

Projet de Programmation

Bibliothèque Python pour la création et la dégradation
d'images de documents pour une utilisation en
deep learning

Lilian BELLINI
Arthur HERMAN

Guillaume FLORET
Quentin LAMU

4 avril 2019

<https://services.emi.u-bordeaux.fr/projet/git/pdp-bfhl>

Sommaire

1	Introduction	6
2	Etude de l'existant	7
2.1	Présentation de DocCreator	7
2.1.1	Dégradation de documents	7
2.1.1.1	Dégradation de caractères par niveaux de gris	7
2.1.1.2	Transparence recto-verso	8
2.1.1.3	Effet de flou	8
2.1.1.4	Dégradation par trous	9
2.1.1.5	Caractères fantômes	9
2.1.1.6	Effet d'ombre	10
2.1.1.7	Déformation 3D	11
2.1.2	Génération de document	11
2.2	Etat de l'art	13
2.2.1	Bibliothèques de <i>data augmentation</i>	13
2.2.2	Wrapper Python	14
2.3	Interface en ligne de commande	14
3	Analyse des besoins	15
3.1	Besoins fonctionnels	15
3.1.1	Bibliothèque Python	15
3.1.1.1	Dégradation d'image	15
3.1.1.1.1	GrayscaleCharsDegradation	15
3.1.1.1.2	BleedThrough	16
3.1.1.1.3	BlurFilter	16
3.1.1.1.4	HoleDegradation	16
3.1.1.1.5	PhantomCharacter	16
3.1.1.1.6	ShadowBinding	17
3.1.1.1.7	Distortion3DModel	17
3.1.1.2	Génération de document	17
3.1.2	Interface en ligne de commande	18
3.1.2.1	Génération semi-synthétique	18
3.1.2.2	Génération synthétique	18
3.1.2.3	Fichier de configuration	18

3.1.3	Installation	19
3.1.3.1	Utilisation dans un script Python	19
3.1.3.2	Packaging	19
3.2	Besoins non-fonctionnels	19
3.2.1	Refactoring C++	19
3.2.1.1	Génération de document	19
3.2.1.2	<i>Distortion3DModel</i>	19
3.2.1.3	Gestion des chemins de <i>PhantomCharacter</i>	20
3.2.1.4	Dégradations avec de l'aléatoire	20
3.2.2	Wrapping Python	20
3.2.3	Portabilité	20
4	Description technique	21
4.1	Travail effectué	21
4.1.1	Récapitulatif	21
4.1.2	Modifications par rapport au code original	22
4.2	Organisation	23
4.2.1	Arborescence	23
4.2.2	Architecture	24
4.3	Implémentation	25
4.3.1	Wrapper Python	25
4.3.1.1	Dégradations	25
4.3.1.1.1	Interface	25
4.3.1.1.2	Fichiers intermédiaires	27
4.3.1.1.3	<i>Distortion3DModel</i>	29
4.3.1.2	Génération	29
4.3.1.2.1	Fonctions	29
4.3.1.2.2	Implémentation	30
4.3.2	Refactoring C++	31
4.3.2.1	Génération de document	31
4.3.2.1.1	Déplacements & Copies	31
4.3.2.1.2	Changements des paramètres	31
4.3.2.1.3	Changements du fonctionnement	31
4.3.2.2	Gestion des chemins de <i>PhantomCharacter</i>	31
4.3.2.3	Dégradations avec de l'aléatoire	32
4.3.3	Interface en ligne de commande	32
4.3.3.1	Principe	33
4.3.3.2	Structure	34
4.3.3.3	Interpréteur	35
4.3.3.4	Contrôleur	36
4.3.3.5	Fichier de configuration	36
4.3.4	Installation via <i>pip</i>	37
4.3.5	Portabilité	37
4.3.5.1	Version de Python	37

4.3.5.2	Système d'exploitation	38
5	Analyse du fonctionnement	39
5.1	Fonctionnement	39
5.1.1	Dégradation d'image	39
5.1.1.1	Utilisation	39
5.1.1.2	Limites & Perspectives	40
5.1.2	Génération de document	41
5.1.2.1	Code C++ refactoré	41
5.1.2.1.1	Utilisation	41
5.1.2.1.2	Limites & Perspectives	41
5.1.2.2	Wrapper Python	41
5.1.2.2.1	Utilisation	41
5.1.2.2.2	Limites & Perspectives	43
5.1.3	Interface en ligne de commande	44
5.1.3.1	Langage & Utilisation	44
5.1.3.1.1	Cycle de dégradation	44
5.1.3.1.2	Fichier de configuration	44
5.1.3.2	Limites & Perspectives	45
5.1.4	Installation via <i>pip</i>	45
5.1.4.1	Utilisation	45
5.1.4.2	Limites & Perspectives	46
5.1.5	Utilisation en <i>data augmentation</i>	46
5.2	Tests	47
5.2.1	Détail	47
5.2.1.1	Wrapper	47
5.2.1.1.1	Fonctionnement	47
5.2.1.1.2	Test de dégradation	47
5.2.1.1.3	Fonctions intermédiaires	48
5.2.1.2	CLI	48
5.2.2	Limites & Perspectives	49
5.2.3	Résultats	49
5.2.3.1	Lancement	49
5.2.3.2	Conditions	50
5.2.3.3	Résultats & Temps d'exécution	50
6	Conclusion	52
	Bibliographie	53
A	Informations pratiques	55

B	Compilation & Dépendances	56
B.1	Dépendances	56
B.1.1	DocCreator	56
B.1.2	Dépendances ajoutées	56
B.1.2.1	Bibliothèque Python	56
B.1.2.2	Interface en ligne de commande	57
B.1.2.3	Tests	57
B.2	Compilation	57
B.3	Installation	57
B.3.1	A la compilation	57
B.3.2	Installation via <i>pip</i>	57
C	Pistes sur <i>Distortion3DModel</i>	58
C.1	Refactoring	58
C.2	OpenGL ES	58
C.3	EGL	58
D	Organisation	60
D.1	Diagrammes de Gantt	60
D.2	Outils utilisés	63

Partie 1

Introduction

DocCreator[1] est un logiciel développé au LaBRI¹ permettant la création de documents synthétiques et leur dégradation. Les clients pour ce projet sont M. Nicholas Journet et M. Boris Mansencal.

Le but ici est de générer des documents ainsi que leur version dégradée pour pouvoir réaliser de la *data augmentation* : entraîner des programmes de deep learning (ou plus généralement de machine learning), en donnant d'une part une version de référence en bon état, et plusieurs autres, abîmées. Des programmes ainsi entraînés seront donc, à terme, capable de reconnaître et déchiffrer de vieux documents en mauvais état. L'intérêt ici est d'éviter de devoir trouver des documents existants déjà dégradés en les générant artificiellement. Dans l'état actuel du logiciel, les procédures de génération et dégradation d'images s'effectuent via une interface graphique. On cherche à minimiser le temps humain pris par cette démarche en automatisant la génération et l'application des dégradations.

Pour cela, les fonctions de DocCreator devront pouvoir être utilisées dans un script. D'autre part, les principaux frameworks de deep learning (tels que *Keras*[2], *TensorFlow*[3], *PyTorch*[4]) ainsi que d'autres bibliothèques de *data augmentation* telles que *ocrodeg*[5], *Augmentor*[6] ou encore *imgaug*[7] (toutes trois étant des outils proposant diverses dégradations d'images, qui pourraient donc être utilisés en complément de DocCreator) étant en Python, la bibliothèque DocCreator contenant les fonctions de génération et de dégradation de documents doit également être en Python.

Les fonctions de génération de documents ainsi que les diverses dégradations qu'il est possible d'y apporter (trous, ombres, tâches d'impression, lettres effacées, déformations du document, flou, ou encore effets de transparence de pages recto-verso) seront scriptables via cette bibliothèque. Pour pouvoir utiliser DocCreator indépendamment, on disposera également d'une interface en ligne de commande (CLI) qui devra rester simple d'utilisation.

1. Laboratoire Bordelais de Recherche en Informatique

Partie 2

Etude de l'existant

2.1 Présentation de DocCreator

La version de DocCreator telle que développée par le LaBRI met à disposition deux fonctionnalités principales : dégradation de documents existants ou génération de documents puis dégradation.

2.1.1 Dégradation de documents

La génération semi-synthétique de document consiste, à partir d'un document donné en entrée, à produire en sortie une ou plusieurs images dégradées de ce même document. Pour cela, 7 méthodes de dégradations sont disponibles.

2.1.1.1 Dégradation de caractères par niveaux de gris

GrayscaleCharsDegradation ajoute des points blancs sur les caractères d'un document et des points noirs autour. Cet effet permet de simuler la détérioration de l'encre au fil du temps. L'effet peut être appliqué avec différents niveaux d'intensité, affectant la lisibilité du contenu.



FIGURE 2.1 – Dégradation de caractère par niveaux de gris

2.1.1.2 Transparence recto-verso

BleedThrough simule un effet de transparence entre deux faces d'un document recto-verso. Le texte du second document (verso) apparaît légèrement sur le premier (recto).

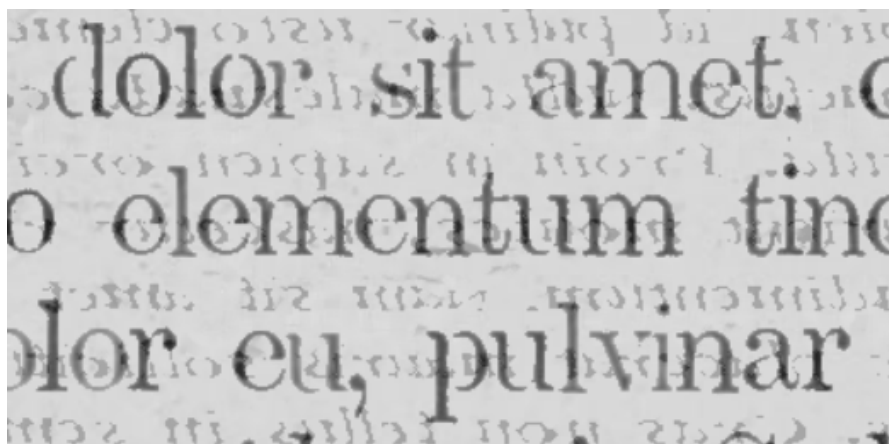


FIGURE 2.2 – Transparence recto-verso

2.1.1.3 Effet de flou

BlurFilter ajoute un effet de flou à tout ou partie d'une image, en offrant la possibilité de régler son intensité. A des niveaux d'intensité élevés, il n'est quasiment plus possible de lire le document.

