Compte Rendu R5A06

GOKCEN Bayram

27 novembre 2023

TP 1

1.1 Génération d'images 2D

1.1.1 Bibliothèque Pillow

La bibliothèque Pillow de Python est une bibliothèque open-source très populaire qui est principalement utilisée pour le traitement d'images. Elle permet de lire, de manipuler et de créer des images dans divers formats, notamment JPEG, PNG ou GIF qui sont les formats qui nous intéressent. Elle est largement utilisée dans le développement d'applications Python qui permettent la manipulation d'images, que ce soit pour le traitement ou la génération.

1.1.2 Structure de données d'une étoile

Pour représenter une étoile, nous avons choisi d'utiliser une classe avec une taille, une couleur et une position. La taille est de type entier, plus cet entier est grand, plus la taille de l'étoile sera grande. La couleur est une liste de 3 entiers, ce qui permet grâce au RGB de donner la couleur que l'on souhaite à l'étoile. En effet, en fonction de la valeur que l'on donne à chaque entier (lié à la "quantité" de couleurs : Rouge, Vert et Bleu), on peut donner à une étoile, la couleur que l'on souhaite. Finalement, la position est un tuple ou une ligne de données qui possède deux entiers pour les positions x et y.

1.1.3 Génération d'une liste d'étoiles

Nous générons alors une liste d'étoiles et nous les positionnons au hasard dans une image RGBA au fond transparent.

Une image RGBA est une image basée sur les quatre canaux : Rouge (R), Vert (G), Bleu (B) et Alpha (A) :

Rouge (R) : Ce canal représente la composante de couleur rouge de chaque pixel. Il détermine la quantité de rouge dans la couleur du pixel.

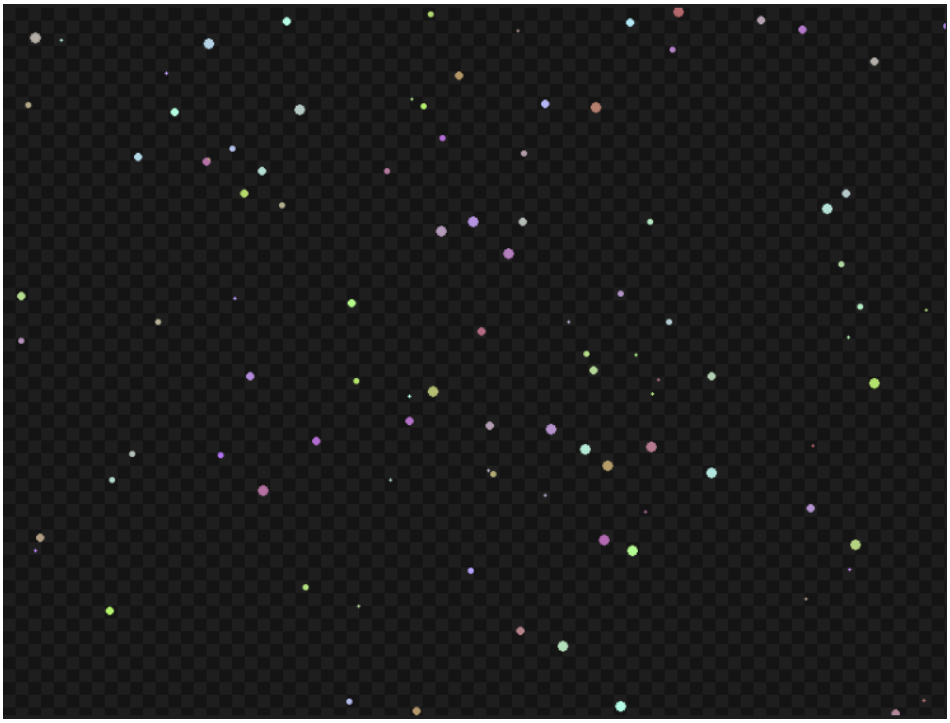
Vert (G) : Le canal vert représente la composante de couleur verte de chaque pixel. Il

détermine la quantité de vert dans la couleur du pixel.

Bleu (B) : Ce canal représente la composante de couleur bleue de chaque pixel. Il détermine la quantité de bleu dans la couleur du pixel.

Alpha (A) : Le terme A ou alpha est utilisé pour représenter la transparence d'un pixel. Un pixel avec une valeur alpha de 0 est totalement transparent alors qu'un pixel avec une valeur alpha de 255 est complètement opaque. Entre les deux, la transparence est partielle.

Nous obtenons alors cette image très simple avec les différentes étoiles de tailles et de couleurs différentes :

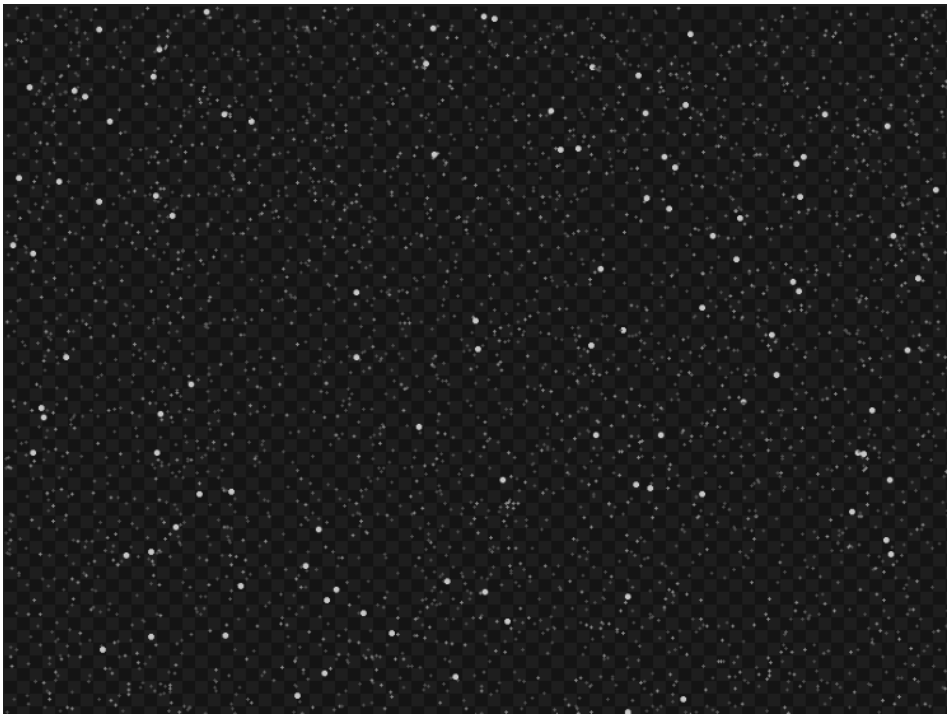


1.1.4 Image finale

Nous avons alors, précédemment, généré une image classique sans fond.

