ANNÉE 2021 - 2022

2281.1 Projet P2 SP – Rapport technique – ISC2il-b

ArcStreamLabs

ArcStreamLabs, un projet, un univers, un logiciel.

Étudiants participant à ce travail :

Alessio Comi, ISC2il-b
Nicolas Aubert, ISC2il-b
Théo Vuilliomenet, ISC2il-b

Présenté à :

Benoit Le Callennec
Stéphane Beurret
Nicolas Minning

Restitution du rapport : **14.06.2022**

Période : **2021 – 2022**

École : **HE-Arc, Neuchâtel**

haute école arc ingénierie
neuchâtel berne jura www.he-arc.ch

1 - Abstract

Se filmer, se mettre en scène, diffuser son image, autant d'activités qui hier encore paraissaient surprenantes, voire saugrenues, et pourtant aujourd'hui c'est d'une banalité sans nom. Que ce soit en photo sur Instagram ou en vidéo sur Twitch ou YouTube de plus en plus de gens cherchent à se mettre en avant et rivalisent d'ingéniosité pour se démarquer. Avec ArcStreamLabs nous offrons à ces créateurs de contenu un nouveau moyen de contrôler et modifier leur image en direct.

Au travers de ce rapport, nous tenterons de vous immerger dans la conception et la réalisation de notre application. Il sera notamment question de flux vidéo, de traitement d'image et de design.

Table des matières

1 - ABSTRACT **1**

2 - GLOSSAIRE **4**

3 - INTRODUCTION **5**

3.1 - CONTEXTE 5
3.2 - PRINCIPE DE NOTRE LOGICIEL 5

4 - ANALYSE **6**

4.1 - OBJECTIFS PRINCIPAUX 6
4.1.1 - Colorimétrie *7*
4.1.2 - Filtres *8*
4.1.3 - Effets spéciaux *8*
4.1.4 - Animations *8*
4.2 - OBJECTIFS SECONDAIRES 9
4.3 - OBJECTIFS TERTIAIRES 9
4.4 - PLANIFICATION 9
4.5 - DIAGRAMME DES CAS D'UTILISATION 9
4.6 - TECHNOLOGIES UTILISÉES 10

5 - CONCEPTION **11**

5.1 - MAQUETTES 11
5.2 - DIAGRAMME DE SÉQUENCE 12
5.2.1 - Les threads sur le buffer circulaire *12*
5.2.2 - Lecture et écriture dans le buffer circulaire *13*
5.3 - DIAGRAMME DE CLASSE 14

6 - RÉALISATION **15**

6.1 - GESTION DE PROJET 15
6.1.1 - Git Workflow suivi *15*
6.1.2 - Issue *15*
6.1.3 - Planification suivie *15*
6.2 - STRUCTURE DU PROJET 16
6.2.1 - CMake *17*
6.2.2 - CI/CD *17*
6.3 - CRÉATION UI 17
6.3.1 - Fenêtre principale *17*
6.3.2 - Fenêtres *19*
6.4 - BUFFER CIRCULAIRE 19
6.4.1 - Tests unitaires *19*
6.4.2 - Implémentation *20*
6.5 - RÉCUPÉRATION DU FLUX VIDÉO 21
6.5.1 - Prise d'instantanés *21*
6.6 - UNDO-REDO 22
6.6.1 - Tests unitaires du undo-redo *23*
6.7 - COLORIMÉTRIE 24
6.8 - FILTRES 24
6.8.1 - Filtre Sobel *24*
6.8.2 - Filtre Exposure *24*
6.8.3 - Filtre Stylization *24*
6.9 - EFFETS SPÉCIAUX 25

6.9.1 - Floutage en mosaïque 25
6.9.2 - Mirroring 26
6.9.3 - Détection faciale et floutage 26
6.10 - ANIMATIONS 27
6.10.1 - Filtre Moustache 27
6.10.2 - Bouncing text 27
6.11 - EXPORTATION / IMPORTATION DES PARAMÈTRES 28
6.11.1 - Exportation 28
6.11.2 - Importation 28

7 - RÉSULTATS 29

7.1 - USER INTERFACE 29
7.2 - COLORIMÉTRIE 30
7.3 - FILTRES 33
7.4 - EFFETS SPÉCIAUX 34
7.5 - ANIMATIONS 34
7.6 - UNDO / REDO 35
7.7 - EXPORTATION / IMPORTATION DES PARAMÈTRES 37

8 - LIMITATIONS ET PERSPECTIVES 38

8.1 - AJOUT DE NOUVEAUX FILTRES / EFFETS 38
8.2 - ÉTENDRE L'EXPORTATION / IMPORTATION DES PARAMÈTRES 38
8.3 - AMÉLIORATION DES MODÈLES DE DÉTECTION 39

9 - CONCLUSION 40

10 - ANNEXES I

10.1 - GUIDE UTILISATEUR I
10.2 - CAHIER DES CHARGES I
10.3 - TABLE DES ILLUSTRATIONS I
10.4 - BIBLIOGRAPHIES ET RÉFÉRENCES II
10.4.1 - Sites Web II

2 - Glossaire

Buffer circulaire	Structure de données circulaire facilitant la lecture et l'écriture par des threads différents, et limitant la mémoire utilisée.
Hover	Permets de sélectionner des éléments lorsque la souris se trouve au-dessus.
Layout	Algorithme simplifiant le placement des éléments d'une interface graphique
Mat	Structure de données propre à OpenCV qui représente une image, sous la forme d'une matrice.
OpenCV	Librairie permettant notamment de faire du traitement d'images.
Qt	Framework pour la programmation événementielle.
UI	*User Interface*, interface utilisateur.
Undo-redo	Pattern de programmation visant à garder une trace des opérations effectuées pour revenir en arrière.

3 - Introduction

3.1 - Contexte

Dans le cadre de notre projet P2 du semestre de printemps, nous avons pour but de réaliser un logiciel de traitement de flux vidéo en temps réel.

3.2 - Principe de notre logiciel

Notre logiciel permettra de lire un flux vidéo en entrée, de le traiter en temps réel et de visualiser le résultat en parallèle. Nous pourrons par exemple augmenter la saturation de l'image, changer les couleurs et appliquer divers filtres. Notre logiciel s'adresse aux créateurs de contenu vidéo en direct, notamment les streameurs.

Figure 1 - Exemple de traitement de flux

4 - Analyse

4.1 - Objectifs principaux

La mise en place et la prise en main de la librairie OpenCV ont été sous-estimées. En effet, comme nous ne l'avions jamais utilisée auparavant, nous ne connaissions pas exactement les possibilités qui s'offraient à nous. Lors de l'implémentation, nous nous sommes rendu compte que certains objectifs étaient hors sujets. Soit ils étaient trop complexes à mettre en place, soit simplement infaisables, avec nos connaissances d'OpenCV. C'est pourquoi ceux-ci, sous l'accord de M. Beurret, ont été mis à jour tout au long du projet.

Le premier point sera de récupérer un flux vidéo capturé en direct (webcam, appareil photo ou caméra) et de l'afficher tel quel.

Un autre objectif sera de permettre à l'utilisateur de prendre des instantanés du rendu vidéo et de l'enregistrer au format PNG.

De plus, nous souhaitons offrir à l'utilisateur la possibilité de modifier le rendu vidéo en temps réel, et ce par le biais de différents panels de réglages identifiés par les thèmes suivants : Colorimétrie, Filtres, Effets spéciaux et Animations. Les deux premières catégories seront composées de slider, de spinbox et de boutons, comme illustrés dans la Figure 2. Tous les réglages pourront être réalisés et appliqués simultanément. Les deux catégories suivantes, quant à elles, seront représentées comme dans la Figure 3. L'utilisateur ne pourra utiliser qu'un seul filtre colorimétrique à la fois, mais pourra cumuler autant d'animations qu'il le souhaite.

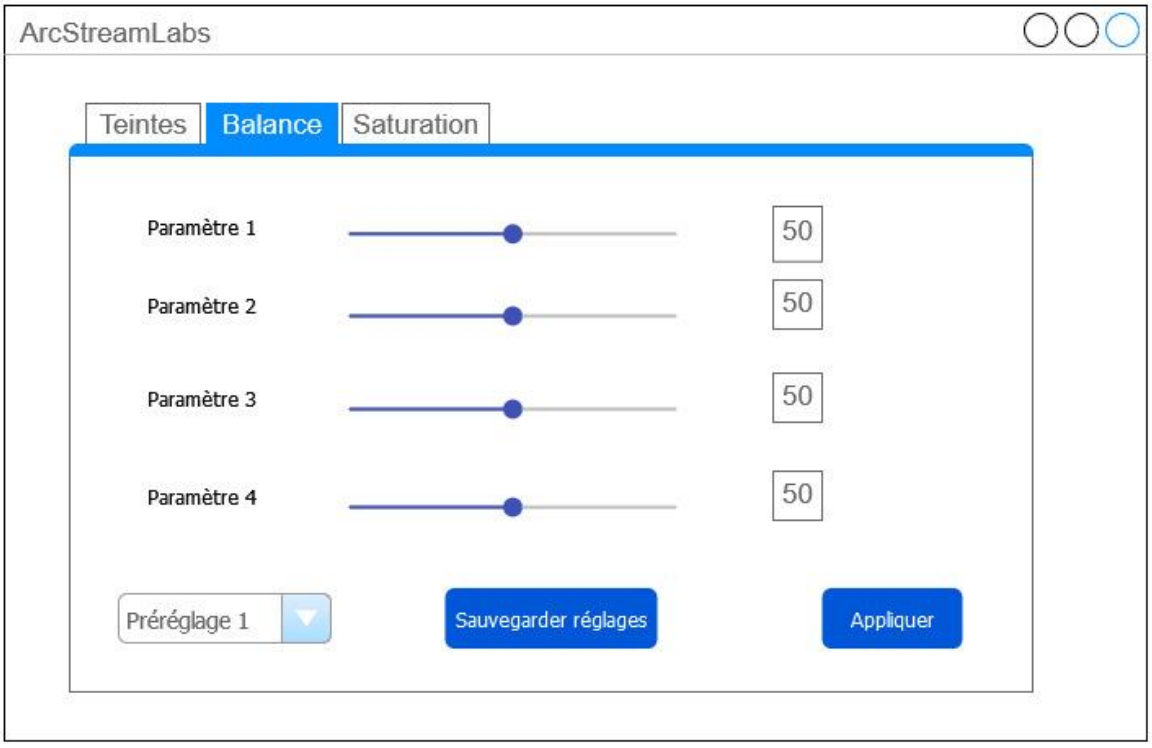

Figure 2 - Maquette des paramètres

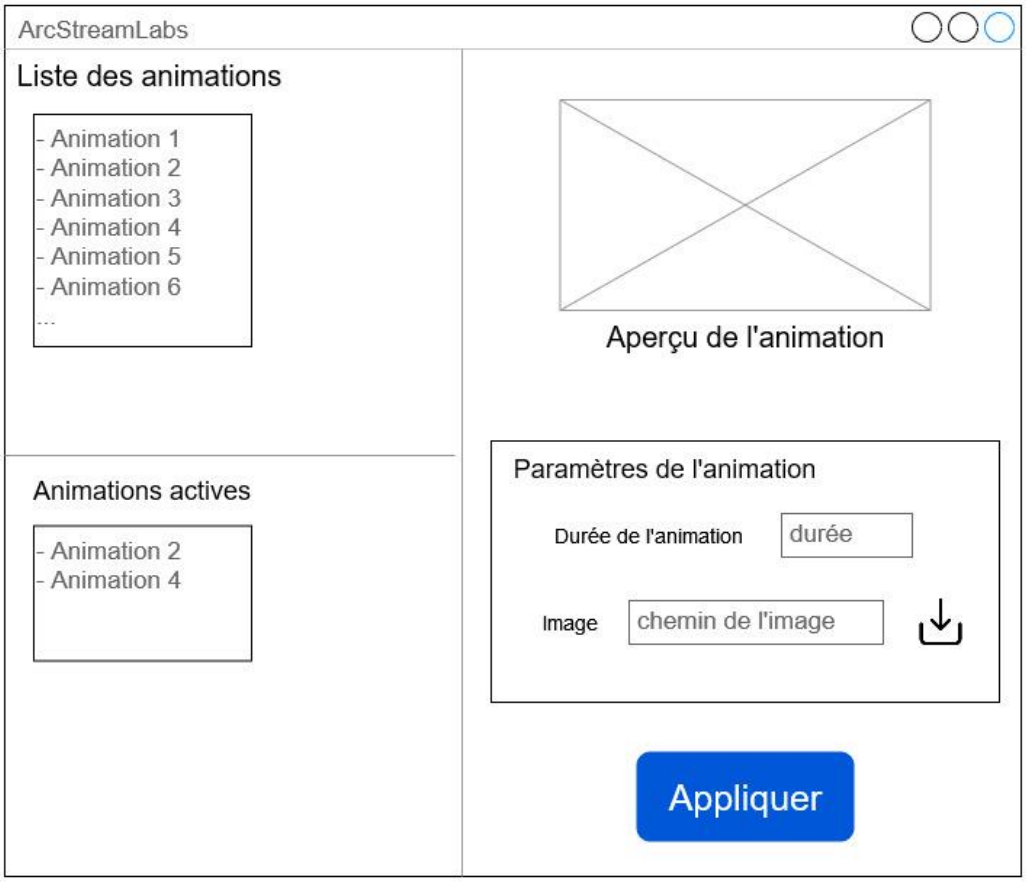

Figure 3 - Maquette des animations

4.1.1 - Colorimétrie

La colorimétrie permettra de retravailler le rendu visuel de l'image à proprement parler. Celle-ci comprendra :

Principaux :

- Balance des couleurs ;
- Luminosité ;
- Paramètres prédéfinis (noir et blanc, sepia, …).

