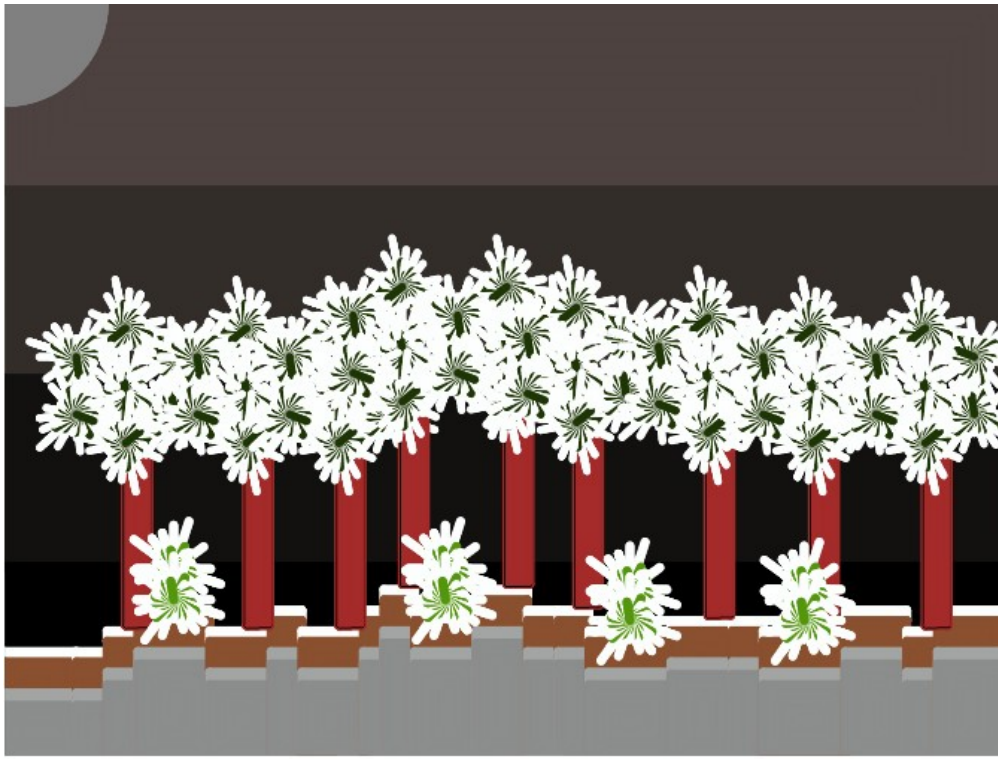
Mascaro Valentin
Vidal-Mazuy Antoine

2019-2020

Projet : TurtleRandomizer



Objectifs / Contraintes : Créer un programme capable à partir d'une suite de mot, de générer un dessin.

- Le dessin, malgré le nom du projet, ne doit pas être généré aléatoirement, le dessin est généré à partir de la suite de mots donné par l'utilisateur.

Ainsi si deux utilisateur donne la même suites de mots, au caractère près, ils obtiendront le même dessin.

- Toute fois, deux suites de caractères PRESQUE différentes, donneront deux dessins complètement différents exemple « aaaba » et « aaaca ».

- Le dessin sera fourni au format .pdf dans le répertoire courant de l'utilisateur (par défaut /Users/Documents OU le dossier depuis lequel le script est lancé).

Introduction :

Nous avions pour but de créer un moyen, à la manière de jeux comme Minecraft, de pouvoir générer une carte, ou un dessin en fonction de suite de chiffre ou de mot.

Le premier défi était donc de trouver une relation qui lie l'entrée - quelconque - au clavier et un dessin.

Ensuite ils nous fallait un moyen d'éviter des dessins trop répétitifs en fonction de l'entrée. Par exemple, que faire si l'utilisateur entre : " J'aime le R " puis "J'aime le C " ? (Ces contraintes nous ont forcé à générer deux dessins très différents pour ces deux entrées pourtant quasi-identique)

Et enfin, créer un dessin cohérent, c'est à dire d'éviter par exemple de créer des dénivelés monstrueux et successifs dans le dessin.

