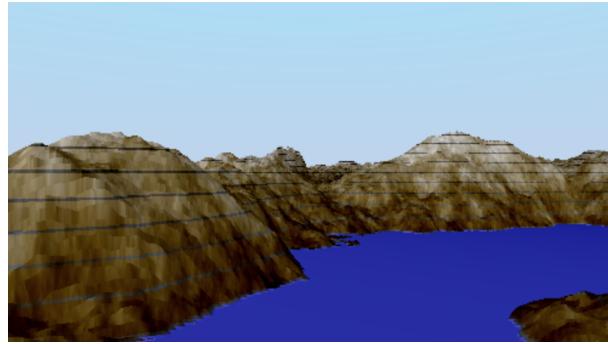


CMSI - Synthèse d'image

Génération d'un paysage virtuel

Dans ce TP de 2 séances, vous allez expérimenter sous ShaderToy quelques méthodes de synthèse d'images. Une évaluation par binôme sera organisée lors de la deuxième séance de TP. Le sujet a été pensé pour un maximum d'autonomie avec des questions/réponses au fur et à mesure de l'avancement pour vous assurer d'avoir bien compris avant de continuer. Jouez le jeu!



1 Fonctionnement du shader

Le shader que vous allez programmer est écrit dans le langage GLSL dont la syntaxe est très proche de celle du langage C. Voici quelques éléments de syntaxe qui vous seront utiles :

- Le typage est fort. Toutes les constantes doivent être systématiquement typées, par exemple un flottant devra s'écrire `1.0` et non `1`. Il existe des types pour les vecteurs : `vec2`, `vec3`, et aussi `vec4`. Pour accéder à un élément de ces vecteurs, il suffit d'écrire le membre, par exemple `a.x` pour la composante `x` du vecteur `a`. On peut construire un sous vecteur en utilisant plusieurs composantes, par exemple `a.xy` est un vecteur de taille 2 qui contient les deux premières composantes du vecteur `a`. La précision double n'existe pas actuellement sur les pipelines graphiques, donc pour coder un réel vous utiliserez un `float`.
- On peut déclarer des paramètres en sortie en utilisant le mot-clé `out`. Par défaut, un paramètre est uniquement en entrée. Le mot-clé `inout` permet d'avoir un paramètre en entrée et en sortie.
- La fonction principale à définir dans le cadre de ShaderToy est `mainImage`. Elle renvoie la couleur `fragColor` sous la forme d'un vecteur de dimension 4 (composantes rouge, vert, bleu et alpha pour la transparence) et prend en entrée la coordonnée du pixel considéré `fragCoord` sous forme d'un vecteur de dimension 2. Il faut donc décrire une image comme étant une fonction du plan :