Optionnels :

- Travail par courbe ;
- Travail par niveau.

4.1.2 - Filtres

Les filtres permettront d'appliquer une liste d'effets en complément des réglages colorimétriques, tels que :

- Filtre *Sobel* ;
- Filtre *Exposure* ;
- Filtre *Stylization*.

4.1.3 - Effets spéciaux

Les effets spéciaux viendront donner un corps à la modification de la vidéo en transformant significativement cette dernière.

- Mirroring (Inversion sur l'axe vertical) ;
- Mosaic blur (floutage sous forme de mosaïque) ;
- Détection faciale & Floutage.

4.1.4 - Animations

Les animations seront des additions d'éléments externes à la vidéo. Il y en aura principalement deux types. Le premier sera de faire bouger une ou plusieurs icônes de 40x40 pixels en suivant des patterns définis ou aléatoires.

- Détection faciale (bouche) & Superposition d'une image de moustache ;
- Bouncing text (Logo DVD).

Le deuxième sera l'utilisation de GIF laissés au choix de l'utilisateur.

4.2 - Objectifs secondaires

Au travers de ce projet, nous souhaitons devenir plus intimes avec le concept d'expérience utilisateur.

Dans un tout autre registre, nous voulons nous confronter à la problématique des ralentis, l'idée est de réussir à avoir un ralenti sans pour autant augmenter le délai capture/diffusion.

Finalement, nous souhaitons explorer notre créativité au travers d'ajout de filtres et d'effets spéciaux.

4.3 - Objectifs tertiaires

Si le temps nous le permet, nous aimerions également pouvoir nous pencher sur le travail du son, l'ajout d'une soundboard, mais aussi l'application de divers filtres sonores.

4.4 - Planification

Vous pouvez retrouver notre planification initiale ainsi que la planification suivie sur notre git en suivant ce lien : planification.

Nous avions largement sous-estimé les tâches liées à des configurations telles qu'OpenCV, CMake, CI/CD et encore les GoogleTests.

Heureusement, pour compenser ce temps perdu, nous avons rapidement mis en place certaines fonctionnalités telles que la colorimétrie, les filtres et les animations.

Il y a aussi certaines tâches que l'on pensait négligeables qui se sont avérées plus longues à implémenter que prévu. Nous les avons donc ajoutés à la planification. Il y avait notamment l'implémentation du buffer circulaire et le undo-redo.

4.5 - Diagramme des cas d'utilisation

Voici le lien sur notre diagramme des cas d'utilisation : use case.

4.6 - Technologies utilisées

Afin de mettre en pratique les connaissances acquises lors du cours de Génie logiciel dispensé par M. Le Callenec, il nous a été demandé d'intégrer au moins une structure de données complexe à notre projet. De plus, le langage de programmation est imposé et il s'agit du C++. Finalement, afin d'utiliser nos connaissances en programmation graphique événementielle, Qt sera le framework pour notre application.

Certains filtres demanderont des traitements assez lourds ; ceux-ci pourront entraîner des délais entre le flux entrant (capturé) et le flux sortant (affiché à l'utilisateur). Afin de limiter le nombre d'images (de frames en attente de traitement) stockées par notre application, nous allons mettre en place un buffer circulaire.

Puis, dans le but d'améliorer l'expérience utilisateur ainsi que pour approfondir nos capacités de conception, nous ajouterons le mécanisme d'undo-redo.

Résumé :

- Framework : Qt
- Langage : C++
- IDE : Qt Creator
- Cible : Windows (exe)
- Librairie : OpenCV
- Structure de données : Buffer circulaire
- Mécanisme complexe : undo-redo

5 - Conception

5.1 - Maquettes

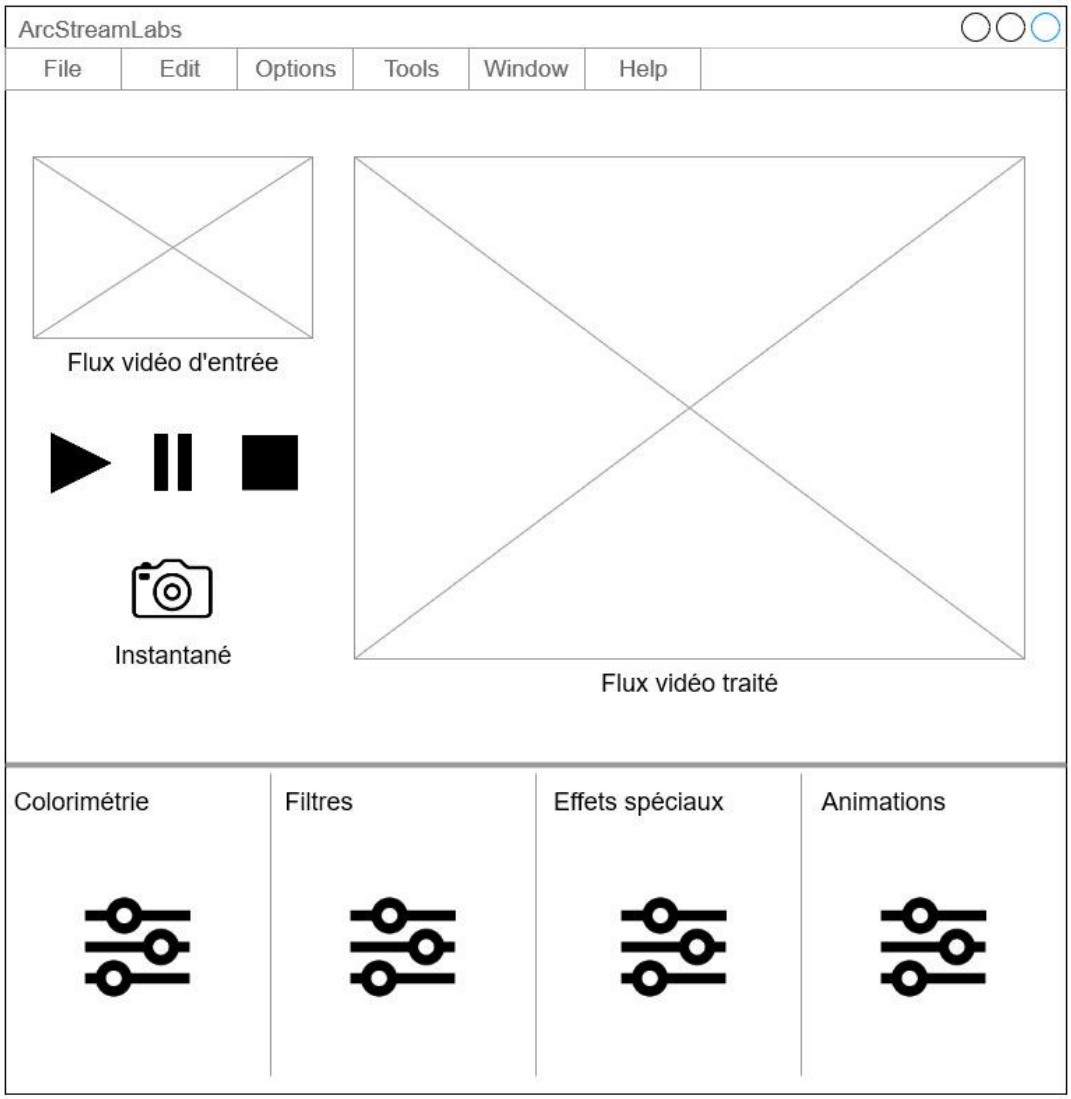

Figure 4 - Maquette de l'application

Pour ce qui est de l'interface principale, nous sommes restés fidèles à la maquette que nous avions produite en début de projet.

5.2 - Diagramme de séquence

5.2.1 - Les threads sur le buffer circulaire

Le diagramme ci-dessous illustre les threads qui vont utiliser le buffer circulaire. On remarque aussi que le buffer possède un *Lock* pour éviter que le *Consumer* puisse lire les données du buffer alors que le *Producer* est en train d'en écrire.

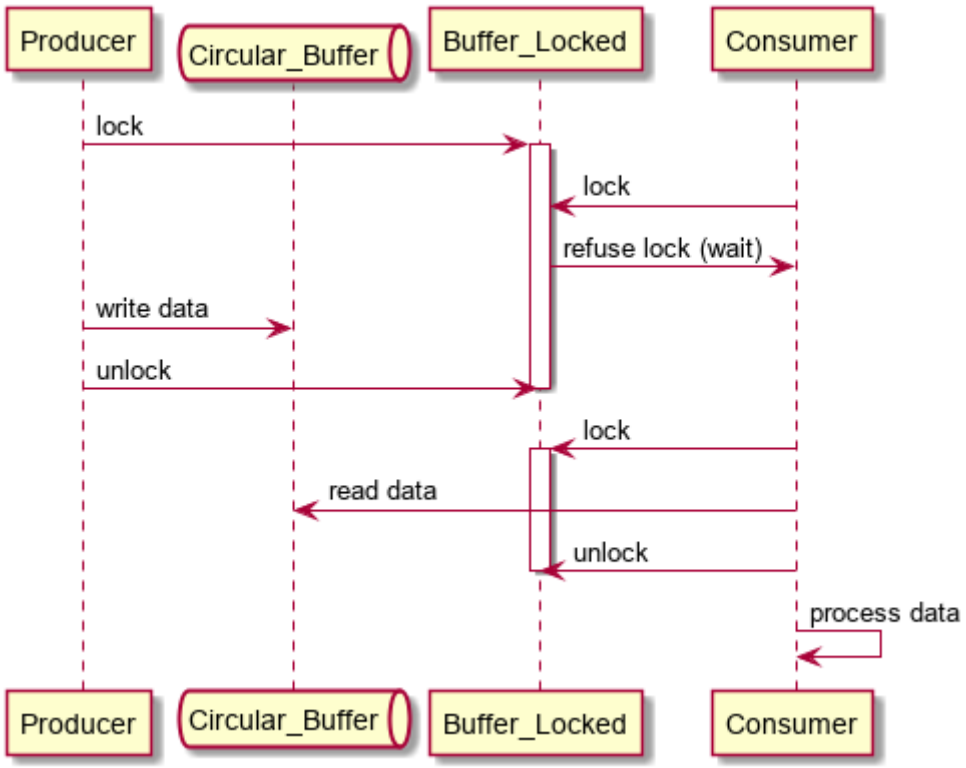

Figure 5 - Diagramme de séquence sur les threads du buffer circulaire

5.2.2 - Lecture et écriture dans le buffer circulaire

Ce deuxième diagramme de séquence détaille le fonctionnement des deux méthodes principales de notre buffer. On constate que la lecture dans le buffer est assez triviale. Cependant, la méthode d'écriture nécessitait une réflexion sur la politique à adopter en cas de buffer plein. Nous avons choisi d'écraser les anciennes données par les nouvelles lorsque le buffer était plein, car dans un contexte de vidéo en temps réel, perdre quelques images, ne devrait pas poser de grand problème.

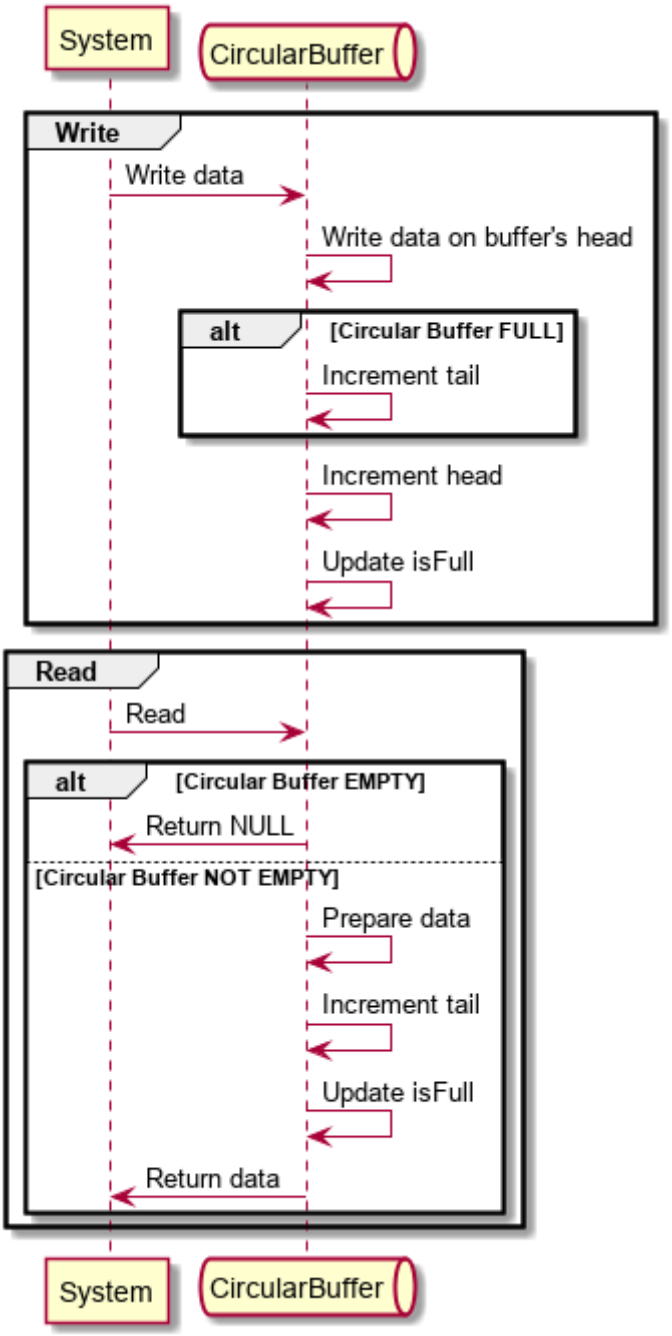

Figure 6 - Diagramme de séquence sur l'écriture/lecture du buffer circulaire

5.3 - Diagramme de classe

Le diagramme de classe représentant la fonctionnalité undo-redo suit le pattern command. Nous nous sommes inspirés d'un exemple trouvé sur le net[1] pour adapter au mieux le pattern command à notre problématique.

