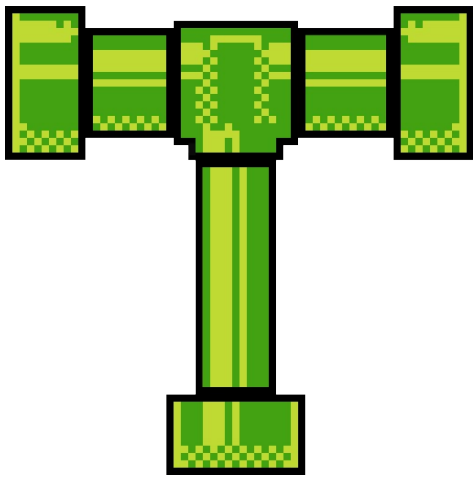


TuttleOFX

Développement d'un Plug-in de Seam Carving

Projet d'étude Master I

Ingénierie des Systèmes Images et Sons
Master ISIS

Étudiants:

Omar ALVAREZ

Valentin NOËL

Encadrant:

Marc-Antoine ARNAUD - MIKROS Image

François-Xavier COUDOUX

Université de Valenciennes

et du Hainaut-Cambrésis

Remerciements

Nous aimerions remercier Marc-Antoine ARNAUD, qui nous a proposé ce passionnant projet et sans qui nous n'aurions pas pu le mener à terme, ainsi que François-Xavier COUDOUX, qui a su transmettre son intérêt pour le sujet et sa motivation pour le projet.

Sommaire

Introduction

Ce projet a pour but le développement d'un Plug-in de *Seam Carving* pour la librairie TuttleOFX, qui est un ensemble d'outils de traitement d'images basé sur le format OpenFX.

Le projet a été réalisé dans le cadre du Master 1 ISIS à l'Université de Valenciennes et du Hainaut-Cambrésis, en une centaine d'heures environ, suivi par Marc-Antoine ARNAUD, ingénieur développeur au sein de MIKROS Image et supervisé par François-Xavier COUDOUX .

Le *Seam Carving* est une technique de redimmensionnement et de recadrages d'image assez peu connu, c'est pourquoi nous ferons une présentation du principe théorique sur lequel elle repose avant de nous intéresser au développement du programme. Enfin, nous discuterons du résultat final et des améliorations envisageables pour ce programme.

Théorie

Principe général

Le ***Seam Carving***, ou recadrage intelligent, est un algorithme de redimensionnement d'image développé par Shai Avidan et Ariel Shamir [1]. Cet algorithme redimensionne, non pas par une mise à l'échelle ou un recadrage classique par découpe, mais par le traitement des pixels dits “de moindre importance”.

L'importance d'un pixel est généralement mesuré par son contraste relatif à ses plus proches voisins, mais d'autres techniques, comme de détections de formes, peuvent être utilisées. L'algorithme du *Seam Carving* analyse les images pour trouver un certain nombre de *seams*, (chemins de pixels de moindre importance) soit pour appliquer une suppression et réduire la taille de l'image, soit pour les appliquer une interpolation permettant l'agrandissement de l'image.

De plus, il est possible de définir, ou de détecter automatiquement, des zones de grandes importances, et ainsi de les protéger de toutes suppressions. À l'inverse, on peut définir des zones devant être retirées en priorité.

Fondements théoriques et mathématiques

Les différentes techniques de redimensionnement d’image se basent sur la réduction ou augmentation du nombre de pixels de l’image, mais, quels sont les pixels à supprimer ou multiplier pour ne pas interagir avec le contenu sémantique de l'image ? Intuitivement, se débarrasser (ou interpoler) des pixels imperceptibles qui se mêlent avec leur environnement, c’est-à-dire les pixels de faible valeur énergétique, semble la meilleure solution.

En effet, on définit à la fonction d’énergie d'une image par l'expression suivante:

$$e_1(I) = \left|\frac{\partial}{\partial x} I\right| + \left|\frac{\partial}{\partial y} I\right|$$

Pour calculer l'énergie de l'image, différentes techniques sont envisageables. On utilisera par exemple le calcul du gradient qui met en évidence les contours de l'image, ou plus précisément les variations franches de luminance. Là encore, différents procédés permettent son calcul, comme par exemple les filtres de Sobel, Laplacien, Prewitt, etc. Par la suite, nous développerons principalement l'utilisation du filtre de Sobel.

Mathématiquement parlant, le filtre Sobel utilise deux masques mh et mv convolués avec l'image d'entrée pour calculer une approximation des dérivées partielles, c'est-à-dire, des changements horizontaux et verticaux de l'image. On obtient le gradient en x, G_X et y, G_Y , grâce à l'implémentation des opérations suivantes :

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

où * représente l'opérateur de convolution.

Pour chaque pixel de l'image, le gradient total est calculé comme une combinaison du gradient en X et Y, en faisant :

$$G = \sqrt{G_X^2 + G_Y^2}$$

Ensuite, on peut calculer l'énergie de l'image.

Dans l'optique de réduire la largeur de l'image, on peut établir plusieurs stratégies. Par exemple, une technique optimale pour préserver l'énergie serait de supprimer les pixels avec la plus basse énergie dans l'ordre croissant et ainsi conserver les pixels d'énergie importante. Le problème de cette technique est la perte de la forme rectangulaire de l'image car le nombre de pixels effacés n'est pas le même pour chaque ligne.

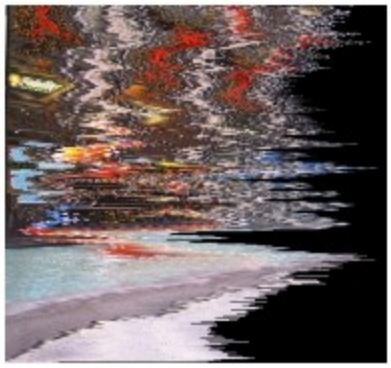


Figure 1. Suppression des pixels avec la plus basse énergie dans l'ordre croissant

Si on veut éviter ce phénomène, on doit supprimer un nombre égal de pixels de faible énergie dans chaque ligne de l'image. Néanmoins, ceci implique la destruction incohérente du contenu de l'image, créant un effet zig-zag.

Figure 2. Effet « zigzag » du à la suppression du même nombre de pixels par ligne

Par ailleurs, si on veut éviter que l'image soit déformé et garder ainsi la cohérence visuelle, on peut utiliser la technique de *Cropping*, qui consiste en un sous-fenêtrage avec la taille cible contenant la plus grande énergie. Cependant, cette technique fait perdre une partie des informations de l'image hors du cadre cible.

Figure 3. Cropping

Alors, une autre possibilité peut être l'élimination des colonnes entières de l'image de plus basse énergie. Mais des artefacts peuvent apparaître dans l'image résultante.

Figure 4. Élimination de colonnes entières

Par conséquent, on va rechercher d'une technique de redimensionnement plus "intelligente" que le *cropping* ou la suppression des colonnes, et qui permet de conserver le contenu de l'image. C'est le but de ce projet : le *Seam Carving*.

Comme son nom l'indique, cette technique fait appel à des chemin des pixels de moindre énergie, appelés *seam*. Pour une image de $n \times m$ pixels, on définit un *seam* vertical par :

$$S^x = \{s_i^x\}_{i=1}^n = \{(x(i), i\}_{i=1}^n \ , \quad \text{avec } \forall i, \ |x(i) - x(i-1)| \leq 1 \ \text{ et } x : [1, ...n] \rightarrow [1, ...m]$$

De même, un *seam* horizontal sera logiquement défini par :

$$S^y = \{s_j^y\}_{j=1}^m = \{(y(j), j\}_{j=1}^m \text{ , avec } \forall j,\ |y(j) - y(j-1)| \leq 1 \text{ et } y : [1, ...m] \rightarrow [1, ...n]$$

Dans notre cas, on cherche à mettre en évidence l'énergie minimum cumulée dans l'image, afin de faciliter et d'optimiser le tracé des *seam*. On calcule donc l'image M de l'énergie minimum cumulée grâce à la fonction :

$$M(i,j) = e(i,j) + min(M(i-1,j-1),\ M(i-1,j),\ M(i-1,j+1))$$

On applique ce tracé à l'image représentant l'énergie minimum cumulée, on sélectionne le *seam* le plus faible de l'image, et on applique le traitement souhaité (suppression ou copie) aux pixels de même coordonnées de l'image initiale. Ci-dessous on peut voir graphiquement l'ensemble des opérations permettant de trouver les *seams*.

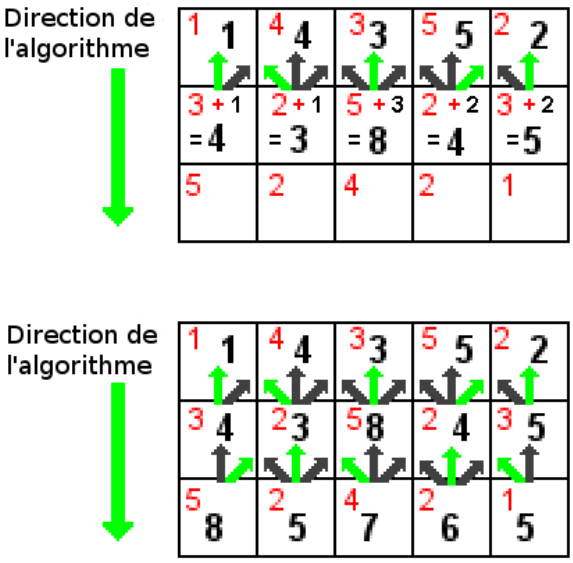


Figure 5a. Calcul de l'énergie minimum cumulée

Les valeurs en rouge correspondent à la valeur du gradient du pixel courant, et les valeurs en noir sont les valeurs de l'énergie cumulée calculés de haut en bas en fonction des pixels voisins. Une fois l'énergie cumulée calculée, on peut parcourir l'image de bas en haut suivant le chemin le moins énergétique.

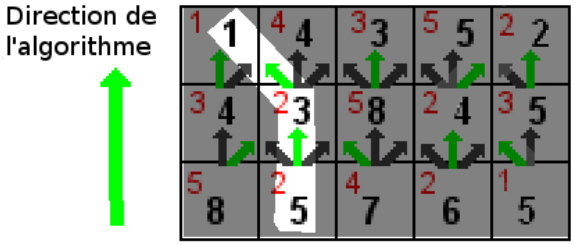


Figure 5b. Tracé du seam

Le processus que l'on vient de décrire réalise le traitement permettant de trouver les *seams* verticales de l'image. La méthode pour calculer les *seams* horizontales est complètement analogue.

Implémentation en MATLAB

Pour une bonne compréhension du sujet, nous avons choisi de développer les étapes de mise en place du processus du *Seam Carving* à travers un exemple d'implémentation dans le logiciel MATLAB.

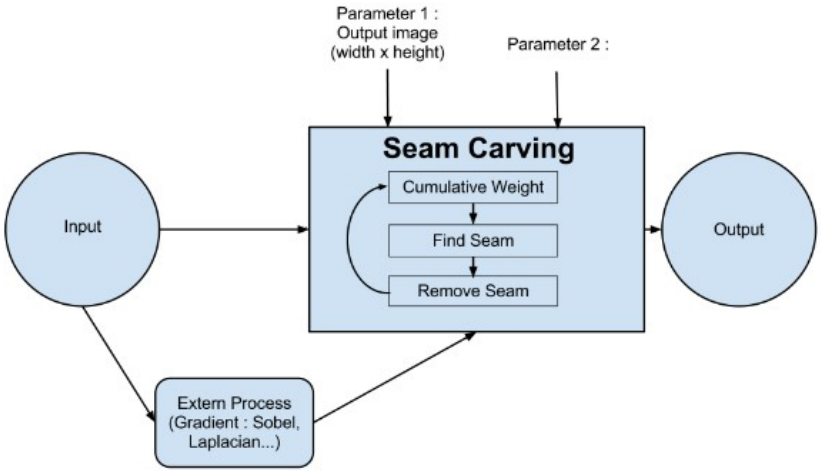


Figure 6. Schéma du processus

Nous avons choisi MATLAB car l'implémentation, les test, la détection et correction d'erreurs et l'optimisation des fonctions est plus facile, rapide et intuitive. L'idée est se familiariser avec

chaque étape du *plug-in SeamCarving*. Ci-dessous, on voit la partie principale de la fonction implémentée :

```
function [Input, X] = seam_carving( Input, iterations ),
      for i = 1: iterations,
            Luminance = rgb2gray( Input );
            lumaNormalize = double( Luminance )/255;
            inputMap = img_gradient( lumaNormalize );
            WeightedMap = processCumulativeWeight( inputMap );
            Seam = find_seam( WeightedMap );
            Input = remove_seam( Input, inputMap, Seam );
            imshow( Input );
            X(i)=getframe;
      end
end
```

La fonction *seam_carving* reçoit deux paramètres : l'image d'entrée, *Input* et le nombre de *seams*, exprimé par la différence entre taille originale et la taille de sortie.

Figure 7. Image d'entrée « Input »

Dans cet exemple, l'algorithme ne travaille que sur une dimension et réalise une réduction de la longueur de l'image. Pour un certain nombre d'itérations, défini par l'utilisateur, on va répéter les étapes suivantes :

- **Calcul de la Luminance :**

```
Luminance = rgb2gray( Input );
```

A partir d'une image RGB, on obtient sa luminance en utilisant une fonction MATLAB.