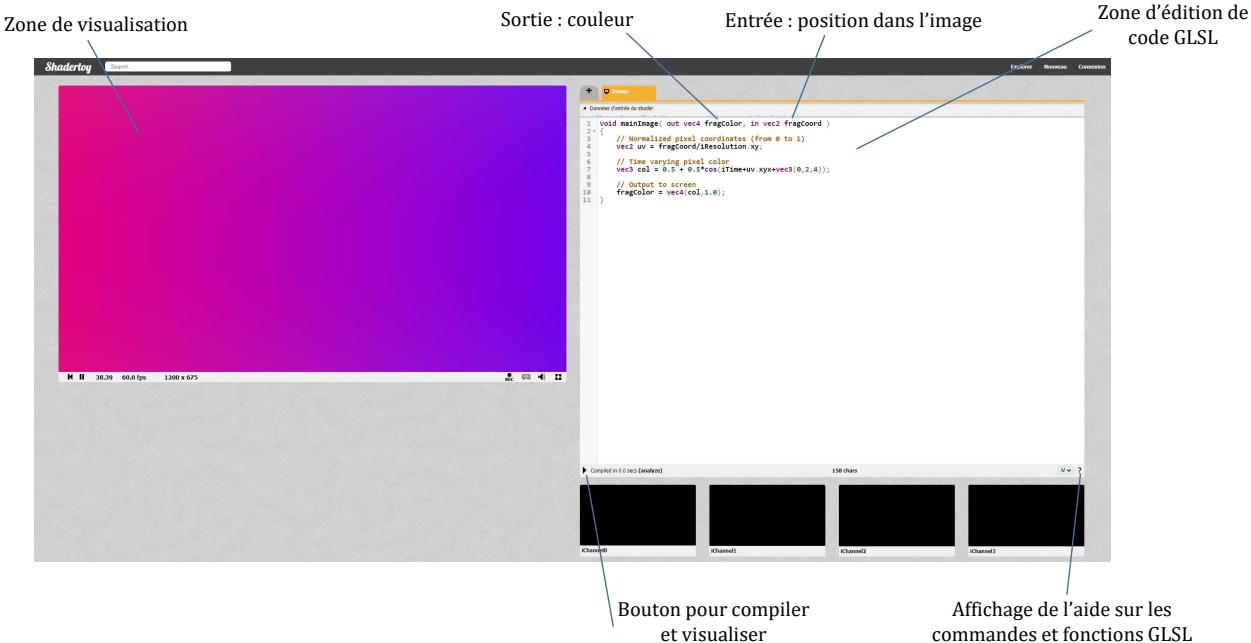
$$\begin{aligned} f : \mathbb{R}^2 &\rightarrow \mathbb{R}^3 \\ \mathbf{p} &\mapsto f(\mathbf{p}) \end{aligned}$$

où `p` représente `fragCoord` et `f(p)` est `fragColor`. Attention, les coordonnées des pixels sont celles des centres de pixels, donc ne sont pas entières mais des demi-s pour exemple (0.5, 0.5) pour le premier pixel.

La programmation d'une telle fonction peut s'avérer difficile pour représenter des objets complexes, en revanche chaque pixel peut être calculé de manière indépendante de son voisin, ce qui fait que le calcul est complètement parallélisable, et c'est ce qui est fait sur la carte graphique.

2 Interface de l'outil

Aller sur la page web <https://www.shadertoy.com/> et cliquer sur nouveau. L'interface ressemble à cela :



Le shader qui vous est proposé fait uniquement une dégradé de couleur en fonction de la position du point dans l'image et de manière animée dans le temps grâce à la variable intégrée `iTime`. Observez le code et comprenez le entièrement avant de continuer.

Questions : (réponses à la fin du sujet)

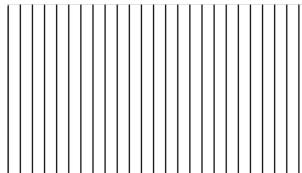
- Q1 À quoi sert la variable `iResolution`? Sa valeur peut-elle changer?
- Q2 Que signifie le symbole `/` dans l'expression `fragCoord/iResolution.xy`?
- Q3 Est-ce facile de débuguer et donc connaître la valeur d'une variable dans le code?
- Q4 Quel est l'intervalle de valeur pour chaque composante couleur de `fragColor`?

3 Premiers pas

Afin de vous familiariser avec l'outil ShaderToy, nous allons commencer par écrire des shaders très simples.

3.1 Texture à rayures

Programmez un shader qui affiche une texture à rayures comme celle-ci :

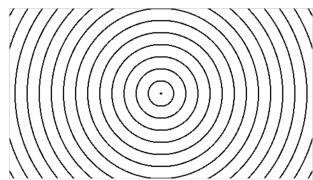


Astuce : pour raisonner en pixel, il est préférable de ne pas normaliser les coordonnées du pixel comme c'était fait dans le shader par défaut.

Enregistrer ce shader sous `step0.gls1` avant de le modifier (il n'y a pas de fonctionnalité pour enregistrer sous dans l'interface, vous devrez faire un copier-coller dans un éditeur de texte).