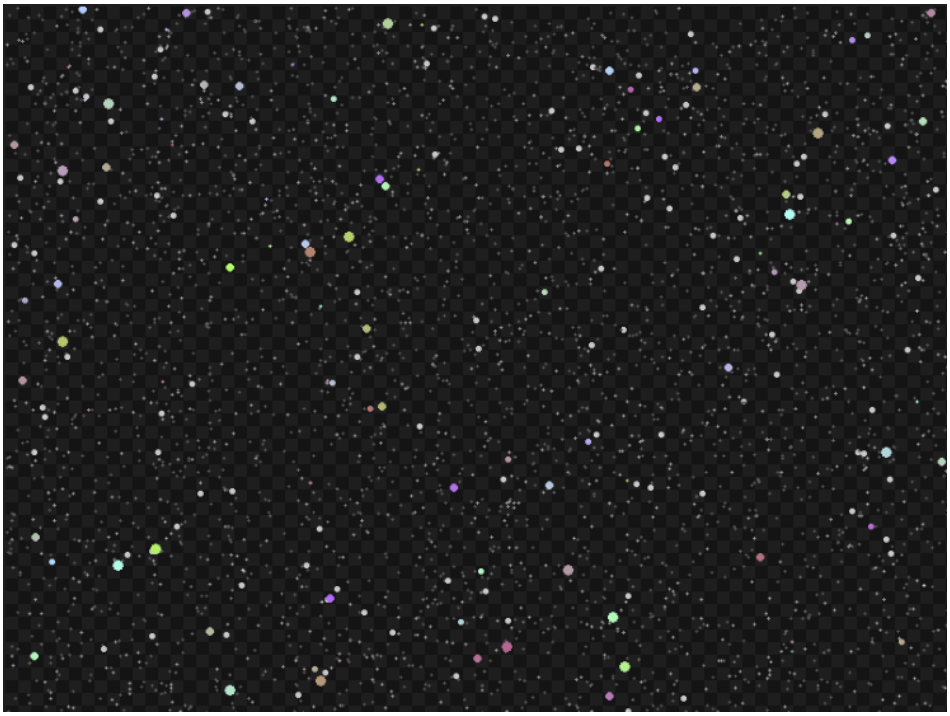
Notre objectif est, alors maintenant, de fusionner l'image que nous avons généré avec une image de fond qui nous est donnée.

Voici notre image de fond :



La bibliothèque Pillow nous permet grâce à la fonction "alpha.composite" de fusionner ces deux images.

Nous obtenons alors l'image suivante :



1.2 Génération d'une animation

L'objectif principal de ce TP est de générer une animation avec plusieurs images d'étoiles. En effet, maintenant que l'on a une image d'étoiles, il faut que l'on fasse déplacer les étoiles de cette image et ceci plusieurs fois pour obtenir une animation.

En somme, on doit obtenir 400 images avec les étoiles qui bougent petit à petit horizontalement. De plus, nous avons décidé de donner à ces étoiles des vitesses différentes en fonction de leur taille. Alors certaines étoiles se déplacent plus vite que d'autres.

Finalement, il faut que cette animation boucle, cela veut dire qu'à la fin des 400 images, il faut que les étoiles soient revenues à leur position de départ pour que l'animation soit infinie.

Pour ceci, j'ai utilisé un programme "etoiles_gif" qui créé, à l'instar de l'image précédente, une première image, avec des étoiles, nommé "étoiles1". Puis, une seconde image "étoiles2" qui possède les mêmes étoiles mais qui se sont déplacées. En effet, en fonction de leur taille (par exemple, si la taille d'une étoile est de 2, alors elle se déplace de 6 vers la droite). Lorsqu'une étoile arrive à l'extrémité droite (donc à la position x=800), elle repart depuis la gauche (donc la position x=0).

Finalement, le programme continue jusqu'à créer 400 images qui forment une boucle.

On utilise alors l'outil ffmpeg pour lier toutes les images et créer une animation gif :

```
\TP1\images>ffmpeg -framerate 25 -i etoiles%d.png -pix_fmt yuv420p output.gif
```

Cette ligne de commande utilise donc l'outil ffmpeg pour créer un fichier GIF animé à partir de 400 images en utilisant un framerate (nombre d'images par seconde) de 25 images par seconde.

Voici une explication détaillée de la commande :

-framerate 25 : Cette option spécifie le nombre d'images par seconde. Dans notre cas, le GIF sera généré avec un taux de 25 images par seconde.

-i 'etoiles%d.png' : Cette option indique à ffmpeg de prendre en compte une séquence d'images en entrée pour créer le GIF. Les images sont nommées 'etoiles0.png', 'etoiles1.png', 'etoiles2.png', et ainsi de suite, avec '%d' permettant de représenter que des nombres entiers (pas de 001).

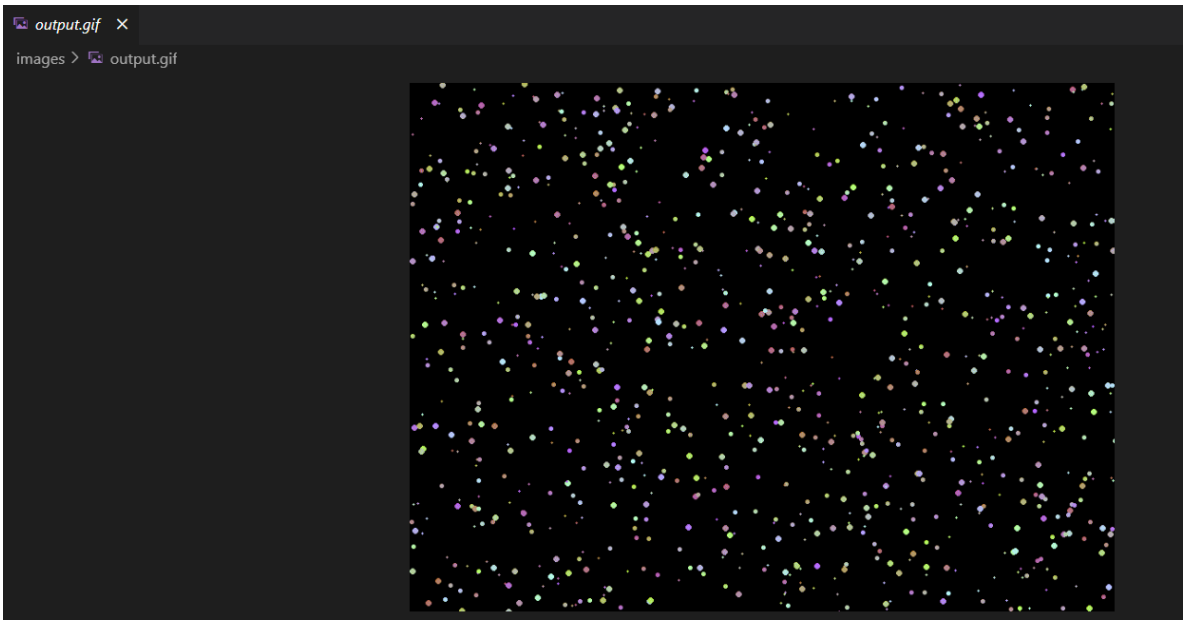
-pix_fmt yuv420p : On spécifie maintenant le format de pixel pour la sortie. Dans ce cas, il est défini sur 'yuv420p', un format couramment utilisé pour les vidéos et les GIF animés.

output.gif : Il s'agit du nom du fichier de sortie. Le GIF résultant sera enregistré sous

ce nom de fichier.