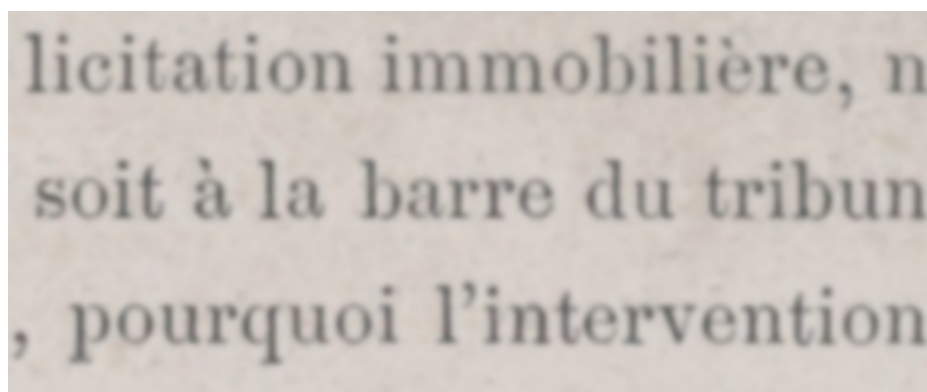


FIGURE 2.3 – Effet de flou, intensité moyenne

2.1.1.4 Dégradation par trous

HoleDegradation simule la présence de trous sur le document. La forme du trou (ou *pattern*) peut être modifiée. Les trous peuvent être appliqués sur les différents bords ou coins de l'image (ou encore au centre). Il est également possible d'appliquer une couleur au trou, ou encore de faire apparaître un second document (comme une seconde page derrière une première, trouée).

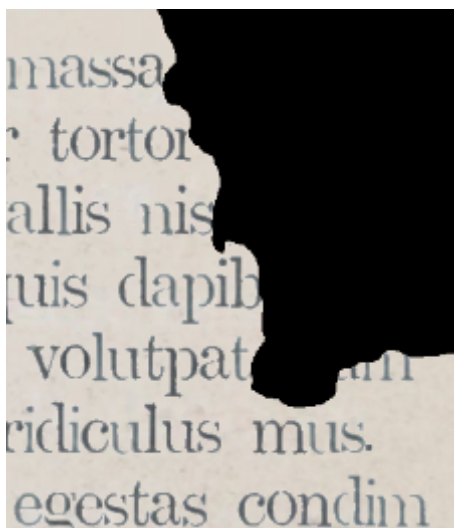


FIGURE 2.4 – Trous appliqués dans les coins d'un document

2.1.1.5 Caractères fantômes

PhantomCharacter ajoute des caractères "fantômes" à l'image. Il s'agit de petites lignes verticales simulant l'usure d'une machine d'impression. Il est possible de choisir la fréquence d'apparition de ces caractères.



FIGURE 2.5 – Caractères fantômes

2.1.1.6 Effet d'ombre

ShadowBinding ajoute une ombre sur le bord d'un document (au choix entre les bords droit, gauche, en haut ou en bas). Il s'agit d'imiter la photocopie d'un livre qui rendrait l'image plus sombre au niveau de la reliure (pour imiter la courbure de la page). On peut régler l'intensité de l'ombre, sa taille, ou encore son angle.

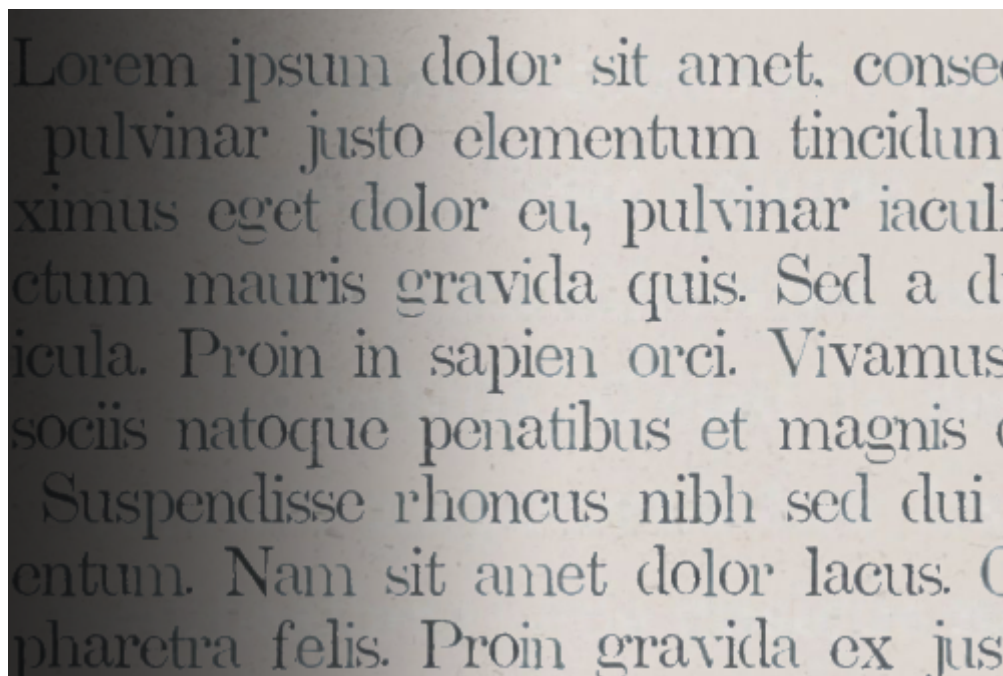


FIGURE 2.6 – Ombre simulant la courbure d'une page d'un livre photocopiée

2.1.1.7 Déformation 3D

Distortion3DModel applique l'image sur un modèle 3D afin de la déformer, permettant ainsi de simuler différents effets, comme une page froissée.

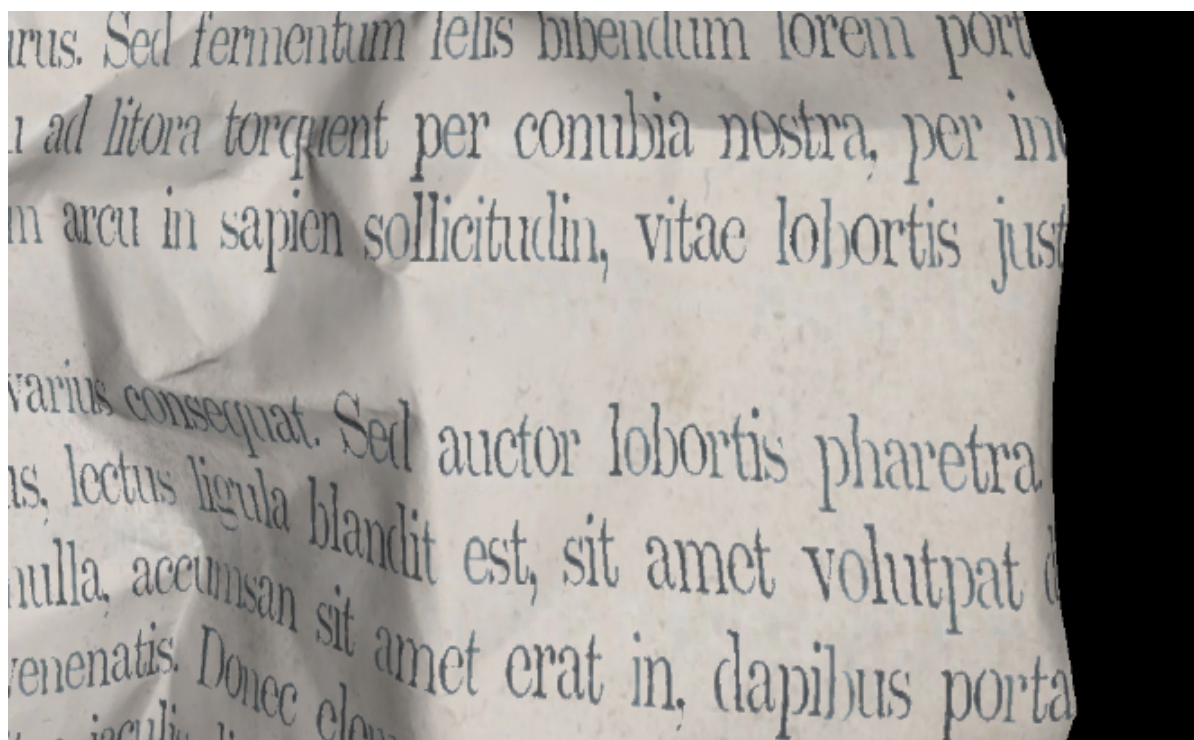


FIGURE 2.7 – Déformation 3D

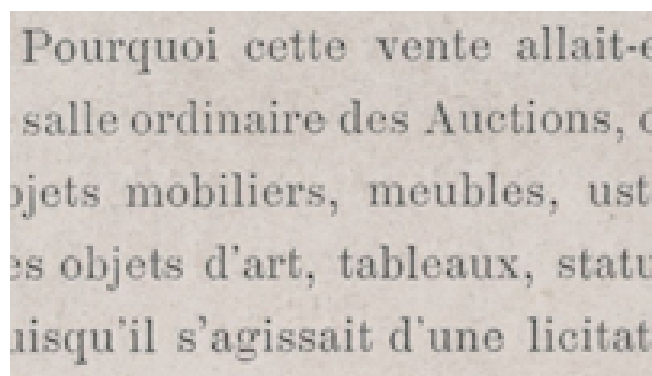
2.1.2 Génération de document

La génération synthétique permet de dégrader des documents, mais ne nécessite pas d'en donner en entrée. Le but ici est justement de générer artificiellement des documents. Cette génération est effectuée à partir de différents éléments :