Cependant, la structure que nous avons implémentée s'en éloigne un peu, car nous devions ajouter certaines méthodes qui ne respectent plus ledit pattern qui doit normalement séparer totalement les *Actions* du *Manager.* Par exemple, nous avons implémenté une méthode qui permet de parcourir le *undo stack* afin d'exécuter à nouveau toutes *Actions* d'animation ou d'effets spéciaux.

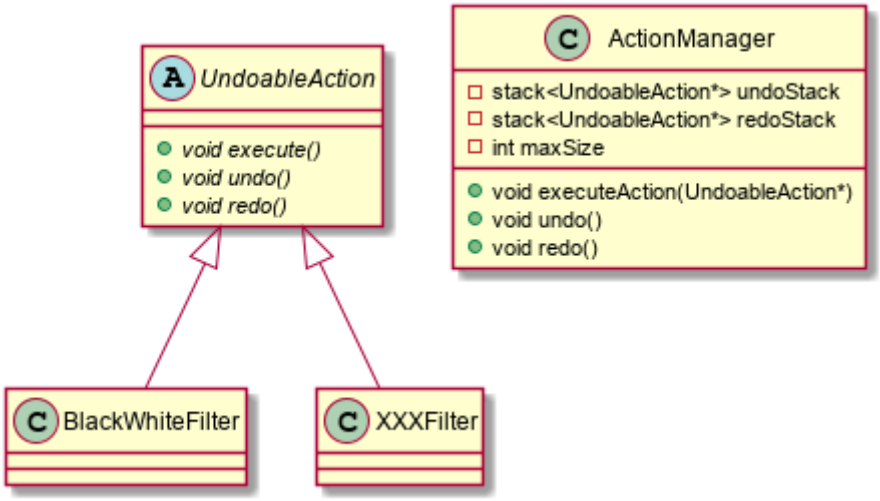

Figure 7 - Diagramme de classe undo-redo

Le diagramme de classe du buffer circulaire correspond à l'implémentation de notre structure de données.

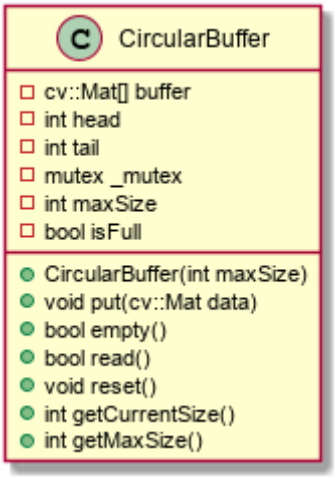

Figure 8 - Diagramme de classe buffer circulaire

[1] gernotklingler.com, Implementing undo/redo with command pattern, url : https://gernotklingler.com/blog/implementing-undoredo-with-the-command-pattern/

6 - Réalisation

6.1 - Gestion de projet

6.1.1 - Git Workflow suivi

À la suite du cours de Génie logiciel dispensé par M Le Callenec, nous avons décidé de suivre le workflow *rebase*.

Ce workflow nous semblait facile à mettre en place et à respecter tout en limitant les grands merges qui pouvaient nous prendre beaucoup de temps à gérer dans des projets antérieurs.

Voici un résumé des commandes que nous avons utilisées pour respecter le workflow :

Situation	**Commandes**
1) Mettre à jour une branche du repo	$ git checkout [branche repo] $ git pull --rebase
2) Créer une nouvelle branche locale	$ git checkout -b [nouvelle branche]

3) Ajouter ses modifications sur le dessus d'une branche du repo	$ git add [fichiers modifiés]
	$ git commit -m "commentaire"
	$ git checkout [branche repo]
	$ git pull –rebase
	$ git checkout [branche locale]
	$ git rebase [branche repo]

4) Fusionner ses modifications et les pousser sur le repo	$ git checkout [branche repo]
	$ git merge –ff-only [branche locale]
	$ git push origin [branche repo]

5) Effacer sa branche locale	$ git branch -d [branche locale]

6.1.2 - Issue

En début de chaque phase de développement (mercredi midi), nous avons pris l'habitude de nous réunir afin de discuter de l'avancée du projet et déterminer les objectifs que nous voulions réaliser durant la session. Ceux-ci ont été ajoutés en tant qu'issues sur GitLab, avec des labels (*To do*, *Doing* et *Done*).

6.1.3 - Planification suivie

Afin de garder une trace visuelle des durées des différentes tâches, nous avons mis à jour une seconde planification (suivie) après chaque session. Celle-ci est disponible dans la section « 4.4 – Planification ».

6.2 - Structure du projet

Voici l'arborescence de notre projet :

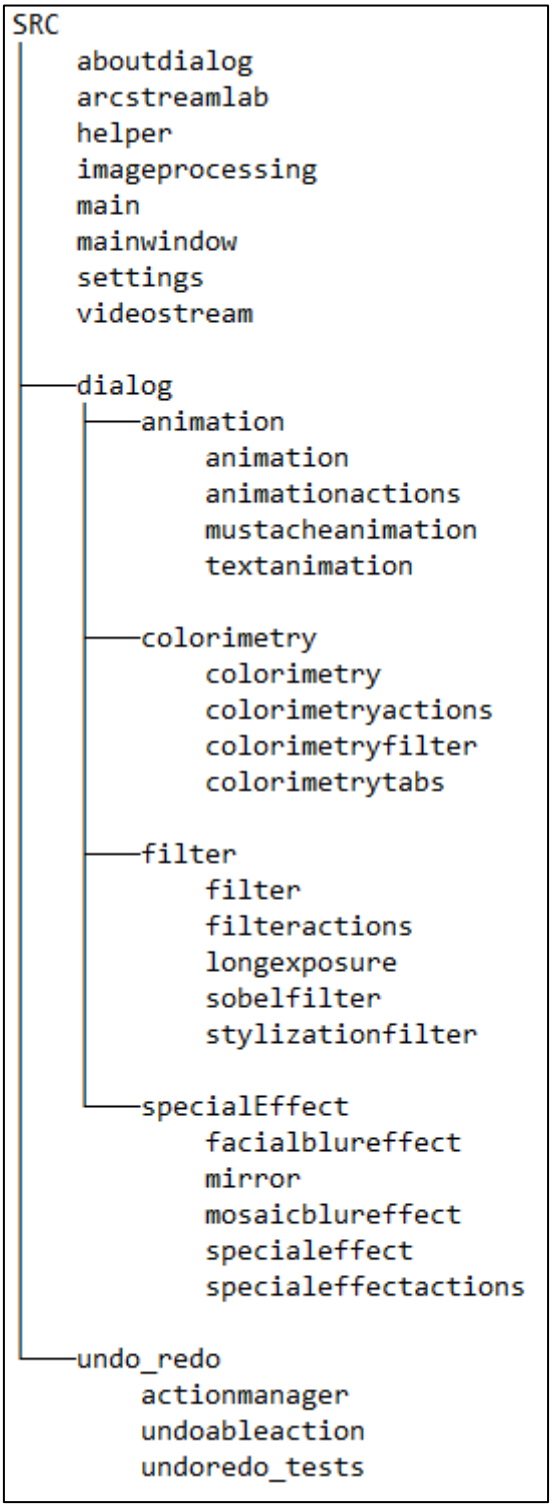

Figure 9 - Structure des fichiers

6.2.1 - CMake

Par habitude, nous déléguions la compilation et la configuration de notre projet à Qt Creator et son fichier .pro. Cependant, afin de mettre en pratique les notions de CMake vues en cours de Génie logiciel, nous devions utiliser ce dernier pour la compilation.

Nous avions à disposition un exemple de projet compilé avec CMake. Nous sommes donc repartis de cet exemple pour lier les tests unitaires et Doxygen à la compilation. Il a aussi fallu lier la librairie OpenCV à notre projet. C'est cette dernière étape qui a été la plus chronophage car nous ne trouvions pas de solution élégante pour lier OpenCV. Finalement, nous avons trouvé un moyen plus simple de lier OpenCV en passant par une variable d'environnement.

6.2.2 - CI/CD

Après avoir longtemps essayé de configurer notre CI/CD avec les outils nécessaires au bon fonctionnement de l'application, nous avons eu besoin d'installer Qt et OpenCV sur la vm. Fort heureusement, d'autres étudiants, qui avait les mêmes besoins que nous en termes d'installation, ont réussi à déployer leur propre image docker avec tous les outils installés. Nous avons pu repartir de leur image afin de l'adapter à notre fichier d'intégration continue.

La génération de la documentation avec Doxygen et l'exécution des tests unitaires avec *GoogleTest* ont aussi été ajoutés au CI/CD. Pour ce faire, nous avons suivi le *template* CPP fourni par M. Le Callennec.

6.3 - Création UI

La création de l'interface graphique de notre application a été réalisée sans l'aide du *Qt Designer* avec les *Widgets* et les *QLayouts*. Nous avons pris comme exempless les maquettes présentées dans le chapitre 4.1 et 5.1.

Un des points phares de l'interface graphique est sa lisibilité. De fait, tous les efforts ont été mis dans ce sens. C'est pourquoi une attention particulière a été portée sur le visuel, et ce dans deux buts. Premièrement, de garantir une unicité sur l'ensemble des fenêtres, mais également de proposer une lecture cohérente et compréhensible à l'utilisateur. Sous une approche plus factuelle, des techniques de mise en évidence simples ont été déployées telles que différencier le fond principal de celui des boutons, utiliser des *hover* ainsi que de créer des styles propres aux clics de souris. Tout ceci dans le but de discrètement guider l'utilisateur lors de ces différentes manipulations.

6.3.1 - Fenêtre principale

Afin de concevoir une UI qui puisse se redimensionner correctement, nous avons abondamment usé des *QLayouts* qui recalculent automatiquement la taille et la position de nos *widgets* d'après certaines règles que nous avons pu définir. Ce fut d'ailleurs la partie la plus longue à mettre en place, car cela demande une grande rigueur dans l'uniformisation des marges et des espaces ainsi que dans l'utilisation des bons *QLayouts* au bon moment.

Les principaux widgets que nous avons utilisés sont les suivants :

- *QGraphicsView* et *QGraphicsPixmapItem* : Affichent la vidéo brute en entrée et la vidéo traitée ;
- *QHBoxLayout* , *QVBoxLayout* et *QGridLayout* : Les *layouts* horizontaux, verticaux et en grille ;
- *QPushButton* : Les boutons ;
- *QLabel* : Les labels.

La majeure partie des agrandissements se font grâce à la propriété *setColumnStretch* des *GridLayout* qui permet de définir la répartition horizontale de manière dynamique.

```
this->displayGridLayout->setColumnStretch(0, 1);
this->displayGridLayout->setColumnStretch(1, 3);
```

En définissant un *stretch* de 1 pour la première colonne et un stretch de 3 pour la deuxième colonne, on fixe la répartition à 25% pour la première colonne et 75% pour la deuxième. Ce qu'on peut observer sur l'image ci-dessous.