- **Normalisation de l'image :**

```
lumaNormalize = double( Luminance )/255;
```

Pour obtenir des niveaux de luminance compris entre 0 et 1 on normalise l'image.

Figure 8. Image de Luminance normalisée « lumaNormalize »

- **Calcul du Gradient de l'image :**

```
function inputMap = img_gradient( Input ),

% using normalized images between [0-1]


mh = [ -1 -2 -1 ;
        0  0  0 ;
        1  2  1 ];

mv = [ -1  0  1 ;
      -2  0  2 ;
      -1  0  1 ];
```

```
GradientY = filter2( mh , Input);
GradientX = filter2( mv , Input );
inputMap = sqrt ( GradientX.^2 + GradientY.^2 );
```

La figure suivant est une représentation du gradient d'une image :

Figure 9. Image du Gradient « inputMap »

- **Matrice « Somme cumulée des variations minimum de Luminance » de l'image :**

```
function WeightedMap = processCumulativeWeight( inputMap ),
      inputSize = size( inputMap );
      WeightedMap = zeros( inputSize );
      WeightedMap( 1, : ) = inputMap( 1, : );
      for i = 2:inputSize(1),
            position = 1;
            WeightedMap( i, position ) = inputMap( i, position ) +...
            min2( WeightedMap( i - 1, position ),...
```

```
                WeightedMap( i - 1, position + 1 ) );
                position = inputSize(2);
                WeightedMap( i, position ) = inputMap(i, position) +...
                min2( WeightedMap( i - 1, position - 1 ),...
                WeightedMap( i - 1, position ) );
                for j = 2:inputSize(2) -1,
                      WeightedMap(i,j) = inputMap(i,j) + …
                      min3( WeightedMap( i - 1, j - 1 ),...
                      WeightedMap( i - 1, j ),...
                      WeightedMap( i - 1, j + 1 ) );
                end
          end
    end
```

Cette matrice représente la répartition de l'énergie minimum cumulée ligne par ligne. Elle est réalisée pixel-après-pixel par la sommation de la valeur du pixel courant avec celle du pixel voisin supérieur ayant la plus petite valeur. Elle permet une première sélection des *seams* dans l'image.

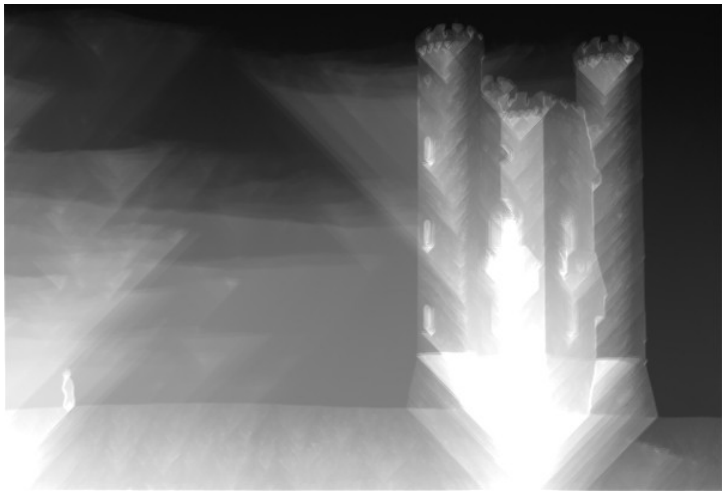


Figure 10. Matrice « Somme cumulée des variations minimum de Luminance » « WeightedMap »

- **Trouver les pixels comportant le moins d'énergie, *seam* :**

```
function Seam = find_seam( WeightedMap ),
      inputSize = size( WeightedMap );
      X = WeightedMap( inputSize(1), :);
      [Y,I] = min( X );
      Seam( 1:inputSize(1) ) = 0;
      for i = inputSize(1):-1:2,
            if(i == inputSize(1)),
                  Seam(i) = I;
            end
            X = WeightedMap(i-1,:);
            if(Seam(i) == 1),
                  Seam(i-1) = Seam(i) + ...
                              seam_min_left(X(Seam(i)),X(Seam(i)+1));
            elseif(Seam(i) == inputSize(2)),
                  Seam(i-1) = Seam(i) + ...
                              seam_min_right(X(Seam(i)-1),X(Seam(i)));
            else
                  Seam(i-1) =Seam(i) + ...
                             seam_min(X(Seam(i)-1), ...
                                         X(Seam(i)),X(Seam(i)+1));
            end
      end
end
```

A partir de la matrice précédente, on trouve la position correspondant à la plus faible valeur de la dernière ligne et on trace le *seam* en remontant ligne par ligne (*bottom-up*). Le *seam* comprend ainsi au chemin des pixels les moins énergétiques.

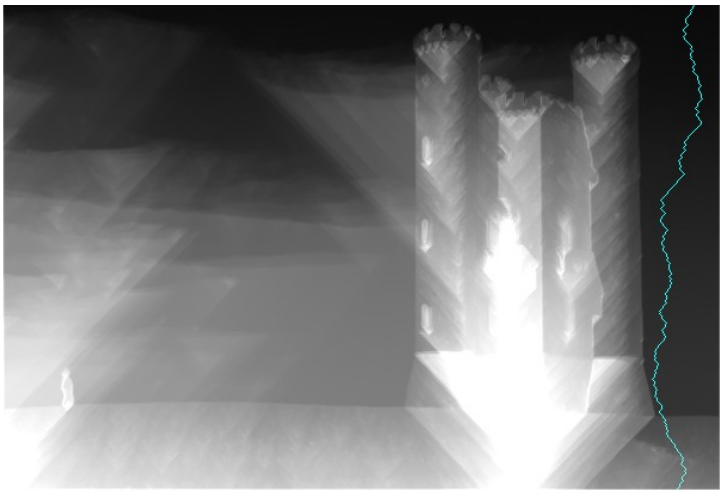


Figure 11. Seam

- **Suppression du Seam :**

Cette dernière étape correspond à la suppression des pixels aux positions contenues dans le vecteur *seam*, c'est-à-dire la position x dans chaque lignes de l'image.

Exemple : Réduction de la largeur de l'image de 300 pixels.

```
I = imread('castle.jpg');
[Output, X] = seam_carving(I,300);
```

Figure 12. Image redimensionnée avec seam_carving

Le Plug-in

Le but de ce projet est l'intégration de l'algorithme précédent dans un *plug-in* de la librairie TuttleOFX, basé sur le standard OpenFX.

TuttleOFX

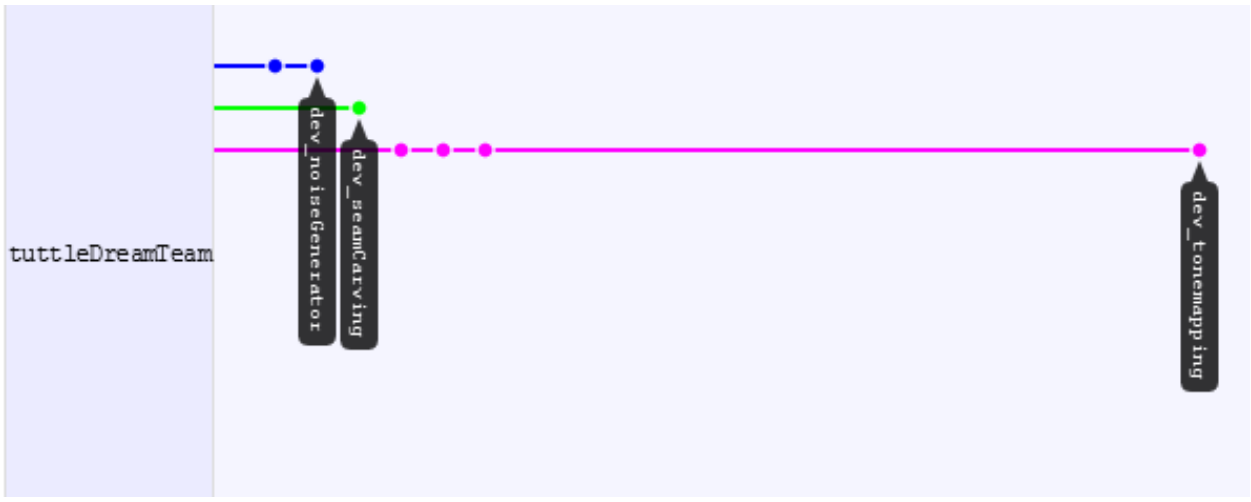
Avant de se plonger dans la programmation du *plug-in* proprement dit, nous avons dédié une partie importante du temps consacré au projet à connaître certains logiciels et librairies sur lesquelles le développement est basé.

Pour commencer, nous avons installé les applications suivantes : Python, Swig, git, gcc/g++, FFMpeg, ImageMagick. La liste de librairies nécessaires est dans le site web de TuttleOFX : https://sites.google.com/site/tuttleofx/development/build/libraries

Comme TuttleOFX est un projet *Open-Source*, il est déposé sur le gestionnaire de projet GitHub.

Git

Nous avons commencé par l'apprentissage de **Git**, lequel est un logiciel libre de gestion décentralisée de versions, distribué selon les termes de la licence publique générale GNU version 2. C'est un outil qui se veut simple et très performant, dont la principale tâche est de gérer l'évolution du contenu d'une arborescence. Une des particularités de Git, c'est l'existence de sites web collaboratifs comme GitHub et Gitorious. Nous particulièrement, on a travaillé avec GitHub. Il est comme une sorte de réseau social pour développeurs, qui ont la possibilité de regarder tous les projets évoluer et décider de participer à l'un d'entre eux si cela les intéresse. Git sait travailler par branches (versions parallèles d'un même projet) de façon très flexible. Chaque *plug-in* du projet TuttleOFX est une branche de la version « Master ». Pour ce qui est de notre projet, nous avons travaillé dans la branche « dev_seamCarving », une branche fille de « tuttleDreamTeam ».



ttps://github.com/tuttleDreamTeam

Git nous a permis de récupérer l'ensemble du projet TuttleOFX. Pour avoir une copie locale du projet, il faut cloner le répertoire TuttleOFX, ce qui se fait avec la commande suivante :

```
git://github.com/tuttleofx/TuttleOFX.git
```

Cette commande crée une copie du projet dans le répertoire courant. On peut vérifier que le clonage est correcte en listant la l'arborescence :

```
omar@omar-alvarez:~/TuttleOFX$ ls
3rdParty      bjam.sh~        INSTALL.bjam   LICENSE.LGPL       SConscript
applications  COPYING         INSTALL.scons  LICENSE.TuttleOFX  SConstruct
AUTHORS       dist            Jamroot        nuke.sh            tools
bin           doc             libraries      plugins            user-config.jam
bjam.sh       finalize.sconf  LICENSE.GPL    README
```

Boost

Boost est un ensemble de bibliothèques C++ gratuites et portables dont certaines seront intégrées au prochain standard C++. On y retrouve donc naturellement les concepts de la bibliothèque standard actuelle, et en particulier ceux de la STL avec laquelle elle se mélange parfaitement.

Boost est très riche et fournit notamment des bibliothèques pour :

- les threads (Boost.Thread)

- les matrices uBLAS et les tableaux à dimensions multiples (Boost.MultiArray)
- les expressions régulières (Boost.Regex)
- la méta-programmation (Boost.Mpl)
- l'utilisation de foncteurs(Boost.lambda, Boost.bind)
- la date et l'heure (Boost.Date_Time)
- les fichiers et les répertoires (Boost Filesystem)
- gérer la mémoire avec des pointeurs intelligents (Smart Pointers)
- faire de la sérialisation en binaire / texte / XML, en particulier sur les conteneurs standards (Boost Serialization)
- manipuler des graphes mathématiques (Boost Graph)
- manipuler les chaînes de caractères (Boost String Algorithms)
- et bien d'autres...

La liste complète des bibliothèques classées par catégories est disponible ici : http://www.boost.org/libs/libraries.htm.

La plupart de ces bibliothèques tentent d'exploiter au maximum les possibilités du langage C++. En fait, Boost se veut un laboratoire d'essais destiné à expérimenter de nouvelles bibliothèques pour le C++. Elle regroupe donc aussi une communauté d'experts (dont plusieurs sont membres du comité ISO de normalisation du C++) qui mettent un point d'honneur à ce qu'un maximum de compilateurs et de systèmes soient supportés. Ils débattent aussi de l'acceptation de nouvelles bibliothèques et l'évolution de celles déjà existantes, préfigurant ainsi ce à quoi ressemblera certainement la prochaine bibliothèque standard du langage C++.

C'est donc là que réside le grand intérêt de Boost. Outre son excellence technique et sa licence très permissive (compatible avec la GPL) qui permet de l'utiliser gratuitement dans des projets commerciaux, Boost est aussi un choix très viable sur le long terme. En effet, on peut légitimement espérer qu'un nombre important de ses bibliothèques soient un jour standardisées, ce qui en fait un outil dans lequel on peut investir du temps (et donc de l'argent) sans craindre de tout perdre au bout de quelques années faute de support ou d'évolution.

Dans le projet TuttleOFX, nous utilisons *Boost.build* (*bjam*) pour gérer la compilation du projet. Fondamentalement, *bjam* cherche dans chaque machine le compilateur : gcc, mingw, msvc... et crée la bonne commande pour compiler notre application en fonction de ce compilateur.