En somme, cette ligne de commande utilise ffmpeg pour créer un GIF animé à partir de 400 images nommées 'etoiles0.png', 'etoiles1.png', etc., avec un taux de 25 images par seconde, en utilisant le format de pixel 'yuv420p', et en sauvegardant le résultat sous le nom de fichier 'output.gif'.

Le résultat obtenu est donc une animation gif d'étoiles en mouvement et cette animation boucle bien sur elle-même.



TP 2

2.1 Le bruit de Perlin

2.1.1 Le ciel étoilé

Le bruit de Perlin est une méthode de génération de motifs cohérent et visuellement agréable qui est couramment utilisée en informatique, en graphisme et en animation. Il a été inventé par Ken Perlin en 1983. Le bruit de Perlin est utilisé pour créer des textures, des terrains, des effets de turbulence, et d'autres types de motifs aléatoires réalistes.

L'idée fondamentale du bruit de Perlin est de créer un motif de bruit qui est généré par superposition de différentes fréquences de sinusoïdes de différentes amplitudes. Le résultat est un bruit qui semble naturel et aléatoire, mais qui conserve une certaine cohérence visuelle.

Le bruit de Perlin est largement utilisé pour simuler des surfaces rugueuses, des textures et des variations aléatoires de manière réaliste, et dans notre cas, un ciel étoilé.

En effet, notre objectif est d'utiliser l'image précédente du ciel étoilé (qui nous est donnée) et d'ajouter du bruit pour agrémenter le ciel, ce bruit donne l'impression de nuages stellaires dans le ciel étoilé, créant ainsi une belle composition visuelle.

Pour générer ce bruit nous utilisons la librairie noise qui s'occupe de créer le bruit de perlin.

Voici l'image initial avec le ciel étoilé :

Notre objectif est alors de créer une nouvelle image vide "imagAvecDuBruit" qui possède les mêmes dimensions que l'image d'étoiles.

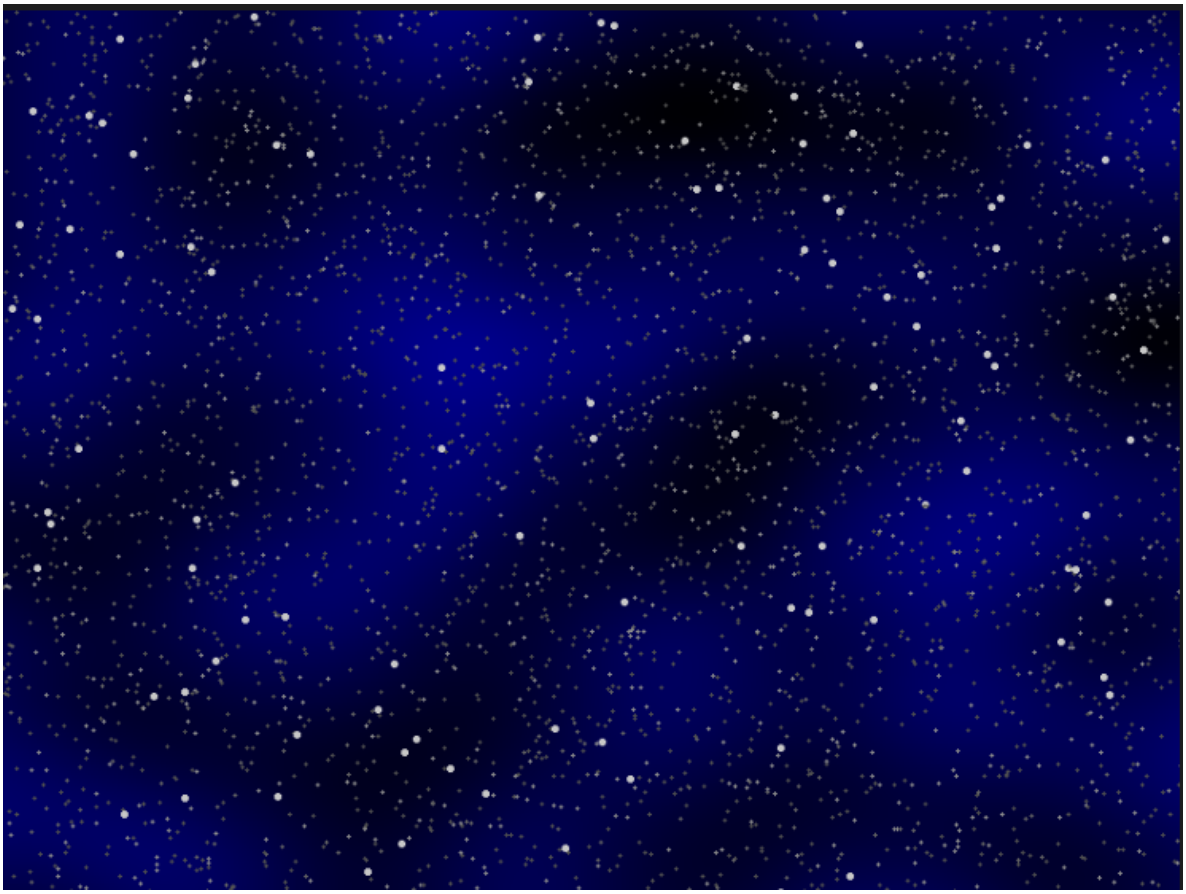
Puis d'ajouter à cette image, pour chaque pixel, (grâce à une boucle qui parcourt tous les pixels de l'image), une valeur de bruit en utilisant la bibliothèque noise, en fonction des coordonnées normalisées du pixel.

La valeur de bruit est ensuite utilisée pour déterminer la composante bleue de la couleur du pixel. En effet, dans le but d'avoir un nombre compris entre 0 et 255 (valeur utilisable pour jouer sur la couleur), la valeur de bruit est ajustée (ValeurDuBruit + 0.5) * 140.

Les composants rouge et vert sont eux laissés à zéro créant ainsi une image avec des tons de bleu définis par le bruit.

Finalement, nous superposons l'image contenant le bruit bleu (ImageAvecDuBruit) sur l'image d'étoiles à l'aide de la fonction Image.alpha_composite. Cela crée une image finale où le bruit est combiné avec l'image d'étoiles, tout cela, en préservant la transparence.

Voici, alors notre image du ciel avec ses "nuages stellaires" :



2.1.2 Génération d'une animation

Nous avons réussi, ci-dessus, à créer une image perturbée. L'objectif est donc maintenant de générer plusieurs images en créant un déplacement, dans le but de générer une animation.