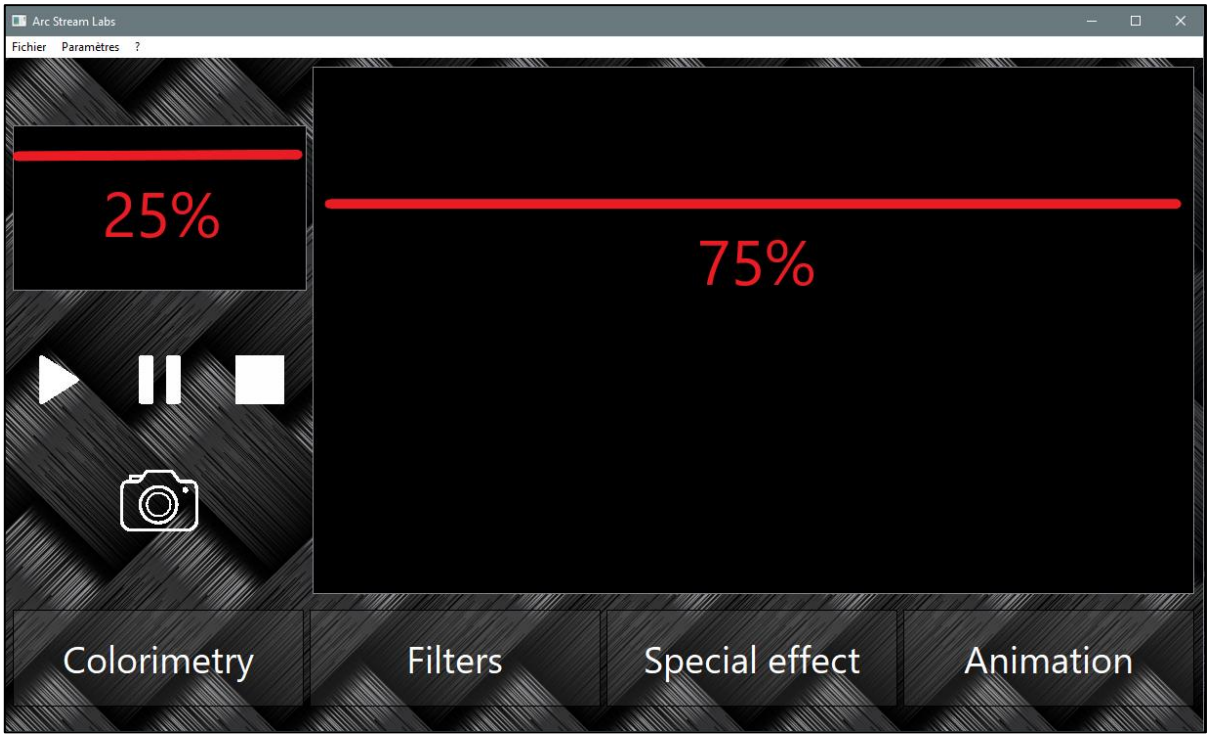

Figure10 - Fenêtre principale, répartition horizontale

Les propriétés *setMaximumHeight*, *setMaximumWidth*, *setMinimumHeight* ainsi que celle du *setMinimumWidth* ont été beaucoup utilisées pour faire en sorte que ce soit l'affichage vidéo qui soit prioritaire dans le redimensionnement et non les paramètres en bas.

6.3.2 - Fenêtres

D'une manière similaire à l'*UI*, les différentes fenêtres sont construites de façon à pouvoir se redimensionner simplement et de manière cohérente. Chacune d'entre elles est indépendante des autres, permettant à l'utilisateur de les disposer à sa convenance. De plus chacune d'entre elles respecte le code graphique et de lecture de l'*UI*.

La fenêtre colorimétrie, contrairement aux autres, a été construite au moyen d'un *QTabWidget*, nous laissant ainsi avoir une page avec les filtres prédéfinis, au moyen d'une liste de bouton, mais également une page contenant les divers curseurs de défilement permettant d'influer directement sur le rendu de chaque paramètre de couleur.

Les autres fenêtres suivent le même patron de conception, une liste verticale de bouton, curseur ou de boîte à texte offrant à l'utilisateur la possibilité de modifier son image.

6.4 - Buffer circulaire

6.4.1 - Tests unitaires

Avant d'implémenter le *buffer circulaire*, nous avons mis en place les tests unitaires afin de vérifier le bon fonctionnement de celui-ci. Ces tests, comme tous les autres, ont été réalisés à l'aide des *GoogleTest*. Pour leur mise en place, nous nous sommes grandement basés sur l'exemple de M Le Callenec, disponible ici.

La structure de fichiers est la suivante :

```
lib/
└── LibCircularBuffer
    ├── CMakeLists.txt
    ├── libcircularbuffer_1.cpp
    ├── libcircularbuffer_1.h
    └── libcircularbuffer_1_tests.cpp
```

Figure 11 - Buffer circulaire, structure des fichiers

Le fichier *CMakeLists.txt* inclut OpenCV et crée une *Target* qu'il inclut dans une librairie.

Puis, dans le *CMakeLists.txt* principal, nous récupérons une archive de *GoogleTest* :

```
include(FetchContent)
FetchContent_Declare(
  googletest
  URL
https://github.com/google/googletest/archive/609281088cfefc76f9d
0ce82e1ff6c30cc3591e5.zip
  )
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)
```

Les fichiers de tests sont ajoutés à l'exécutable :

```
add_executable(tests test/tests.cpp ${LibCircularBuffer_TESTS}
src/undo_redo/undoredo_tests.cpp)
```

Les sources sont ajoutées à la *Target* :

```
target_include_directories(tests PUBLIC
${CMAKE_CURRENT_SOURCE_DIR})
target_link_libraries(tests gtest_main
Qt${QT_VERSION_MAJOR}::Widgets ${OpenCV_LIBS} LibCircularBuffer)
```

Finalement, les *GoogleTest* sont inclus et les tests exécutés :

```
include(GoogleTest)
gtest_discover_tests(tests)
enable_testing()
```

6.4.2 - Implémentation

La structure de notre classe *CircularBuffer* correspond à celle explicitée dans son diagramme de classe, disponible dans la section 5.3.

La fonction *put* permet d'ajouter une matrice au buffer. Si le pointeur d'écriture (*head*) a atteint la taille maximale du *buffer*, celui-ci est remis à 0 : c'est cela qui permet la circularité.

```
this->head = (this->head + 1) % this->maxSize;
```

Si le pointeur d'écriture atteint le pointeur de lecture (le rattrape), cela signifie que le buffer est plein. Dans ce cas, le pointeur d'écriture incrémente le pointeur de lecture et écrit (il le pousse en avant).

```
if(this->isFull) this->tail = (this->tail + 1) % this->maxSize;
```

Ce mécanisme peut entraîner la perte de quelques images, dans le pire cas on peut envisager une perte d'un nombre élevé d'images, mais ce faisant la mémoire utilisée reste limitée.

6.5 - Récupération du flux vidéo

La récupération du flux vidéo se fait dans la méthode *run* de la classe *VideoStream*, héritant de la classe *QThread*.

OpenCV met à disposition une classe *VideoCapture* permettant de récupérer une image à partir d'une caméra (webcam dans notre cas). Il a alors suffi de placer cette récupération d'image à l'intérieur d'une boucle infinie, et d'émettre un signal une fois celle-ci récupérée.

Comme la classe *VideoStream* hérite de *QThread*, celle-ci peut être démarrée dans un second Thread, afin de ne pas figer le programme lors de la récupération.

Dans notre classe principale, une fois le signal de *VideoStream* émis, l'image est placée dans le buffer circulaire, puis affichée à l'écran, à l'intérieur d'un *QGraphicsView*.

6.5.1 - Prise d'instantanés

Une fois la récupération du flux mis en place, la prise d'instantané s'est avérée extrêmement simple et rapide à implémenter. En effet, il a suffi de récupérer l'image actuellement affichée à l'écran, puis de l'enregistrer au format *PNG* à l'emplacement choisi par l'utilisateur.

6.6 - Undo-Redo

Le *pattern undo-redo* mémorise toutes les actions effectuées à l'exécution de notre logiciel. Ce qui permet à l'utilisateur de naviguer dans l'historique des actions à l'aide des commandes CTRL + Z et CTRL + Y. Ce *pattern* est très répandu et implémenté dans la quasi-totalité des applications déployées sur le marché. Cependant, son implémentation n'est pas aussi triviale que l'on pourrait le croire.

Dans notre cas, nous avions plusieurs types d'actions à gérer. La classe abstraite regroupant ces actions s'appelle *UndoableAction* et hérite de *QObject* afin d'utiliser les *slots* de Qt. Quel que soit le type d'action, elle doit être capable de s'exécuter et de se retirer pour revenir à l'état précédent.

Afin de séparer les différents types d'actions, nous avons établi un premier niveau de spécialisation qui comprend quatre classes abstraites qui héritent de *UndoableAction*.

- *ColorimetryAction* : Les actions de ce type vont mémoriser une matrice qui contient les paramètres de colorimétrie actuels (pour le undo) et une autre matrice qui contient les nouveaux paramètres à utiliser.
- *FilterAction* : Les actions de ce type vont travailler avec un Mat en entrée et un Mat en sortie pour y appliquer le filtre.
- *SpecialEffectAction* : Tout comme les *FilterAction*, ce type d'action va prendre un Mat en entrée et retourner un Mat en sortie.
- *AnimationAction* : Encore une fois, un Mat en entrée et un Mat en sortie.

Cette séparation nous permettra aussi, par la suite, de *caster* avec la classe abstraite pour récupérer le type des actions dans la liste de toutes les actions.

Afin de tout gérer et de garder en mémoire l'historique des actions, il a fallu créer une classe *ActionManager*. Cette dernière possède deux *stacks*. Une première pour stocker les actions courantes, celles qui seront potentiellement annulées, qui donc nous permet de revenir en arrière (*undostack*) et une deuxième qui, à l'inverse, va contenir toutes les actions qui ont été annulées pour pouvoir les restaurer (*redostack*). De plus, c'est ce manager qui possède les méthodes principales du *pattern* qui sont bien évidemment *undo* et *redo*. Les deux méthodes fonctionnent de manière similaire. Dans *undo*, on va dépiler le premier élément du *undostack* et le placer sur le *redostack* et dans *redo*, c'est l'inverse.

Par la suite, nous avons tout de même ajouté quelques fonctions permettant de parcourir et d'utiliser certains types de *UndoableAction*. Nous en avions besoin, car notre application traite en permanence des flux et lors de chaque traitement il faut exécuter certaines actions.

L'ajout des fonctions de parcours va à l'encontre du *pattern undo-redo* que nous avions imaginé. Cependant, il a bien fallu adapter le *pattern* à notre cas qui contenait du traitement temps-réel qui n'est pas pris en compte dans le *pattern* de base. Nous aurions peut-être pu trouver une meilleure conception plus spécifique à notre cas d'utilisation. Malgré cela, le *undo-redo* fonctionne pour toutes les actions de notre logiciel.

6.6.1 - Tests unitaires du undo-redo

Le fonctionnement de notre structure *undo-redo* est dépendante d'autres classes héritantes de *UndoableAction*. C'est pourquoi, afin de tester notre *ActionManager* sans coupler les tests à nos futures classes qui auront des actions, nous avons créé une classe héritant de *UndoableAction* qui possède un comportement trivial.

```
private:
    int backup;
    int newValue;
    int * value;

public:
    UndoableActionTest(int newValue, int * value);
    virtual ~UndoableActionTest() override;

    void execute() override;
    void undo() override;
```

Un objet de cette classe est créé à partir d'un pointeur sur la valeur actuelle d'un entier et de la nouvelle valeur qu'on souhaite y mettre. Grâce à nos trois attributs, il est maintenant possible de modifier la nouvelle valeur de l'entier et de la restaurer avec la variable backup que nous avons créée.

Afin de tester que l'action pouvait être annulée et restaurée, nous avons implémenté le test suivant :

```
ActionManager actionManager;
int *value; *value = 10;
UndoableActionTest * uat = new UndoableActionTest(20, value);
int v1 = *value;

actionManager.addAction(uat);
int v2 = *value;

actionManager.undo();
int v3 = *value;

actionManager.redo();
int v4 = *value;

EXPECT_EQ(v1 != v2, true);
EXPECT_EQ(v1 == v3, true);
EXPECT_EQ(v1 != v4, true);

delete uat; uat = nullptr;
```

Par la suite, toutes les classes utilisant le *undo-redo* se sont inspirées de cette structure afin de respecter au mieux le *pattern*.

Certaines actions comme les filtres ont nécessité une autre implémentation, car ce ne sont plus de simples valeurs, mais des fonctions à effectuer sur des valeurs. C'est pourquoi nous avons aussi ajouté des tests sur ces implémentations.

6.7 - Colorimétrie

Grâce à la fonction *Filter2D* d'OpenCV, il est possible d'appliquer nos propres filtres linéaires. Cela permet entre autres de jouer avec les couleurs de l'image, pour, par exemple, appliquer un filtre rouge, ou encore un filtre sépia.

Pour plus d'informations, nous vous renvoyons sur le tutoriel officiel, disponible sur la documentation d'OpenCV.[2]

6.8 - Filtres

6.8.1 - Filtre Sobel

Le premier filtre qui a été implémenté est le « *sobel filter* ». L'implémentation du filtre en soi vient de la documentation officielle de OpenCV[3]. Nous avons commencé par copier le code afin d'avoir rapidement un rendu fonctionnel, puis nous l'avons pris en main, mais adapté à notre convenance. Son principe est détaillé sur la référence que nous avons utilisée.

La difficulté a surtout été d'ajouter l'action *sobel filter*, à notre architecture *undo-redo*. Pour ce faire, nous avons créé une classe *SobelFilter* qui hérite de *FilterAction*. Tout le code lié au filtre se trouve donc dans la méthode *execute* de la classe *SobelFilter*. Après cela, nous avons redirigé les signaux liés à l'activation de ce filtre jusqu'à la classe mère *ArcStreamLab* qui s'occupent de l'ajouter au *ActionManager*. Par la suite, afin de continuer d'utiliser le *sobel filter* à chaque rafraichissement de notre flux vidéo, nous parcourons la liste des filtres actifs afin de traiter l'image.

6.8.2 - Filtre Exposure

Ce filtre permet de rajouter des images en transparence sur le flux de sortie. Pour ce faire nous travaillons avec des images PNG, ce format n'étant nécessaire que pour obtenir un résultat plus séduisant. Une fois celle-ci sélectionnée, nous ajoutons à la matrice du flux vidéo les matrices de chaque image afin de pouvoir les ajouter aux flux et qu'elle soit intégrée lors d'une capture.

6.8.3 - Filtre Stylization

Ce filtre, de la famille des filtres au « rendu non photoréaliste », produit une image avec un effet de peinture à l'eau. Pour le mettre en place, nous avons suivi une procédure similaire à celle des filtres précédents, tout en nous aidant de la documentation.