Ci-dessous, un exemple du processus de compilation avec *bjam* :

```
omar@omar-alvarez:~/TuttleOFX$ ./bjam.sh
plugins/image/process/geometry/SeamCarving/ variant=release
address-model=32 python=2.7
```

Le fichier *bjam.sh* contient :

```
export BOOST_ROOT=`pwd`/3rdParty/boost
$BOOST_ROOT/bjam --user-config=user-config.jam --toolset=gcc
--disable-icu -j2 $*
```

On peut aussi modifier les différents paramètres de configuration du développeur. Particulièrement, les machine dans lesquelles nous avons développé notre programme ont les paramètres suivants :

```
using python
  : 2.7                      # version
  : /usr/bin/python2.7       # bin directory
  : /usr/include/python2.7   # includes
  : /usr/lib                 # libs
  :
        <os>LINUX
        <address-model>32
  ;
```

En résumé, *Boost.build* compile le projet en fonction des paramètres de la machine du développeur.

GIL

Le projet TuttleOFX se base également sur l'utilisation de la librairie générique d'images GIL, qui a pour caractéristiques :

- Faire une abstraction des algorithmes de traitement d'images. Cela permet réutiliser un morceau de code et le faire travailler pour différentes types d'images.
- La vitesse des différents algorithmes de traitement d'images est élevée.
- La librairie permet à n'importe quel paramètre de l'image la possibilité d'être modifié lors de l'exécution avec un coût mineur.
- La librairie GIL peut être mis à jour et devenir plus performante en termes tels que : nouveaux espaces de couleurs, itérateurs, vues des images, locateurs, et d'autres concepts GIL.
- Elle a été conçue comme un complément STL. Les algorithmes génériques STL peuvent être utilisés pour la manipulation de pixels, particulièrement l'optimisation. La librairie GIL fonctionne avec les données existantes (pixels) provenant d'une autre bibliothèque de traitement d'images.

GIL travaille avec images et vues. Une image GIL c'est un conteneur de pixels. Elle permet de savoir la quantité de pixels et connaître la taille des images. Néanmoins, une vue est un concept purement virtuel qui permet de placer une camera (virtuelle également) sur les images. Une vue peut être lue et écrite. Dans l'algorithme, on travaille avec les vues des images. Quand on déclare une vue, il faut définir la zone de travail que l'on veut traiter.

Figure 13. Différence entre image et vue en GIL

Exemple :

On crée la vue de l'image à traiter et son gradient (map) :

```
View src = subimage_view( this->_srcView, procWindowOutput.x1,
procWindowOutput.y1, procWindowSize.x, procWindowSize.y );

MapView map = subimage_view( _mapView, procWindowOutput.x1,
procWindowOutput.y1, procWindowSize.x, procWindowSize.y );
```

Pour le calcul de l'énergie *cumulSum*, on doit créer une image et une vue de type gray32f (32 bits).

```
gray32f_image_t cumulSum ( map.width(), map.height(), 0 );
gray32f_view_t viewCumulSum ( view(cumulSum) );
```

Dans le *plug-in* de *Seam Carving*, les vues ont la même taille que les images, ce qui signifie que l'on applique le traitement à toute l'image.

Une autre fonction utilisée de la librairie GIL est `get_color()`. Elle nous permet l'accès à la valeur des pixels. Les paramètres nécessaires sont l'image de laquelle on veut connaître la valeur avec les coordonnées du pixel et la composant désirée. Par exemple, si on veut savoir la valeur du pixel courant du *map* (gradient) de chaque pixel de l'image, on écrit :

```
for(int x = 1; x < map.width(); x++ )
{
      for(int y = 1; y < map.height(); y++ )
      {
            currentP = get_color( map(x,y), gray_color_t() );
      }
}
```

On remarque que le choix du deuxième paramètre doit être logique par rapport type d'image. En effet, on ne peut pas obtenir une composante couleur (rouge, vert ou bleu) sur une image ne comprenant pas cette composante. Dans l'exemple précédent, comme `map` est une image qui contient un seul canal (*gray*), on passe comme paramètre `gray_color_t()` (et non pas `red_color_t()`, par exemple).

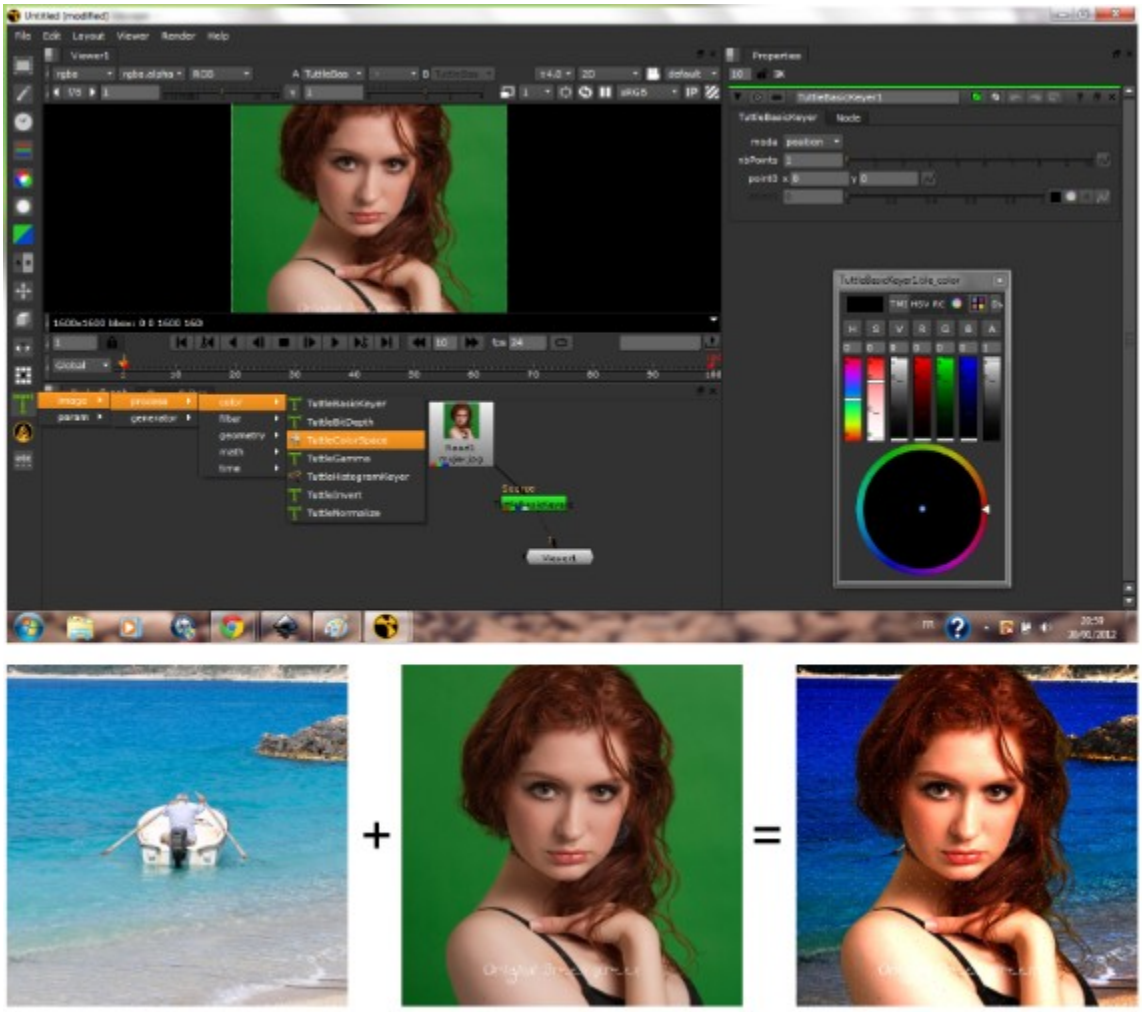
GIL permet aussi faire copies et conversions d'images. De la même manière que l'on peut définir l'entrée du *plug-in* (`src`), on peut déclarer l'image de sortie (`dst`). Après avoir fait tout le processus qui permet de recadrer l'image dans la vue `processingSrc`, si on veut visualiser la nouvelle image, il suffit de copier l'image de travail vers l'image de sortie. Pour cela, on utilise la fonction :

```
copy_and_convert_pixels( processingSrc, dst);
```

Encore une fois, il faut vérifier la cohérence des types des deux images entrées en paramètres de cette fonction.

NUKE

Nuke est un logiciel de *compositing* numérique nodal contenant un ensemble de méthodes numériques consistant à mélanger plusieurs sources d'images et rendre une seule image (ou séquence d'images, pour la vidéo).



http://www.thefoundry.co.uk/products/nuke/

Présentation de la structure du Plug-in

Pour débuter la mise en place du *plug-in* sur de bonnes bases, nous avons choisi de conserver la structure du code source du *plug-in Resize* déjà existant dans la librairie TuttleOFX. En effet, s'agissant de traitement géométrique de l'image, la structure fonctionnelle de ce programme s'approche assez de celui Seam Carving.

Le code source du *plug-in* se décompose en plusieurs fichiers :

- mainEntry.cpp
- SeamCarvingDefinitions.hpp
- SeamCarvingPluginFactory.hpp
- SeamCarvingPluginFactory.cpp
- SeamCarvingPlugin.hpp
- SeamCarvingPlugin.cpp
- SeamCarvingProcess.hpp
- SeamCarvingProcess.tcc

MainEntry

Contient les identifiants du *plug-in* et permet l'accès du logiciel hôte (ici, Nuke) d'accéder aux données du *plug-in*.

Definitions

Contient les définitions et déclarations des données visibles du *plug-in*. C'est ici que l'on définira les commandes (boutons, switch...) sur lesquels l'utilisateur peut intervenir pour modifier facilement les paramètres entrant dans l'algorithme du *plug-in*.

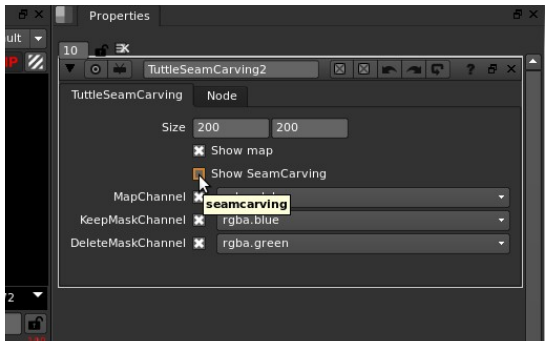


Figure 14. Panel de commande de Nuke pour le plugin de Seam Carving

PluginFactory

Ce fichier contient une description globale du programme. Il est composé de deux fonctions *describe* et *describeInContext*.

La fonction *describe* comporte les informations générales telles que les noms donnés au programme (*labels*), le groupe de *plug-in* auquel il appartient (*PluginGrouping*) et sa place dans l'arborescence des fichiers (par exemple, dans notre cas : *tuttle/image/process/geometry*), et éventuellement les informations d'aide à l'utilisateur.
Elle permet également de déclarer les données supportées par le programme (*image components, bit depth, tile rendering, contexts...*).

La fonction *describeInContext* permet quant à elle d'ajuster les composants du *plug-ins* selon son contexte d'utilisation, de déclarer les attributs du programmes (*clips* et paramètres) et de définir leurs propriétés.
Par exemple, on définira les composantes de l'image d'entrée supportées par le programme (RGB, RGBA, Alpha), ou les propriétés des paramètres d'entrées (définis dans *Definitions*) telles que les valeurs par défaut, le type de variables associées (entier, réel, booléen...) ou l'intitulé affiché sur la commande.

Plugin

Cette partie du programme sert à définir les fonctions du *plug-in* dans l'environnement nodal.

Dans un premier temps, on peut trouver le constructeur de la classe *SeamCarvingPlugin*. Celui-ci gère la définition des entrées du nœud (objet de la classe), notamment les images (*clip*)

autres que l'image à traiter et les variables associés aux paramètres du panel de commande du *plug-in*.

Dans notre cas, nous avons défini l'entrée de la matrice correspondant au gradient de l'image *« map »*. Nous avons également définit les variables d'entrée affectant les fonctionnalités du greffon et l'affichage en sortie du nœud (*size*, *showMap*, *showSeamCarving*).

Ensuite, la fonction d'acquisition des paramètres d'entrée est alors nécessaire. C'est le rôle de la fonction *getProcessParams* qui crée le liens entre les paramètres et les valeurs d'entrée, et l'utilisation de la fonction GIL *getValue()* permet de simplifier son implémentation.

Suivent ensuite une fonction d'acquisition des préférences (type d'image, composantes, résolution...), une fonction de paramétrage de la *Region of Definition*, c'est-à-dire la partie utile de l'image à traiter, et les fonctions relative au *rendering*, c'est-à-dire aux sorties du nœud.

Process

Ce fichier contient le cœur du programme, l'algorithme de la fonction du *plug-in*. Il s'agit d'un fichier .tcc (template C++) dont la fonction *multiThreadProcessImages* (méthode de la classe *SeamCarvingProcess*) contenant l'algorithme n'est créée que dans la fonction *render* de la classe *Plugin*. Elle n'existe que temporairement, le temps du calcul du traitement de l'image.