- un extrait de texte (encodé en UTF-8)
- une police d'écriture
- un arrière plan
- la taille des marges désirées
- le nombre de lignes et de colonnes
- l'interligne
- l'espacement entre les mots
- la taille de l'image qui sera générée

Ensuite, il suffit de choisir les dégradations à appliquer, on aura alors en sortie une image par dégradation sélectionnée. Les polices d'écritures utilisées ne sont pas des formats standards. Elles sont extraites de documents anciens, avec plusieurs instances différentes pour chaque caractère. Elles sont contenues dans le dossier */data* de DocCreator :

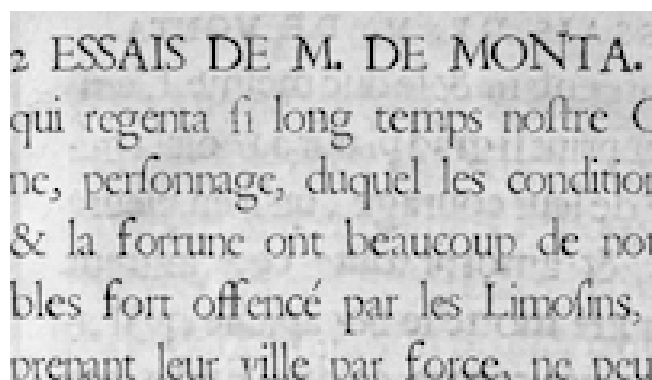
— Jules Vernes (*JulesVerne.of*) :



Pourquoi cette vente allait-elle
salle ordinaire des Auctions, c
objets mobiliers, meubles, ust
es objets d'art, tableaux, statu
puisqu'il s'agissait d'une licitat

FIGURE 2.8 – Police "Jules Vernes"

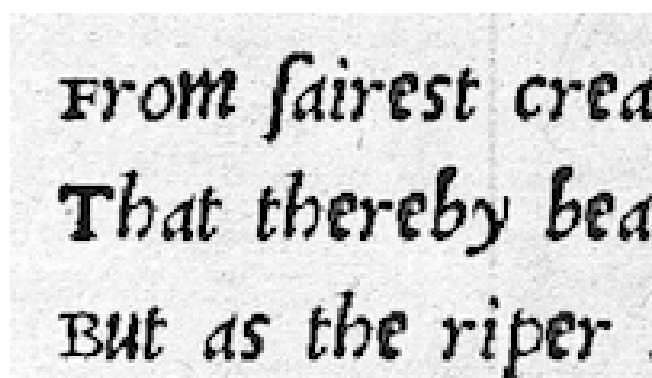
— Montaigne (*montaigne.of*) :



2 ESSAIS DE M. DE MONTA.
qui regenta si long temps nostre C
ne, personnage, duquel les condition
& la fortune ont beaucoup de nob
bles fort offensé par les Limosins,
prenant leur ville par force, ne peu

FIGURE 2.9 – Police "Montaigne"

— William Shakespeare Sonnet (*BMTcomplete.of*) :



From fairest crea
That thereby bea
But as the riper

FIGURE 2.10 – Police "William Shakespeare Sonnet"

— Handwritten (*manuscript.of*) :

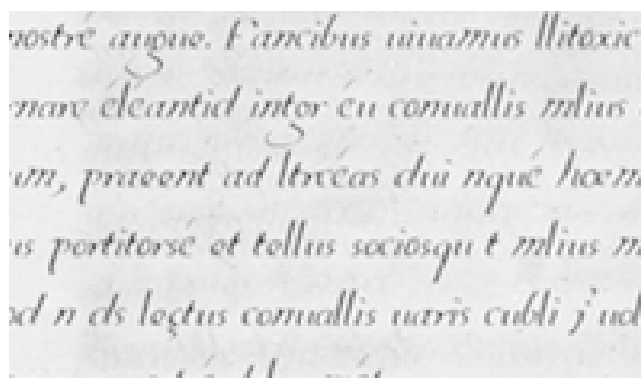


FIGURE 2.11 – Police "Handwritten"

— Manto (*manto.of*) :

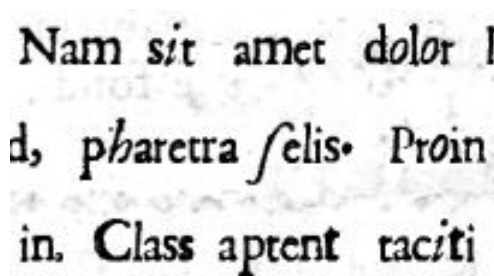


FIGURE 2.12 – Police "Manto"

2.2 Etat de l'art

2.2.1 Bibliothèques de *data augmentation*

La bibliothèque Python de DocCreator pourra être amenée à être utilisée en complément d'autres bibliothèques Python de *data augmentation*. Parmi elles, on peut citer ocodeg, Augmentor ou imgaug qui peuvent servir de modèle sur le résultat souhaité pour la dégradation de documents. Ces bibliothèques partagent des caractéristiques communes que l'on veut retrouver dans DocCreator. Le schéma d'utilisation est le suivant : on donne en entrée une image ou un ensemble d'image, ou applique une dégradation ou un ensemble de dégradations, puis on renvoie les images correspondantes en sortie. Le fichier d'origine n'est pas modifié. D'autre part, il suffit d'importer ces bibliothèques pour pouvoir les utiliser dans un script Python, cela permet donc d'automatiser les démarches de dégradation, notamment si l'on souhaite dégrader un grand nombre d'images

2.2.2 Wrapper Python

Pour pouvoir profiter des fonctions écrites en C++ dans un code Python, il est nécessaire de réaliser un *wrapper* autour du code de DocCreator. Pour cela, différents outils existent, tels que SWIG[8]. SWIG permet de générer à la compilation des interfaces entre des programmes en langage C ou C++ vers d'autres langages, notamment des langages de script, tels que Perl, Python, Ruby ou encore Tcl. A partir de fonctions définies dans un *header* (en C++, un .hpp), SWIG va générer les fonctions Python (dans notre cas) permettant d'appeler les fonctions du code C++. Cela signifie cependant que, bien que l'on puisse profiter des capacités du C++ dans un script Python, il sera toujours nécessaire de compiler DocCreator pour pouvoir utiliser une bibliothèque compatible sur l'OS utilisé. On ne pourra donc pas bénéficier pleinement de la portabilité du langage Python.

2.3 Interface en ligne de commande

Pour réaliser des interfaces en ligne de commande, différents outils existent, tels que le module par défaut de gestion d'argument de Python, à savoir *argparse*[9]. Cet outil permet de spécifier les arguments voulus qui seront ensuite récupérés via ligne de commande, puis utilisés librement dans le script. L'ajout de descriptions, de messages d'aide ou de messages d'erreurs sont également pris en charge par *argparse*, permettant ainsi de gérer les informations invalides rentrées par l'utilisateur.

Partie 3

Analyse des besoins

3.1 Besoins fonctionnels

3.1.1 Bibliothèque Python

Un des objectifs du projet est de pouvoir utiliser DocCreator dans un script, qui puisse être utilisé dans des frameworks de machine learning en Python, et en complément avec des bibliothèques de dégradations d'image en Python également. Pour répondre à ce besoin, les fonctions de dégradation et de génération de document seront portées dans une bibliothèque Python. La priorité client est l'inclusion de la dégradation *GrayscaleCharsDegradationModel* dans cette bibliothèque, suivie des 5 autres dégradations, de la génération de document, puis enfin de la dernière dégradation : *Distortion3DModel*.

3.1.1.1 Dégradation d'image

La génération semi-synthétique sera utilisable via la bibliothèque Python. Cette génération permet de créer des images à partir d'un document déjà existant auquel on a appliqué différentes dégradations. Toutes les dégradations renvoient 1 image en sortie, elles sont au nombre de 7 et se caractérisent comme suit (pour une description plus illustrative des effets des dégradations, voir section 2.1.1) :

3.1.1.1.1 GrayscaleCharsDegradation Affiche les caractères de l'image comme si l'encre avait été dégradée par le temps grâce à l'ajout de points en niveaux de gris sur et autour les caractères du document. Les paramètres sont :

- 1 image en entrée : `imgIn`
- un niveau d'intensité : `level`
- le pourcentage de "independant points" : `I`, les points noirs autours des caractères
- le pourcentage de "overlapping points" : `O`, les points blancs qui chevauchent des caractères sans les couper
- le pourcentage de "disconnection points" : `D`, les points blancs qui chevauchent des caractères en les coupant en deux

La somme des pourcentages des trois types de points doit être égale à 100

3.1.1.1.2 BleedThrough Simule un effet de transparence entre les deux faces d'un même document recto-verso. Le texte du second document (verso) apparaît légèrement sur le premier (recto). Les paramètres sont :

- 2 images en entrées : `imgIn` et `img2`
- le nombre d'itérations de l'algorithme à appliquer : `nblter`, correspondant au niveau d'opacité de la transparence
- (optionnel) la position relative de la seconde image par rapport à la première : `x=0` et `y=0` (par défaut)
- (optionnel) le nombre de threads à utiliser durant l'exécution : `nbThreads=-1` (par défaut), si laissé à sa valeur par défaut, il sera choisi automatiquement