Solution :

Afin de transformer une entrée au clavier en donnée utilisable pour générer un dessin, on converti simplement toute l'entrée dans le code Ascii correspondant (Réf : Fonction : CreationDuvecteurTemporaire | Ligne 86). Les chiffres obtenus pourront ensuite être traités comme des longueurs et des hauteurs dans les futurs objets composants le dessin.

Afin d'éviter d'obtenir des dessins quasiment identique lorsque les entrées sont elles aussi presque identiques, le programme applique ce que l'on appelle un hachage. Une fois l'entrée au clavier transformé en suite de chiffre, on multiplie le résultat à plusieurs reprises, puis on prend un chiffre tout les x-chiffres du résultat. Ainsi une différence - même infime - dans l'entrée de l'utilisateur, devient après cette fonction une énorme différence. (Réf : hashDuSeed | Ligne 104).

Enfin puisque ce résultat est composé de chiffres censés être des hauteurs et des largeurs pour les objets de notre dessin, on applique une règle afin d'éviter des différences de tailles trop grandes (Réf : creationDuSeed | Ligne 57-84).

Composition du dessin :

Le dessin est composé de 4 parties :

Ciel :

Comme tout devoir d'élève d'école primaire, le dessin possède un ciel dont la couleur et l'astre seront définis en fonction de ce qu'on expliquera plus tard : le seed (correspond à l'entrée au clavier). Ainsi le dessin est soit composé d'un soleil, soit d'une Lune, qui peux se trouver à gauche, au centre ou à droite (simulation d'une heure).

Arbre:

Viens ensuite les arbres, dont la taille et la position dépend du seed (comme tout en réalité). Un arbre n'apparaît que si le sol qui le soutient le permet.
De plus on impose qu'il est impossible que deux arbres apparaissent côte à côte (tronc contre tronc).

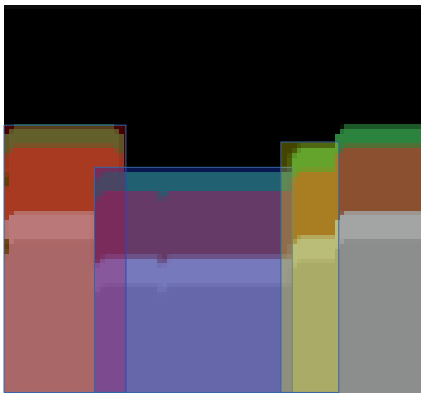
Buisson:

On ajoute des buissons qu'y n'apparaissent que si le taille du terrain sur lequel il est positionné le permet. Toutefois au contraire des arbres, plusieurs buissons peuvent apparaître l'un à côté de l'autre.

Chunk:

Le chunk est un morceau de terrain composé de 3 parties, l'herbe, la terre puis la roche. Le seed défini la possibilité qu'un arbre ou qu'un buisson apparaisse. Dans le cas du chunk, Le seed définit la hauteur et la largeur du bloc de terrain (chunk).

On impose la règle suivante :



Il est impossible que la différence de hauteur entre deux blocs de chunk successifs soit supérieur à deux.
Comme le montre le schéma ci-contre, la différence de hauteur entre le bloc rouge et le bloc bleu est de deux. Il s'agit de la différence maximum entre deux blocs.
La différence entre le bloc bleu et le bloc jaune est de un.
Enfin le bloc rouge n'a aucun impact sur le bloc jaune.

Finalement ces 4 objets auront des couleurs différentes en fonction du biome qui sera déterminé par le seed.

Explication du code :

Pour lancer le programme de dessin il suffit de lancer la fonction DoIt(). Cette fonction lance simplement toutes les fonctions nécessaires à la création du dessin. Les caractères avec accents ne sont pas valide en entrée, ils généreront une erreur, car ils n'ont pas de valeurs lors de l'utilisation de la fonction utf8ToInt.

Le paramètre taille est toujours égal à 40. On a laissé la possibilité de modifier ce paramètre en vue d'une modification de la taille des seeds, mais aussi pour pouvoir effectuer des tests afin de valider nos fonctions.