3.2 Texture à rayures radiales

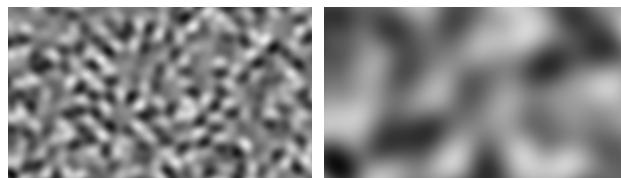
Modifier ce shader pour qu'il affiche une texture à rayures radiales comme celle-ci :



Enregistrer ce shader sous `step0b.gls1` avant de le modifier.

3.3 Fonction de bruit

Vous allez maintenant afficher une fonction de bruit en utilisant le code fourni (fichier `turbulence.gls1` sur moodle). La fonction `noise` prend en argument une position 2D et renvoie des valeurs aléatoires interpolées dans l'intervalle $[-1, 1]$. La fréquence spatiale de cette fonction est d'environ 1 ce qui signifie que si vous déplacez l'entrée d'une distance de 1 la fonction oscille en moyenne une fois. Appeler `noise(2.0*p)` revient donc à multiplier cette fréquence par 2 (deux fois plus d'oscillations). Essayez d'obtenir ces résultats en faisant appel à la fonction `noise` et en lui donnant comme argument la position 2D du pixel multipliée par une fréquence à déterminer :



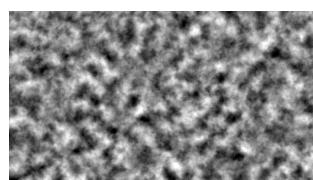
Pour obtenir des choses plus riches visuellement, nous allons maintenant mélanger ces bruits de base à plusieurs fréquences différentes, grâce à la fonction `turbulence` fournie. Observez la fonction et répondez aux questions suivantes.

Q5 Les paramètres `amplitude` et `attenuation` sont liés au réglage de l'amplitude, quelle est leur signification ?

Q6 Les paramètres `fbase` et `noctave` sont plutôt liés à la fréquence, quelle est leur signification ?

Q7 Donner des intervalles de valeurs pour les paramètres `attenuation` et `noctave`

Vous pouvez maintenant essayer d'obtenir le résultat suivant en utilisant la fonction `turbulence` et les paramètres adéquats :



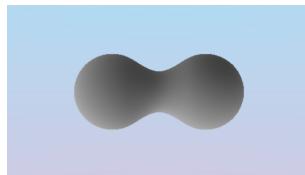
4 Ray marching

Nous allons maintenant passer à un shader plus complexe qui permet de faire du ray-marching. Le ray-marching est un dérivé du lancer de rayon. Il consiste à se déplacer le long d'un rayon jusqu'à ce que la surface (typiquement une surface implicite) soit traversée. Les fonctions implicites sont expliquées dans le chapitre *Modélisation implicite* du cours, vous pouvez vous y reporter. Retenez bien qu'à partir de maintenant, nous avons deux types de coordonnées qui sont utilisées :

- coordonnées 2D dans l'espace de l'écran, ces coordonnées servent à calculer les paramètres du rayon (origine et direction) qui va venir intersester la scène;
- coordonnées 3D dans l'espace de modélisation, celui de la fonction implicite. Dans cet espace la direction verticale est la deuxième coordonnée y .

Il est important de ne pas les confondre !