Process.tcc contient le constructeur de la classe *SeamCarvingProcess*, la fonction *setup* qui initialise les paramètres de la classe relatif au traitement, la fonction de traitement *multiThreadProcessImages* et l'ensemble des fonctions nécessaires à ce traitement, appelées par la fonction principale.

Développement du plug-in

Une fois l'environnement de travail maîtrisé et la structure du code mise en place, nous avons pu commencer l'incrémentation de l'algorithme de *Seam Carving* que nous avions mis au point préalablement sous MATLAB.

Nous avons débuté avec l'algorithme de *Seam Carving* vertical, et découpé les grandes étapes qui le compose en différentes fonctions :

- *ProcessVerticalCumulSum,*
- *ProcessFoundVerticalSeam*,
- *ProcessRemoveVerticalSeam*.

ProcessVerticalCumulSum

Cette fonction a pour but de calculer l'énergie minimum cumulée de l'image à traiter. Pour cela, elle prend en paramètre le gradient de l'image *map*. Cette variable pointe vers l'entrée *Map,* qui reçoit l'image calculé par des nœuds externes au *plug-in*, et correspond à un objet de la classe « templatée » MapView, capable de supporter différents types d'image (en terme de composante et de résolution).

La fonction reprend le principe expliqué en première partie de ce rapport pour calculer ligne-par-ligne, pixel-par-pixel, l'image de l'énergie minimum cumulée *viewCumulSum*. Dans cette opération, on diffère les pixels placés aux extrémités de l'image utile des autres pixels, puisque leur position ne les rend adjacents qu'avec deux pixels supérieurs, contrairement aux autres, qui eux en comptent trois.

ProcessVerticalCumulSum permet également de créer une matrice de direction *dirMap*, donnant pour chaque pixel de *viewCumulSum*, la position du pixel supérieur de moindre énergie. Ainsi, on conserve une trace des « chemins de moindre énergie » utilisé dans la suite du programme.

ProcessFoundVerticalSeam

Cette fonction a pour objectif de tracer le *seam*, modélisé par un vecteur de position dans l'image : l'indice des composantes indique la ligne du pixel, et leur valeur correspond à la colonne.

Une fois trouvé, dans la dernière ligne de l'image, le pixel correspondant à la valeur la plus faible de l'énergie minimum cumulée, le vecteur se remplit en remontant l'image ligne par ligne et en suivant les indications de directions de *dirMap* pour trouver la position des pixels de moindre énergie.

Sans cette matrice de direction, l'algorithme devrait comprendre une série de tests semblables au calcul de l'énergie cumulée. Par son utilisation, on évite la redondance, on simplifie l'algorithme et optimise le temps de calcul.

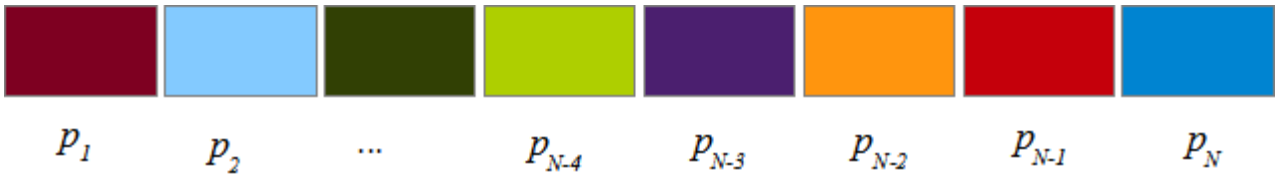
Au final, le vecteur comprend l'ensemble des positions des pixels du chemin de moindre énergie, c'est à dire les coordonnées du *seam.*

Il faut préciser que l'utilisation de la classe *vector* de la librairie standard du C++ pour le *seam* permet de simplifier certains aspects du code. Notamment, il ne nécessite pas qu'on lui assigne

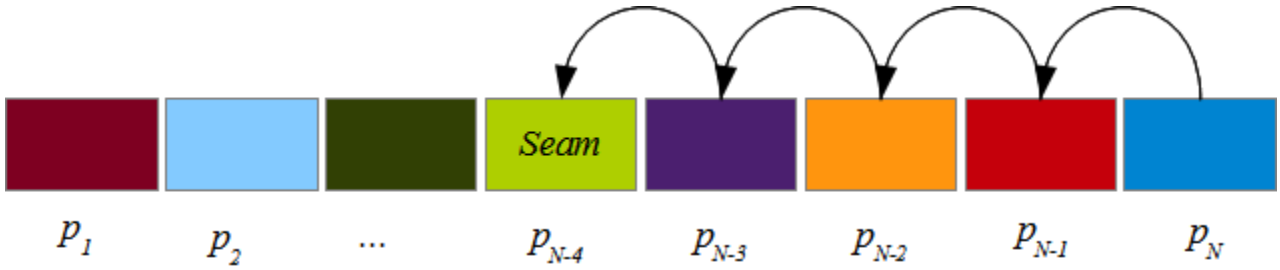
une taille initiale (à la déclaration) et permet donc de s'adapter à n'importe quelle taille d'image. De même, l'utilisation de certaines fonctions membres de la classe simplifie l'écriture.

ProcessRemoveVerticalSeam

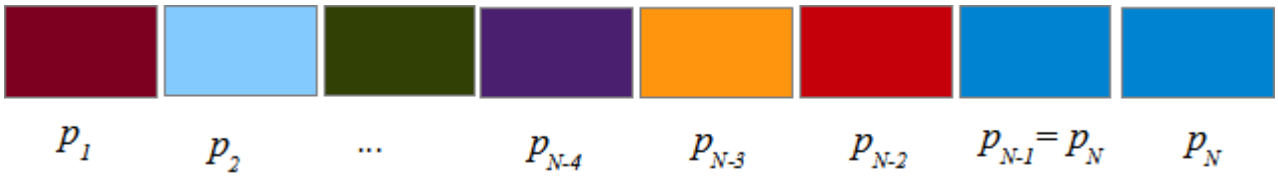
Cette dernière fonction permet la suppression du *seam* tracé précédemment. On crée pour cela une nouvelle image *processingSrc* qui est une copie de l'image source *src*, et on décale l'ensemble des pixels compris entre le *seam* et le bord droit de l'image d'un pixel vers la gauche.



Ligne de l'image initiale de largeur N pixels p_n



Décalage des pixels placés entre le seam *et le bord droit de l'image*



Ligne de l'image finale : le pixel appartenant au seam *est supprimé*

Après décalage des pixels, les pixels compris dans le *seam* n'apparaissent plus dans l'image. Ce processus peut-être répété autant de fois que nécessaire (dans la limite, bien entendu, de la largeur de l'image).

On remarque néanmoins qu'en fin de traitement, les derniers pixels de la ligne sont identiques. En effet, le *plug-in* ne permet pas la suppression des pixels de l'image, et nécessite l'ajout d'un traitement de *Cropping*, dans le cadre du fonctionnement nodal de Nuke.

Seam Carving horizontal

Le principe de fonctionnement du *Seam Carving* horizontal est le même que le processus vertical, mais les traitements sont réalisés de gauche à droite de l'image. Ainsi, l'algorithme est découpé en trois fonctions :

- *ProcessHorizontalCumulSum,*
- *ProcessFoundHorizontalSeam*,
- *ProcessRemoveHorizontalSeam*.

Ces trois fonctions appliquent de manière horizontal ce que font les fonctions de l'algorithme vertical.

Seam Carving bidirectionnel

Pour le cas du *Seam Carving* dans les deux directions, nous avons choisi d'alterner les traitements verticaux et horizontaux autant que possible. En effet, dans l'optique de rendre le résultat le plus propre possible visuellement, cette alternance semble plus logique et plus optimisée que d'effectuer les processus à la suite. Elle permet de minimiser l'impact d'un traitement directionnel sur l'autre, et donc réduire le risque d'apparition d'artefacts dans l'image finale et conserver une certaine cohérence. Néanmoins, une fois le nombre d'itération du processus d'une direction effectué, le traitement doit continuer dans l'autre direction uniquement.

Pour réaliser ceci, nous avons simplement effectué une série de test de comparaison des itérations des deux dimensions et envisagé tous les cas possibles :

- nombre d'itérations verticales supérieurs aux itérations horizontales,
- nombre d'itérations verticales inférieurs aux itérations horizontales,

- nombre d'itérations verticales égal aux itérations horizontales.

Gestion d'erreur

Lors de la mise au point de notre programme, nous avons souhaité tester ce qui se passerait si l'utilisateur entrait une taille de sortie de l'image supérieure à sa taille initiale. Résultat du test : Nuke plaint et se ferme directement. Nous avons donc voulu contrer ce défaut en mettant un simple test qui vérifie que les valeurs rentrées par l'utilisateur sont bien inférieures aux dimensions de l'image source. Si cette condition n'est respectée le processus du *Seam Carving* ne démarre pas, et l'utilisateur peut toujours voir à l'écran l'image source (ou l'image du gradient, selon les options cochées).

Utilisation du Plug-in

Enfin, il nous a fallu mettre au point un mode d'utilisation du programme qui soit assez simple et intuitif pour l'utilisateur (on est tout de même loin de ce qu'aurait pu apporter une étude d'ergonomie, mais ce n'était pas le but de ce projet). Nous avons choisi de proposer à l'utilisateur les commandes suivantes :

- Entrer la taille de l'image souhaitée en sortie du programme : les deux cases à droite de l'intitulé *Size* reçoivent respectivement la largeur et la longueur de l'image de sortie.
- Voir l'image du gradient de l'image source : cocher la case fait varier la valeur d'un paramètre booléen, et un test de vérité autorise ou non la copie de l'entrée *Map* dans l'image de destination affichée *dst.*
- Voir le résultat du traitement *Seam Carving* : cocher la case fait varier la valeur d'un autre booléen, dont le test de vérité autorise ou non le lancement du calcul du *Seam Carving* suivi son affichage.

Par défaut, la taille de sortie est initialisée à 600x400 pixels, les *switches* d'affichages décochés et l'image source est affichée.

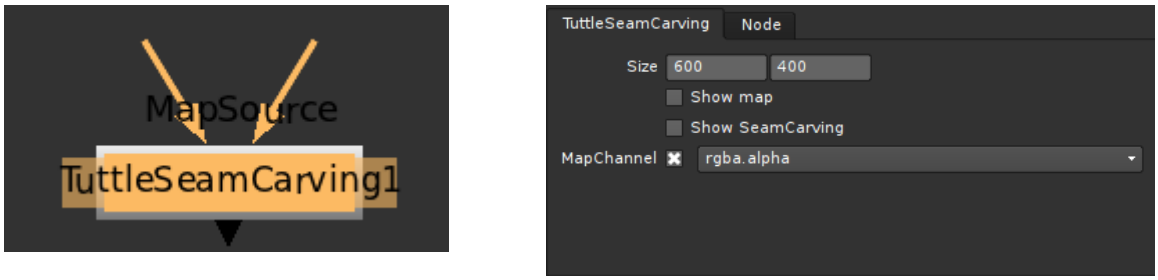


Figure 15. Nœud TuttleSeamCarving et paramètres par défaut

Tests et résultats

Dans cette partie, nous allons présenter quelques résultats des tests réalisés avec le *plug-in*.

Nous avons utilisé l'expérimentation pour trouver des solutions aux problèmes rencontrés pendant le développement des algorithmes, et pour penser aux futures lignes de travail. Le fait de tester le *plug-in* avec différentes types d'images nous a permis de mettre en évidence les faiblesses et points forts de notre implémentation.

Nous allons maintenant analyser un certain nombre d'images et leurs recadrages respectifs avec la technique du *Seam Carving* : à gauche, on visualise l'image originale, et à droite l'image redimensionnée.

Figure 16. Images de Test pour le plug-in SeamCarving

On peut observer dans certains cas, l'apparition d'artefacts et la déformation du contenu. Ceci prouve que, pour trouver les *seams*, le processus ne privilégie aucune zone et ne connaît pas l'importance des différentes parties de l'image.

Masques de protection et de suppression contrôlée

L'algorithme de *Seam Carving* implémenté est capable d'obtenir l'information de l'énergie d'une image en fonction d'un gradient. Si on manipule l'information du gradient, on est capables de « tromper » l'information d'énergie et donc de privilégier certaines zones pour que les *seams* ne les traversent jamais, ou au contraire, pour que les premières *seams* calculés suppriment en priorité certaines zones indésirables.

Un nœud du logiciel Nuke appelé *Bezier* nous permet de dessiner des courbes dans l'image. Avec cet outil, on peut colorer certaines zones du gradient et manipuler l'information qu'ils apportent avant de faire le calcul du minimum énergie cumulée. Si par exemple on applique un niveau élevé (alpha = 10) dans une certaine zone de l'image, on va la privilégier en augmentant son énergie et donc les *seams* ne seront pas présents dans cette zone. De la même manière, si le niveau appliqué est faible (alpha = -10), son énergie sera tellement faible que les seams calculés vont passer obligatoirement par là.