Partie Creation du Seed:

DoIt_Seed<-function(taille)

L'objectif de cette fonction est simplement de lancer toutes les fonctions nécessaire à la création du seed. (Cela permet surtout de pouvoir gérer le projet en séparant chaque partie du programme en petite fonction).

Le paramètre vecteur est un vecteur de ' taille ' éléments numérique, tous initialisé à 0.

On retourne un vecteur contenant une suite de chiffres et dont la différence entre deux chiffres d'indice pair successif est de 2 max.

Exemple invalide : 2 8 6 7 Exemple valide : 2 8 4 7. (Explication ci-dessous)

EntreeClavier<-function()

L'objectif de cette fonction est d'obtenir l'entrée de l'utilisateur, qui servira à la conception du dessin.

Aucun paramètre.

On retourne la variable **scan**, qui contient l'entrée clavier de l'utilisateur, la taille maximum de cette entrée est de 30 caractères. 30 caractères suffisent largement à obtenir de nombreux dessins différents. Et un nombre trop élevé de caractère viendrai ralentir le programme inutilement.

CreationDuvecteurTemporaire<-function(scan,taille)

L'objectif de cette fonction est de transformer le **scan** (chaîne de caractère), en un vecteur de 40 chiffres.

Le paramètre **scan**, contient la suite de caractère entrée au clavier par l'utilisateur.

La variable **tmp** est la conversion de **scan** en int, multipliée une première fois par 1234 pour augmenter le nombre de chiffre dans le futur vecteur. On multiplie par la suite ce résultat par 5678 + 33 (par tour de boucle) tant que le nombre de chiffre n'est pas suffisant.

```
hasard<-5678
while(nchar(tmp)<taille)
{
  tmp<-paste(tmp,as.character(utf8ToInt(scan)*hasard),sep="",collapse=NULL)
  hasard<-hasard+33

}
```

Cette suite d'instruction génère un Warning. En effet la condition contrôle la taille (nchar) de tmp.
Or tmp est un vecteur de caractère `"119698550766553967" "120932556444559678" "122166562122565389"`

Ici on peux voir le résultat de la chaîne de caractère "abc" après 2 tours de boucle.
nchar(tmp) retourne ici 18 car ne controle que **tmp**[1]. Ce qui déclenche le warning. " Attention, seul la première valeur de **tmp** est contrôler ", or c'est bien le but afin de pouvoir utiliser la fonction hashDuSeed() à son plein potentiel.

hashDuSeed <- function(tmp,taille)

Cette fonction est simplement une sous partie de la fonction CreationDuVecteurTemporaire() son objectif est le même. Mais afin de pouvoir tester les instructions indépendamment du travaille fait précédemment, il est fait dans une fonction différente.

Ici tmp correspond au tmp présent dans la fonction CreationDuVecteurTemporaire().

Lorsque CreationDuVecteurTemporaire() appelle hashDuSeed(), nchar(**tmp**) renvoie 40 éléments. Toutefois ces 40 éléments sont présents dans la première case de **tmp**.
Ainsi pour "abc" on obtiens **tmp** `"11969855076655396755396755716856036956357O" "120932556444559678559678562912566146569380" "122166562122565389565389568656571923575190"`
= On a bien 40 chiffres pour 'a', 40 pour 'b' et 40 pour 'c'. La première boucle **for** de cette fonction va stocker toute les valeurs de **tmp** - regroupés en 1 variable **tmp_entier** - dans un vecteur de int.
La deuxième boucle **for** va stocker 40 éléments de **tmp_entier** selon sa taille.

Pour schématiser :

- Parmi 120 éléments on stocke tous les 120/40 = 3 chiffres

- Parmi 160 éléments on stocke tous les 160/40 = 4 chiffres

Donc pour "abc" on va garder le 3éme chiffre, puis le 6éme, puis le 9éme et ainsi de suite jusqu'à obtenir 40 chiffres.
Pour "abcd" on va garder le 4éme, puis le 8éme, puis le 12éme etc. Ainsi le Warning déclenché lors de l'exécution est prévu puisque l'on cherche bien à obtenir 40 chiffres par caractère.

creationDuSeed<-function(scan,vecteur,taille)

L'objectif ici est de retourner un vecteur conforme pour la fonction DoIt_Seed().