3.1.1.1.3 BlurFilter Ajoute un effet de flou à l'image. Les paramètres sont :

- 1 image en entrée : `imgIn`
- l'intensité pouvant prendre des valeurs impaire de 1 à 21 : `intensity`
- l'algorithme à utiliser : `method` :
 0. *gaussian* : convolution gaussienne (avec des moyennes pondérées en fonction de la distance)
 1. *median* : convolution par médianne
 2. *normal* : convolution par moyenne

3.1.1.1.4 HoleDegradation Simule la présence de trous sur le document. Les paramètres sont :

- 1 image en entrée : `imgIn`
- 1 image correspondant au pattern à utiliser : `pattern`
- la position du trou : `xOrigin` et `yOrigin`
- la taille du trou : `size`
- la couleur du trou `color`
- (optionnel) : 1 seconde image à faire apparaître derrière le trou : `img2=None` (par défaut)
- (optionnel) : la largeur : `width=0` (par défaut)
- (optionnel) : l'intensité : `intensity=1000` (par défaut)

3.1.1.1.5 PhantomCharacter Ajoute des caractères fantômes à l'image (lignes verticales simulant l'usure d'une machine d'impression). Les paramètres sont :

- 1 image en entrée : `imgIn`
- la fréquence :
 0. *rare* : 15%
 1. *frequent* : 40%
 2. *very frequent* : 70%

3.1.1.1.6 ShadowBinding Ajoute une ombre sur le bord d'un document pour imiter la courbure d'une page photocopiée. Les paramètres sont :

- 1 image en entrée : `imgIn`
- le bord choisi : `border` :
 - 0. haut
 - 1. droite
 - 2. bas
 - 3. gauche
- la taille : `distance`
- (optionnel) l'intensité : `intensity=0.5` (par défaut) pouvant aller de 0 à 1
- (optionnel) l'angle : `angle=30` (par défaut) pouvant aller de 0 à 90

3.1.1.1.7 Distortion3DModel Applique l'image sur un modèle 3D. Les paramètres sont :

- 1 image en entrée : `imgIn`
- le chemin menant à un répertoire qui contient les modèles 3D à appliquer
- le chemin menant aux arrières-plan que l'on souhaiterait utiliser (optionnel)

Cette dégradation est un cas particulier vis à vis des autres. En effet, les 6 premières dégradations sont rassemblées dans un même dossier : `/framework`, mais *Distortion3D* est dans le dossier `/software`. D'autre part, les fonctions de cette dégradation utilisent OpenGL[10]. Hors, étant donné que l'on souhaite utiliser la bibliothèque Python dans un script, on ne peut contraindre l'utilisateur à travailler dans un environnement avec fenêtre. En plus de devoir effectuer un déplacement de *Distortion3D* dans le code original de DocCreator, il est nécessaire de rendre ses fonctions exécutables sans serveur X. Pour ce faire, il est possible d'envisager la solution EGL[11].

EGL est une interface entre les applications de rendus (comme OpenGL) et un système de fenêtre. Cet outil est notamment connu pour pouvoir être utilisé sur des serveurs, en se dispensant d'environnement graphique. Cependant, EGL a une compatibilité restreinte à OpenGL ES, une version plus légère et plus portable d'OpenGL.

3.1.1.2 Génération de document

La génération synthétique consiste en la création d'un document avant de lui appliquer les dégradations précédemment citées. Les paramètres sont :

- le chemin du dossier contenant des fichiers texte (encodés en UTF-8) existants (optionnel, un faux-texte "lorem ipsum" peut être choisi à la place)
- le chemin du dossier contenant les polices d'écriture que l'on souhaite utiliser
- le chemin du dossier contenant les arrières-plan que l'on souhaite utiliser
- la taille des marges
- le nombre de colonnes et de lignes
- la taille de l'interligne

- l'espacement entre les mots (pourcentage de blocs vides)
- la taille de l'image à générer
- le chemin du dossier cible des images de texte
- la liste des dégradations à appliquer et leurs arguments
- le chemin du dossier cible des images dégradées

3.1.2 Interface en ligne de commande

La version Python de DocCreator pourra être utilisée via une interface en ligne de commande (CLI), ce qui permettra de directement profiter de la bibliothèque sans passer par l'écriture d'un script, elle ne nécessitera pas de connaissances en Python pour son utilisation. La CLI détectera les paramètres entrés par l'utilisateur et vérifiera leur validité, en envoyant des messages d'erreurs et d'aides associés. La priorité de cette tâche pour les clients est secondaire.

3.1.2.1 Génération semi-synthétique

Il sera possible via l'interface graphique d'appeler les différentes fonctions de dégradation d'image dans la bibliothèque Python. On pourra également choisir le nombre d'images voulues en sortie. 1 image sera générée par dégradation et par nombre d'image spécifié. Par exemple, si l'on veut appliquer *GrayscaleCharsDegradation* et *BlurFilter*, et que l'on choisit de générer 3 images à chaque fois, on aura en sortie 6 images. L'image donnée en entrée ne sera pas modifiée, de nouvelles seront créées dans le dossier de sortie spécifié. Il sera possible d'enchaîner les dégradations, en les appliquant à la suite sur un même document. On pourra donc ainsi produire une image floue, trouée, avec des caractères usés et une ombre sur la reliure par exemple.

3.1.2.2 Génération synthétique

Les fonctions de génération de documents synthétiques seront également accessibles via la CLI, en entrant les mêmes paramètres que pour les fonctions de base (voir section 3 sur les dégradations).

3.1.2.3 Fichier de configuration

Pour faciliter l'utilisation il sera possible d'utiliser des fichiers de configuration. La vérification des arguments se déroulera de la même manière que pour une utilisation classique, mais cela permettra à l'utilisateur de sauvegarder différents appels à la CLI, et de perdre moins de temps à passer des arguments.

3.1.3 Installation

3.1.3.1 Utilisation dans un script Python

Les fonctions de la bibliothèque Python DocCreator devront pouvoir être appelées comme pour n'importe quelle autre bibliothèque Python, avec :

```
import DocCreator
```

3.1.3.2 Packaging

Pour faciliter l'installation du projet, la bibliothèque Python et la CLI seront incluses dans un paquet, qui pourra être installée via *pip*. Ce besoin est de basse priorité client.

3.2 Besoins non-fonctionnels

3.2.1 Refactoring C++

Pour pouvoir constituer une bibliothèque Python, on opérera sur un sous-ensemble de DocCreator, avec un nombre minimisé de dépendances (il ne sera pas utile dans notre cas de restituer toutes les fonctionnalités liées à l'interface graphique). Le besoin système d'un refactoring du code C++ s'impose donc. Il s'agit de rassembler en un même endroit (dossier */framework* de DocCreator) toutes les fonctions de dégradation et de génération d'image. Les dégradations *GrayScaleCharsDegradation*, *BleedThrough*, *BlurFilter*, *HoleDegradation*, *PhantomCharacter* et *ShadowBinding* se trouvent déjà dans le dossier */framework*. Cela consistera donc à isoler puis à déplacer le code de la génération de document et celui de la 7e dégradation : *Distortion3DModel*.

3.2.1.1 Génération de document

La génération de document est actuellement définie dans */software*. Pour pouvoir utiliser ses fonctions uniquement avec */framework*, il est nécessaire d'isoler le code de la génération des autres fonctionnalités, comme celles de l'UI, puis de le déplacer. Le code concerné se trouve principalement dans */software/src/Degradations/ImageGenerationFromDirDialog.hpp/cpp*.

3.2.1.2 *Distortion3DModel*

La dégradation *Distortion3DModel* se trouve dans le dossier */software* du code de DocCreator, contrairement aux autres qui se trouvent dans le dossier */framework*. Avant d'inclure cette dégradation dans la bibliothèque Python, il faut donc effectuer un déplacement du code de cette dégradation pour la regrouper avec les autres. Actuellement, les fichiers qui composent *Distortion3DModel* (dans le dossier */software/src/Degradations/Distortion3DModel*) sont au nombre de 1765, répartis dans 148 dossiers ou sous-dossiers.

3.2.1.3 Gestion des chemins de *PhantomCharacter*

Dans la version de base du logiciel, la dégradation *PhantomCharacter* se limite à la bibliothèque de patterns de DocCreator et ne fonctionne pas sans initialiser les chemins, action qui se fait dans le fichier *Assistant.cpp*, lié à l'interface graphique. Cet aspect sera modifié pour que le chemin des patterns soit un paramètre de la dégradation, et que l'on puisse utiliser librement n'importe quel dossier contenant des patterns.

3.2.1.4 Dégradations avec de l'aléatoire

Pour pouvoir tester le bon fonctionnement de la bibliothèque Python, il sera utile de pouvoir effectuer des séries de tests de comparaison d'image pixel par pixel entre les images générées par les fonctions d'origine en C++ et les fonctions Python. Cependant, les dégradations *GrayScaleCharsDegradation* et *PhantomCharacter* contiennent de l'aléatoire, rendant invalide ce type de test. Dans le cadre du refactoring, un paramètre sera ajouté aux dégradations concernées permettant de fixer la seed, et ainsi supprimer temporairement l'aléatoire pour les tests.

3.2.2 Wrapping Python

Les fonctions de dégradation et de génération du code C++ doivent pouvoir être utilisées en Python. Pour cela on réalisera un wrapper Python autour du code d'origine. Il s'agit de générer une interface Python qui servira à appeler les fonctions en C++. La bibliothèque ainsi constituée sera utilisable dans des scripts Python. Le but ici étant de pouvoir l'utiliser avec les frameworks de Deep Learning communs (tels que *Keras*, *Tensorflow* ou *Pytorch*, ...) et en complément d'autres bibliothèques de dégradation d'image pour une utilisation en *data augmentation* telles que *ocrodeg*, *Augmentor* ou *imgaug* précédemment citées. Cette bibliothèque sera également utilisable dans la CLI évoquée.