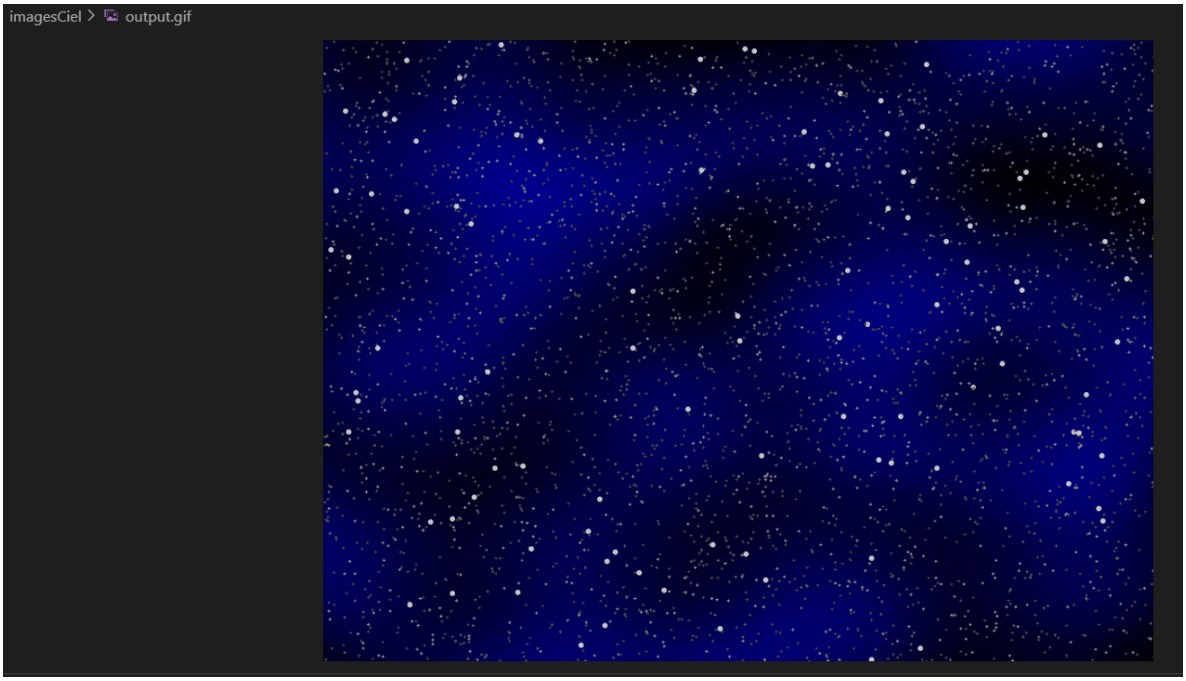
Pour ceci, nous avons décidé de générer 400 images et pour chaque image de modifier (de façon évoluante) la position du bruit.

Alors, nous créons dans un premier temps, l'image0 qui est à la position initiale puis l'image1 qui est à 0.005 plus en haut à gauche, puis l'image2 qui est à 0.01 plus en haut à gauche et ceci jusqu'à obtenir 400 images.

Puis, il nous suffit de lier ces 400 images pour obtenir un gif qui boucle sur lui-même, avec la commande ffmpeg que j'ai expliqué précédemment.

```
\images>ffmpeg -framerate 25 -i image%03d.png -pix_fmt yuv420p output.gif
```

On obtient alors une animation gif :



2.1.3 Image sinusoïdal

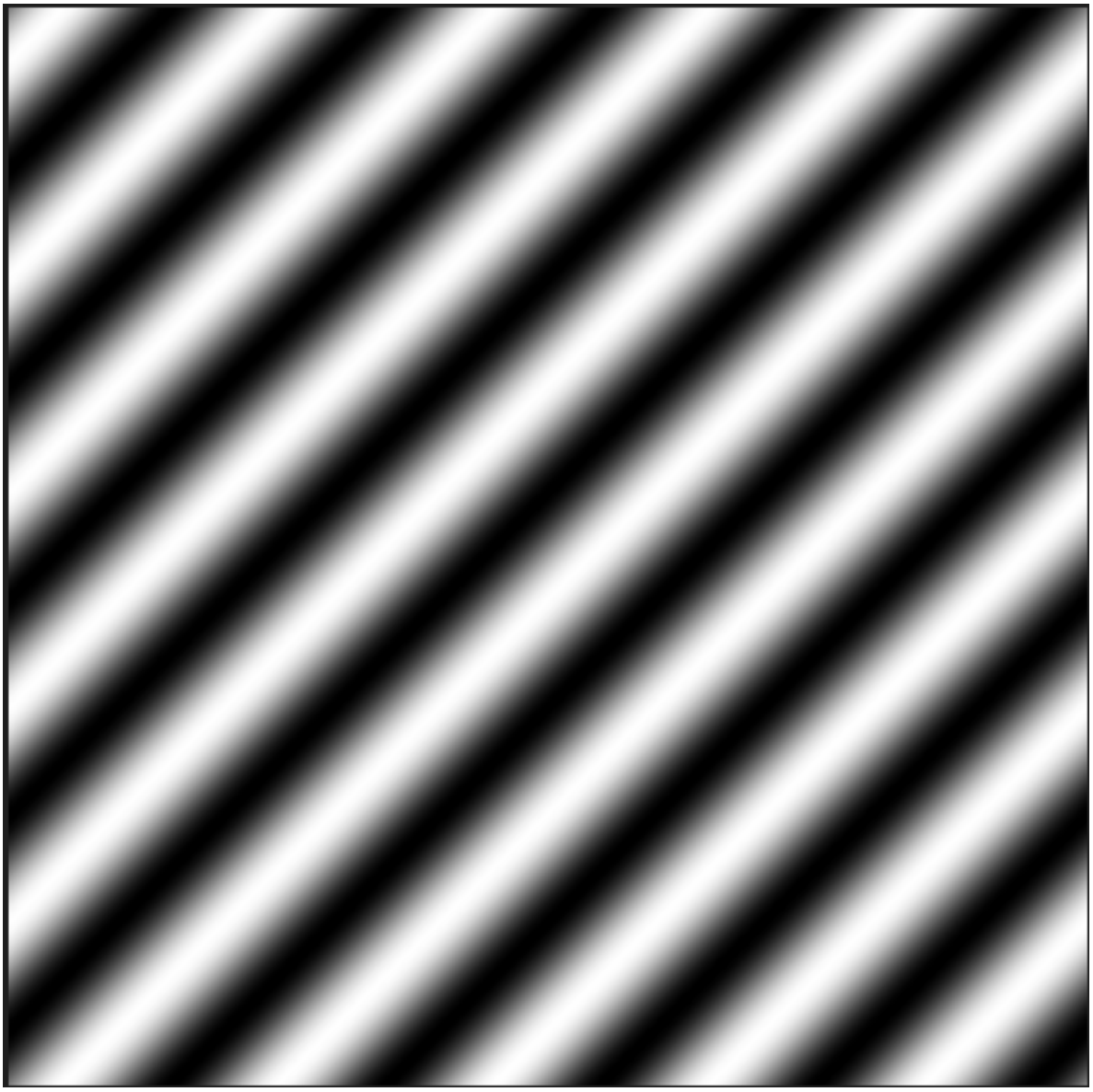
Nous pouvons, alors, utiliser une image où la couleur du pixel x,y varie selon le sinus d'une fonction de x+y, pour lui ajouter du bruit de Perlin.