Prenez le code de départ¹ qui se trouve sur moodle (fichier `ray-marching.gls1`) et mettez le dans l'interface de ShaderToy. Vous devriez voir ce résultat :



Ce code possède déjà les fonctions suivantes implémentées :

- `mainImage` est le point d'entrée. Cette fonction permet de calculer un point de vue (variable `ro`), une direction en fonction du pixel (variable `rd`). Elle appelle ensuite la fonction `Trace` et en fonction du résultat va afficher le fond `background` ou la surface shadée. Le point de vue dans ce code de départ dépend de la position y de la souris de telle sorte qu'un zoom est possible.
- `Trace` est la boucle de ray marching. Elle évalue la fonction `object` le long de la trajectoire des rayons avec un pas `Epsilon` jusqu'à ce que la fonction change de signe ou que l'on soit trop loin (variable `rB`).
- `Shade` permet de donner la couleur de la surface en fonction de la position `p`, de la normale `n` et du nombre de pas utilisés avant de frapper la surface `s`.
- `object` est la fonction implicite à visualiser. Par défaut dans le code donné, il s'agit d'un mélange de deux primitives ponctuelles.

Pour comprendre un peu mieux le fonctionnement de ce shader, vous allez faire les expérimentations suivantes :

- Essayez de diminuer la valeur de la constante `Epsilon` qui se trouve au début du programme en la positionnant à 0.01. Que se passe-t-il ?
- Pour faire réapparaître l'objet, vous allez devoir augmenter la valeur de la constante `Steps` au début du programme aussi (mettez là à 10000). L'objet a réapparu, mais que constatez-vous ?
- Pour revenir à des performances raisonnables, nous allons modifier une ligne du programme, la ligne 94 :
`t += max(Epsilon, -v/2.0);`

Q8 Tenter de donner une interprétation de l'amélioration des performances.

Q9 Que représente la constante `2.0` ?

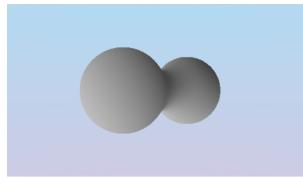
Vous pouvez maintenant remettre `Steps` à une valeur plus raisonnable de 1000.

4.1 Rotation

Implémenter la fonction `rotateY` qui prend en argument le point `p` et un angle (dans le code distribué cette fonction existe mais renvoie le même point que celui donné en argument). Cette fonction doit effectuer une rotation autour de l'axe y du repère 3D. La fonction `mainImage` fait déjà appel à cette fonction de telle sorte que vous n'avez qu'à implémenter la rotation pour que cela soit effectif (la position x de la souris donne l'angle de rotation).

Une fois que cela est fait, vous devriez pouvoir tourner autour de l'objet :

1. Eric Galin est en grande partie l'auteur de ce fichier de départ



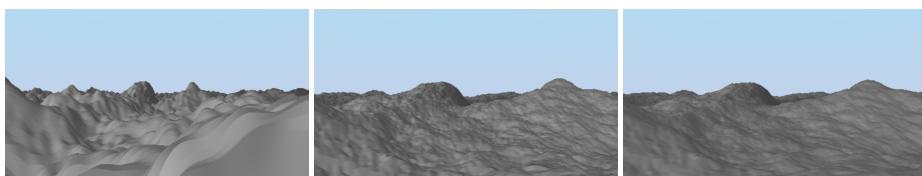
Enregistrer ce shader sous `step3.gls1` avant de le modifier.

5 Terrain par somme de bruits

Nous allons maintenant remplacer la surface visualisée par celle d'un terrain, qui sera calculé par une somme de bruits (fonction `turbulence` utilisée précédemment).

Dans la fonction `object`, remplacer le contenu par un appel à la fonction `Terrain` que vous allez implémenter. Cette fonction doit renvoyer une valeur qui indique si on est à l'extérieur (valeur négative) ou à l'intérieur (valeur positive) du terrain. Pour un terrain qui n'est pas un objet fermé cela n'a pas beaucoup de sens, donc on va considérer que les valeurs négatives correspondent à des points au dessus du terrain et positive en dessous de celui-ci. Déduire quelle va être l'expression à utiliser pour avoir ces propriétés. Attention, la fonction doit être continue, il est donc exclu de faire un test et de renvoyer `-1` ou `1` par exemple.