3.2.3 Portabilité

La bibliothèque Python fonctionnera sur les versions de Python 2.7 et 3.5. D'autre part, elle pourra être utilisée sur Ubuntu 16.04, Ubuntu 18.04, Mac OS 10.13 et Windows 10 (dans cet ordre de priorité).

Partie 4

Description technique

4.1 Travail effectué

4.1.1 Récapitulatif

Voici un tableau récapitulatif des besoins qui ont été implémentés dans la version actuelle du projet (les tâches marquées de ✓ ont été terminées, celles marquées de ~, partiellement, et celles avec ✗ n'ont pas été effectuées ; les priorités ont été indiquées ainsi : 1 : plus haute priorité, 5 : plus basse priorité) :

Fonctionnalité	Statut	Testé	Priorité
Wrapper : <i>GrayscaleCharsDegradationModel</i>	✓	✓	1
Wrapper : <i>BleedThrough</i>	✓	✓	2
Wrapper : <i>BlurFilter</i>	✓	✓	2
Wrapper : <i>HoleDegradation</i>	✓	✓	2
Wrapper : <i>PhantomCharacter</i>	✓	✓	2
Wrapper : <i>ShadowBinding</i>	✓	✓	2
Wrapper : <i>Distortion3DModel</i>	✗	✗	5
Refactoring génération	✓	✗	3
Wrapper : Génération	✓	✗	3
Refactoring phantomPatternsPath	✓	✓	3
Refactoring aléatoire	✓	✓	2
CLI : Dégradation	✓	✓	3
CLI : Génération	✓	✗	3
CLI : Fichier de configuration	✓	✗	3
Portabilité : Linux	✓	✓	1
Portabilité : Mac	✓	✓	4
Portabilité : Windows	~	✗	4
Installation via PIP	~	~	4

4.1.2 Modifications par rapport au code original

Comme notre projet est basé sur un logiciel existant, cette section a pour but de lister les fichiers issus de notre travail, pour les distinguer des fichiers de DocCreator d'origine.

Les fichiers que nous avons créés :

- Tout le contenu de wrapper, excepté numpy.i, et le contenu de Generation_aux.cpp est constitué de fonctions copiées de software/DocCreator/src/DocCreator.cpp
- Tout le contenu de tests
- framework/src/RandomDocument/, la majorité du code vient de software/DocCreator/src/Degradations/ImageGenerationFromDirDialog.cpp
- DocCreatorPackage (depuis la racine)
- doc (depuis la racine)
- Tous les autres fichiers à la racine du dépôt

Les fichiers qui existaient mais que nous avons modifié pour intégrer notre travail ou corriger des bugs :

- CMakeLists.txt
- framework/CMakeLists.txt
- framework/src/Degradations/GrayscaleCharsDegradationModel.cpp
- framework/src/Degradations/PhantomCharacter.cpp
- software/CMakeLists.txt
- software/DocCreator/CMakeLists.txt

Les fichiers que nous avons déplacés (leur position actuelle) :

- framework/src/Utils/FontUtils.cpp
- framework/src/Utils/FontUtils.hpp
- framework/src/Utils/RandomElement.cpp
- framework/src/Utils/RandomElement.hpp
- framework/src/appconstants.h

Les autres fichiers étaient présents à l'origine et n'ont pas été modifiés.

4.2 Organisation

4.2.1 Arborescence

Notre projet se base sur le code d'origine de DocCreator. Outre quelques changements ou corrections mineures dans le dossier `/software` et le `CMakeLists.txt` à la racine de `/Doc-Creator`, la majorité de notre travail se trouve dans les dossiers `/wrapper/python` (pour la bibliothèque Python et la CLI), `/framework` (pour le refactoring de la génération de document : `/src/RandomDocument`) et `/tests` pour les tests. Le "cœur" de notre projet se présente donc ainsi :

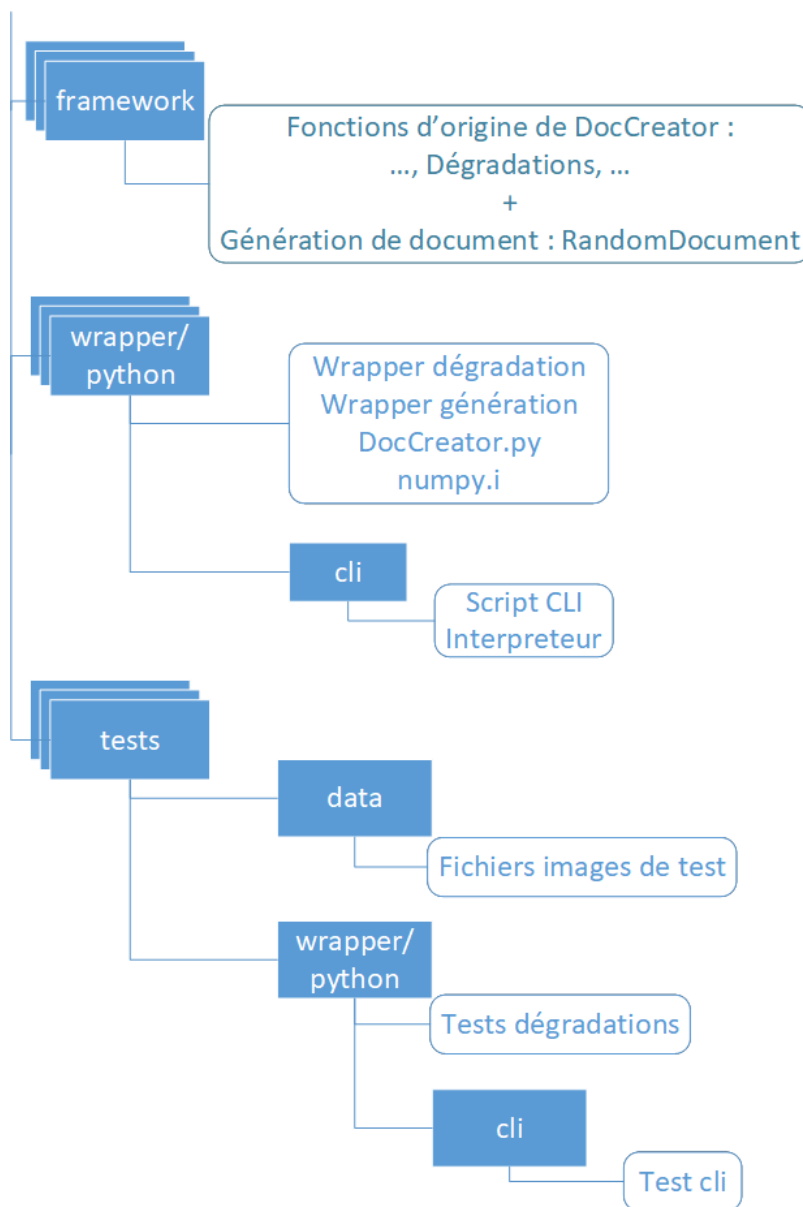


FIGURE 4.1 – Contenu des dossiers principaux du projet

D'autre part, le dossier */DocCreatorPackage* contient les fichiers nécessaires à la création du paquet pip, le dossier */doc* contient le cahier des charges et ce mémoire. A la racine se trouvent les scripts d'installation des dépendances et de compilation : *install-dependencies.sh* et *compile-project.sh*.

4.2.2 Architecture

Notre bibliothèque Python est constituée à partir d'interfaces (générées par SWIG) appelant le code C++ de DocCreator. L'interface en ligne de commande (voir section 4.3.3) utilise cette bibliothèque pour fonctionner. La bibliothèque et la CLI forment ensuite le paquet DocCreator, qui a été mis sur *PyPi.org*.

