en contact avec le premier polygone. En fait, avec n_side comme numéro du côté par lequel on inverse, c'est la face (m-1)-(n_side+1) qui est en contact avec le premier polygone.

Pour régler ce problème, il faudrait tourner le nouveau polygone (m-1)-(n_side+1) fois dans le sens trigonométrique. On fait cela en décalant chaque élément vers la gauche de (m-1)-(n_side+1) après l'inversion du tableau. On a désormais une méthode inverseSide(n_side) qui inverse un polygone selon son côté n_side.

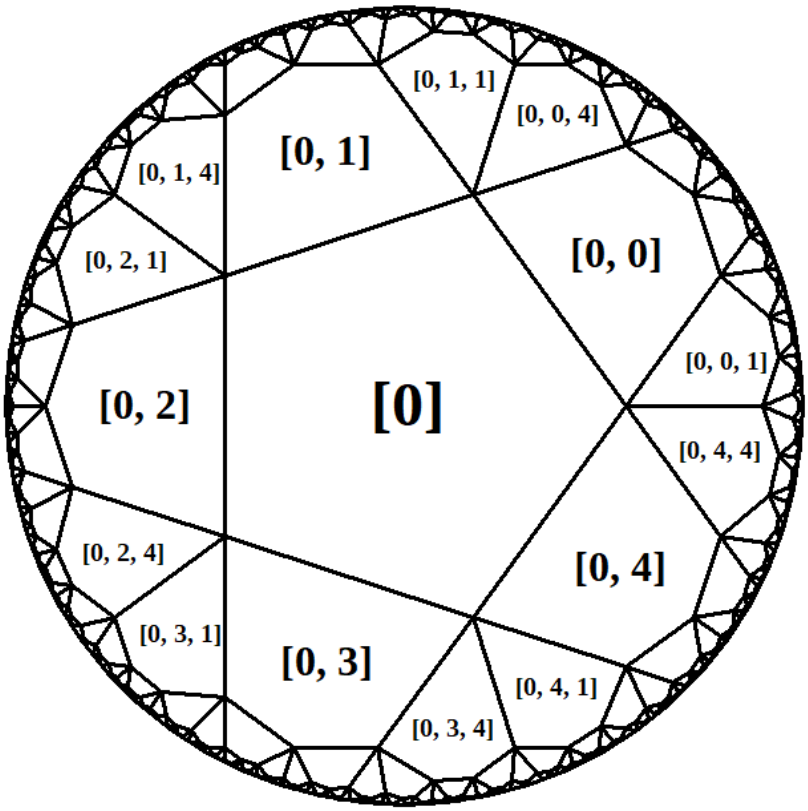


Régler le problème de changement de sens de rotation causé par la réflexion

Pour chaque polygone p, on peut stocker ses informations en créant une instance d'une classe Polygone. Les adresses des voisins de p sont stockées dans une liste p.neighbours. Aussi, une liste p.sides_with_children contient tous les numéros des côtés contenant un successeur. Pour stocker les coordonnées des points dans le bon ordre de passage, on utilise une liste p.points.

Pour paver le disque en entier, on inverse le premier polygone selon ses côtés où il y a de la place, on stocke les adresses des voisins et des successeurs, puis on répète récursivement. **Pour savoir où il y a de la place**, il suffit de créer le polygone qui devrait aller à cet endroit, puis de comparer le centre de ce polygone avec celui de tous les autres polygones. **Pour comparer des points flottants, il ne faut pas utiliser le symbole ==, mais plutôt une fonction qui retourne si la distance entre deux points est assez petite (<0,0001)**. Même si il est coûteux de comparer le centre de chaque polygone avec celui de chaque autre polygone, ce processus n'est fait que lors du démarrage et du changement de pavage, car après avoir stocké toutes les relations entre les polygones et leurs voisins, on n'a plus besoin de comparer les centres. Appelons calculateTreeForFirstTime() la fonction qui inverse les polygones et stocke leurs adresses et leurs coordonnées, et calculateTree() la fonction qui ne fait qu'inverser les polygones et stocker leurs coordonnées.