Il peut être simplement appliqué grâce à la fonction *stylization*, qui prend comme paramètres la matrice source, la matrice de sortie, ainsi que de plusieurs valeurs permettant de varier le

[2] OpenCV, Making your own linear filters!, url : https://docs.opencv.org/4.x/d4/dbd/tutorial_filter_2d.html
[3] OpenCV, Sobel Derivatives, url : https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html

résultat. Pour plus d'informations quant à son utilisation de la paramétrisation, nous vous renvoyons sur la documentation.[4]

Malheureusement, après l'avoir mis en place, nous avons constaté qu'il consommait énormément de ressources, ce qui limite les images par seconde du flux de sortie (création de latence). Toutefois, cela démontre le bon fonctionnement de notre buffer circulaire, qui écrase les données en cas de traitement trop long, afin de limiter la mémoire utilisée.

Ces latences influent également sur le bon fonctionnement du *undo-redo*, ce que nous ne pouvons pas expliquer, même si nous pensons que cela est lié à Qt plus qu'à notre implémentation.

6.9 - Effets spéciaux

6.9.1 - Floutage en mosaïque

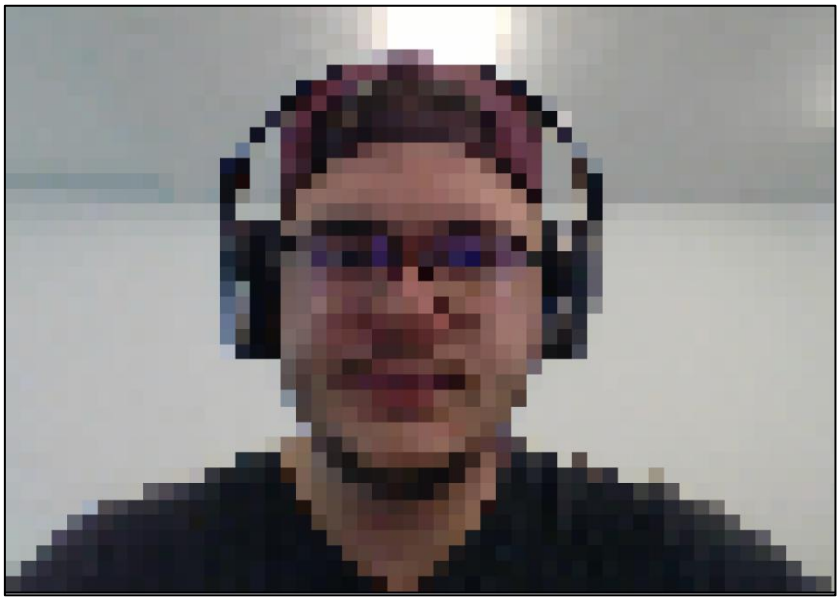

Figure 12 - Exemple de floutage en mosaïque

Le floutage en mosaïque, illustré par l'image ci-dessus, peut être réalisé avec l'extrait de code suivant :

```
cv::Mat src, temp, output;

cv::resize(this->mat, temp, cv::Size(cols, rows), 0, 0,
cv::INTER_LINEAR);

cv::resize(temp, output, cv::Size(matWidth, matHeight), 0, 0,
cv::INTER_NEAREST);
```

[4] OpenCV, Non-Photorealistic Rendering, url : https://docs.opencv.org/4.x/df/dac/group__photo__render.html#gacb0f7324017df153d7b5d095aed53206

La matrice *src* est la matrice source. La matrice *temp* est une matrice temporaire. La matrice *output* est la matrice résultante, avec le floutage appliqué.

Cols et *rows* sont le nombre de carrés (pixels) souhaité sur l'axe horizontal et l'axe vertical. La fonction *resize* d'OpenCV permet de, comme son nom l'indique, modifier les dimensions d'une matrice.

Ensuite, il suffit de redonner sa taille initiale à la matrice *temp* (la ragrandir), qui vient d'être rétrécie, pour que l'effet de pixélisation apparaisse.

6.9.2 - Mirroring

Le mirroring est une fonction par défaut d'OpenCV. Il nous a juste suffi d'utiliser la méthode « flip » et l'image est retournée.

```
cv::Mat output;
cv::flip(this->mat, output, 1);
```

6.9.3 - Détection faciale et floutage

OpenCV met à disposition différents modèles préentraînés, sous forme de fichier XML, permettant de détecter différents objets (bouche, tête, plaque d'immatriculation …) sur une image (matrice).

Ces modèles peuvent être utilisés à l'aide de la classe *CascadeClassifier*, mise à disposition par OpenCV. Premièrement, il est nécessaire de charger le modèle à utiliser, à l'aide de la méthode *load*.

```
cv::CascadeClassifier cascadeFrontalFace;
cascadeFrontalFace.load("path/to/file");
```

La détection se fait alors comme dans l'exemple suivant :

```
Std::vector<Rect> frontalFaces;
cascadeFrontalFace.detectMultiScale(matFrameGraySmall,
frontalFaces);
```

Les différentes zones dans lesquelles des objets ont été détectés sont stockées sous forme de *Rect* dans un vecteur. Elles peuvent être parcourues avec une simple boucle, dans laquelle des traitements peuvent être appliqués. Dans l'exemple suivant, un filtre gaussien est appliqué sur l'image initiale, à l'emplacement de chaque zone de détection :

```
for (size_t i = 0; i < frontalFaces.size(); i++)
{
    cv::Rect r = frontalFaces[i];
    cv::GaussianBlur(this->mat(r), this->mat(r));
}
```

6.10 - Animations

6.10.1 - Filtre Moustache

Sur le principe, ce filtre fonctionne de la même façon que la détection faciale, cependant au lieu de flouter le visage de l'utilisateur, le filtre va ajouter une image de moustache.

Afin de placer une moustache au niveau de la bouche, nous avons commencé par récupérer un fichier XML fourni par OpenCV qui est en réalité un modèle entrainé à la détection de la bouche.

Par la suite, le principe est de zoomer à l'intérieur de l'image afin de trouver une zone qui puisse correspondre à une bouche et d'en récupérer la position.

Finalement une fois la position idéale trouvée, il faut encore transformer l'image de moustache afin qu'elle soit ajoutable par-dessus l'image de base.

Afin de réaliser cette animation, nous avons suivi un tutoriel[5] spécifique à cette problématique.

6.10.2 - Bouncing text

Tout le monde a sûrement déjà pu voir le fameux logo DVD qui se balade sur un écran en veille et rebondissant sur tous les côtés. C'est ce que nous allons décrire dans cette partie. En effet, nous nous sommes inspirés de l'animation DVD pour faire la nôtre. Néanmoins, nous n'allons pas écrire DVD, mais laisser la possibilité à l'utilisateur d'inscrire son propre texte dans un *QLineEdit* et d'en récupérer la valeur pour l'afficher par-dessus l'image.

L'animation fonctionne de la manière suivante :

1. On définit de manière aléatoire la position initiale du texte ;
2. On récupère le texte saisi par l'utilisateur pour le mettre par-dessus le Mat avec quelques paramètres supplémentaires comme la police ;

```
cv::putText(this->mat, this->text.toStdString(), position,
cv::FONT_HERSHEY_SIMPLEX, 0.8, cv::Scalar(255, 255, 255), 2);
```

3. On met à jour la future nouvelle position en fonction de la vitesse sur l'axe x et sur l'axe y tout en vérifiant qu'on ne sorte pas de l'image.

Le point trois ainsi que l'affichage du texte sont réalisés à chaque fois que l'image est rafraichie.

[5] Datahacker, How to create Instagram like filters, mustaches, glasses, and masks?, url : https://datahacker.rs/003-opencv-projects-how-to-create-instagram-like-filters-mustaches-glasses-and-masks/

6.11 - Exportation / Importation des paramètres

6.11.1 - Exportation

Afin de permettre l'exportation des paramètres de nos classes, il était nécessaire de les rendre sérialisables. Nous avons décidé de déléguer cette tâche à une classe globale, nommée *Settings*.

Elle possède une référence des instances des classes *ColorDialog*, *FilterDialog*, *SpecialEffectDialog* et *AnimationDialog*, afin de pouvoir accéder à leurs différents paramètres.

Lors de l'enregistrement sur un fichier (*extension.asls*, propre à notre logiciel), les paramètres sont récupérés, sérialisés puis ajoutés au fichier. Cela est fait grâce à la classe *QDataStream* de Qt.

Par manque de temps, les seuls paramètres enregistrés lors de l'exportation sont ceux de la fenêtre *Colorimétrie*. Cependant, la mise en place de l'exportation des paramètres des autres fenêtres serait très similaire, c'est pourquoi nous avons décidé de prendre ce temps pour implémenter d'autres fonctionnalités.

6.11.2 - Importation

L'importation fonctionne comme l'exportation, mais dans le sens inverse : le contenu du fichier d'export (.*asls*) est lu, les paramètres désérialisés, puis appliqués à la fenêtre correspondante.

7 - Résultats

Ici, nous allons vous présenter visuellement le rendu de notre application ainsi que des exemples d'utilisation des traitements d'image que nous avons mis en place.

7.1 - User Interface

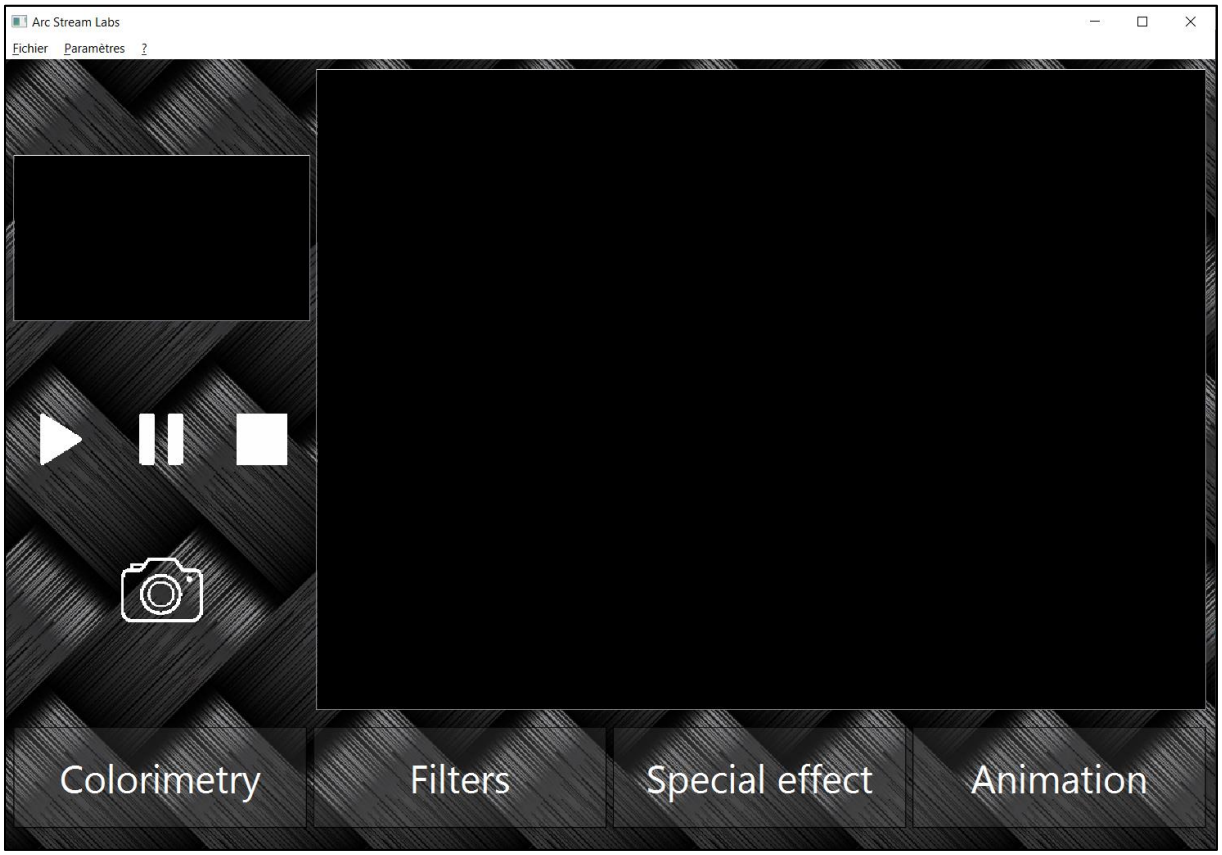

Figure 13 - Fenêtre principale

L'user interface est la fenêtre principale de notre application, cette dernière contient deux emplacements pour la lecture vidéo, brute en haut à gauche ainsi que le rendu au milieu à droite. De plus elle comprend huit boutons (quatre graphiques et quatre textuels).