Pour ceci nous devons, dans un premier temps, créer cette image simplement, puis lui ajouter de bruit de Perlin.

L'image que nous voulons créer est une image pour laquelle les pixels situés dans le sinus d'une fonction x+y sont noirs. Pour espacer ces bandes noires qui sont créées, nous avons choisi une périodicité de 100. En fonction de cette périodicité, les bandes diagonales qui apparaissent sont plus ou moins espacées.

Nous calculons alors, la valeur du sinus de cette fonction (avec la bonne périodicité), puis nous normalisons cette valeur, pour qu'elle soit comprise entre 0 et 255 (ce qui nous permet de jouer sur les couleurs).

Finalement, nous créons une image d'après cette valeur du sinus normalisée :



L'objectif est donc maintenant, de perturber cette image grâce au bruit de Perlin.

Pour ceci, on commence par créer une grille de points en x et y couvrant toute la taille de l'image et tous les pixels de l'image. Cette grille de points servira de base pour générer les valeurs du sinus perturbé par le bruit.

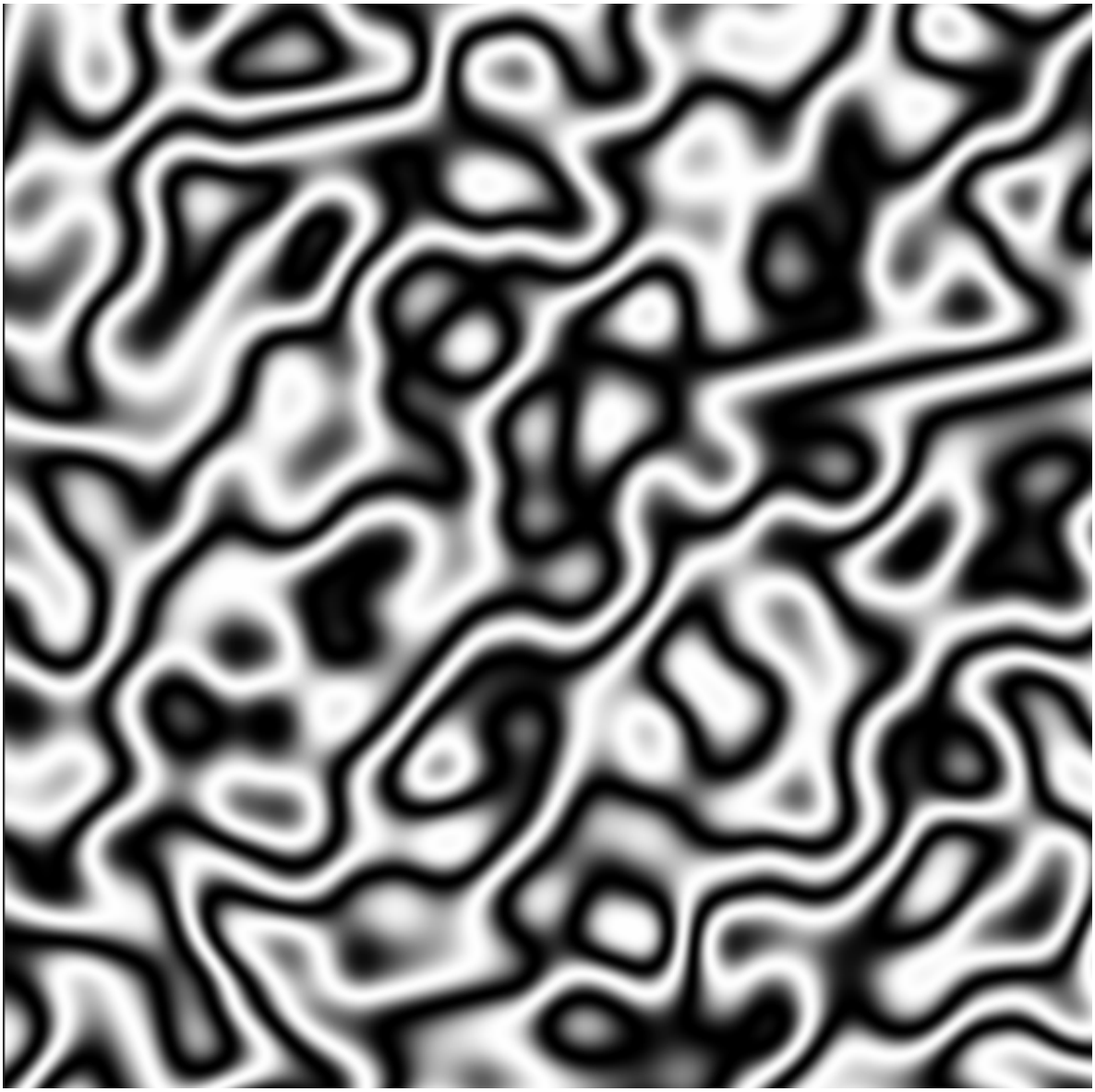
Ensuite, le bruit de Perlin est généré à l'intérieur d'une boucle for imbriquée qui parcoure les pixels de l'image. Le bruit est, donc, calculé pour chaque pixel à l'aide de la fonction noise.pnoise2.

Le bruit de Perlin est ensuite ajouté aux coordonnées x et y de la grille de points (qu'on a créé précédemment), créant ainsi de nouvelles coordonnées x_with_noise et y_with_noise.

Finalement, La fonction sinus perturbée est calculée pour chaque pixel en utilisant les

coordonnées x_with_noise et y_with_noise avec la périodicité (100 pour l'instant). Les valeurs de sinus sont aussi normalisées. (pour que l'on jour sur les couleurs).

On obtient alors une image perturbée avec du bruit de Perlin :



2.1.4 Génération d'une animation

Nous avons réussi, ci-dessus, à créer une image perturbée. L'objectif est donc maintenant de générer plusieurs images avec une périodicité évoluante, dans le but de générer une animation.

Pour ceci, comme pour l'animation des étoiles, nous avons décidé de générer 400 images et pour chaque image de modifier (de façon évoluante) la périodicité du sinus.