On peut nommer un polygone selon le nom du prédécesseur plus le numéro du côté emprunté. J'ai choisi d'appeler le premier polygone [0]. Pour obtenir un polygone à partir de son nom, il suffit de commencer au premier polygone, d'enlever le zéro de départ, puis de considérer le reste du nom comme un chemin de faces à emprunter. Ainsi, dans le pavage (5, 5), le polygone [0, 0, 4] peut aussi être représenté par [0, 1, 1, 1]. **On peut désormais créer et se repérer dans n'importe quel pavage hyperbolique centré.**



Noms des polygones dans le pavage (5, 5)

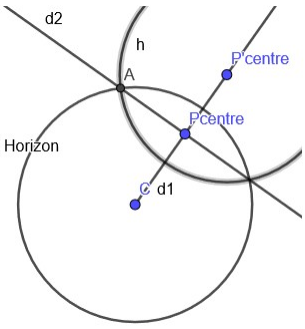
Deuxième partie : Bouger les polygones

Maintenant que l'on peut créer des pavages centrés, **on voudrait créer une méthode move() capable de bouger le centre de n'importe quel polygone à n'importe quel point P_{centre} du disque** (sauf au bord, appelé horizon, car il représente des points infiniment lointains). **Le résultat de la méthode move() sur le premier polygone peut être vu dans mon programme en mettant la variable « test » à True.** Au lieu que le Snake se lance, le centre du premier polygone suivra la souris.

Malheureusement, pour déplacer un polygone, on ne peut pas juste décaler tous ses points dans une même direction puis appliquer calculateTree(), car le premier polygone doit rétrécir lorsqu'il s'éloigne du centre. À la place, **il faut trouver une droite hyperbolique h qui transforme l'origine en P_{centre}, puis inverser getCenterPolygon(0) selon h.** Ensuite on pourra utiliser calculateTree().

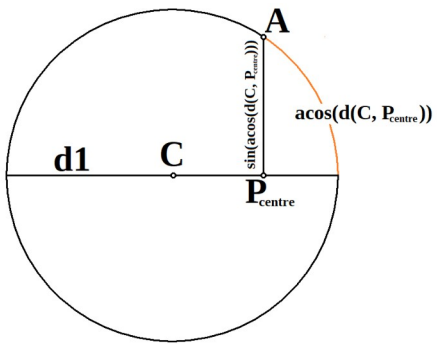
Déjà, on a le cas de base où P_{centre} est au centre du disque. Dans ce cas, le polygone choisi est égal à getCenterPolygon(0).

Pour résoudre les autres cas graphiquement, on trace une droite euclidienne d_1 qui passe par P_{centre} et le centre du disque (qu'on appellera C). Puis on trace la perpendiculaire d_2 de d_1 passant par P_{centre}. Appelons P un des deux points d'intersections entre d_2 et l'horizon, et P'_{centre} l'inverse de P_{centre} par rapport à l'horizon. h est le cercle qui a comme centre P'_{centre} et passant par A :



Trouver h graphiquement

Pour calculer ce cercle algorithmiquement, il faut calculer P'_{centre} et la distance entre P'_{centre} et A. Pour calculer cette distance, il suffit de réaliser que la distance entre A et P_{centre} est égale au sinus de l'arccos de la distance entre C et P_{centre}. Ensuite, on calcule la distance entre P_{centre} et P'_{centre}. Avec le théorème de Pythagore, on peut calculer la distance entre P'_{centre} et A.

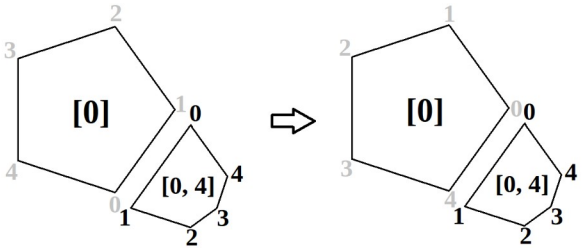


La distance entre A et P_{centre} est égale au sinus de l'arccos de la distance entre C et P_{centre}

Après avoir fait une inversion de getCenterPolygon(0) selon h, l**es points du polygone obtenu sont dans le sens de rotation opposé que ceux du premier polygone**. On fixe cela en inversant l'ordre de placement des points, comme dans la première partie.

Pour l'instant, la méthode ne fonctionne que sur le premier polygone. Si on applique la méthode sur un polygone différent, cela ne change pas le premier polygone, et puisqu'on calcule le pavage à partir du premier polygone, le pavage n'aura pas changé. **Il faut donc trouver un moyen de propager les changements du polygone choisi vers le premier polygone.** Pour cela, après avoir bougé le polygone choisi, on calcule les nouvelles coordonnées de son prédécesseur en utilisant l'inversion à la face 0, puis on calcule le prédécesseur de ce prédécesseur, et ainsi de suite jusqu'à être arrivé au premier polygone. **C'est ce que fait la méthode propagation().** Maintenant que l'on a changé correctement les coordonnées du premier polygone selon les changements du polygone choisi, on peut utiliser calculateTree() sur le premier polygone.