En jouant sur la fréquence de base, l'amplitude, le nombre d'octaves et l'atténuation, essayez d'obtenir les résultats semblables à ceux-là :



Vous pourrez avoir recours à la modification des lignes suivantes afin de modifier le point de vue :

```
vec3 rd = vec3(aspect*pixel.x, pixel.y, -4.0);
vec3 ro = vec3(0.0, 0.0, 15.0);
```

Le vecteur `rd` donne la direction du rayon (que vous pouvez monter ou baisser en changeant sa deuxième composante) et le vecteur `ro` indique l'origine du rayon que vous pouvez aussi changer pour voir la scène de plus ou moins haut.

Enregistrer ce shader sous `step4.gls1`.

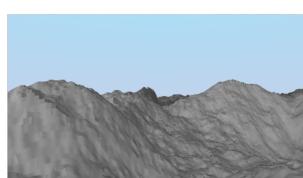
5.1 Ridged noise

Afin de voir apparaître des crêtes sur votre terrain, vous allez créer une fonction de bruit que vous appellerez `ridged` et qui va introduire une valeur absolue dans son calcul :

$$r(p) = 2(0.5 - |0.5 - n(p)|)$$

où `n` est la fonction de bruit classique `noise` et `r` est la nouvelle fonction de bruit.

Modifiez la fonction `turbulence` pour qu'elle fasse appel à cette nouvelle fonction de bruit, vous devriez obtenir un terrain comme celui-là :



Enregistrer ce shader sous `step5.gls1` avant de le modifier.

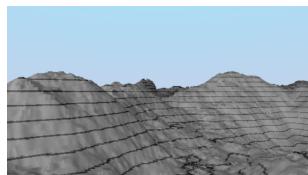
6 Construction d'une texture

Nous allons ici travailler sur les couleurs du terrain. Pour l'instant, la couleur implémentée dans la fonction `Shade` utilise toujours la même couleur grise de base, et permet d'afficher un *shading* diffus en fonction d'une direction de lumière.

6.1 Textures hypsométriques

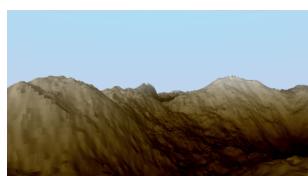
Nous allons construire des textures dont la couleur dépend uniquement de l'altitude (qui est représentée par la composante y dans notre cas).

Modifier la fonction `Shade` pour qu'elle affiche des courbes de niveau :



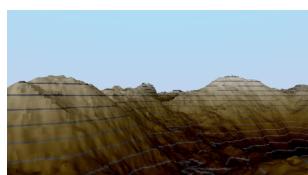
Enregistrer ce shader sous `step6.gls1` avant de le modifier.

Modifier la fonction `Shade` pour qu'elle affiche un dégradé de couleurs en fonction de l'altitude :



Enregistrer ce shader sous `step7.gls1` avant de le modifier.

Modifier le shader pour qu'il combine les deux effets précédents :

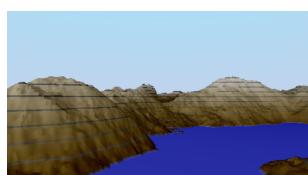


Enregistrer ce shader sous `step8.gls1` avant de le modifier.

7 Bonus

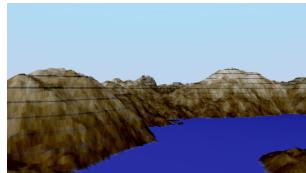
7.1 Mer

Modifier les fonctions `Terrain` et `Shade` afin d'afficher une zone d'eau :



7.2 Perturbation de couleurs

Modifier la fonction `Shade` afin de perturber les couleurs avec la fonction de turbulence :



7.3 Autres

Il est possible de faire toutes sortes de choses, laissez place à votre imagination (reflet spéculaire sur la mer, nuages, programmation d'autres primitives implicites, animations en fonction du temps, brouillard, etc.).