Le paramètre **scan** correspond au **scan** obtenu par EntreeClavier().
Le paramètre vecteur correspond au vecteur de taille 40 de la fonction principale.

On commence par transformer **scan** en un vecteur temporaire (Ici un vecteur non conforme, est un vecteur où les différences de hauteur des chiffres ne sont pas valides pour faire un dessin 'propre' et 'jolie). Cette transformation se fait avec les fonctions expliqués précédemment.
Viens ensuite la boucle **while** dont le but est de niveler le vecteur en diminuant les différences de hauteurs des chiffres des indices paires. De plus on retire les 1, jugé trop petit pour nos fonctions de dessins.

Par exemple : 1 2 5 5 8 6 7 9 deviens 3 2 5 4 8 6 7 8

Partie Dessin de la tortue :

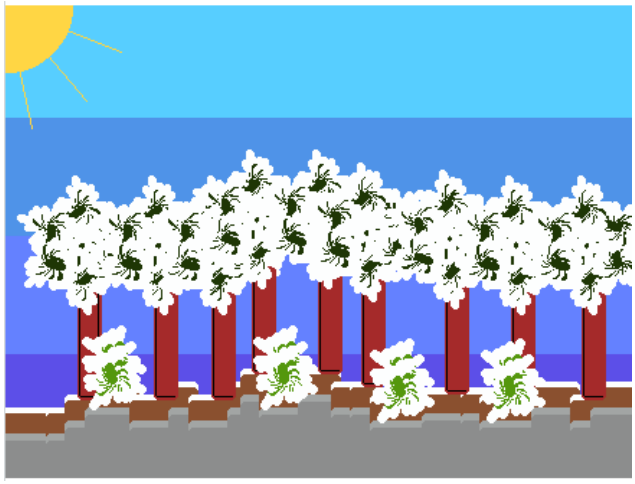
DoIt_Dessin <- function(vecteur,taille_vecteur)

Comme la plupart des fonctions DoIt() de notre projet, cette fonction lance successivement toute les fonctions nécessaires à la création du dessin.

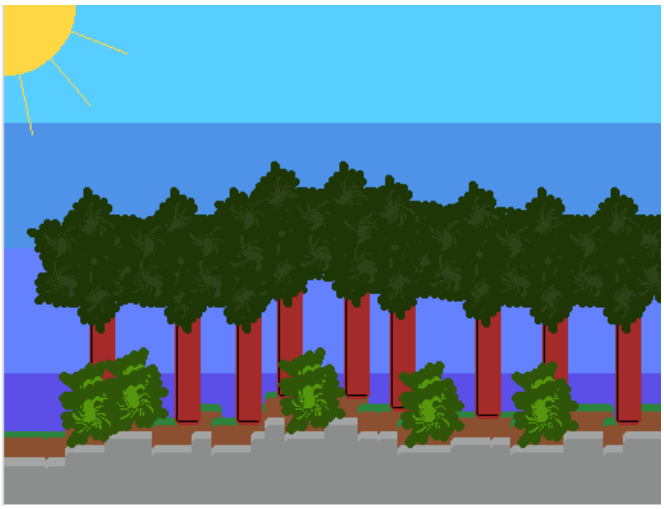
Biome <- function(vecteur)

La fonction permet de déduire selon la valeur d'une case de vecteur choisi d'avance, le type du biome du futur dessin(aucune raison particulière dans le choix du numéro de la case). 3 biomes possibles :

Neige



Normal



Forêt corompue

(Les 3 dessins sont, à l'exception de leur biome, quasiment identique, le dessin neige est généré à partir du mot "nuit" et les 2 suivants sont simplement une modification du vecteur)

Ci dessous : vecteur neige > clair > corrompu. Seul le chiffre en rouge est modifié.