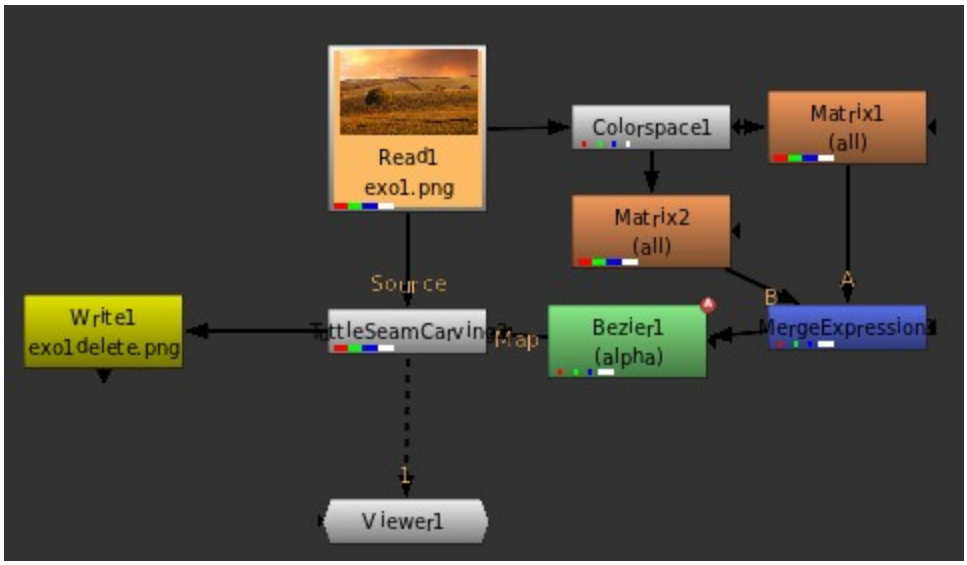


Figure 17. Nœuds de traitement nécessaires au bon fonctionnement du plug-in dans Nuke

Protection de zones importantes de l'image

Il s'agit de colorer la partie de l'image que l'on veut protéger avec un valeur de alpha élevé :

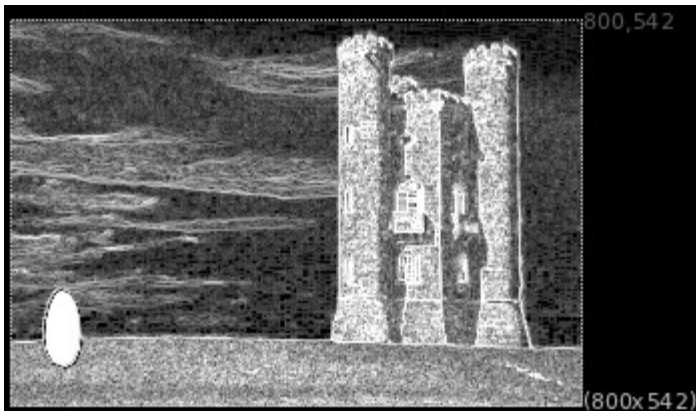
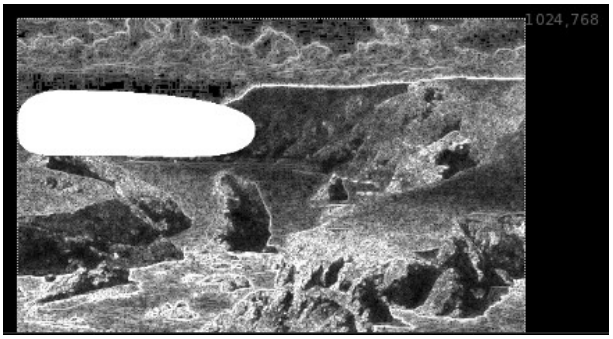


Figure 18. Masque de *protection avec courbe Bezier*

La zone colorée sera la plus importante et sera toujours conservée n'importe quel taille de sortie, c'est-à-dire, ses pixels seront les dernières censés d'être supprimés. L'image d'entrée a une taille de 800x542. Si on utilise le nœud SeamCarving pour réduire sa taille à 100x100 on peut observer que notre personnage, placé sous la zone colorée, reste intacte.

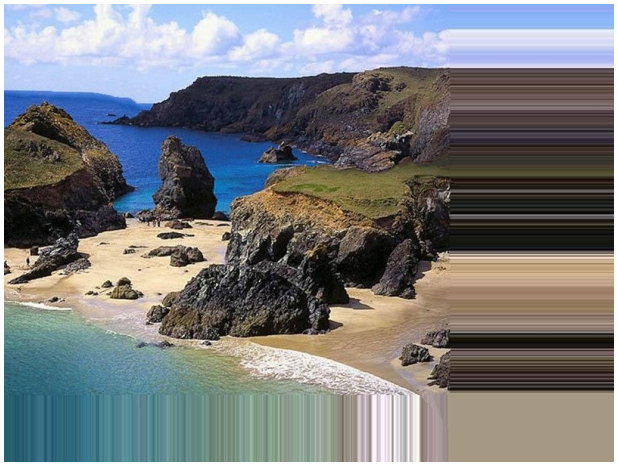
Figure 19. Zone de l'image protégée

Cette méthode est fondamentale pour préserver l'horizon dans les images. Si on colore les pixels voisins des lignes d'horizon, on peut éviter la distorsion.



Ci-dessous une comparaison entre le processus sans et avec masques.

Sans masque



Avec masque

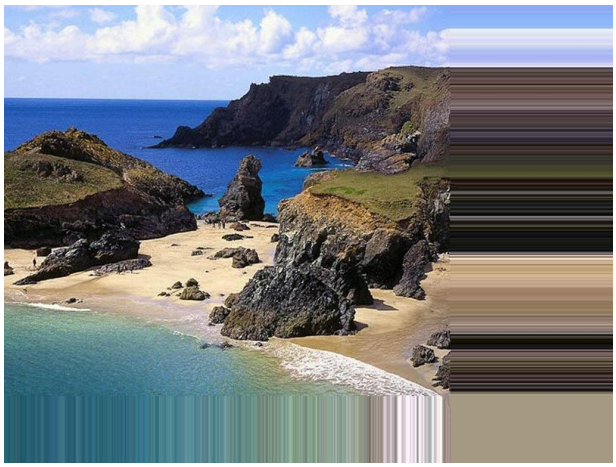


Figure 20. Utilisation des masques pour préserver l'horizon

Suppression contrôlée de zones l'image

Il s'agit de colorer la partie de l'image que l'on veut supprimer avec une valeur de alpha négative.

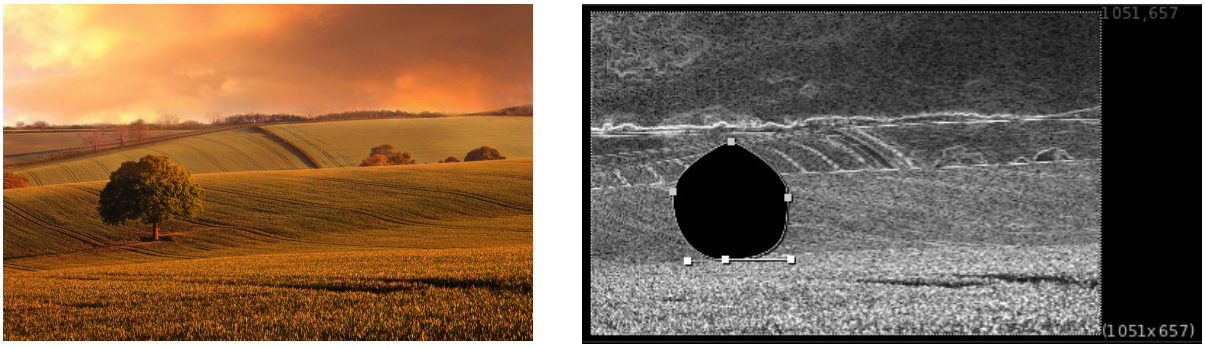


Figure 21. Masque de suppression

Dans ce cas, les *seams* calculés traverseront en priorité la zone colorée, car son énergie devient plus faible par rapport aux autres zones de l'image. On peut ainsi éliminer des objets d'une image à volonté.

Figure 22. Zone de l'image supprimée (arbre)

Dans cet exemple, on peut remarquer l'apparition des artefacts et un résultat avec une incohérence géométrique importante. La solution peut être combiner les deux méthodes précédentes afin de supprimer les zones indésirables mais en gardant la cohérence de l'horizon et d'autres zones d'importance.

Choix du calcul de l'énergie

L'opérateur que nous venons de présenter peut être optimisé et/ou adapté pour fonctionner avec d'autres techniques de calculs d'énergie d'images, comme des cartes de saillance ou d'autres algorithmes de détection de contours (Laplacien, *Laplacien of Gradient LoG*, Prewit, etc).

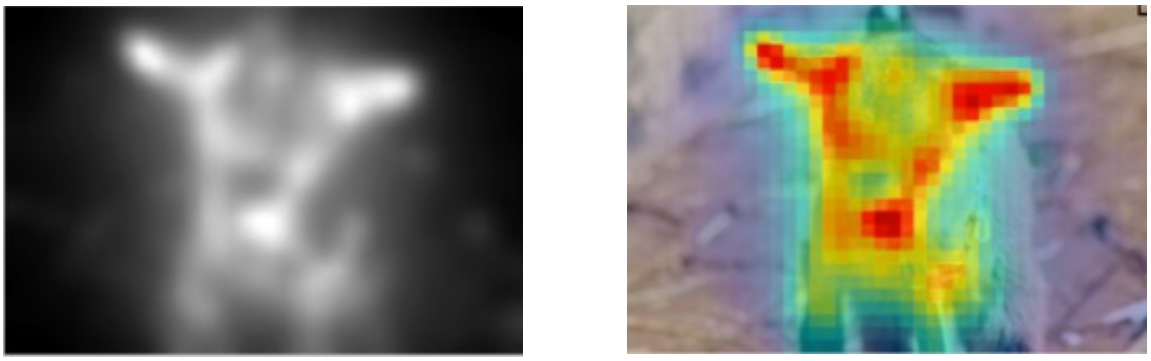


Figure 23. Exemples de carte de saillance

Le choix de ces différentes techniques peut permettre l'amélioration du traitement par le *plug-in*, car l'utilisation du gradient n'est pas toujours très adapté au contenu sémantique de l'image. Ceci peut également permettre de s'affranchir de certains artefacts du au traitement. Enfin, le fait de pouvoir adapter l'algorithme en fonction de l'image d'entrée est très intéressant.

Problèmes, limitations, améliorations et avenir du projet

Problèmes rencontrés et solutions

Durant la phase de développement, nous avons été confrontés à un certain nombre d'obstacles dans l'avancement de notre projet.

Tout d'abord, la compréhension de l'environnement TuttleOFX et la prise en main de la structure complexe du *plug-in* a pris un certain temps. En effet, n'ayant que des notions de base des processus de programmation, compilation, débogage, et du langage C++, et malgré notre maîtrise théorique du sujet, le début du projet se serait avéré très difficile sans l'aide de notre tuteur de projet (Marc-Antoine ARNAUD). Grâce à ses interventions à l'université et ses explications, nous avons pu commencer le projet de développement, et au fur-et-à-mesure de son avancée, nous avons élargi nos connaissances sur le fonctionnement du *plug-in* et sur le langage de programmation C++.

Ensuite, l'intégration de l'algorithme de Seam Carving dans le *plug-in*, qui traite d'images, n'a pas été aussi simple que dans MATLAB, qui lui traite de simples matrices. Ainsi, nous avons été confrontés à la gestion des *templates* d'images, et des librairies graphique GIL et Terry. Pour ce qui est des *templates*, les explications de notre tuteur ont suffis à nous faire comprendre leur fonctionnement. Néanmoins, pour ce qui est des librairies graphiques, nous avons eu des difficultés à trouver une documentation assez détaillées pour notre utilisation. A plusieurs reprises, il nous a fallu faire de nombreux tests, chercher des solutions dans d'autres plug-in de la suite TuttleOFX ou sur l'Internet pour comprendre le fonctionnement de certaines fonctions spécifiques.

Limitations, optimisations et améliorations du projet

Malgré l'opérabilité de notre programme, il demeure des limites d'utilisation et de fonctionnalité. Il convient de les citer afin de pouvoir cibler les éléments à améliorer dans le *plug-in*.

Tout d'abord, notre programme ne gère que le Seam Carving de réduction, pourtant les publications concernant cette technique de redimensionnement d'image prévoient l'agrandissement. Bien que le principe de base reste le même, le Seam Carving d'agrandissement implique de recréer des pixels de moindre énergie dans l'image. Ceci peut être effectué par simple copie, mais pour un résultat visuellement plus propre, il semble indispensable de réaliser une interpolation entre les pixels voisins. Le Seam Carving n'est plus alors un traitement uniquement géométrique (n'influant que sur la position des pixels) mais intervient sur l'information de l'image (valeurs des pixels). Cette troisième dimension demanderait donc d'effectuer un algorithme d'interpolation pour l'ensemble des types d'images supportées par le logiciel hôte (*8, 16, 32, 32f bits, RGB, RGBA, gray...*). Ceci reste faisable, mais augmenterait considérablement la complexité de l'implémentation.