Il faut cependant faire une petite modification après avoir utilisé l'inversion. Ici, on veut déplacer le polygone [0, 4]. On inverse donc vers son côté 0 pour obtenir [0]. Mais dans, ce cas, **[0] devrait montrer son côté 4 à [0, 4]. Or, il montre son côté 0.** Il faut donc tourner le prédécesseur une fois dans le sens trigonométrique. Si n_side est le côté que l'on voudrait que le polygone prédécesseur montre, alors il faut décaler chaque élément du tableau d'ordre de placement de m-n_side :



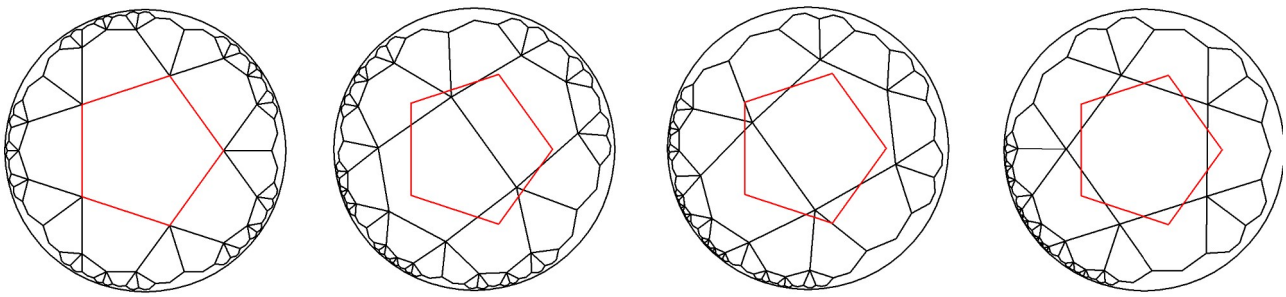
Obtenir le polygone prédécesseur de [0, 4]

Troisième partie : Passer d’une case à l’autre

On peut maintenant avoir n’importe quel pavage hyperbolique autour de n’importe quel point. **On voudrait maintenant faire une fonction moveTowardsSide(n_side) qui déplace le voisin n_side du polygone central vers le centre.**

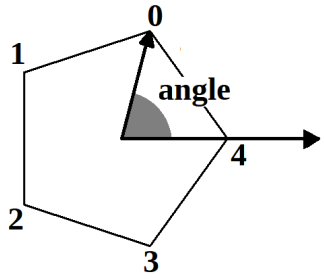
Si on veut qu’il y ait trois images par déplacement, il faut utiliser move() deux fois sur le polygone voisin. Cependant, en utilisant move(), on a fait une inversion, puis on a inversé l’ordre de placement. Mais on a vu dans la première partie qu’en faisant cela, le polygone voisin montrerait son côté (m-1)-(x+1) au polygone central. Or, on voudrait que le polygone voisin montre son côté en commun n_side_common avec le polygone central. Pour cela, on décale chaque élément du tableau d’ordre de placement de (m-1)-(x+1) - n_side_common.

À la dernière étape de moveTowardsSide(), on veut mettre le polygone voisin au centre. Si on utilise move((0, 0)) sur le polygone voisin, cela retournera le getCenterPolygon(0). Mais dans le cas où m est impair, on ne veut pas que le polygone voisin devienne ce polygone :



Utilisation de moveTowardsSide(0) avec [0] comme polygone central, donc le polygone [0, 0] se déplace vers le centre. La première image est prise avant l’appel de la méthode. Le polygone getCenterPolygon(0) est représenté en rouge. On voit que le polygone [0, 1] ne doit pas se superposer au polygone rouge à la dernière étape.

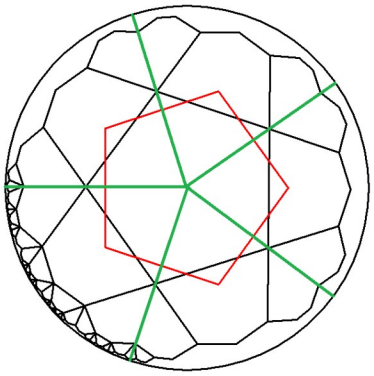
Pour **créer le polygone à la dernière étape**, on crée un polygone temporaire qui a les mêmes coordonnées que le polygone voisin si il avait parcouru 99,9% du trajet. Ce polygone est quasiment identique à ce que l'on veut, mais il n'est pas centré. C'est pour cela qu'au lieu de l'utiliser comme tel, on va juste utiliser son angle de rotation. **L'angle de rotation d'un polygone**, c'est l'angle qui est formé entre le vecteur qui commence en (0;0) et finit en (1;0), et le vecteur qui commence en (0;0) et finit par le point 0 du polygone.