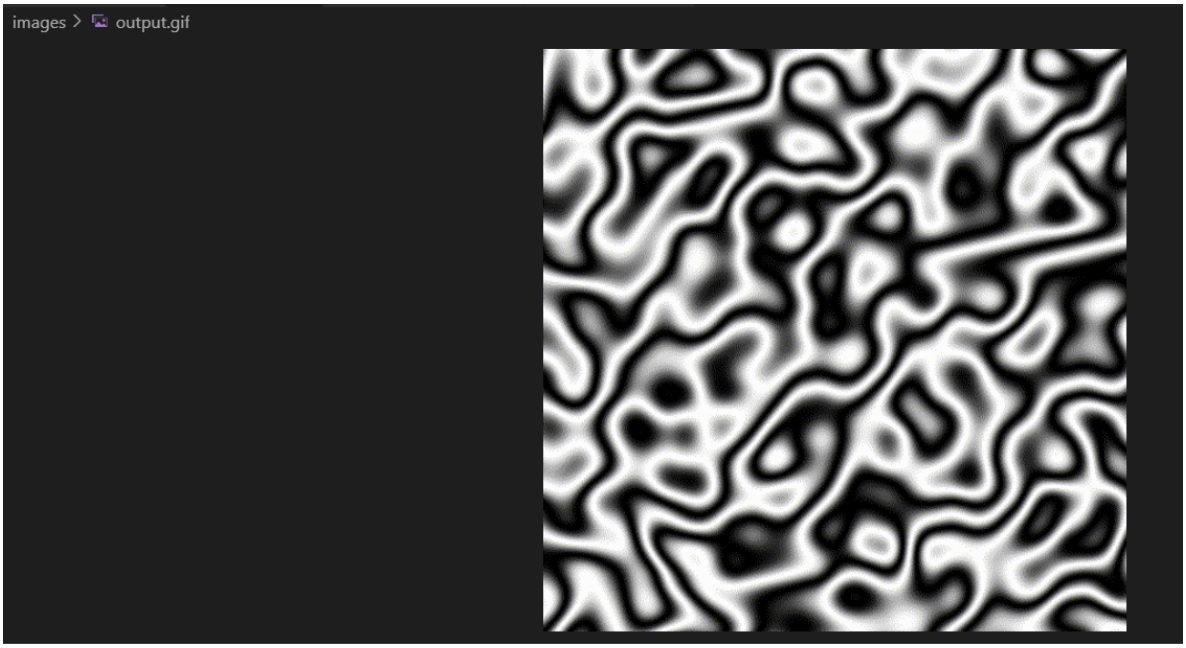
Alors, nous créons dans un premier temps, l'image0 avec une périodicité de 100 puis

l'image1 avec une périodicité de 100,1, puis l'image2 avec une périodicité de 100,2 et ceci jusqu'à obtenir 400 images.

Puis, il nous suffit de lier ces 400 images pour obtenir un gif qui boucle sur lui-même, avec la commande ffmpeg que j'ai expliqué précédemment.

```
\images>ffmpeg -framerate 25 -i image%03d.png -pix_fmt yuv420p output.gif
```

On obtient alors une animation gif :

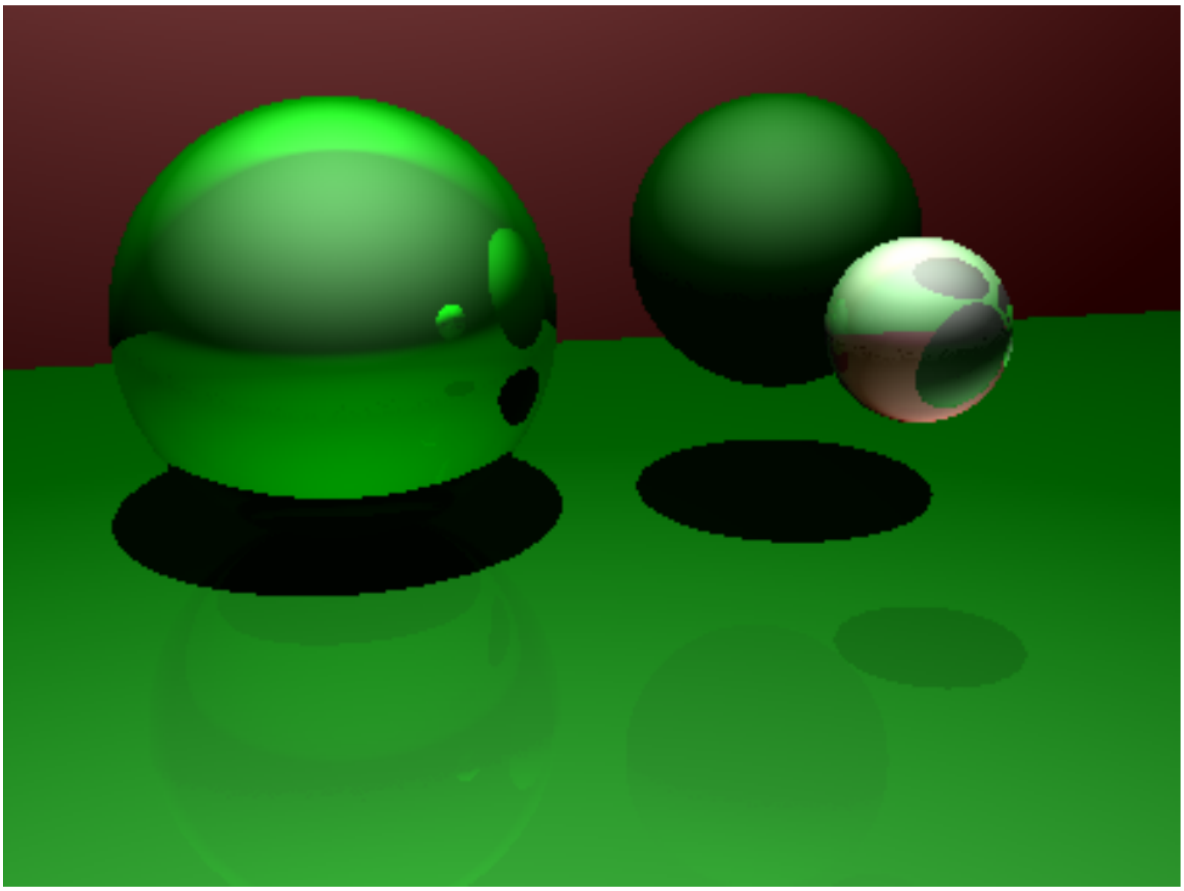


TP 3

3.1 Lancer de rayons

Dans ce troisième TP, nous allons travailler sur les textures, en particulier sur la transition des textures 3D en textures 2D.

Pour ceci, nous possédons une image initiale qui ne possède aucune texture particulière :



Notre objectif est alors de changer les textures de l'image partie par partie

- Le plan vertical : le mur
- Le plan horizontal : le sol
- Les 3 sphères de tailles et de matériaux différents

3.1.1 Le mur

Nous avons alors dans un premier temps créer une classe "RVB Damier" qui possède une fonction nommée getColor. Cette fonction donne au mur la texture d'un damier.