L'amélioration du *plug-in* pourrait également passer par l'adaptation à la taille de sortie de l'image. En effet, Nuke étant un logiciel utilisant un système nodal, chaque nœud doit avoir une fonction spécifique et facilement paramétrable. Dans cet optique, notre programme, qui remplit bien le caractère de spécificité, demande l'ajout d'un nœud de *Cropping* afin d'ajuster la taille d'affichage et de rendu à la taille de l'image après traitement. Car en sortie du nœud que nous avons développé (et de la même manière que les *plug-ins* de *Resize* et *Crop* de la librairie TuttleOFX), on remarque la présence des lignes et colonnes copiées, résultantes du processus de redimensionnement. L'ajout d'une fonction d'ajustement de la taille serait donc un plus dans la fonctionnalité et la prise en main de notre programme.

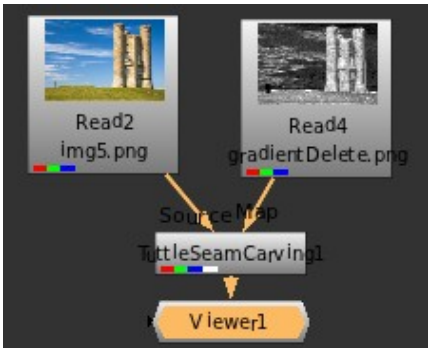


Figure 24. Résultat du plug-in avec et sans l'ajout d'un nœud Crop

Le temps de calcul est aussi un paramètre qui pourrait être amélioré. Ceci passerait par une importante étape d'optimisation du code et des algorithmes. Ce temps de calcul est un critère non négligeable de l'efficacité du programme, et sa diminution serait un avantage, notamment pour le traitement des images de grande taille (format cinéma), et pour ce qui concerne le développement du Seam Carving pour la vidéo (voir plus bas).

Enfin, la gestion des masques de suppression ou conservation pourrait également être améliorée. En effet, afin d'optimiser l'effet de ces masques, il serait judicieux de leur introduire un paramètre de directionnalité. Ainsi, un masque ne serait visible pour le programme que si ça direction est identique à celle du traitement : le masque vertical n'aurait d'effet que sur le processus de *Seam Carving* vertical, et le masque horizontal sur le traitement horizontal. Ceci permettrait surtout d'optimiser la recherche de *seam* et donc de minimiser les artefacts visibles.

Cette amélioration pourrait être suivit de l'intégration de la gestion des masques directement dans le *plug-in*, ce qui rendrait leur utilisation plus intuitive.

Seam Carving pour la vidéo...

Le *Seam Carving* étant opérationnel pour l'image fixe, la question d'appliquer cette technique à une séquence vidéo se pose assez naturellement dans le prolongement du projet, d'autant que Nuke est un logiciel de *compositing* utilisé pour la vidéo et le cinéma.

A première vue, le *Seam Carving* pour la vidéo pourrait n'être que le simple enchaînement du traitement à l'ensemble des images d'une séquence filmée. Il n'en est rien. En effet, la vidéo, apportant la troisième dimension qu'est le temps, implique que les images de la séquence sont généralement toutes différentes les unes des autres et donc que le traitement du *Seam Carving* diffèrent pour chacune d'entre elles.

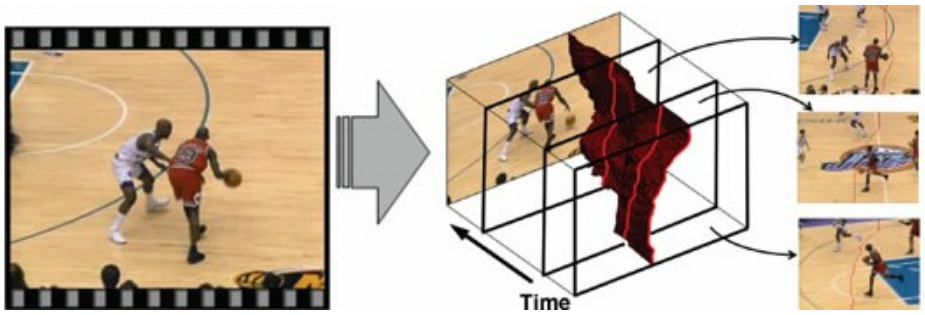


Figure 25. Idée de base pour le *SeamCarving pour la vidéo*

Parti de ce constat, on peut très aisément imaginer les conséquences qu'aurait le *Seam Carving* image-par-image sur la vidéo : les pixels supprimés (ou ajoutés) de l'image ne seraient pas les mêmes d'une image à l'autre, les parties non traitées changeraient de position à chaque image, et toute la cohérence de la scène serait perdue.

Ainsi, il convient de considérer le problème différemment, et le *Seam Carving* vidéo semble être un sujet suffisamment complexe pour être étudier dans un projet placé directement dans le prolongement de celui-ci.

Conclusion

Nous venons de présenter un programme basé sur un algorithme de redimensionnement d'images utilisant la technique du *Seam Carving*, qui peut être appliqué pour une grande variété de manipulations d'images: changement de formats, amplification des contenus, suppression d'objets.

Le *plug-in* est finalement totalement opérationnel et reproduit assez bien les fonctions de base relatives au *Seam Carving*. Sa nature nodale, qui implique un certain nombre de contraintes d'adaptabilité et de compatibilité, est tout à fait assumée, son utilisation est relativement simple, et la combinaison des processus vertical et horizontal est réalisée de manière quasi-optimale.

Néanmoins, ce programme nécessite un certain nombre d'améliorations, et ce à différents niveaux (syntaxe, implémentation, débogage, fonctionnalité, utilisation), pour le rendre à la fois fonctionnel, ergonomique et solide.

Ainsi, ce projet, bien que inachevé, fut très enrichissant autant en ce qui concerne le sujet que les techniques utilisées et le cadre de travail. Il nous a permit d'appréhender les techniques de traitement numérique des images, de découvrir l'intégration d'un *plug-in* et d'élargir nos connaissances en programmation informatique.

De plus, s'agissant d'un sujet relativement récent, les futures lignes de travail sont nombreuses dans le prolongement de ce projet, notamment celle d'envisager l'implémentation de cette technique pour le recadrage vidéo en temps réel, qui est un sujet de recherche d'actualité.

Bibliographie

1. AVIDAN S. and SHAMIR A., “Seam Carving for Content-Aware Image Resizing”, ACM Transactions Graphics, vol. 26, no.10, (2007).

2. M. RUBINSTEIN, D. GUTIERREZ, O. SORKINE and A. SHAMIR, “A Comparative Study of Image Retargeting”, ACM Transactions on Graphics, Volume 29, Number 5, Proceedings Siggraph Asia 2010.

3. Seam Carving, From Wikipedia, the free encyclopedia :
   http://en.wikipedia.org/wiki/Seam_carving

4. CAIR - Content Aware Image Resizer :
   https://sites.google.com/site/brainrecall/cair

5. Philippe THOMIN, “*C dans la poche*”, V 5.0

6. Alain GIBAUD, “*C++ compact*”, version 0.6

7. Saliency Map Algorithm : MATLAB Source Code :
   http://www.klab.caltech.edu/~harel/share/gbvs.php

ANNEXES

MainEntry.cpp

```
#define OFXPLUGIN_VERSION_MAJOR 0
#define OFXPLUGIN_VERSION_MINOR 0

#include "SeamCarvingPluginFactory.hpp"
#include <tuttle/plugin/Plugin.hpp>

namespace OFX {
namespace Plugin {

void getPluginIDs( OFX::PluginFactoryArray& ids )
{
    mAppendPluginFactory( ids, tuttle::plugin::seamcarving::SeamCarvingPluginFactory,
"tuttle.seamcarving" );
}

}
}
```

SeamCarvingDefinitions.hpp

```
#ifndef _TUTTLE_PLUGIN_SEAMCARVING_DEFINITIONS_HPP_
#define _TUTTLE_PLUGIN_SEAMCARVING_DEFINITIONS_HPP_

#include <tuttle/plugin/global.hpp>
#include <tuttle/plugin/context/ResolutionDefinition.hpp>
#include <tuttle/plugin/context/SamplerDefinition.hpp>


namespace tuttle {
namespace plugin {
namespace seamcarving {

static const std::string kClipMap        = "Map";

static const std::string kParamSize = "size";
static const std::string kParamSizeLabel = "Size";

static const std::string kParamMap = "map";
static const std::string kParamMapLabel = "Show map";

static const std::string kParamSeamCarving = "seamcarving";
static const std::string kParamSeamCarvingLabel = "Show SeamCarving";

}
}
}

#endif
```

SeamCarvingPluginFactory.hpp

```
#ifndef _TUTTLE_PLUGIN_SEAMCARVINGPLUGINFACTORY_HPP_
#define _TUTTLE_PLUGIN_SEAMCARVINGPLUGINFACTORY_HPP_

#include <ofxsImageEffect.h>

namespace tuttle {
namespace plugin {
namespace seamcarving {

mDeclarePluginFactory( SeamCarvingPluginFactory, {}, {} );

}
}
}

#endif
```

SeamCarvingPluginFactory.cpp

```
#include "SeamCarvingPluginFactory.hpp"
#include "SeamCarvingPlugin.hpp"
#include "SeamCarvingDefinitions.hpp"

#include <tuttle/plugin/context/SamplerPluginFactory.hpp>

#include <limits>


namespace tuttle {
namespace plugin {
namespace seamcarving {

static const bool kSupportTiles = false;

/**
 * @brief Function called to describe the plugin main features.
 * @param[in, out] desc Effect descriptor
 */
void SeamCarvingPluginFactory::describe( OFX::ImageEffectDescriptor& desc )
{
    desc.setLabels( "TuttleSeamCarving", "SeamCarving", "Image Retargeting" );
    desc.setPluginGrouping( "tuttle/image/process/geometry" );

    // add the supported contexts, only filter at the moment
    desc.addSupportedContext( OFX::eContextFilter );
    desc.addSupportedContext( OFX::eContextGeneral );

    // add supported pixel depths
    desc.addSupportedBitDepth( OFX::eBitDepthUByte );
    desc.addSupportedBitDepth( OFX::eBitDepthUShort );
    desc.addSupportedBitDepth( OFX::eBitDepthFloat );

    // plugin flags
```

```
        desc.setSupportsMultipleClipDepths( true );
    desc.setSupportsTiles( kSupportTiles );
    desc.setRenderThreadSafety( OFX::eRenderFullySafe );
}

/**
 * @brief Function called to describe the plugin controls and features.
 * @param[in, out]   desc       Effect descriptor
 * @param[in]        context    Application context
 */
void SeamCarvingPluginFactory::describeInContext( OFX::ImageEffectDescriptor& desc,
OFX::EContext context )
{
        OFX::ClipDescriptor* srcClip =
desc.defineClip( kOfxImageEffectSimpleSourceClipName );
    srcClip->addSupportedComponent( OFX::ePixelComponentRGBA );
    srcClip->addSupportedComponent( OFX::ePixelComponentRGB );
    srcClip->addSupportedComponent( OFX::ePixelComponentAlpha );
    srcClip->setSupportsTiles( kSupportTiles );

    // Create the mandated output clip
    OFX::ClipDescriptor* dstClip = desc.defineClip( kOfxImageEffectOutputClipName );
    dstClip->addSupportedComponent( OFX::ePixelComponentRGBA );
    dstClip->addSupportedComponent( OFX::ePixelComponentRGB );
    dstClip->addSupportedComponent( OFX::ePixelComponentAlpha );
    dstClip->setSupportsTiles( kSupportTiles );

    OFX::ClipDescriptor* mapClip = desc.defineClip( kClipMap );
        mapClip->addSupportedComponent( OFX::ePixelComponentAlpha );
    mapClip->setSupportsTiles( kSupportTiles );

    OFX::Int2DParamDescriptor* size = desc.defineInt2DParam( kParamSize );
    size->setLabel( kParamSizeLabel );
    size->setDefault( 600, 400 );
    size->setRange( 1, 1, std::numeric_limits<int>::max(),
std::numeric_limits<int>::max() );

    OFX::BooleanParamDescriptor* showMap = desc.defineBooleanParam( kParamMap );
    showMap->setLabel( kParamMapLabel );
    showMap->setDefault( false );

    //switch d'affichage de la source :
    OFX::BooleanParamDescriptor* showSeamCarving =
desc.defineBooleanParam( kParamSeamCarving );
    showSeamCarving->setLabel( kParamSeamCarvingLabel );
    showSeamCarving->setDefault( false );


}

/**
 * @brief Function called to create a plugin effect instance
 * @param[in] handle  Effect handle
 * @param[in] context Application context
 * @return  plugin instance
 */
OFX::ImageEffect* SeamCarvingPluginFactory::createInstance( OfxImageEffectHandle
handle,
                                OFX::EContext context )
```

```
{
    return new SeamCarvingPlugin( handle );
}

}
}
}
```

