Profilage
Espaces de couleur

1 Profilage

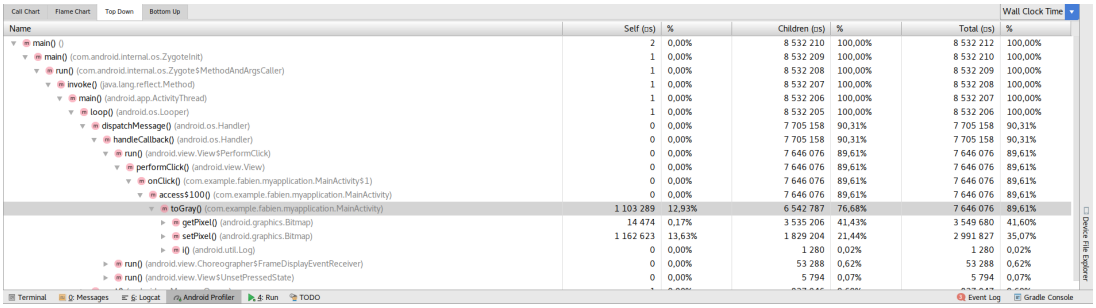
Il est primordial dans tout développement informatique d'étudier le comportement de son code, que ce soit en terme de temps de calcul (afin de déterminer les portions de codes coûteuses) ou en terme d'utilisation mémoire. C'est d'autant plus important lorsque l'on fait du traitement d'image car on implémente des algorithmes dont la complexité est toujours au moins linéaire, sur des données volumineuses. De même, c'est primordial pour les développements sur smartphones car les ressources sont limitées.

Android Studio intègre des outils de *profilage* (ainsi qu'un *profiler* réseau qui ne nous est pas utile). Voir `https://developer.android.com/studio/profile/`.

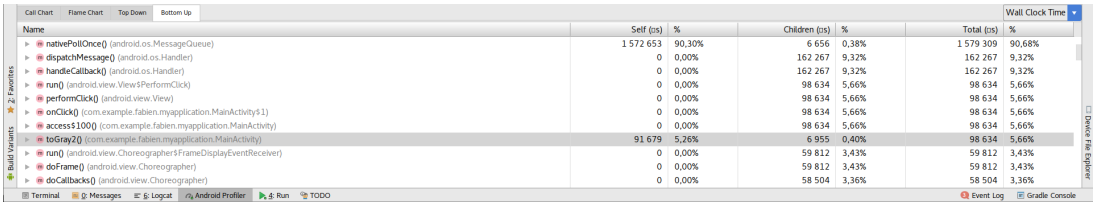
Profil CPU Le profil CPU va permettre de localiser les portions de code qui mériteraient d'être optimisées (si c'est possible !). Attention, certaines optimisations rendent le code beaucoup moins lisible, et il faut donc s'assurer que le besoin et le gain sont réels avant de l'inclure dans une release.

Une façon immédiate d'estimer un temps d'exécution consiste à utiliser l'appel `System.nanoTime()` pour mesurer le temps qui s'écoule entre deux instants. Cette solution ne permet toutefois pas d'analyser finement ce qui prend du temps dans un algorithme, à moins de rajouter des compteurs pour chaque ligne. Elle n'est donc pas utilisable en pratique pour un code conséquent.

Il est alors nécessaire d'utiliser un outil de profilage qui va permettre de suivre tous les appels de fonctions, ainsi que les temps passés dans chacune. L'utilisation du *profiler* CPU permet de voir en détail qu'effectivement la première version de la fonction de conversion en niveaux de gris prend la majorité du temps d'exécution, et ce à cause des fonctions `getPixel()` et `setPixel()`.



Cet outil permet donc de localiser le problème, et d'y remédier en remplaçant les appels coûteux, par `getPixels()` et `setPixels()`. Si on réalise les mêmes mesures sur la deuxième version du passage en niveau de gris, on obtient un temps d'exécution nettement inférieur, et on peut vérifier que le temps passé dans la fonction de conversion en niveaux de gris correspond en majorité à la boucle de l'algorithme (temps d'exécution des fils quasi nul). On ne peut donc plus améliorer, sauf à paralléliser la boucle.



Profil Mémoire Le profil mémoire permet de suivre toute les allocations/libérations de la mémoire. Cela permet de vérifier qu'il n'y a pas de fuite et surtout que l'on alloue que ce qui est nécessaire. Vérifiez notamment que vous n'allouez pas plus de bitmaps que nécessaire. Leur nombre dépend des fonctionnalités données à l'utilisateur (par exemple une fonction `undo` va justifier des duplications d'images).

2 Application

Tous les algorithmes de traitement d'image que vous écrirez seront testés via les outils de *profilage* d'Android Studio. Vous devrez bien faire apparaitre ces tests dans vos compte-rendus (notamment mesures de temps avec des images de différentes tailles, en précisant quel téléphone et quelle version d'Android ont été utilisés).

2.1 Coloriser l'image

On veut obtenir l'effet suivant :

Avant Après

Les pixels dans l'image résultat ont la même luminosité que dans l'image initiale mais ils ont changé de teinte.

— Ecrire une méthode Java `void colorize (Bitmap bmp)` qui applique cet effet au Bitmap passé en paramètre. La teinte appliquée sera choisie aléatoirement.

Indication : Les méthodes `RGBToHSV` et `HSVToColor` permettent de passer de l'espace de couleur RGB à HSV et réciproquement.
Analyser les temps de calcul. Est-il intéressant de réécrire les conversions RGB/HSV ? Justifier.

— Réécrire les conversions RGB/HSV en Java. Comparer les performances.

2.2 Conserver une couleur

— Proposer un algorithme permettant d'obtenir l'effet suivant :

Avant

Après