Angle d'un polygone

Ensuite, on appelle getCenterPolygon avec l'angle de rotation du polygone temporaire, et on transforme le polygone voisin en ce polygone. On arrive ainsi à afficher la dernière image

Il ne reste plus qu'un dernier problème. On veut que le polygone voisin se rapproche du centre en **montrant un de ses côtés**. Mais si on utilise move(p) sur un point n'étant pas compris sur une des cinq demi-droites euclidiennes représentées en vert ci dessous, alors le polygone voisin se rapproche du centre en **montrant un de ses coins**. Il faut donc que le polygone voisin commence sur une de ces droites, donc que le polygone par lequel on inverse dans l'algorithme de la deuxième partie (celui surligné en orange aux pages 12 et 13) soit confondu avec le polygone central. Il suffit juste de remplacer son angle par celui du polygone central.



Il faut que le polygone rouge soit confondu avec le polygone central, pour que les droites vertes passent par les centres des polygones voisins, pour que le point de départ du polygone à déplacer au centre soit sur une des droites vertes.

Quatrième partie : Implémenter le Snake

Dans un Snake, **les cases peuvent soit être vides, soit êtres remplies par une partie du Snake, soit être remplie par le fruit**. Ces états peuvent être représentés par les valeurs 0, 1, et 2 dans la variable polygon.state :

J'ai décidé d'ajouter la valeur 3 qui correspond à quand **la case est un cul-de-sac**. Cela veut donc dire qu'elle ne doit pas être choisie pour placer le fruit, et elle n'a pas besoin d'être remplie pour finir le jeu. En fait, on ne l'affiche même pas lorsque le mode test est désactivé. **Pour décider quelles cases sont des culs-de-sac**, on parcours tous les polygones en regardant à chaque fois le nombre de voisins n'étant pas à l'état 3. Si ce nombre est en dessous de 2, on met le polygone à l'état 3. **Parcourir la liste une seule fois n'est pas suffisant**, car il se peut qu'en mettant un polygone à l'état 3, on transforme d'autres polygones en culs-de-sac. On parcours donc tous les polygones tant qu'il y a eu au moins un polygone qui est devenu à l'état 3 la fois précédente. **Il se peut aussi qu'en faisant cela, on réalise que tous les polygones sont des culs-de-sac.** Dans ce cas, on affiche un message dans le menu, et on empêche le menu d'être quitté tant qu'une autre option de pavage n'a pas été choisie ou que la récursivité n'a pas été augmentée.

Pour **remplir et vider les cases selon les actions du Snake**, j'utilise une **file**. Quand le Snake se déplace vers un polygone d'état 1 ou 3, la partie se quitte. Si il se déplace vers un polygone d'état 0 ou 2, ce polygone devient à l'état 3 et entre dans une liste. Si ce polygone était à l'état 0, alors le Snake n'a pas mangé de fruit, donc on défile la liste, et le polygone défilé est mis à l'état 0. Si ce polygone était à l'état 2, on place un nouveau fruit.

Pour **placer le fruit**, on utilise un tableau t_all qui contient tous les polygones. Dans une fonction chooseRandomFruit(), on choisis un élément au hasard de t_all grâce à la librairie random. Si l'état du polygone est égal à 0, alors on met son état à 2. Sinon, on rappelle chooseRandomFruit() récursivement. En dehors de chooseRandomFruit(), on vérifie si elle cause une RecursionError. Si il y en a une, alors c'est que la fonction n'a pas réussi à placer de fruit, donc que les cases étaient très probablement toutes pleines, donc que le jeu est gagné.

Lorsque l'on gagne ou perd, on affiche un message de victoire ou d'échec, puis on utilise la fonction exitonclick() de Turtle pour quitter le programme une fois que le joueur clique sur la fenêtre.

Sitographie :

Une vidéo expliquant bien l'inversion :
https://www.youtube.com/watch?v=6z3rQb3fUMw

Le site expliquant la création du polygone central, et comment sait-on qu'un polygone peut paver le plan hyperbolique :
http://www.malinc.se

L'inspiration de ce projet, la série sur le développement du jeu Hyperbolica :
https://www.youtube.com/watch?v=EMKLeS-Uq_8&list=PLh9DXIT3m6N4qJK9GKQB3yk61tVe6qJvA

Le site expliquant l'histoire de la géométrie non euclidienne :
https://fr.wikipedia.org/wiki/Géométrie_non_euclidienne

Le site expliquant ce qu'est un disque de Poincaré :
https://fr.wikipedia.org/wiki/Géométrie_hyperbolique

Le site expliquant comment trouver le centre d'un cercle passant par 3 points :
https://cral-perso.univ-lyon1.fr/labo/fc/Ateliers_archives/ateliers_2005-06/cercle_3pts.pdf