7 2 3 2 3 4 7 6 6 4 3 6 6 4 2 6 3 8 6 6 5 8 3 6 3 6 8 4 6 5 3 5 8 4 6 6 3 4 6 6
7 2 3 2 6 4 7 6 6 4 3 6 6 4 2 6 3 8 6 6 5 8 3 6 3 6 8 4 6 5 3 5 8 4 6 6 3 4 6 6
7 2 3 2 8 4 7 6 6 4 3 6 6 4 2 6 3 8 6 6 5 8 3 6 3 6 8 4 6 5 3 5 8 4 6 6 3 4 6 6

Initialisation <- function(taille_monde)

Paramètre taille_monde, il s'agit de la somme du vecteurs, multiplié par 10.

Cette fonction permet de générer le Terrarium, en bref, la toile sur laquelle la tortue va dessiner. On génère le terrarium en format paysage, d'où le turtle_init(taille_monde , taille_monde * ¾). On rajoute "clip" dans l'appel, cela permet à la tortue de déborder du terrarium sans générer d'erreur. Cela peut arrivé lors du coloriage, notamment celui du ciel.

Ciel <- function(taille_monde)

Le paramètre taille_monde est le même que pour la fonction initialisation (Intitialisation). Ce paramètre permet de connaître la taille du monde (largeur et longueur).

On commence par à nouveau déduire la taille Y du monde et de la stocker dans taille_mondeY. Puis on en déduit la taille Y du monde, divisé par 4 dans taille_mondeDecalageY. C'est avec cette valeur qu'il est possible de créer les dégradés de couleurs. (le dégradé est en réalité une suite de 4 rectangles de couleur différentes).

En fonction de la taille du monde, la fonction va maintenant créer 4 rectangles de couleur fixes, qui donne une impression de dégradé. Pour ce faire, on commence par les rectangles du haut.

Puis on fait le rectangle du dessous. On soustrait à taille_mondeY la valeur de taille_mondeDecalageY pour faire descendre le rectangle représentant le ciel.

Chunk <- function(vecteur,taille_vecteur,biome)

Le but de cette fonction est d'appeler successivement la fonction OutilsChunk afin de créer le sol - les blocs - qui compose le dessin, selon une couleur défini par le paramètre biome.

On donne à la fonction OutilsChunks la positionX (vecteur d'indice impair) la positionY(vecteur d'indice pair), la longueur du chunk, sa hauteur, puis la couleur du biome.

La variable compteurX est l'indice de tous les éléments impairs de vecteur (vecteur[1], vecteur[3]) correspondant aux longueurs des chunks.

CompteurY correspond aux éléments pairs, correspondant aux hauteurs des chunks.

La variable PosX permet de décaler la création des chunks de gauche à droite, on l'incrémente de la largeur des chunks.

Position de posX 1ere boucle. Position de posX 2éme boucle

Arbre <- function(vecteur,taille_monde,taille_vecteur,biome)

Même paramètres que précédemment. PosX, CompteurX, compteurY fonctionne de la meme façon.

On parcours à nouveau tout le dessin et on pose un arbre selon les conditions suivantes : La valeur de vecteur[compteurY] doit être supérieure à 3 (pour éviter de poser un arbre sur une cases très basse), la hauteur du prochain chunk (vecteur[compteurY+2])doit être supérieur à 2, pour éviter de poser un arbre puis que la hauteur de la case suivante soit trop basse ET le chunk précédent ne doit pas posséder d'arbre (variable presence_arbre_position_precedente == 0).

Buisson <- function(vecteur,taille_monde,taille_vecteur,biome)

Même principe que les arbres. A l'exception qu'on ne prend plus en compte vecteur[compteurY] dans les conditions vecteur[compteurX] et on ne se souci pas de savoir si un buisson existe déjà dans le chunk précédent.
(Les buissons s'avère être un prototype d'arbre raté dans lequel le tronc apparaissait sous le sol, laissant simplement apparaître le feuillage. On a fini par supprimé le tronc, et renommé la fonction Buisson).

Astre<-function(taille_monde,vecteur,taille)

Cette fonction prend en paramètre tout le vecteur (et sa taille). Ainsi que la taille du monde.