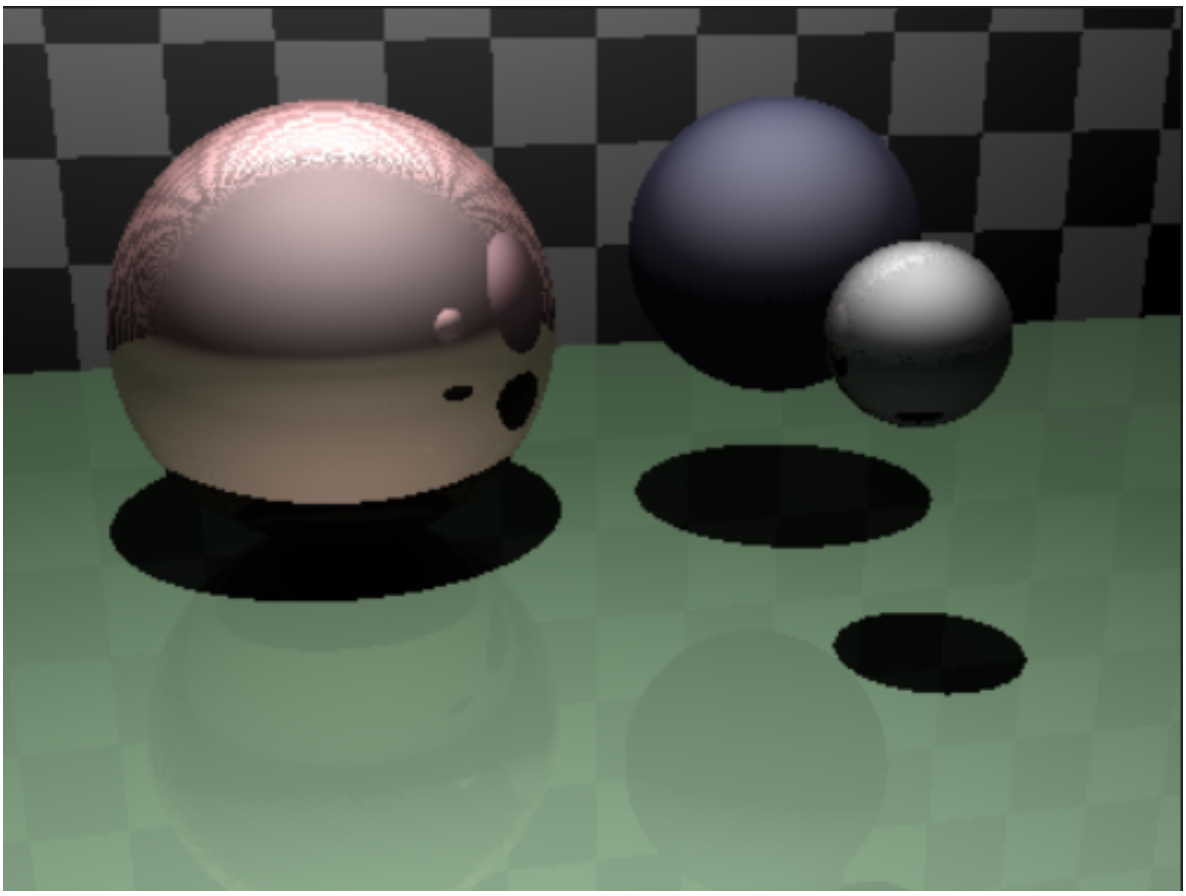
Cette fonction calcule donc les variables "celluleX" et "celluleY". Ces valeurs sont calculées en prenant la partie entière des coordonnées "UEchelle" et "VEchelle",.

Ensuite, ces valeurs sont prises modulo 2 pour obtenir un résultat qui alterne entre 0 et 1.

Finalement, la couleur du point dans le damier est sélectionnée en fonction de la valeur de la cellule. En effet, si la somme de "celluleX" et "celluleY" est un nombre pair (0 ou 2), alors la case est noire, sinon, la couleur est blanche.

Si l'on donne au mur cette texture grâce à la commande "setTexture (new Damier ..)"

On obtient cette image :



Pour donner cette texture au mur, nous avons alors créé nous-même le damier et l'avons "posé" sur le mur.

Pour autant, les textures suivantes, contrairement au damier, seront des images (préexistantes) que l'on va poser sur différentes formes de façon à les habiller.

3.1.2 Les textures à partir d'une image

Pour les textures restantes (des sphères et du sol), nous avons utilisé une classe "RVB Bitmap" avec la fonction getColor.

Cette fonction permet d'obtenir la couleur d'un point dans une image bitmap en fonction de ses coordonnées.

On calcule alors les variables "UEchantillon" et "VEchantillon" (qui sont des coordonnées d'échantillon de l'image bitmap) en fonction des coordonnées initiales de l'image et en ajoutant un décalage (offsetX et offsetY qui permettent par exemple de lier l'origine (0,0) de l'image bitmap au point d'origine de la texture, tout ceci en fonction de l'échelle.

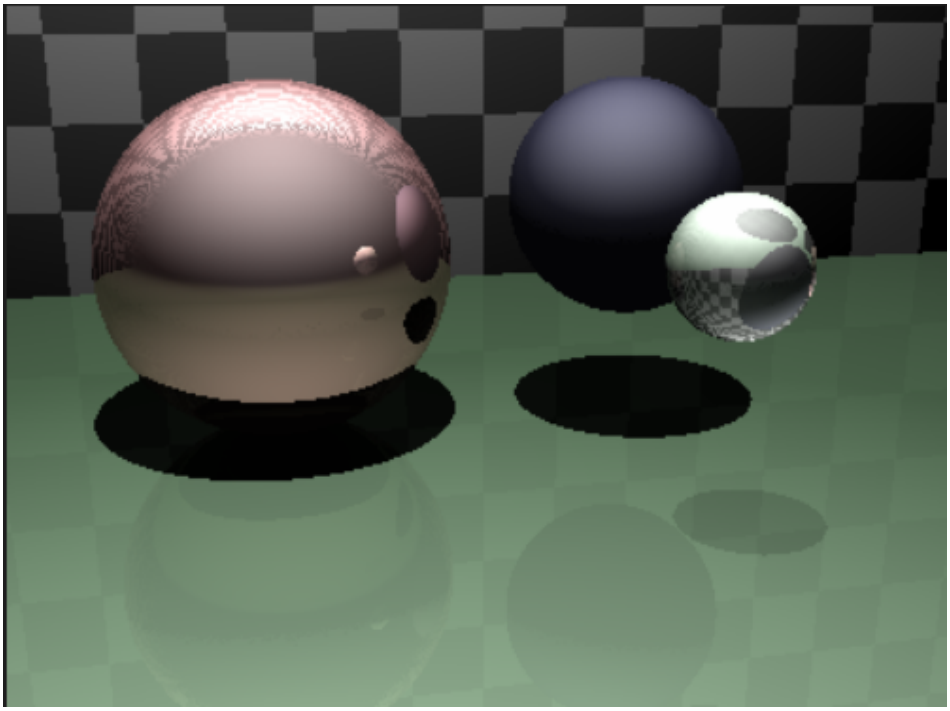
On normalise alors ces valeurs pour qu'elles appartiennent à l'intervalle [0,1].

Puis, nous calculons "bitmapX" et "bitmapY", ces valeurs s'obtiennent en multipliant les coordonnées d'échantillon en fonction des dimensions de l'image bitmap. Cela permet de traduire les coordonnées d'échantillon en coordonnées de pixel dans l'image bitmap.