7.2 - Colorimétrie

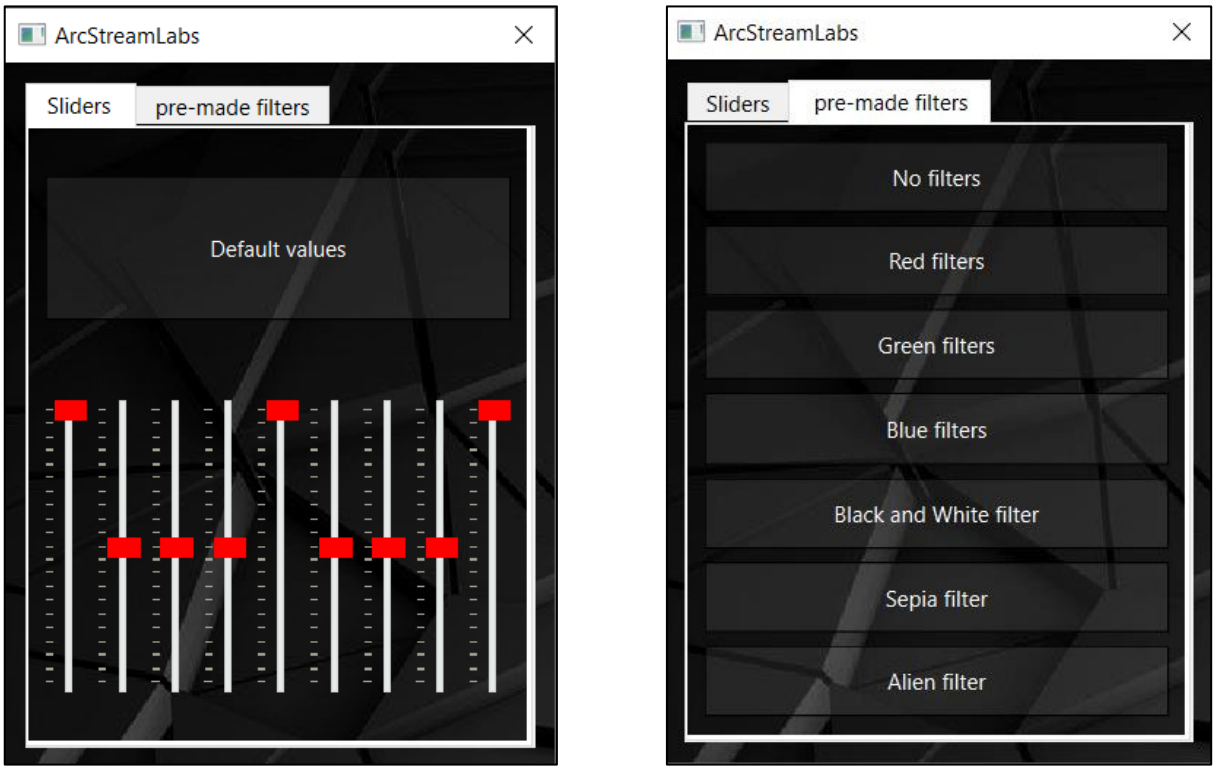

Figure 14 - Fenêtre colorimétrie, onglets "Sliders" et "Pre-made filters"

Comme cité plus haut la fenêtre colorimétrie est séparé en deux sous-ensembles. Le premier, sur la gauche permet de manuellement sélectionner les valeurs de chaque curseur afin d'obtenir un réglage fin sur la couleur souhaitée, le deuxième permet de choisir un filtre tout fait pour directement avoir un résultat. Il est intéressant de noter que les filtres déjà faits modifieront les curseurs, permettant ainsi à l'utilisateur d'affiner son choix sur un préréglage.

Les prochaines captures illustrent certains effets obtenus avec *ArcStreamLab*.

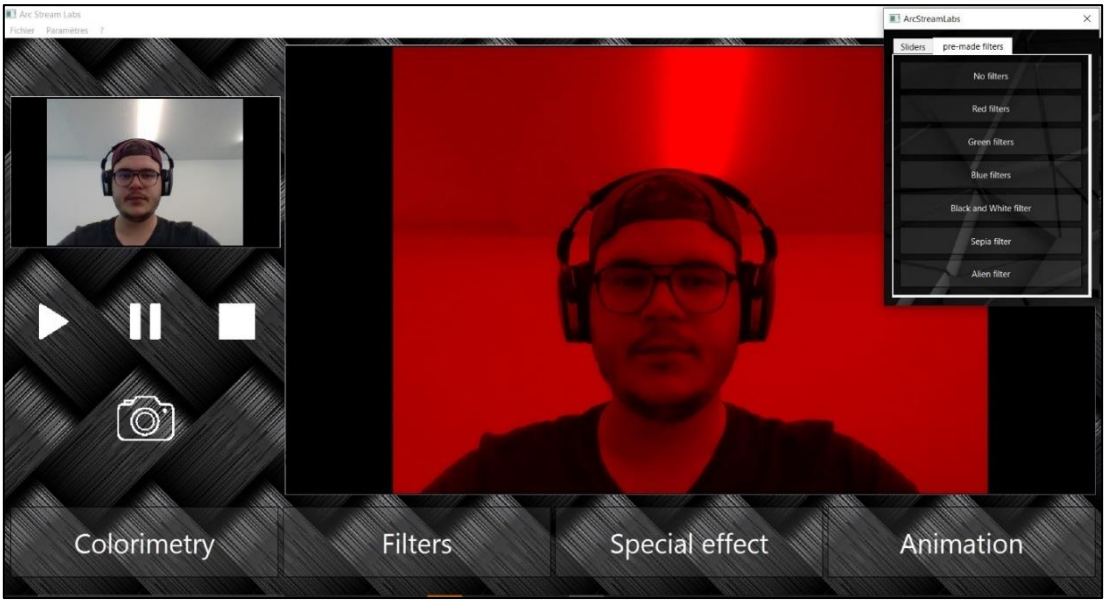

Figure 15 - Exemple colorimétrie 1 (rouge)

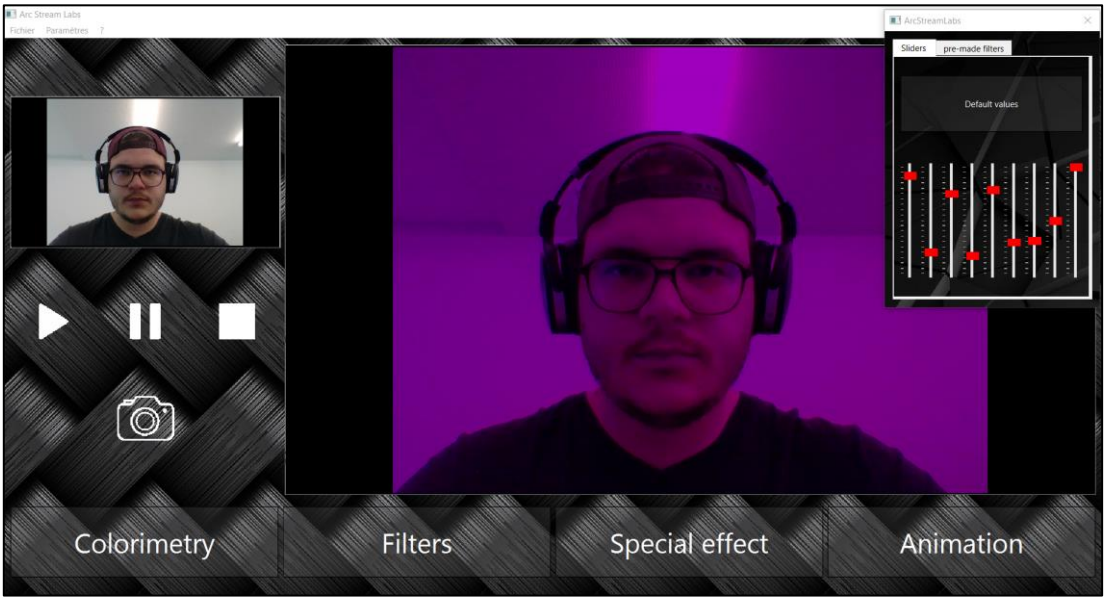

Figure 16 - Exemple colorimétrie 2 (violet)

Les deux premières captures illustrent la colorimétrie, *Red Filters* dans le premier cas ainsi qu'une approche manuelle dans le deuxième.

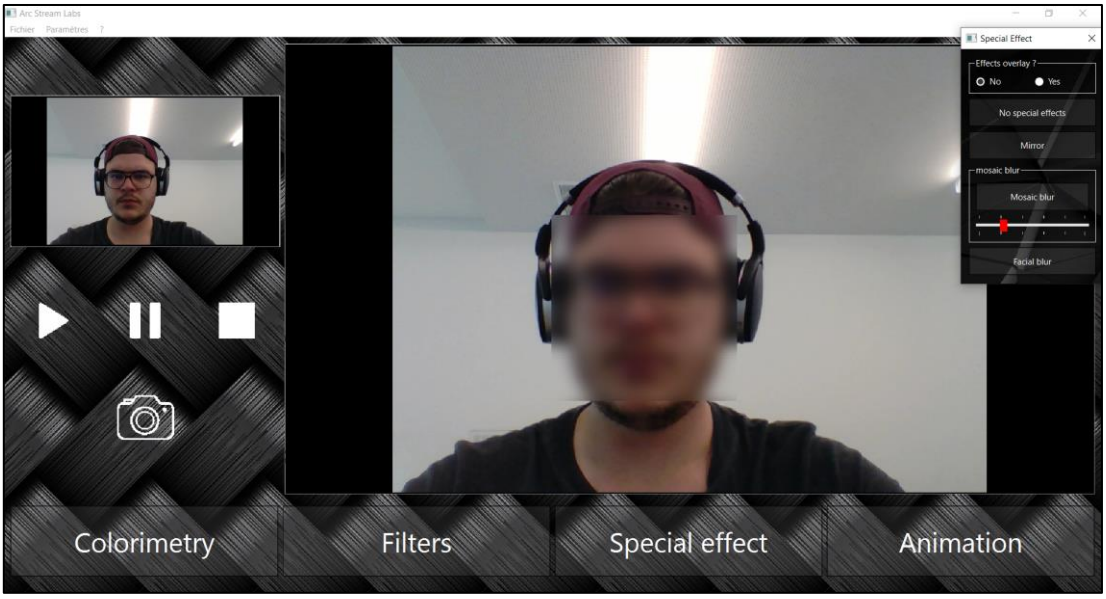

Figure 17 - Exemple, floutage facial

La capture ci-dessus illustre le *Facial blur*.

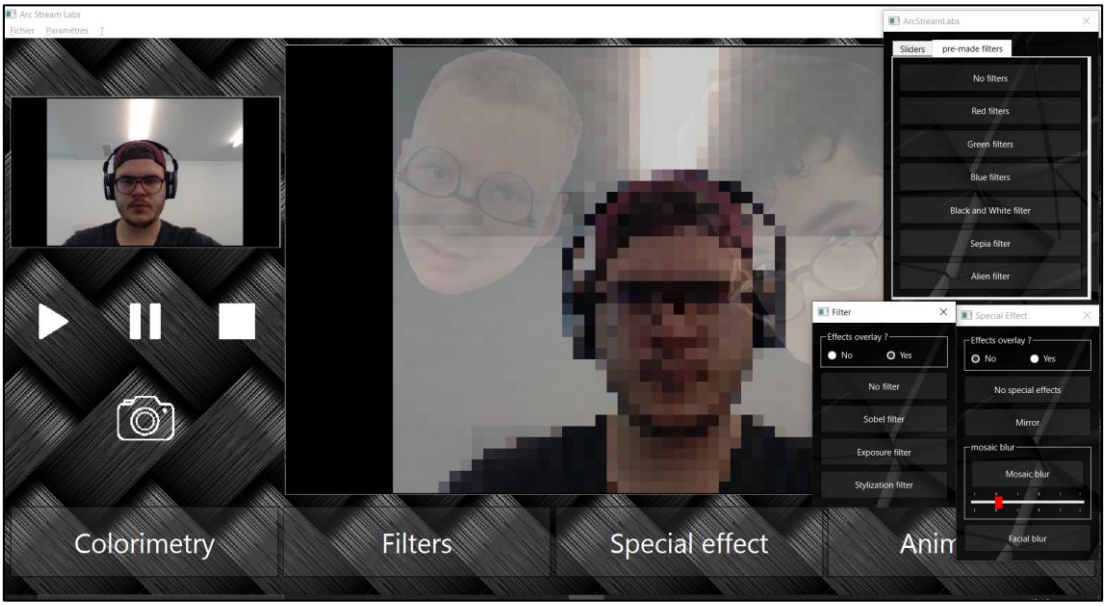

Figure 18 : Exemple, filtre « exposure » et mosaic blur

Ci-dessus, nous avons un mix du filtre *exposure* ainsi que d'un *Mosaic blur*.

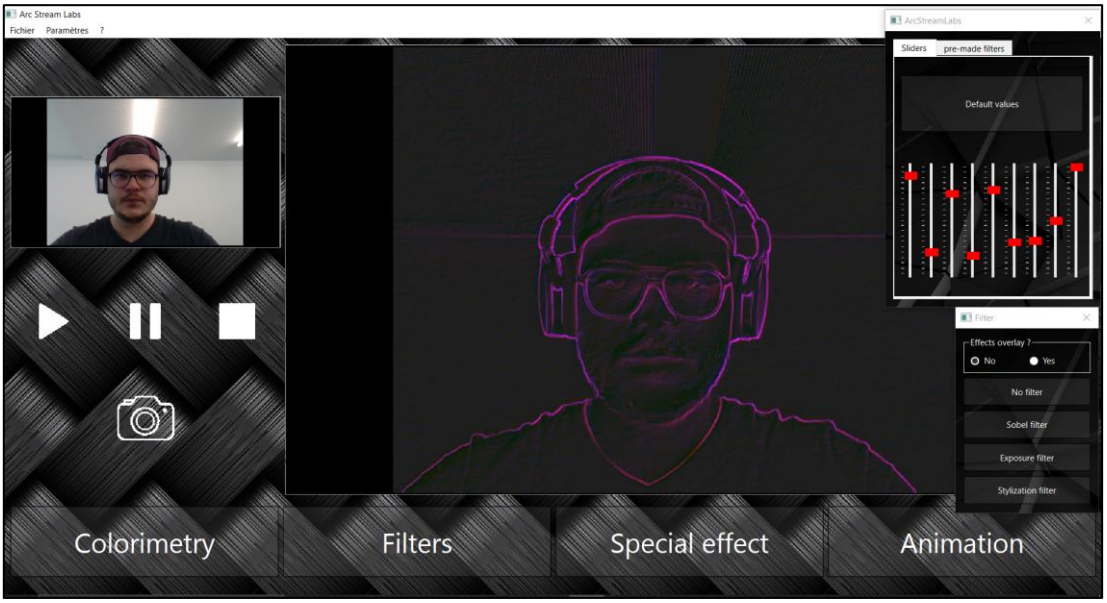

Figure 19 - Exemple colorimétrie (violet) et filtre « sobel »

Cette dernière capture illustre le mélange entre une colorimétrie personnalisée ainsi que le filtre *sobel*.