En fonction de la valeur de la dernière case du vecteur et de la première cette fonction positionne l'astre dont le type (lune ou soleil) est décidé par la fonction OutilsAstre dans la partie Outils.

Partie Outils :

Ceette partie contient toutes les fonctions nécessaires à la tortue. Donc on retrouvera une fonction qui s'occupera des arbres, des buissons des astres etc.…

* Les positions données aux fonctions de DessinTortue.R (exemple (2/11)* taille_monde ou encore posX+40+30) ont été trouvées à talonnement et témoigne de l'approximation faite pour rendre le dessin le plus beaux possible. Il existe probablement de meilleur façon de le faire, mais dans le cadre de ce mini-projet, on considère çela suffisant.

** Je ferais une mention spéciale aux parties qui gèrent les Biomes en début des fonctions OutilsChunk, OutilsArbre et OutilsBuisson.

OutilsChunk ← function(posX,posY,Longueur,Hauteur,biome)

L'objectif de cette fonction est de dessiner le sol du dessin.

Les paramètres posX et posY servent à position le chunk. La Longueur définit par le vecteur permet d'avoir la longueur du chunk. La Hauteur définit aussi par le vecteur permet d'avoir la Hauteur à la quelle se positionne le chunk. Le biome est utilisé pour définir les couleurs du sol.

OutilsChunk() dessine entièrement le chunk. Elle définit Les couleurs et appelle 4 fois la fonction Quadri, ainsi que la fonction setpos.

OutilsArbre <- function(posX,posY,taille_monde,vecteur,taille_vecteur,biome)

L'objectif de cette fonction est de dessiner le tronc et d'appeler la fonction OutilsFeuillage qui s'occupera de faire les feuilles de l'arbre.

Les paramètres posX et posY servent à positionner l'arbre. La taille_monde permet à l'arbre d'être à l'échelle du monde. Le vecteur est passé à la fonctions OutilsFeuillage. La taille_vecteur est passé à la fonctions OutilsFeuillage. Le biome est passé à la fonctions OutilsFeuillage.

OutilsArbre() dessine le tronc composé de 3 rectangles utilisant la fonction Quadri. Elle finit par repositionner la tortue avec la fonction setpos, avant d'appeler OutilsFeuillage.

OutilsFeuillage<-function(vecteur,taille_monde,taille_vecteur,couleur_biome,biome)

L'objectif de cette fonction est de dessiner les feuilles de l'arbre. Pour ce faire, la fonction va dessiner un hexagone et placer un Oursin à chaque sommet en plus d'un, placé au milieu.

Le vecteur est utilisé dans les boucles for, afin de faire varier les piques des oursins. La taille_monde permet aux feuillages d'être à l'échelle du monde. La taille_vecteur est utilisé dans les boucles for des oursins, pour avoir un nombre de piques variant selon la taille du monde. La couleur_biome - comme son nom l'indique - définit la couleur des oursins selon le biome. Le biome est utilisé afin de définir une 2ème couleur pour les oursins.

OutilsFeuillage() dessine les feuilles de l'arbre, et commence donc par dessiner un oursin au dessus de l'arbre, (première boucle **for**) et ensuite va faire 1 oursin à chaque sommet de l'hexagone. (deuxième boucle **for** imbriquée).

OutilsBuisson <- function(posX,posY,vecteur,taille_vecteur,biome)

L'objectif de la fonction est de dessiner les buissons.

Les paramètres **posX** et **posY** définissent la position des buissons. Les paramètres **vecteur** et **taille_vecteur** sont passés en paramètre à OursinBuisson(). Le **biome** permet de définir les couleurs du buisson.

Cette fonction se contente d'appeler la fonction OursinBuisson(), avec 2 postions différentes.

OutilsBuisson <- function(posX,posY,vecteur,taille_vecteur,biome)

L'objectif de la fonction est très similaire à OutilsFeuillage(). Elle va dessiner 2 oursins superposés de différentes couleurs.