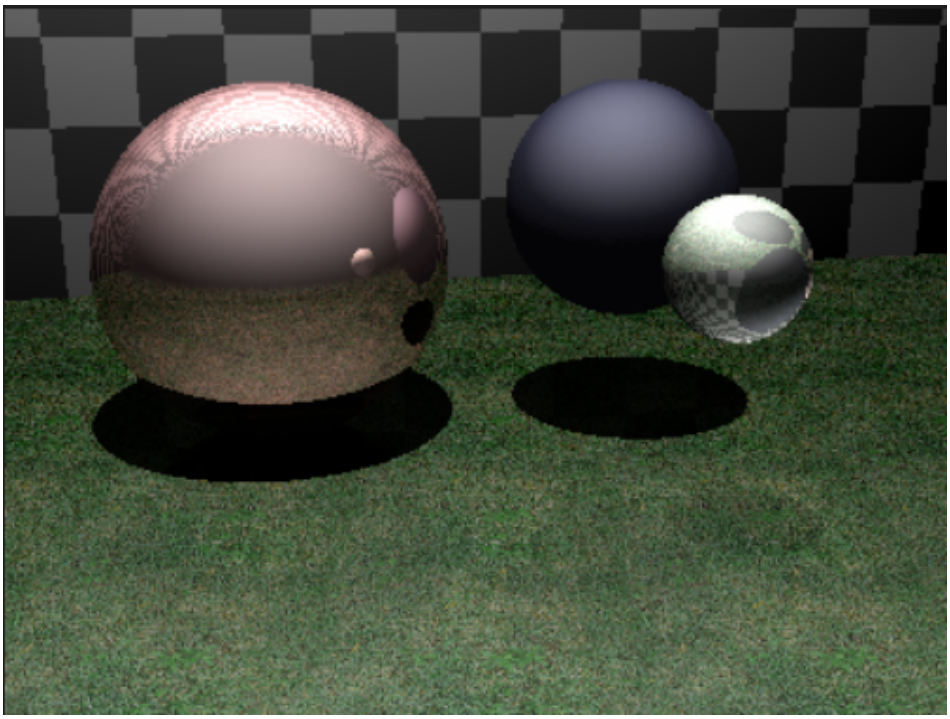
Finalement, la couleur du pixel est extraite de l'image bitmap en utilisant les coordonnées calculées précédemment avec la fonction "map.MapRVB." Cette fonction renvoie une couleur RVB qui correspond au pixel de l'image bitmap.

En parallèle, quand on instancie un objet comme le sol par exemple, on lui donne la texture (grâce à la fonction "setTexture") avec une image Bitmap de l'image qui doit servir de texture("Textures/herbe.ppm" par exemple).

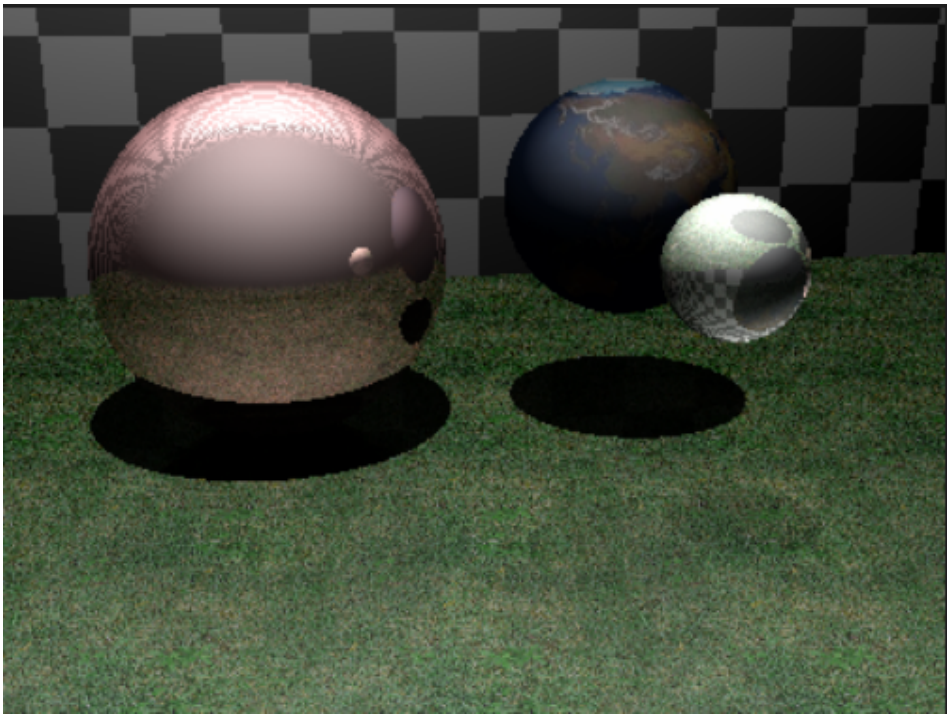
Voici l'image avec la sphère en verre :



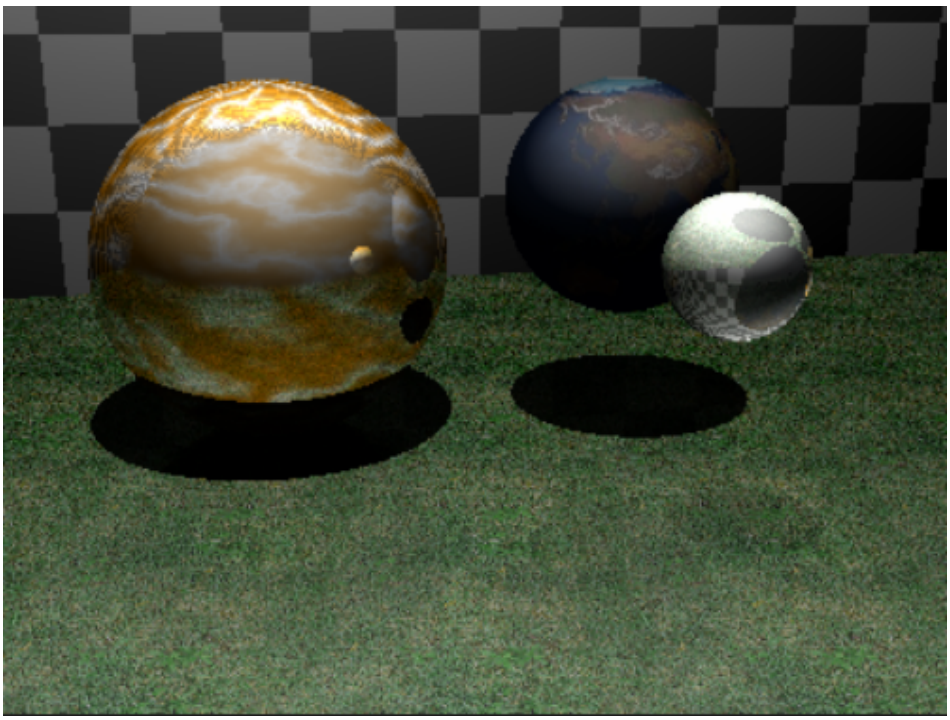
Voici l'image avec le sol :



Voici l'image avec la sphère qui représente la Terre :



Voici l'image avec la grosse sphère :



Nous pouvons alors voir ci-dessus l'image finale avec toutes les textures. On lui a alors donné les textures des images bitmap (pour la Terre ou le sol par exemple) et on a créé nous-mêmes les textures pour le mur en damier.