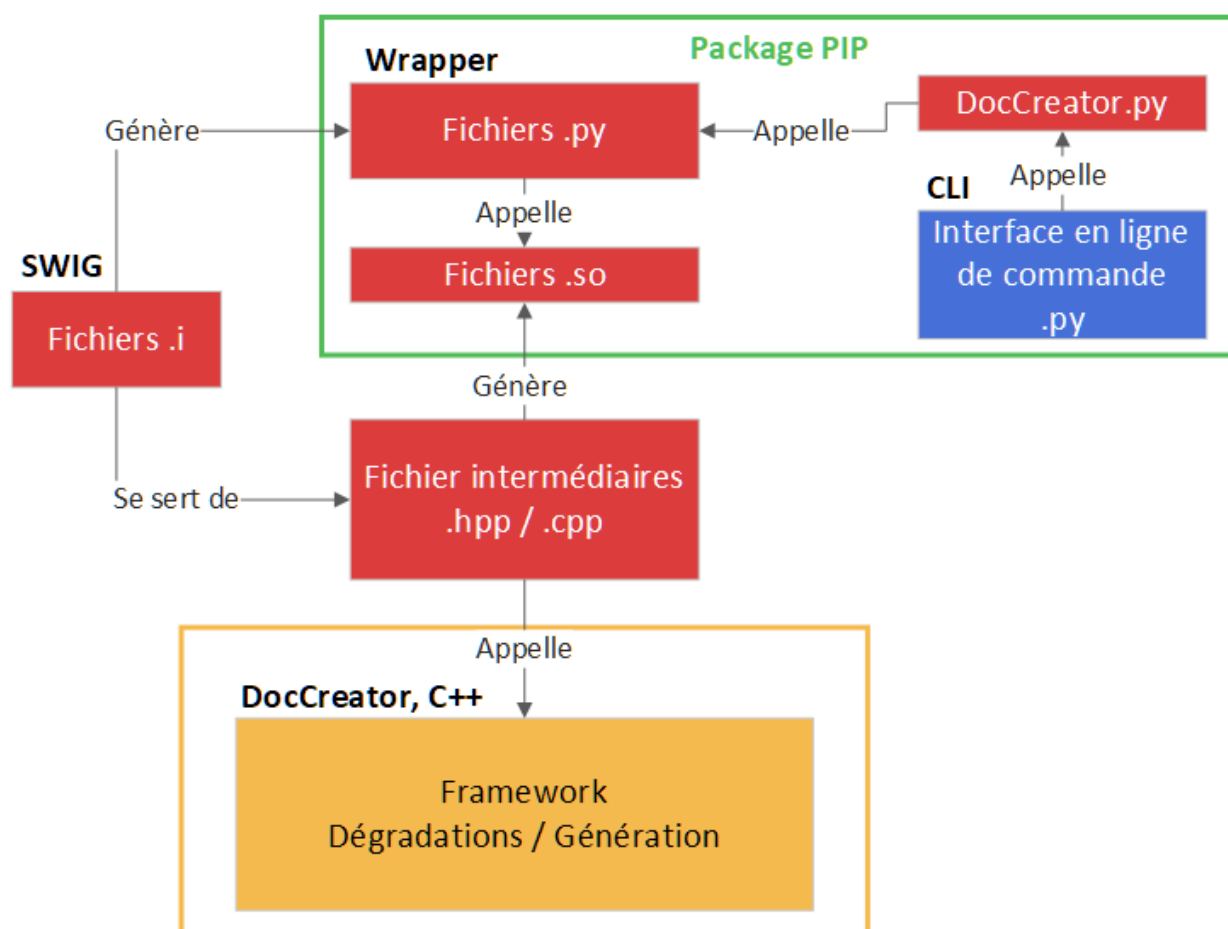


FIGURE 4.2 – Architecture (les figures en jaune représentent le code C++ de DocCreator; en rouge : le wrapper Python; en bleu : la CLI et en vert : le package PIP)

4.3 Implémentation

4.3.1 Wrapper Python

La bibliothèque Python de DocCreator est créée via SWIG, qui construit une interface autour du code C++ d'origine. SWIG est lié au *CMakeLists.txt*, à la compilation de DocCreator, deux fichiers sont générés par wrapper (deux pour les dégradations, deux pour la génération) : un fichier *.py* correspondant aux fonctions utilisables en Python, et un fichier *.so* contenant les fonctions C++ compilées. Nous avons rassemblé les fonctions Python dans un fichier intermédiaire *DocCreator.py*, c'est ce fichier qu'il est nécessaire d'importer pour utiliser la bibliothèque Python DocCreator.

4.3.1.1 Dégradations

Le wrapper construit une interface Python pour les 6 dégradations du dossier */framework*. SWIG est lié au *CMakeLists.txt*, à la compilation, deux fichiers sont générés : un fichier *.py* correspondant aux fonctions utilisables en Python, et un fichier *.so* contenant les fonctions C++ compilées.

Les dégradations wrappées sont toutes celles définies dans l'analyse des besoins (section 3) excepté *Distortion3DModel*. Toutes les dégradations renvoient en sortie une image dégradée. L'image d'origine n'est pas modifiée.

4.3.1.1.1 Interface Le wrapper Python créé par SWIG se définit grâce à un fichier *Degradations.i*. C'est dans cette interface que l'on importe le header qui contient les dégradations à wrapper, *Degradations.hpp*, car SWIG a besoin du prototype des fonctions.

Types En Python, il n'est pas possible de créer des *cv::Mat* nativement, en effet, le module *opencv-python* (*cv2*) renvoie les images en tant que *np arrays* donc il nous a fallu trouver un moyen de passer ces *np arrays* aux fonctions C++. Pour ce faire, nous avons décidé d'utiliser *numpy.i*[12], une bibliothèque de *typemaps* SWIG qui permet de faire passer un *np array* vers le code C++ en le convertissant en 4 arguments :

- *uint8_t** *imgIn* : tableau de pixel de l'image
- *int* *nbRows* : nombre de lignes
- *int* *nbCols* : nombre de colonnes
- *int* *nbChannels* : nombre de canaux de l'image (1 : niveaux de gris, 4 : avec canal alpha pour l'opacité, etc ...). Nos fonctions de dégradations ne manipulent cependant que des images 3 canaux.

Ces conversions sont effectuées après l'appel de fonction côté Python (où l'utilisateur fournit un *np array*) et avant l'exécution côté C++. Nous avons donc défini des fonctions en C++ (une pour chaque fonction de dégradation) dans des fichiers intermédiaires *Degradations.cpp/.hpp* qui prennent en paramètres les 4 arguments cités précédemment que l'on utilise pour créer une *cv::Mat*. Les fonctions de dégradation de DocCreator sont ensuite

appelées sur ces cv::Mat. Cette *typemap* est appliquée dans notre fichier interface *Degradations.i*, et ne s'applique qu'à un *np array* à 3 dimensions.

Le choix de *NumPy* se justifie également par sa popularité, et notamment dans le domaine du Machine Learning en Python. Etant donné qu'il s'agit là d'un des domaines "cibles" de notre bibliothèque, cela nous permet de prétendre à une bonne compatibilité dans l'utilisation de la bibliothèque avec d'autres outils de Machine Learning.

Arguments *imgOut* et *imgOutLength* sont les "arguments de sortie" utilisés par *numpy.i*, *imgOut* ne fait pas partie de l'input. Dans le code C++ de *Degradations.cpp*, on effectue un *memcpy*, ce qui nous permet de retourner dans *imgOut* l'image dégradée, sans avoir modifié l'image d'origine.

A l'appel d'une fonction de dégradation, il est nécessaire de spécifier la taille de l'image en 1D (*imgOutLength*). Cependant, comme cette valeur est égale au nombre de pixel de l'image d'entrée, on fait en sorte de simplifier la démarche à l'utilisateur en le calculant dans le fichier intermédiaire Python : *DocCreator.py*. Dû à des limitations de SWIG, il n'est pas possible de retourner un tableau à plusieurs dimensions sans avoir la taille de ces dimensions fixée à l'avance. La valeur de retour est donc *imgOut*, l'image dégradée en 1D. Pour éviter à l'utilisateur de recevoir une image en sortie ayant des dimensions différentes de celle donnée en entrée, on effectue un *reshape* dans *DocCreator.py* avant de la retourner.

L'interface contient également des *kwargs*, des *keywords arguments*, qui nous permettent de spécifier le nom d'un paramètre avant de lui affecter une valeur, permettant ainsi de les passer dans n'importe quel ordre lors à condition de les nommer de l'utilisation d'une fonction de dégradation en Python, exemple :

```
img = DocCreator.holeDegradation(  
    imgIn = img,  
    size = 0,  
    pattern = pattern,  
    yOrigin = 50,  
    color = [220, 60, 60]  
    xOrigin = 50,  
    holetype = 1,  
    side = 1,  
)
```

4.3.1.1.2 Fichiers intermédiaires Pour utiliser le wrapper, 3 fichiers intermédiaires sont mis en place. Ces fichiers intermédiaires, outre leurs avantages spécifiques, présentent également l'intérêt de pouvoir centraliser le wrapper Python et le séparer du reste du code. De cette manière le code original de DocCreator n'est pas mêlé avec les fichiers du wrapper et la maintenabilité du logiciel à l'avenir n'en sera que plus aisée.

Degradations.cpp/.hpp Le fichier header *Degradations.hpp* permet de déclarer à SWIG quelles fonctions wrapper. Le fichier *Degradations.cpp* associé permet de garder "intacts" les fichiers de dégradation originaux, sans y ajouter les modifications nécessaires au bon fonctionnement du wrapper. Les fonctions d'origine utilisent des classes C++ en tant que types, qui n'existent pas en Python. Pour faciliter l'intégration dans l'environnement Python, on remplace ces arguments (notamment les `cv::Mat` et les *QImage* pour que, depuis Python, ne soit passé qu'un *np.array*. On effectue ce remplacement via SWIG grâce aux *typemaps* de *numpy.i*. Un *np.array* sera converti en 4 arguments (*imgIn*, *nbRows*, *nbCols* et *nbChannels*) qui pourront être utilisés pour initialiser une `cv::Mat` qui sera utilisée pour appeler les fonctions de dégradation en C++. Le code de ces fonctions suit un même schéma, par exemple, pour *PhantomCharacter* on a :

```
void phantomCharacter(
    uint8_t *imgIn ,
    int nbRows ,
    int nbCols ,
    int nbChannels ,
    uint8_t *imgOut ,
    int imgOutLength ,
    int frequency ,
    char *phantomPatternsPath
){
    /*
        On verifie que l'image n'est pas vide :
        les degradations utilisent des images a 3 canaux
    */
    assert(imgIn && nbRows>0 && nbCols>0 &&nbChannels==3);

    /*
        On verifie que l'image de sortie a bien
        les memes dimensions que l'image d'entree
    */
    assert(imgOutLength == nbRows * nbCols * nbChannels);

    /*
        On commence par passer le tableau a une cv\hc\hc Mat
        CV_8UC3 correspond a une image en uint 8 bit a 3 canaux
    */
}
```

```

cv\hc\hc Mat img(nbRows, nbCols, CV_8UC3, imgIn);

/*
   On appelle la fonction de dégradation
   du code de DocCreator
*/
cv\hc\hc Mat res = phantomCharacter(
    img,
    (Frequency) frequency,
    (QString) phantomPatternsPath
);

/*
   On écrit le résultat dans imgOut,
   qui sera l'image retournée
*/
uint8_t* dataPointer=reinterpret_cast<uint8_t*>(res.data);
memcpy(imgOut, dataPointer, imgOutLength*sizeof(uint8_t));
}

```

DocCreator.py Le 3e fichier intermédiaire est *DocCreator.py*. Ce fichier Python permet d'utiliser les *typemaps* prédéfinies de *numpy.i* et évite ainsi l'utilisateur d'avoir à faire des *reshape* à chaque appel d'une dégradation. Le schéma des fonctions intermédiaires est relativement identique pour toutes les dégradations (excepté pour *holeDegradation*), par exemple, le code de *phantomCharacter* dans ce fichier est :

```

def phantomCharacter(imgIn, frequency, phantomPatternsPath):

    imgOutLength =
        imgIn.shape[0]
        * imgIn.shape[1]
        * imgIn.shape[2]

    res = Degradations.phantomCharacter(
        imgIn,
        imgOutLength,
        frequency,
        phantomPatternsPath
    )

    return res.reshape(imgIn.shape)

```

HoleDegradation est un cas particulier car la fonction peut prendre une seconde image en entrée, en tant qu'argument optionnel. La valeur par défaut est "None", hors les fonctions

C++ n'acceptent pas le type *Python None*, qui n'est pas converti par SWIG, il faut donc rajouter une condition dans le code pour détecter la présence ou l'absence de l'argument *img2*.