Réponses aux questions

Q1 La variable `iResolution` correspond à la taille de la fenêtre visualisée en pixel. C'est un vecteur de dimension deux. Cette taille change quand la fenêtre du navigateur est redimensionnée ou lorsqu'on met en plein écran la visualisation.

Q2 L'opérateur `/` fait une opération dite *pointwise*, c'est à dire qu'il prend en argument deux vecteurs de dimension deux et renvoie un autre vecteur de dimension 2, où chacune des composante est traitée séparément. Autrement dit c'est équivalent à écrire en plus court le code suivant :

```
vec2 uv;
uv.x = fragCoord.x/iResolution.x;
uv.y = fragCoord.y/iResolution.y;
```

Q3 C'est tout sauf facile! Le seul moyen de connaître la valeur d'une variable est de la visualiser par l'intermédiaire de `fragColor`. Vous avez à votre disposition trois composantes qui peuvent aider à visualiser différentes variables, mais il faudra être capable d'interpréter les couleurs qui sont affichées.

Q4 Chaque composante de couleur est entre 0 et 1, on le voit bien dans la formule qui définit `col` du fait que le cosinus est entre -1 et 1.

Q5 `amplitude` règle l'amplitude générale que l'on veut donner au signal, tandis qu'`attenuation` donne l'atténuation que l'on va avoir entre deux octaves différentes.

Q6 `fbase` est la fréquence de base, toutes les autres fréquences ajoutées seront des multiples (puissances de 2) de cette fréquence de base. `noctave` donne le nombre d'octaves qui seront sommés. Plus on ajoute d'octaves plus il y aura de détails dans le fonction générée.

Q7 `attenuation` doit être entre 0 (surface très lisse) et 1 (surface très rugueuse). La valeur standard est de 0.5. `noctave` ne doit pas être trop élevé sinon les performances vont être dégradées. Une bonne valeur est 8, mais on peut aller jusqu'à 10. Dans le cas d'une texture on peut s'arrêter quand la fréquence engendre des détails plus petits que la taille d'un pixel et ne sera pas visible.

Q8 Dans la version initiale du ray-marching, on avance d'un pas fixe `Epsilon` à chaque itération. Ce pas est aussi la précision avec laquelle on obtiendra la position de l'intersection avec l'objet. Plus `Epsilon` est petit, plus la précision est bonne mais plus les performances sont mauvaises. D'autre part, pour ne pas faire trop chuter les performances, il y a un autre paramètre qui permet de donner le maximum d'itération possible en chaque pixel, c'est la constante `Steps`. Quand on a diminué `Epsilon`, ce nombre de pas a été atteint avant de toucher l'objet, qui a donc disparu. Quand on change la ligne pour adapter le pas en fonction de la valeur de la fonction implicite, on augmente le saut fait à chaque itération, donc les performances sont meilleures. Elles ne peuvent pas être moins bonnes car on prend le maximum de `Epsilon` et d'une autre valeur.

Q9 Cette constante représente la borne de Lipschitz de la fonction implicite, c'est-à-dire la norme maximum du gradient de la fonction. Si on note ρ cette borne, on a la propriété suivante :

$$\forall \mathbf{p}, \mathbf{q} \in \mathbb{R}^3 \times \mathbb{R}^3, |f(\mathbf{p}) - f(\mathbf{q})| < \rho d(\mathbf{p}, \mathbf{q})$$

Lorsque la valeur de la fonction est v , alors on sait qu'en parcourant une distance de $-v/\rho$, on atteindra pas la valeur nulle de la fonction. Ici v est négatif, d'où le signe moins devant la formule. Si vous changez la fonction implicite visualisée, vous pouvez être amené à modifier cette valeur, par exemple dans le cas d'un terrain qui aurait une pente très élevée, votre fonction pourrait avoir une borne de Lipschitz plus grande que 2.