Les paramètres **posX** et **posY** permettent de dessiner l'oursin aux bonnes postions, x et y. Le **vecteur** est utilisé dans les boucles for, afin de faire varier les piques des oursins. La **taille_vecteur** est utilisé dans les boucles for des oursins, pour avoir un nombre de piques variant selon la taille du monde. Enfin **biome** va permettre de changer les couleurs des oursins en fonction des Biomes.

Cette fonction agit de manière similaire à la fonction Outils Feuillage, mais sans les boucles **for**. Elle se contente de dessiner 2 oursins de taille différentes.

Quadri <- function(largeur,hauteur,i,colour,epaisseur)

L'objectif de cette fonction est de dessiner un quadrilatère, puis de le colorier. En réalité elle ne fait que dessiner en spirale, avec un trait d'épaisseur suffisamment épais pour pouvoir tout colorier.

Les paramètres **largeur** et **hauteur** définissent la taille du quadrilatère. Le paramètre **i** donne la précision qui par défaut est toujours égal à 0,5. Le paramètre **colour**, donne la couleur de quadrilatère. L'**épaisseur**, change l'épaisseur du trait de la tortue.

La fonction commence par faire un quadrilatère incomplet, qui donne la forme global du quadrilatère, Et ensuite tant que la hauteur ou la largeur (**k** et **l**) sont supérieur à 0 on va continuer la spirale.

OutilsAstre ← function(posX,posY,taille_monde)

OutilsAstre est appelé par la fonction Astre pour choisir la position de l'astre.

Les paramètres **posX** et **posY** permettent de placer l'astre a l'endroit souhaité (gauche / milieu ou droite). Le paramètre **taille_monde** permet de dessiner l'astre à l'échelle et de déterminé s'il s'agit d'une lune ou d'un soleil.

Cette fonction commence par positionner la tortue à l'endroit souhaité (avec **posX** et **posY**). Selon le paramètre taille_monde on décide de la couleur, jaune ou grisâtre, de l'astre. L'astre n'est qu'un rond. Dans le cas où taille_monde est supérieur à 1010, l'astre sera jaune et on y ajoute des rayons (Le Soleil). Dans le cas contraire, l'astre sera grisâtre et sans rayon (La Lune).

setpos ← function(x,y)

L'objectif de cette fonction est juste de recréer la fonction turtle_setpos() mais en utilisant les 2 méthodes turtle_up() et turtle_down() pour éviter de dessiner quand on déplace la tortue.

SommeVecteur <- function(vecteur,taille_vecteur)

L'objectif de cette fonction est de retourner la somme des indices impair de la fonction. En effet cette somme est utilisée pour connaître la taille monde qui est basiquement (somme*10).

Le seed est utilisé de la façon suivante : les indices pairs sont utilisés pour définir la taille Y des chunks et les indices impairs la taille X des chunks et donc *in fine* on peut connaître la taille totale de X.

*** En effet la fonction biome() renvoie 1 si c'est neige 2 si c'est corruption et 3 si c'est normal. Pour le biome neige aucun problème, la couleur qui est changé est uniquement le blanc. Mais pour corruption et normal il y a deux couleurs à changer, d'où la présence de couleur et couleur2. On aurait du faire une fonction qui gère les couleurs. On s'est rabattu sur cette solution car ce n'était pas prévu à la base il aurait fallu changer plusieurs grosses parties de code.*

Conclusion:

Nous avons réussi à générer des vecteurs utilisables aisément pour nos fonctions dessins, et qui nous permettent d'avoir un grand nombre de dessin possible. Le tout avec seulement un scan de 30 caractères.

Pour ce qui est des dessins, le rendu reste agréable. Seul bémol, la taille des dessins.

En effet les objets qui composent le dessins sont pensés pour être fait sur des dessins dont le vecteur est de 40 (et dont la taille du monde tourne entre 800 et 1200 en général). En retravaillant les tailles de chaque dessins pour les rendre dépendante de la taille du monde, on pourrait générer des dessins bien plus grands, avec beaucoup plus de détails (et la possibilité de rajouter des éléments autres que seulement buisson et arbre).

En conclusion nous avons réussi notre objectif principal, à savoir créer des dessins en fonction de petite phrase, à la façon des biomes Minecraft.