SeamCarvingPlugin.hpp

```
#ifndef _TUTTLE_PLUGIN_SEAMCARVING_PLUGIN_HPP_
#define _TUTTLE_PLUGIN_SEAMCARVING_PLUGIN_HPP_

#include "SeamCarvingDefinitions.hpp"

#include <tuttle/plugin/ImageEffectGilPlugin.hpp>
#include <tuttle/plugin/context/SamplerPlugin.hpp>

namespace tuttle {
namespace plugin {
namespace seamcarving {

template<typename Scalar>
struct SeamCarvingProcessParams
{
    OFX::Clip* _clipMap;

    boost::gil::point2<Scalar> _outputSize;
    bool                       _showMap;
    bool                       _showSeamCarving;
};

/**
 * @brief SeamCarving plugin
 */
class SeamCarvingPlugin : public ImageEffectGilPlugin
{
public:
    typedef float Scalar;
    typedef boost::gil::point2<double> Point2;

public:
    SeamCarvingPlugin( OfxImageEffectHandle handle );

public:
    SeamCarvingProcessParams<Scalar> getProcessParams( const OfxPointD& renderScale =
OFX::kNoRenderScale ) const;


    void changedParam          ( const OFX::InstanceChangedArgs &args, const
std::string &paramName );

        void getClipPreferences    ( OFX::ClipPreferencesSetter& clipPreferences );
    bool getRegionOfDefinition ( const OFX::RegionOfDefinitionArguments& args,
OfxRectD& rod );
    void getRegionsOfInterest  ( const OFX::RegionsOfInterestArguments& args,
OFX::RegionOfInterestSetter& rois );
```

```
    bool isIdentity            ( const OFX::RenderArguments& args, OFX::Clip*&
identityClip, double& identityTime );

    void render                ( const OFX::RenderArguments &args );

private:

        template< template<class, class> class Process, class MapLayout, class
BitDepth, class DstLayout, class Plugin >
        void renderMapComponentBitDepthDstComponent( Plugin& plugin, const
OFX::RenderArguments& args);

        template< template<class, class> class Process, class MapLayout, class
BitDepth, class Plugin >
        void renderMapComponentBitDepth( Plugin& plugin, const OFX::RenderArguments&
args, OFX::EPixelComponent dstComponents );
public:
    OFX::Clip* _mapClip;

    OFX::Int2DParam*    _paramSize;
    OFX::BooleanParam*  _paramShowMap;
    OFX::BooleanParam*  _paramShowSeamCarving;
};

}
}
}

#endif
```

SeamCarvingPlugin.cpp

```
#include "SeamCarvingPlugin.hpp"
#include "SeamCarvingProcess.hpp"
#include "SeamCarvingDefinitions.hpp"

#include <tuttle/plugin/ofxToGil/point.hpp>

#include <terry/sampler/sampler.hpp>
#include <terry/point/operations.hpp>

#include <boost/gil/gil_all.hpp>

#include <boost/numeric/conversion/cast.hpp>

namespace tuttle {
namespace plugin {
namespace seamcarving {

using namespace ::terry::sampler;
using boost::numeric_cast;

SeamCarvingPlugin::SeamCarvingPlugin( OfxImageEffectHandle handle )
    : ImageEffectGilPlugin( handle )
{
    _mapClip        = fetchClip ( kClipMap );
    _paramSize      = fetchInt2DParam ( kParamSize );
```

```
    _paramShowMap   = fetchBooleanParam ( kParamMap );
    _paramShowSeamCarving= fetchBooleanParam ( kParamSeamCarving );
}


SeamCarvingProcessParams<SeamCarvingPlugin::Scalar>
SeamCarvingPlugin::getProcessParams( const OfxPointD& renderScale ) const
{
    SeamCarvingProcessParams<Scalar> params;

    OfxPointI size = _paramSize->getValue();
    params._outputSize.x = size.x;
    params._outputSize.y = size.y;

    params._showMap = _paramShowMap->getValue();
    params._showSeamCarving = _paramShowSeamCarving->getValue();

    return params;
}

void SeamCarvingPlugin::getClipPreferences( OFX::ClipPreferencesSetter&
clipPreferences )
{
        clipPreferences.setClipBitDepth( *this->_clipDst, OFX::eBitDepthFloat );
        clipPreferences.setClipComponents( *this->_clipDst, OFX::ePixelComponentRGBA
);
        clipPreferences.setPixelAspectRatio( *this->_clipDst, 1.0 ); /// @todo tuttle:
retrieve info from exr
}

void SeamCarvingPlugin::changedParam( const OFX::InstanceChangedArgs &args, const
std::string &paramName )
{

}

bool SeamCarvingPlugin::getRegionOfDefinition( const OFX::RegionOfDefinitionArguments&
args, OfxRectD& rod )
{
    using namespace boost::gil;

    const OfxRectD srcRod = _clipSrc->getCanonicalRod( args.time );
    const Point2   srcRodSize( srcRod.x2 - srcRod.x1, srcRod.y2 - srcRod.y1 );

    const OfxPointI s = _paramSize->getValue();
    rod.x1 = 0;
    rod.y1 = 0;
    rod.x2 = s.x;
    rod.y2 = s.y;

    rod.x2 = srcRodSize.x;
    rod.y2 = srcRodSize.y;

    return true;

}

void SeamCarvingPlugin::getRegionsOfInterest( const OFX::RegionsOfInterestArguments&
args, OFX::RegionOfInterestSetter& rois )
```

```
{

}

bool SeamCarvingPlugin::isIdentity( const OFX::RenderArguments& args, OFX::Clip*&
identityClip, double& identityTime )
{
//    SeamCarvingProcessParams<Scalar> params = getProcessParams();
//    if( params._in == params._out )
//    {
//        identityClip = _clipSrc;
//        identityTime = args.time;
//        return true;
//    }
    return false;
}



template< template<class, class> class Process, class MapLayout, class BitDepth, class
DstLayout, class Plugin >
void SeamCarvingPlugin::renderMapComponentBitDepthDstComponent( Plugin& plugin, const
OFX::RenderArguments& args )
{
    typedef boost::gil::pixel<BitDepth, MapLayout> MapPixel;
    typedef boost::gil::image<MapPixel, false> MapImage;
    typedef typename MapImage::view_t MapView;

    typedef boost::gil::pixel<BitDepth, DstLayout> DPixel;
    typedef boost::gil::image<DPixel, false> DImage;
    typedef typename DImage::view_t DView;

    Process<MapView, DView> procObj( plugin );

    procObj.setupAndProcess( args );
}


template< template<class, class> class Process, class MapLayout, class BitDepth, class
Plugin >
void SeamCarvingPlugin::renderMapComponentBitDepth( Plugin& plugin, const
OFX::RenderArguments& args, OFX::EPixelComponent dstComponents )
{
    switch( dstComponents )
    {
            case OFX::ePixelComponentAlpha:
            {
                    renderMapComponentBitDepthDstComponent<Process, MapLayout,
BitDepth, boost::gil::gray_layout_t>( plugin, args );
                    return;
            }
            case OFX::ePixelComponentRGB:
            {
                    renderMapComponentBitDepthDstComponent<Process, MapLayout,
BitDepth, boost::gil::rgb_layout_t>( plugin, args );
                    return;
            }
            case OFX::ePixelComponentRGBA:
            {
```

```
                    renderMapComponentBitDepthDstComponent<Process, MapLayout,
BitDepth, boost::gil::rgba_layout_t>( plugin, args );
                    return;
            }
            case OFX::ePixelComponentCustom:
            case OFX::ePixelComponentNone:
            {
                    BOOST_THROW_EXCEPTION( exception::Unsupported()
                            << exception::user() + "Pixel components (" +
mapPixelComponentEnumToString(dstComponents) + ") not supported by the plugin." );
            }
    }
}
```

```
/**
 * @brief The overridden render function
 * @param[in]   args     Rendering parameters
 */
void SeamCarvingPlugin::render( const OFX::RenderArguments &args )
{
    OFX::EBitDepth       bitDepth          = _clipDst->getPixelDepth();

    OFX::EPixelComponent mapComponents        = _mapClip->getPixelComponents();

    OFX::EPixelComponent dstComponents        = _clipDst->getPixelComponents();

    switch( mapComponents )
    {
            case OFX::ePixelComponentAlpha:
            {
                switch( bitDepth )
                {
                    case OFX::eBitDepthUByte:
                    {
                        renderMapComponentBitDepth<SeamCarvingProcess,
boost::gil::gray_layout_t, boost::gil::bits8>( *this, args, dstComponents );
                        return;
                    }
                    case OFX::eBitDepthUShort:
                    {
                        renderMapComponentBitDepth<SeamCarvingProcess,
boost::gil::gray_layout_t, boost::gil::bits16>( *this, args, dstComponents );
                        return;
                    }
                    case OFX::eBitDepthFloat:
                    {
                        renderMapComponentBitDepth<SeamCarvingProcess,
boost::gil::gray_layout_t, boost::gil::bits32f>( *this, args, dstComponents );
                        return;
                    }
                    case OFX::eBitDepthCustom:
                    case OFX::eBitDepthNone:
```

```
                    {
                        BOOST_THROW_EXCEPTION( exception::Unsupported()
                                               << exception::user() + "Bit depth (" +
mapBitDepthEnumToString(bitDepth) + ") not recognized by the plugin." );
                    }
                }
                return;
            }
            case OFX::ePixelComponentRGBA:
            case OFX::ePixelComponentRGB:
            case OFX::ePixelComponentCustom:
            case OFX::ePixelComponentNone:
            {
                    BOOST_THROW_EXCEPTION( exception::Unsupported()
                            << exception::user() + "Pixel components (" +
mapPixelComponentEnumToString(mapComponents) + ") not supported by the plugin." );
            }
    }
    BOOST_THROW_EXCEPTION( exception::Unknown() );
}
}
}
}
```

SeamCarvingProcess.hpp

```
#ifndef _TUTTLE_PLUGIN_SEAMCARVING_PROCESS_HPP_
#define _TUTTLE_PLUGIN_SEAMCARVING_PROCESS_HPP_

#include <tuttle/plugin/ImageGilFilterProcessor.hpp>

namespace tuttle {
namespace plugin {
namespace seamcarving {

/**
 * @brief SeamCarving process
 *
 */
template<class MapView, class View>
class SeamCarvingProcess : public ImageGilFilterProcessor<View>
{
public:
    typedef typename View::value_type Pixel;
    typedef typename boost::gil::channel_type<View>::type Channel;
    typedef float Scalar;

        /*View _srcView; ///< Source view
        OFX::Image* _srcImg;
        OfxRectI _srcPixelRod;*/

        MapView _mapView; ///< Map view
        OFX::Image* _mapImg;
        OfxRectI _mapPixelRod;

protected:
    SeamCarvingPlugin&            _plugin;    ///< Rendering plugin
```

```
    SeamCarvingProcessParams<Scalar>    _params;    ///< parameters

public:
    SeamCarvingProcess( SeamCarvingPlugin& effect );

    void setup( const OFX::RenderArguments& args );

    void multiThreadProcessImages( const OfxRectI& procWindowRoW );
};

}
}
}

#include "SeamCarvingProcess.tcc"

#endif
```