7.3 - Filtres

Figure 20 - Fenêtre filtres

Effects overlay ou superposition des effets en français permet à l'utilisateur de choisir s'il souhaite ou non combiner plusieurs filtres ensemble.

7.4 - Effets spéciaux

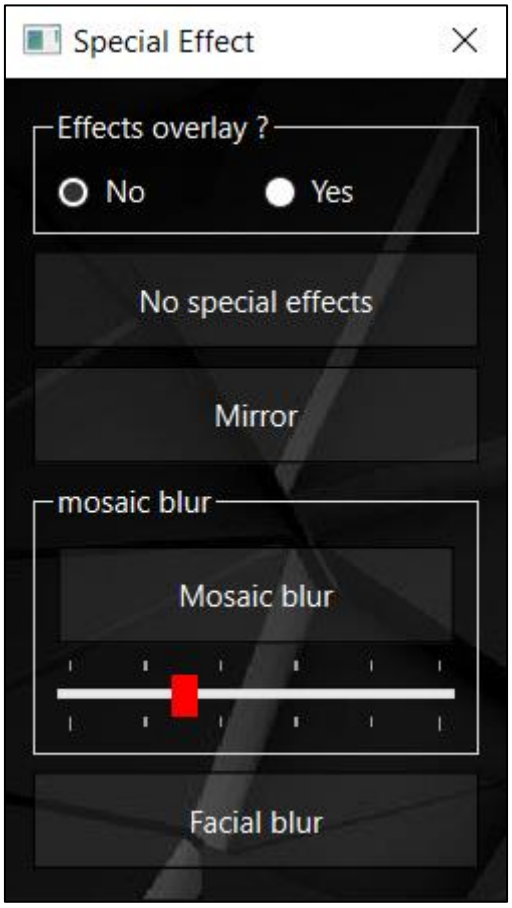

Figure 21 - Fenêtre effets spéciaux

7.5 - Animations

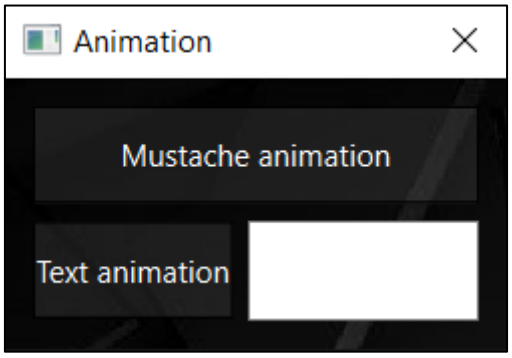

Figure 22 - Fenêtre animations

7.6 - Undo / Redo

Afin de montrer le fonctionnement du *undo-redo* voici une première capture lors de laquelle le *filtre sobel* est activé.

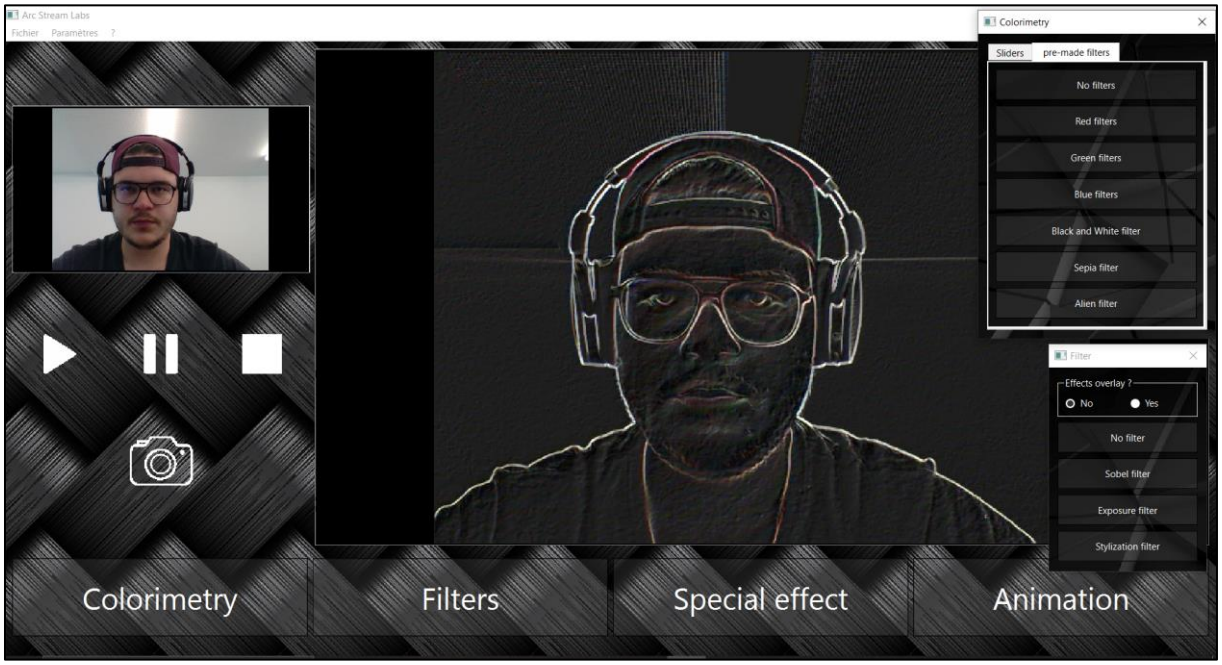

Figure 23 - Undo-redo sobel initial

Voici une deuxième capture sur laquelle nous avons ajouté le paramètre de colorimétrie rouge.

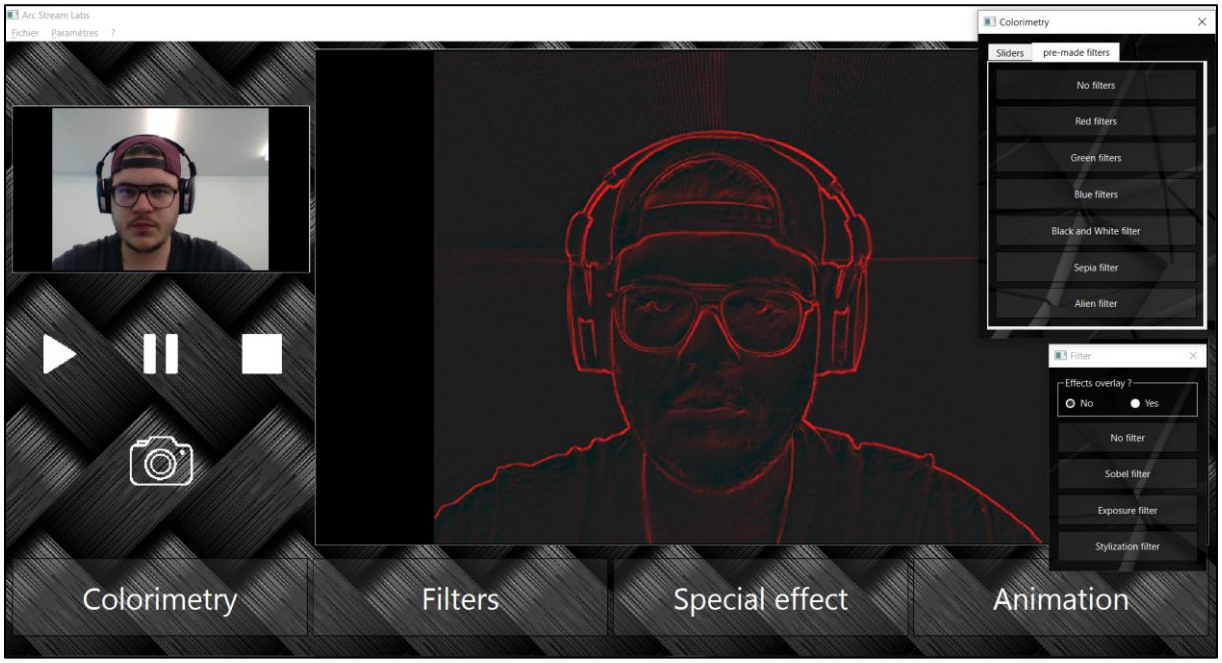

Figure 24 - Undo-redo sobel + rouge

Il suffit maintenant d'un simple CTRL + Z pour revenir à la configuration précédente qui ne contenait pas le filtre rouge.

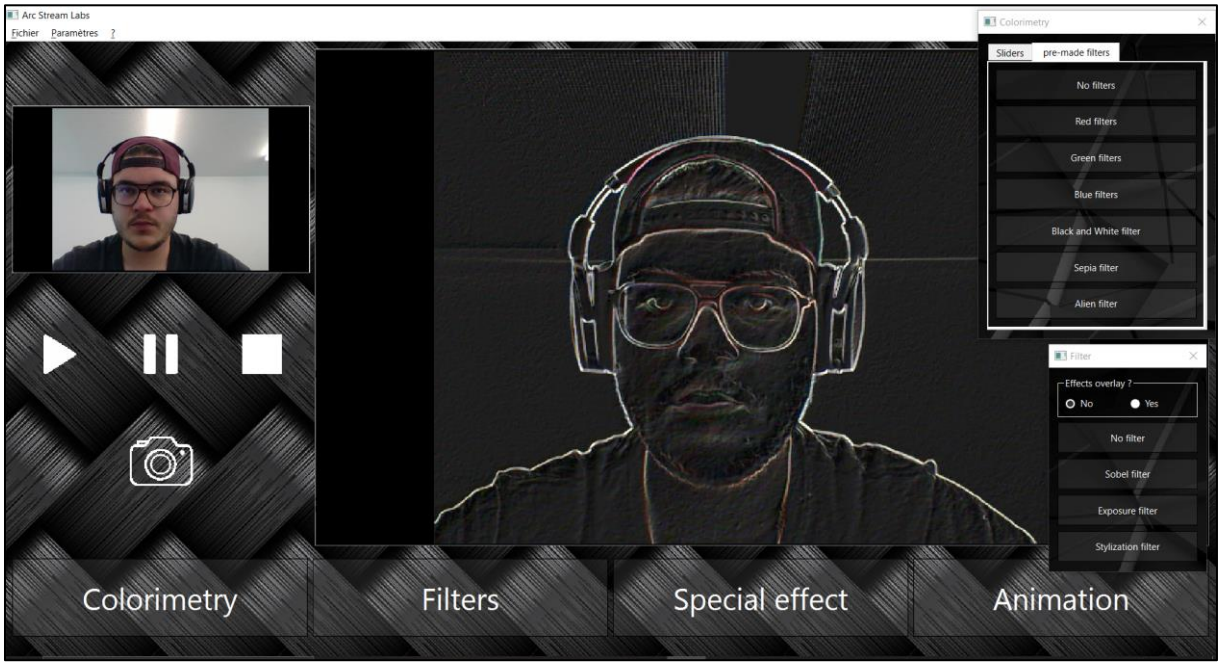

Figure 25 - Undo-redo undo rouge

Mais il est aussi possible à l'aide d'un CTRL + Y de retourner en avant dans les actions et de restaurer le filtre rouge.