4.3.1.1.3 *Distortion3DModel* La dégradation *Distortion3DModel* n'est pas présente dans la version actuelle du wrapper par manque de temps. Cependant, les différentes étapes de son inclusion future dans la bibliothèque Python ont été étudiées : voir annexe C.

4.3.1.2 Génération

4.3.1.2.1 Fonctions Depuis Python, l'utilisateur crée un dictionnaire listant tous les arguments de la génération, ces arguments sont les suivants :

- *outputFolderPath* : le chemin du dossier de sortie. Si il n'existe pas, il est créé à l'exécution
- *backgroundList* : la liste des arrière-plans que l'on souhaite utiliser. Il est possible de préciser un chemin personnalisé pour ces arrière-plans, sinon, par défaut, la fonction ira chercher ceux du dossier */data* de DocCreator.
- *fontList* : la liste des polices que l'on souhaite utiliser. Il est possible de préciser un chemin personnalisé pour ces polices, sinon, par défaut, la fonction ira chercher celles du dossier */data* de DocCreator.
- (optionnel) *txtDirPath* : le chemin d'un dossier contenant des fichiers texte qui serviront de contenu pour le document. Si cet argument n'est pas précisé, un faux-texte "Lorem Ipsum" sera appliqué à la place.
- (optionnel) *bottomMarginMin*, *bottomMarginMax*, *leftMarginMin*, *leftMarginMax*, *rightMarginMin*, *rightMarginMax*, *topMarginMin*, *topMarginMax* : les marges min et max des différents bords du document, par défaut ces paramètres valent tous 50

puis on appelle une des 4 fonctions de génération, telles que déclarées dans *DocCreator.py* :

- *create* : qui va créer 1 document par texte en entrée à partir d'une police et d'un arrière-plan choisis aléatoirement dans leur listes respectives
- *createAllFontsOneBackground* : qui va créer autant de documents (par texte) qu'il y a de polices dans la liste donnée en paramètre, et en utilisant un arrière-plan choisi au hasard parmi ceux passés en argument
- *createOneFontAllBackgrouds* : qui va créer autant de documents (par texte) qu'il y a d'arrière-plans dans la liste donnée en paramètre, et en utilisant une police choisie au hasard parmi celles passées en argument
- *createAllFontsAllBackgrouds* : qui va créer des documents à partir de tous les arrière-plans, polices et textes donnés en entrée

Si un document ne rentre pas sur une seule page, plusieurs images seront générées (une par page), dépendant de la police utilisée, des marges, etc ...

L'intégration de la bibliothèque *Lipsum4Qt* posait des problèmes lors de l'intégration au wrapper, rendant son exécution impossible, le faux-texte a donc été directement inclus dans notre code.

4.3.1.2.2 Implémentation

Déclaration des fonctions Les fichiers *Generation.cpp/.hpp* sont les appels aux fonctions de générations refactorées, c'est dans ce fichier que l'on différencie les 4 fonctions *create* différentes à l'aide de l'argument *createFunctionID* (entier allant de 0 à 3). Ces fonctions sont ensuite déclarées dans *DocCreator.py* côté Python, qui se contente d'appeler le constructeur *Generation*, puis la même fonction *create* en faisant varier cet ID. Ceci permet d'éviter à l'utilisateur de connaître un paramètre en plus, il lui suffit donc de distinguer les différentes générations grâce au nom de fonction.

Dictionnaire Comme notre fonction C++ ne prend pas directement un dictionnaire Python en argument, on utilise à nouveau le fichier *DocCreator.py* pour transformer ce dictionnaire en une liste d'arguments via la syntaxe Python ***dictionnaire* (nommé ***params* dans le code). Les images en sortie (au format .png) sont directement enregistrées sur le disque. En effet, nous avons dans un premier temps pensé à les renvoyer vers Python pour pouvoir les manipuler de la même manière que pour les dégradations, mais son utilisation future dans un script pourra rendre cette manipulation problématique. En effet, si un utilisateur décide de générer plusieurs milliers d'images d'un seul coup, il y a un risque de saturer la mémoire (certains arrières-plans dépassant les 20 Mo).

Fichiers & fonctions auxiliaires Le fichier intermédiaire *Generation_aux.cpp* initialise le *configManager*, ce qui nous permet d'établir les chemins vers le dossier */data*, qui contient les polices et les arrières-plans de base de DocCreator.

Nous avons ajouté les fonctions *loadBackground* et *loadFont* dans *Generation.cpp* qui nous permettent d'aller chercher des arrières-plans et des polices dans n'importe quel dossier (et non pas de se restreindre au dossier */data*).

Le fonctionnement du wrapper reste le même que pour les dégradations, deux fichiers *Generation.py* et *_Generation.so* sont générés à la compilation, et les appels de fonctions se font via le fichier *DocCreator.py*.

4.3.2 Refactoring C++

Cette section regroupe les changements que nous avons appliqué au code C++ d'origine de DocCreator.

4.3.2.1 Génération de document

4.3.2.1.1 Déplacements & Copies Pour que la génération de document puisse être utilisée avec */framework* uniquement, et ainsi minimiser les dépendances, nous avons créé deux fichiers */framework/src/RandomDocument/DocumentGeneration.cpp/.hpp* qui contiennent une partie du code de */software/src/Degradations/ImageGenerationFromDirDialog.hpp/.cpp*. Toutes les dépendances au *DocumentController* ainsi que les appels à l'UI ont été supprimés.

Nous avons également déplacé les fichiers */software/src/Utils/FontUtils.cpp/.hpp* et */software/src/Utils/RandomDocument.cpp/.hpp* vers */framework/src/Utils*. Ces fichiers contiennent des fonctions qui nous servent à calculer le meilleur Line Spacing selon le font choisi, et à prendre un élément aléatoire dans un objet donné.

Le fichier *Generation_aux.cpp* contient des fonctions copiées depuis */software/DocCreator/src/DocCreator.cpp*.

4.3.2.1.2 Changements des paramètres Les fonts, textes et backgrounds étaient gérées par l'UI, il nous a donc fallu les ajouter. Par défaut, les documents générés étaient stockés dans le même dossier que les .txt d'entrée. Nous avons rajouté le paramètre *output-Folder*, qui permet de choisir le dossier de sortie des images générées. En outre, les autres paramètres : marges et espacements étaient définis en dur dans le code, ils ont donc été ajoutés aux arguments.

4.3.2.1.3 Changements du fonctionnement Dans la version d'origine de la génération, les fonctions généraient un fichier .xml qui était ensuite utilisé pour générer une image. Dans la version actuelle, on génère directement l'image avec la fonction *saveImage* pour éviter des écritures sur disque excessives.

4.3.2.2 Gestion des chemins de *PhantomCharacter*

La version de base de la dégradation *PhantomCharacter* de DocCreator récupérait systématiquement ses patterns depuis le dossier */data*, ce qui nécessitait d'initialiser le *configManager* de DocCreator. Pour s'éviter cette démarche dans le wrapper, nous avons effectué des modifications au code original en ajoutant un argument à *PhantomCharacter* : le chemin vers un dossier de patterns. Il nous a également fallu modifier tous les appels à cette fonction, et on peut à présent se passer de la dépendance au *configManager*.

4.3.2.3 Dégradations avec de l'aléatoire

Pour pouvoir effectuer des tests correctement, il a été nécessaire d'ajouter la possibilité de fixer la *seed* de l'aléatoire des dégradations *GrayscaleCharsDegradationModel* et *PhantomCharacter*. Pour cela, nous avons effectué des modifications dans le code C++ de ces deux dégradations pour que l'aléatoire soit présent ou non en fonction des options de compilation à l'aide de *ifdefs* :

```
#ifdef WITHOUT_RANDOM
    srand(0);
#else
    srand(time(nullptr));
#endif // WITHOUT_RANDOM
```

Si l'on veut effectuer des tests sur les deux dégradations concernées, il est nécessaire de compiler avec :

```
$ cmake .. -DWITHOUT_RANDOM=ON
$ make
```

L'exécution des tests inclura alors automatiquement *GrayscaleCharsDegradationModel* et *PhantomCharacter* (qui ne sont pas testées par *make test* en temp normal).