SeamCarvingProcess.tcc

```
#include <tuttle/plugin/ofxToGil/rect.hpp>
#include <terry/geometry/affine.hpp>
#include<time.h>

namespace tuttle {
namespace plugin {
namespace seamcarving {

template<class MapView, class View>
SeamCarvingProcess<MapView, View>::SeamCarvingProcess( SeamCarvingPlugin &effect )
        : ImageGilFilterProcessor<View>( effect , eImageOrientationFromTopToBottom )
    , _plugin( effect )
{
    this->setNoMultiThreading();
}

template<class MapView, class View>
void SeamCarvingProcess<MapView, View>::setup( const OFX::RenderArguments& args )
{
        ImageGilFilterProcessor<View>::setup( args );
        _params = _plugin.getProcessParams( args.renderScale );

    // clip Map
        _mapImg = _plugin._mapClip->fetchImage( args.time );
        if( _mapImg == NULL )
        BOOST_THROW_EXCEPTION( exception::ImageNotReady() );
        if( _mapImg->getRowBytes() == 0 )
        BOOST_THROW_EXCEPTION( exception::WrongRowBytes() );

        _mapPixelRod = _plugin._mapClip->getPixelRod( args.time, args.renderScale );
        _mapView = tuttle::plugin::getGilView<MapView>( _mapImg, _mapPixelRod,
eImageOrientationFromTopToBottom );
}

template<class MapView>
void ProcessVerticalCumulSum( MapView& map, boost::gil::gray32f_view_t& viewDirMap,
boost::gil::gray32f_view_t& viewCumulSum, int widthProcess)
{
    using namespace terry;
    float currentP;
    float topLeft;
    float top;
    float topRight;
    float minValue;
    int dir;

    for(int x = 0; x < widthProcess; x++ )
    {
        viewCumulSum(x,0) = map(x,0);
    }
    for(int y = 1; y < map.height(); y++ )
    {
        //First Column
        currentP = get_color( map(0,y), gray_color_t() );
        top      = get_color( viewCumulSum(0  ,y-1), gray_color_t() );
```

```
            topRight = get_color( viewCumulSum(1,y-1), gray_color_t() );

            if( topRight < top )
            {
                minValue = topRight;
                dir = 1;
            }
            else
            {
                minValue = top;
                dir = 0;
            }
            viewCumulSum(0,y) = currentP + minValue;
            viewDirMap(0,y) = dir;

            //Last Column
            currentP = get_color( map(widthProcess-1,y), gray_color_t() );
            top      = get_color( viewCumulSum(widthProcess-1  ,y-1), gray_color_t() );
            topLeft  = get_color( viewCumulSum(widthProcess-2,y-1), gray_color_t() );
            if( topLeft < top )
            {
                minValue = topLeft;
                dir = -1;
            }
            else
            {
                minValue = top;
                dir = 0;
            }
            viewCumulSum(widthProcess-1,y) = currentP + minValue;
            viewDirMap(widthProcess-1,y) = dir;

            //Other Column
            for(int x = 1; x < widthProcess-1; x++ )
            {
                currentP = get_color( map(x,y), gray_color_t() );
                topLeft  = get_color( viewCumulSum(x-1,y-1), gray_color_t() );
                top      = get_color( viewCumulSum(x  ,y-1), gray_color_t() );
                topRight = get_color( viewCumulSum(x+1,y-1), gray_color_t() );
                minValue = top;
                dir = 0;

                if( topLeft < minValue )
                {
                    minValue = topLeft;
                    dir = -1;
                }
                else if( topRight < minValue)
                {
                    minValue = topRight;
                    dir = 1;
                }
                viewCumulSum(x,y) = currentP + minValue;
                viewDirMap(x,y) = dir;
            }
        }
    }
```

```
void ProcessFoundVerticalSeam( boost::gil::gray32f_view_t& viewDirMap,
boost::gil::gray32f_view_t& viewCumulSum, std::vector<int>& verticalSeamPosition, int
widthProcess )
{
    using namespace terry;
    /* Find the last seam element. (pixel to remove in the last arrow of the image)*/
    float min = get_color( viewCumulSum(0, viewCumulSum.height()-1), gray_color_t() )
;
    int index = 0;
    for(int x = 1; x < widthProcess; x++ )
    {
        if ( get_color( viewCumulSum(x, viewCumulSum.height()-1), gray_color_t() ) <
min )
        {
            min = get_color( viewCumulSum(x, viewCumulSum.height()-1), gray_color_t()
) ;
            index = x;
        }
    }
    verticalSeamPosition.at(viewCumulSum.height()-1) = index;
    /* Find the other seam elements */
    for(int y = 0 ; y <= viewCumulSum.height()-2 ; y++ )
    {
        verticalSeamPosition.at(viewCumulSum.height()-2 - y) =
verticalSeamPosition.at(viewCumulSum.height()- 1 - y) +
get_color( viewDirMap( verticalSeamPosition.at(viewCumulSum.height()- 1 - y)
,viewCumulSum.height()- 1 - y ), gray_color_t() ) ;
    }
}

template<class View, class MapView>
void ProcessRemoveVerticalSeam( View& processingSrc, MapView& map, std::vector<int>&
verticalSeamPosition)
{
    using namespace terry;
    for (int y = 0; y < processingSrc.height(); y++)
    {
        for (int x = verticalSeamPosition.at(y); x < processingSrc.width()-1; x++)
        {
            processingSrc(x,y) = processingSrc(x+1,y);
            map(x,y) = map(x+1,y);
        }
    }
//     TUTTLE_COUT("The seam is removed");
}

template<class MapView>
void ProcessHorizontalCumulSum( MapView& map, boost::gil::gray32f_view_t& viewDirMap,
boost::gil::gray32f_view_t& viewCumulSum, int heightProcess)
{
    using namespace terry;
    float currentP;
    float topUp;
    float top;
    float topDown;
    float minValue;
    int dir;

    for(int y = 0; y < heightProcess; y++ )
```

```
{
    viewCumulSum(0,y) = map(0,y);
}
for(int x = 1; x < map.width(); x++ )
{
    //First arrow
    currentP = get_color( map(x,0), gray_color_t() );
    top      = get_color( viewCumulSum(x-1,0), gray_color_t() );
    topDown     = get_color( viewCumulSum(x-1,1), gray_color_t() );

    if( topDown < top )
    {
        minValue = topDown;
        dir = 1;
    }
    else
    {
        minValue = top;
        dir = 0;
    }
    viewCumulSum(x,0) = currentP + minValue;
    viewDirMap(x,0) = dir;

    //Last arrow
    currentP = get_color( map(x,heightProcess-1), gray_color_t() );
    top      = get_color( viewCumulSum(x-1,heightProcess-1), gray_color_t() );
    topUp     = get_color( viewCumulSum(x-1,heightProcess-2), gray_color_t() );

    if( topUp < top )
    {
        minValue = topUp;
        dir = -1;
    }
    else
    {
        minValue = top;
        dir = 0;
    }
    viewCumulSum(x,heightProcess-1) = currentP + minValue;
    viewDirMap(x,heightProcess-1) = dir;

    //Other arrow
    for(int y = 1; y < heightProcess-1; y++ )
    {
        currentP = get_color( map(x,y), gray_color_t() );
        topUp     = get_color( viewCumulSum(x-1,y-1), gray_color_t() );
        top      = get_color( viewCumulSum(x-1,y), gray_color_t() );
        topDown     = get_color( viewCumulSum(x-1,y+1), gray_color_t() );
        minValue = top;
        dir = 0;

        if( topUp < minValue )
        {
            minValue = topUp;
            dir = -1;
        }
        else if( topDown < minValue)
        {
            minValue = topDown;
```

```
                dir = 1;
            }
            viewCumulSum(x,y) = currentP + minValue;
            viewDirMap(x,y) = dir;
        }
    }
}

void ProcessFoundHorizontalSeam( boost::gil::gray32f_view_t& viewDirMap,
boost::gil::gray32f_view_t& viewCumulSum, std::vector<int>& horizontalSeamPosition,
int heightProcess )
{
    using namespace terry;
    /* Find the last seam element. (pixel to remove in the last arrow of the image)*/
    float min = get_color( viewCumulSum(viewCumulSum.width()-1,0), gray_color_t() ) ;
    int index = 0;
    for(int y = 1; y < heightProcess; y++ )
    {
        if ( get_color( viewCumulSum(viewCumulSum.width()-1,y), gray_color_t() ) < min
)
        {
            min = get_color( viewCumulSum(viewCumulSum.width()-1,y), gray_color_t() )
;
            index = y;
        }
    }
    horizontalSeamPosition.at(viewCumulSum.width()-1) = index;

    /* Find the other seam elements */
    for(int x = 0 ; x <= viewCumulSum.width()-2 ; x++ )
    {
        horizontalSeamPosition.at(viewCumulSum.width()-2 - x) =
horizontalSeamPosition.at(viewCumulSum.width()- 1 - x) +
get_color( viewDirMap( viewCumulSum.width()- 1 - x,
horizontalSeamPosition.at(viewCumulSum.width()- 1 - x)), gray_color_t() ) ;
    }
}

template<class View, class MapView>
void ProcessRemoveHorizontalSeam( View& processingSrc, MapView& map, std::vector<int>&
horizontalSeamPosition)
{
    using namespace terry;
    for (int x = 0; x < processingSrc.width(); x++)
    {
        for (int y = horizontalSeamPosition.at(x); y < processingSrc.height()-1; y++)
        {
            processingSrc(x,y) = processingSrc(x,y+1);
            map(x,y) = map(x,y+1);
        }
    }

    //TUTTLE_COUT("The seam is removed");
}


/**
 * @brief Function called by rendering thread each time a process must be done.
 * @param[in] procWindowRoW  Processing window
```

```
 */
template<class MapView, class View>
void SeamCarvingProcess<MapView, View>::multiThreadProcessImages( const OfxRectI&
procWindowRoW )
{
    using namespace terry;

    time_t start,end;
    double dif;

    OfxRectI procWindowOutput = this-
>translateRoWToOutputClipCoordinates( procWindowRoW );
    OfxPointI procWindowSize = { procWindowOutput.x2 - procWindowOutput.x1,
procWindowOutput.y2 - procWindowOutput.y1 };

    /* Affichage de src et dst */
    View src = subimage_view( this->_srcView, procWindowOutput.x1,
procWindowOutput.y1, procWindowSize.x, procWindowSize.y );
    View dst = subimage_view( this->_dstView, procWindowOutput.x1,
procWindowOutput.y1, procWindowSize.x, procWindowSize.y );


    /* Affichage de map --> gradient */
        MapView map = subimage_view( _mapView, procWindowOutput.x1,
procWindowOutput.y1, procWindowSize.x, procWindowSize.y );

    /* Affichage de CumulSum image */
    gray32f_image_t cumulSum ( map.width(), map.height(), 0 );
    gray32f_view_t viewCumulSum ( view(cumulSum) );

    /* Affichage de direction map */
    gray32f_image_t dirMap ( map.width(), map.height(), 0 );
    gray32f_view_t viewDirMap ( view( dirMap ) );

    /* Image temporaire pendant SeamCarving Process : */
    View processingSrc = subimage_view( this->_srcView, procWindowOutput.x1,
procWindowOutput.y1, procWindowSize.x, procWindowSize.y );


    std::vector<int> verticalSeamPosition( viewCumulSum.height() );     // vecteur
seam vertical
    std::vector<int> horizontalSeamPosition( viewCumulSum.width() );     // vecteur
seam horizontal


    int iteration_x = src.width() - _params._outputSize.x;
    int widthProcess = 0;
    int iteration_y = src.height() - _params._outputSize.y;
    int heightProcess = 0;


    //copy_and_convert_pixels( map, dst );

    if( _params._showSeamCarving )
    {
        TUTTLE_COUT("show Seam Carving");
        TUTTLE_COUT_VAR(iteration_x);
        TUTTLE_COUT_VAR(iteration_y);
        time (&start);
```

```
        if ( iteration_x > iteration_y )
        {
            int itDiffx = iteration_x - iteration_y;
            TUTTLE_COUT_VAR(itDiffx);
            for (int i = 0; i < iteration_y; i++)
            {
                //Vertical reduction
                widthProcess = src.width() - i;
                ProcessVerticalCumulSum( map, viewDirMap, viewCumulSum, widthProcess);
                ProcessFoundVerticalSeam( viewDirMap, viewCumulSum,
verticalSeamPosition, widthProcess);
                ProcessRemoveVerticalSeam(processingSrc, map, verticalSeamPosition);

                //Horizontal reduction
                heightProcess = src.height() - i;
                ProcessHorizontalCumulSum( map, viewDirMap, viewCumulSum,
heightProcess);
                ProcessFoundHorizontalSeam( viewDirMap, viewCumulSum,
horizontalSeamPosition, heightProcess );
                ProcessRemoveHorizontalSeam(processingSrc, map,
horizontalSeamPosition);
            }

            for (int i = iteration_y; i < iteration_x; i++)
            {
                //Vertical reduction
                widthProcess = src.width() - i;
                ProcessVerticalCumulSum( map, viewDirMap, viewCumulSum, widthProcess);
                ProcessFoundVerticalSeam( viewDirMap, viewCumulSum,
verticalSeamPosition, widthProcess);
                ProcessRemoveVerticalSeam(processingSrc, map, verticalSeamPosition);
            }
        }


        if ( iteration_y > iteration_x )
        {
            int itDiffy = iteration_y - iteration_x;
            TUTTLE_COUT_VAR(itDiffy);
            for (int i = 0; i < iteration_x; i++)
            {
                //Horizontal reduction
                heightProcess = src.height() - i;
                ProcessHorizontalCumulSum( map, viewDirMap, viewCumulSum,
heightProcess);
                ProcessFoundHorizontalSeam( viewDirMap, viewCumulSum,
horizontalSeamPosition, heightProcess );
                ProcessRemoveHorizontalSeam(processingSrc, map,
horizontalSeamPosition);

                //Vertical reduction
                widthProcess = src.width() - i;
                ProcessVerticalCumulSum( map, viewDirMap, viewCumulSum, widthProcess);
                ProcessFoundVerticalSeam( viewDirMap, viewCumulSum,
verticalSeamPosition, widthProcess);
                ProcessRemoveVerticalSeam(processingSrc, map, verticalSeamPosition);
            }
```

```
            for (int i = iteration_x; i < iteration_y; i++)
            {
                //Horizontal reduction
                heightProcess = src.height() - i;
                ProcessHorizontalCumulSum( map, viewDirMap, viewCumulSum,
heightProcess);
                ProcessFoundHorizontalSeam( viewDirMap, viewCumulSum,
horizontalSeamPosition, heightProcess );
                ProcessRemoveHorizontalSeam(processingSrc, map,
horizontalSeamPosition);
            }
        }

        if ( iteration_y == iteration_x )
        {
            for (int i = 0; i < iteration_x; i++)
            {
                //Vertical reduction
                widthProcess = src.width() - i;
                ProcessVerticalCumulSum( map, viewDirMap, viewCumulSum, widthProcess);
                ProcessFoundVerticalSeam( viewDirMap, viewCumulSum,
verticalSeamPosition, widthProcess);
                ProcessRemoveVerticalSeam(processingSrc, map, verticalSeamPosition);

                //Horizontal reduction
                heightProcess = src.height() - i;
                ProcessHorizontalCumulSum( map, viewDirMap, viewCumulSum,
heightProcess);
                ProcessFoundHorizontalSeam( viewDirMap, viewCumulSum,
horizontalSeamPosition, heightProcess );
                ProcessRemoveHorizontalSeam(processingSrc, map,
horizontalSeamPosition);
            }
        }

        copy_and_convert_pixels( processingSrc, dst);

        time (&end);
        dif = difftime (end,start);
        TUTTLE_COUT_VAR(dif);

        }
        else if( _params._showMap )
        {
        copy_and_convert_pixels( map, dst );
        TUTTLE_COUT("showMap active");
    }
    else if( !_params._showSeamCarving )
    {
        copy_and_convert_pixels( src, dst );
    }
}

}
}
}
```