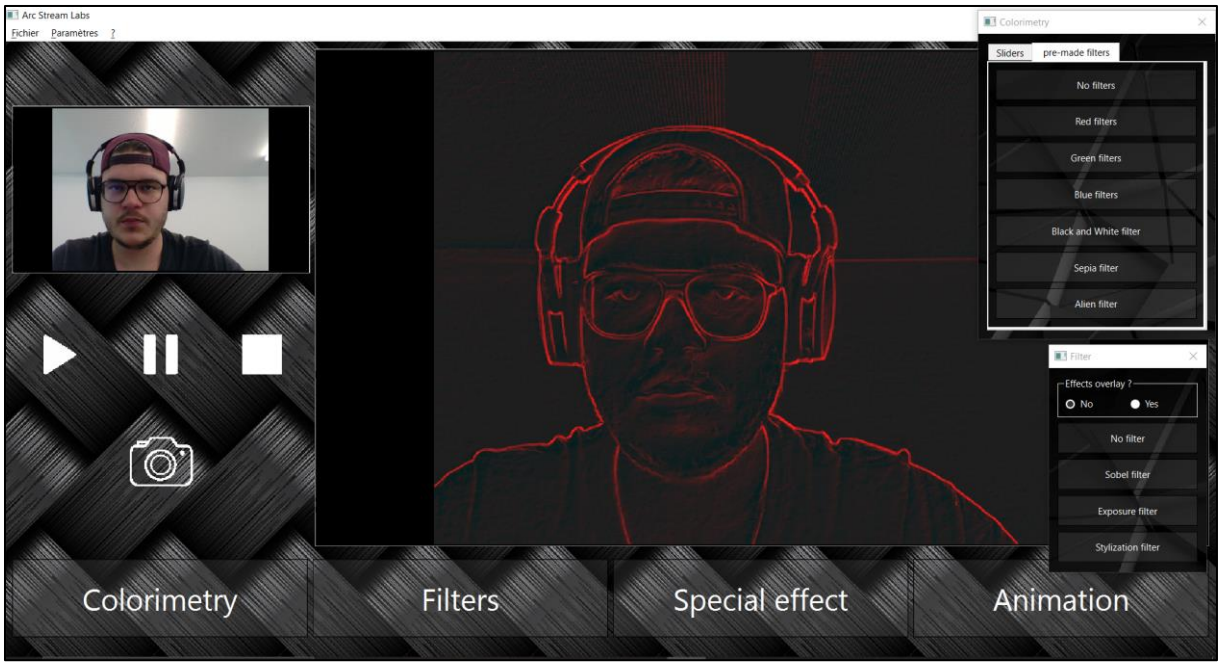

Figure 26 - Undo-redo redo rouge

7.7 - Exportation / Importation des paramètres

Il est possible de sauvegarder les configurations de colorimétrie en les exportant par le biais de la barre des menus, sous la rubrique paramètres ou avec un simple CTRL + S.

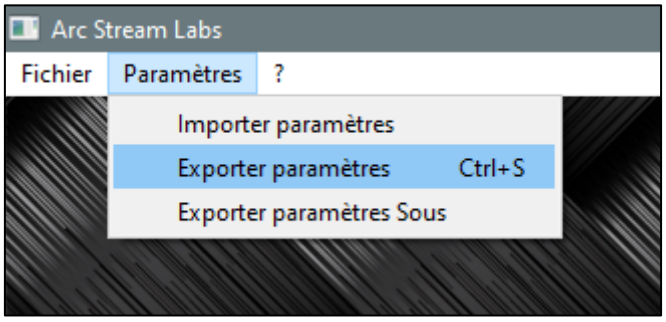

Figure 27 - Exportation des paramètres

Cette Action nous redirigera directement sur une boîte de dialogue qui permet d'enregistrer un fichier de dessin au format **.asls* pour *ArcStreamLabs Settings*.

Il aussi possible d'importer des paramètres de la même manière que l'exportation avec l'action « Importer paramètres » de la capture d'écran ci-dessus. Cette action ouvrira une boîte de dialogue qui nous permettra de choisir le fichier à importer.

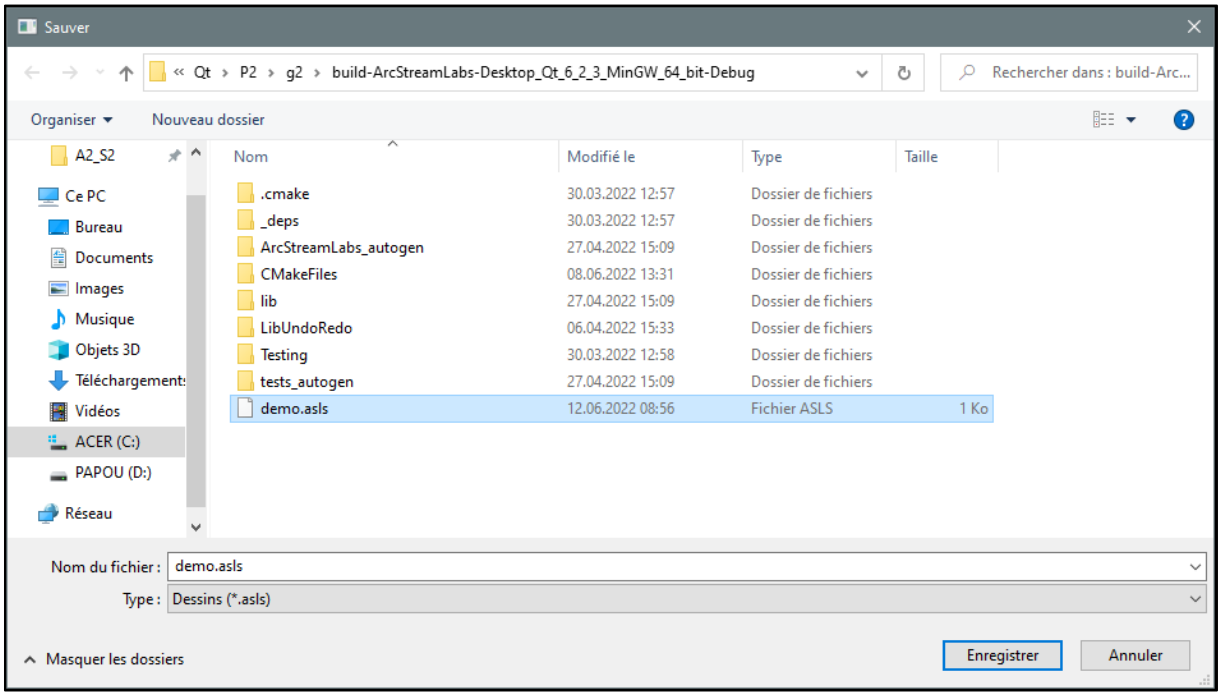

Figure 28 - Importation des paramètres

8 - Limitations et perspectives

8.1 - Ajout de nouveaux filtres / effets

Open CV offre des possibilités quasiment infinies quant à l'ajout de filtres, malheureusement c'est le manque de temps qui nous a empêché d'en ajouter plus. Cependant, notre implémentation a été faite de telle sorte que l'ajout de nouveaux filtres reste simple. Par analogie, on peut comparer l'ajout d'un filtre à l'ajout d'une fonction. Il est à noter qu'une certaine rigueur est nécessaire afin de garder le *undo-redo* fonctionnel.

8.2 - Étendre l'exportation / importation des paramètres

À l'heure actuelle, nous ne pouvons importer et exporter que les paramètres de colorimétrie. En effet, ces paramètres ne sont qu'une simple matrice, il est donc facile de les écrire dans un fichier et de les lire par la suite.

Il n'en va pas de même pour les effets spéciaux, les filtres et les animations qui sont des fonctions à effectuer sur l'image. C'est pourquoi ces paramètres ne sont pas encore enregistrés lors de l'exportation.

Une manière de sauvegarder ces paramètres serait de renseigner l'état de chacune de ces fonctions à l'aide d'une sorte de booléen. Nous pourrions ainsi savoir quelles fonctions étaient actives lors de l'exportation. Par la suite, il nous suffirait donc d'utiliser ces fonctions pour revenir au même résultat.

8.3 - Amélioration des modèles de détection

Les modèles de détection fournis par OpenCV ont été entraînés avec environ 7000 échantillons. Malgré ça, nous avons constaté un manque certain de fiabilité de ces modèles. Nous souhaitions donc changer de modèles pour mieux détecter les visages et y appliquer des filtres supplémentaires.

Nous nous sommes renseignés sur diverses librairies et projets qui avaient réalisé de bons modèles parmi lesquels se trouvait MediaPipe[6]. Malheureusement, nous avons rencontré des difficultés lors de l'installation et de la configuration. Nous avons donc pris la décision de ne pas l'utiliser.

Cependant, au vu des possibilités offertes par cette librairie, nous pensons qu'il aurait été intéressant de l'utiliser afin d'améliorer les détections faciales déjà présentes dans notre projet et pourquoi ne pas ajouter d'autres filtres liés à la détection faciale.

Voici un petit exemple de la précision qu'offre MediaPipe au niveau de la détection faciale.

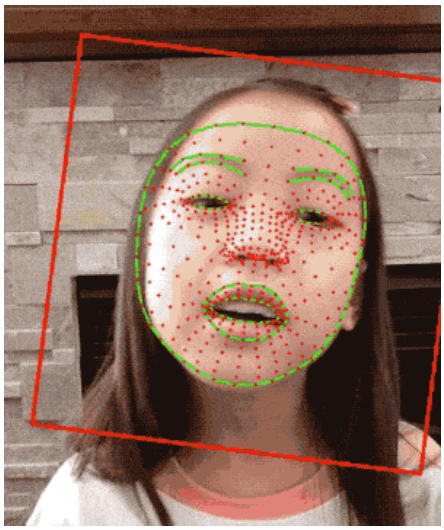

Figure 29 - GIF face mesh de MediaPipe

[6] MediaPipe, url : https://mediapipe.dev/

9 - Conclusion

Comme la librairie OpenCV a apporté plus de difficulté que prévu à être prise en main, les objectifs initiaux ont dû être revus. Ces nouveaux objectifs, détaillés dans l'analyse, ont tous été atteints. De manière similaire, les objectifs secondaires ont également subi leurs lots de changement. Cependant, une majorité d'entre eux ont été réalisés.

Nous avons aussi réussi à implémenter nos propres versions du buffer circulaire ainsi que du undo-redo. Bien que nous nous soyons basés sur diverses sources notamment pour le principe, l'implémentation reste la nôtre.

Les différents filtres ou effets appliqués, dérivent également de sources externes, qui ont été adaptées pour notre utilisation.

Concernant les graphismes, nous avons pris le parti de créer nos styles afin de pouvoir obtenir un rendu uniforme et unique.

10 - Annexes

10.1 - Guide utilisateur

Vous pouvez retrouver notre guide utilisateur sur notre wiki avec le lien suivant : lien sur le guide utilisateur.

10.2 - Cahier des charges

Vous pouvez retrouver notre cahier des charges sur notre wiki avec le lien suivant : lien sur le cahier des charges.

10.3 - Table des illustrations

Figure 1 - Exemple de traitement de flux 5
Figure 2 - Maquette des paramètres 6
Figure 3 - Maquette des animations 7
Figure 4 - Maquette de l'application 11
Figure 5 - Diagramme de séquence sur les threads du buffer circulaire 12
Figure 6 - Diagramme de séquence sur l'écriture/lecture du buffer circulaire 13
Figure 7 - Diagramme de classe undo-redo 14
Figure 8 - Diagramme de classe buffer circulaire 14
Figure 9 - Structure des fichiers 16
Figure10 - Fenêtre principale, répartition horizontale 18
Figure 11 - Buffer circulaire, structure des fichiers 19
Figure 12 - Exemple de floutage en mosaïque 25
Figure 13 - Fenêtre principale 29
Figure 14 - Fenêtre colorimétrie, onglets "Sliders" et "Pre-made filters" 30
Figure 15 - Exemple colorimétrie 1 (rouge) 30
Figure 16 - Exemple colorimétrie 2 (violet) 31
Figure 17 - Exemple, floutage facial 31
Figure 18 : Exemple, filtre « exposure » et mosaic blur 32
Figure 19 - Exemple colorimétrie (violet) et filtre « sobel » 32
Figure 20 - Fenêtre filtres 33
Figure 21 - Fenêtre effets spéciaux 34
Figure 22 - Fenêtre animations 34
Figure 23 - Undo-redo sobel initial 35
Figure 24 - Undo-redo sobel + rouge 35
Figure 25 - Undo-redo undo rouge 36
Figure 26 - Undo-redo redo rouge 36
Figure 27 - Exportation des paramètres 37
Figure 28 - Importation des paramètres 37
Figure 29 - GIF face mesh de MediaPipe 39

10.4 - Bibliographies et références

10.4.1 - Sites Web

OpenCV : https://opencv.org/

Qt Camera Overview : https://doc.qt.io/qt-6/cameraoverview.html

OpenCV with Qt : https://wiki.qt.io/OpenCV_with_Qt

How to setup Qt and openCV on Windows : https://wiki.qt.io/How_to_setup_Qt_and_openCV_on_Windows

Floyd–Steinberg dithering : https://en.wikipedia.org/wiki/Floyd%E2%80%93Steinberg_dithering

OpenCV C++ installation on Windows with MinGW : https://medium.com/csmadeeasy/opencv-c-installation-on-windows-with-mingw-c0fc1499f39

circular buffer video recording : https://camera-sdk.com/p_6554-how-to-implement-circular-buffer-video-recording-in-c-sharp.html

Creating a Circular Buffer : https://embeddedartistry.com/blog/2017/05/17/creating-a-circular-buffer-in-c-and-c/

Changing the contrast and brightness of an image! : https://docs.opencv.org/4.x/d3/dc1/tutorial_basic_linear_transform.html

Undo/redo with Command Pattern : https://gernotklingler.com/blog/implementing-undoredo-with-the-command-pattern/

Implement Undo and Redo features of a Text Editor : https://www.geeksforgeeks.org/implement-undo-and-redo-features-of-a-text-editor/

Command Pattern : https://www.javatpoint.com/command-pattern

Doxygen : https://www.doxygen.nl/download.html

Image Processing in OpenCV : https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html

Getting Started with OpenCV : https://learnopencv.com/getting-started-with-opencv/

LearnOpenCV : https://github.com/spmallick/learnopencv/blob/master/README.md?ck_subscriber_id=1673430036

Making your own linear filters! : https://docs.opencv.org/4.x/d4/dbd/tutorial_filter_2d.html

Blur and anonymize faces with OpenCV and Python : https://pyimagesearch.com/2020/04/06/blur-and-anonymize-faces-with-opencv-and-python/