4.3.3 Interface en ligne de commande

L'interface en ligne de commande (*Command Line Interface* : CLI) devait répondre au besoin de pouvoir utiliser les fonctions de dégradation et de génération de la bibliothèque Python sans passer par l'écriture d'un script, et sans avoir de connaissances particulières en Python (ou dans un autre langage de programmation). La bibliothèque DocCreator étant en Python, la CLI l'est aussi. Nous avons utilisé deux bibliothèques supplémentaires :

- *PLY* (Python Lex-Yacc), pour les outils de génération d'analyseurs lexicaux et syntaxiques, et pour la manipulation de grammaires
- *argparse* pour le passage et la gestion d'arguments dans un programme Python

4.3.3.1 Principe

La CLI fonctionne sur la reconnaissance d'un langage. Un mot de ce langage est une succession de fonctions de dégradation ou génération de la bibliothèque Python DocCreator. Les mots sont encadrés par, soit un répertoire d'entrée contenant des images à dégrader, soit la fonction de génération de document, et par un répertoire de sortie. L'automate représentant ce langage est le suivant :

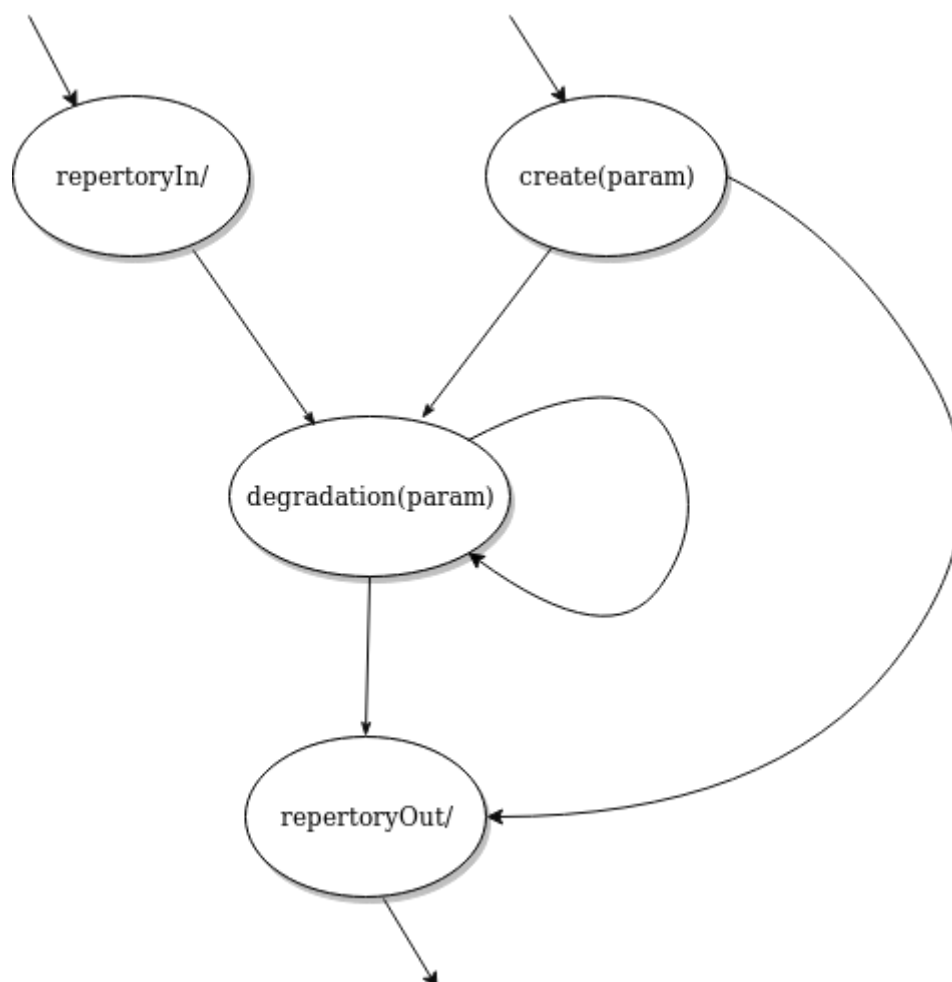


FIGURE 4.3 – Automate du langage de la CLI

L'état entrant est soit un répertoire d'image à dégrader, soit une fonction de génération de document. Ensuite, on entre dans une chaîne de dégradation (les images sont stockées dans un buffer entre chaque état) : toutes les images sont dégradées, puis on passe à la dégradation suivante, etc ... Quand on sort du cycle, l'état sortant est le dossier de sortie dans lequel les images qui ont été générées et/ou dégradées seront placées.

4.3.3.2 Structure

La CLI s'organise en 3 fichiers. Le fichier *cli.py* est le contrôleur, c'est lui qui se charge d'importer *DocCreator.py* et de récupérer les arguments grâce au module *argparse*. Le fichier *lexYaccAnalyse.py* contient la classe *lexYaccAnalyse* qui possède tous les outils d'analyse syntaxique pour l'interprétation de l'expression passée en paramètre. C'est dans ce fichier que se fait l'implémentation de l'algorithme présenté par l'automate montré précédemment. La classe *listeDegradation* (contenue dans le fichier du même nom) permet de stocker le résultat de l'interprétation. La structure de la CLI peut se résumer avec le diagramme suivant :

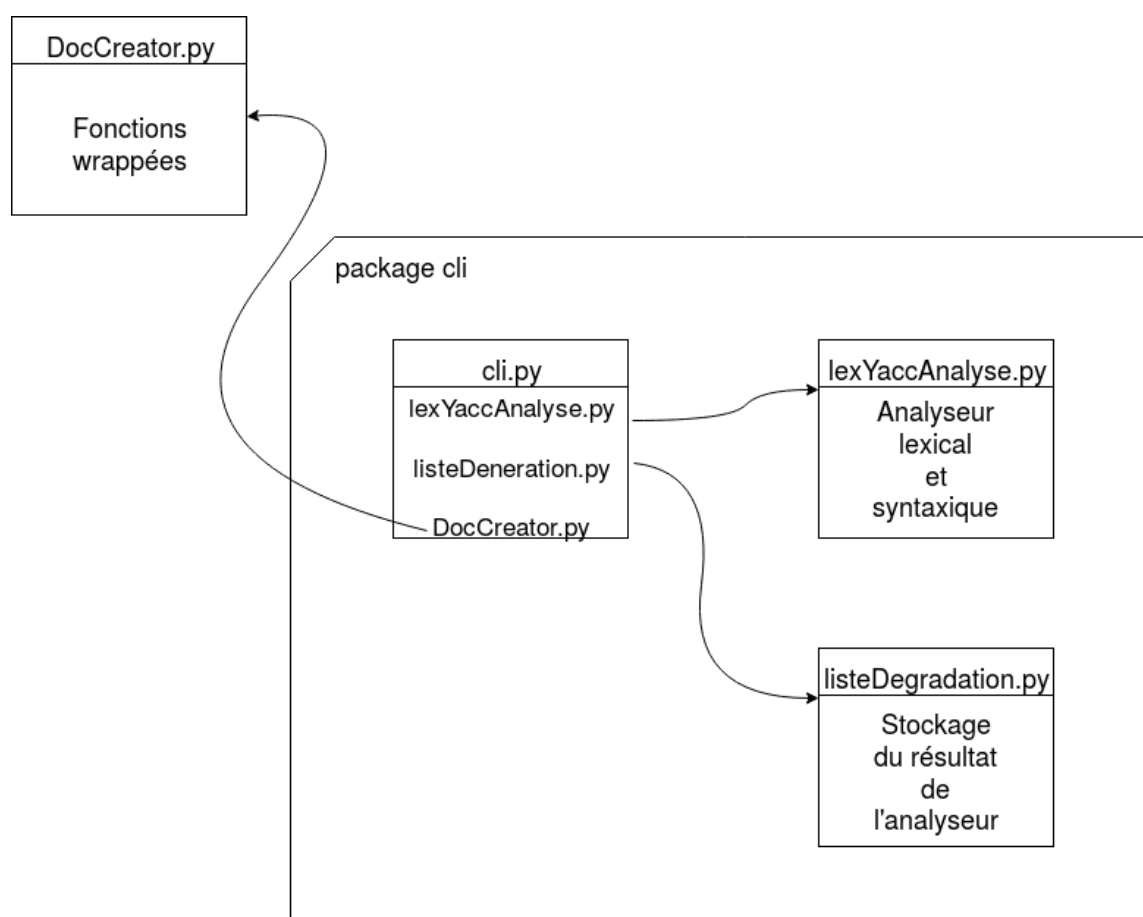


FIGURE 4.4 – Architecture de la CLI

4.3.3.3 Interpréteur

L'interpréteur lexical va découper la chaîne de l'expression en un certain nombre de *tokens*. Un *token* est un objet qui reconnaît une expression régulière, par exemple, le *token* "FUNCTION" reconnaît et enregistre les différentes fonctions de DocCreator. Ensuite, l'analyseur va vérifier que les *tokens* sont dans le bon ordre, ou qu'il n'en manque pas. Par exemple, l'absence d'un repertoire de sortie empêcherai la validation de l'expression. Après la vérification, l'interpréteur va transcrire les différentes informations contenues dans la *listeDegradation*. Dans le code, on utilise la bibliothèque Python *PLY* : Python Lex-Yacc pour gérer l'interpréteur de la manière suivante :



FIGURE 4.5 – Utilisation du module Lex-Yacc dans le code

4.3.3.4 Contrôleur

Suite à la génération de la *listeDegradation*, la CLI construit une liste de buffers d'images en utilisant *opencv* pour l'ouverture. Ces buffers contiennent les images de la génération ou du répertoire d'entrée. Ensuite, ce buffer est modifié, dégradation par dégradation (toutes les images du buffer sont dégradées avant de passer à la dégradation suivante) avec les fonctions de dégradation contenues dans *DocCreator.py*. L'application d'une dégradation au buffer s'implémente de la manière suivante :



FIGURE 4.6 – Application des dégradations au buffer

4.3.3.5 Fichier de configuration

La CLI peut lire des fichiers de configuration. Un fichier de configuration est composé d'une ou plusieurs expressions, séparées par un retour à la ligne. Ces expressions suivent le même langage que pour une utilisation en ligne de commande. Un repertoire de sortie d'une ligne peut être utilisé en temps que repertoire d'entrée pour une expression dans une ligne suivante. Le fichier de configuration doit être au format texte et respecter les règles d'utilisation du langage, mais l'extension du fichier n'est pas discriminante pour son utilisation. Le fichier remplace l'expression dans la ligne de commande :

```
$ python cli.py [fichier]
```

4.3.4 Installation via *pip*

Pour installer facilement notre bibliothèque Python ainsi que notre CLI, nous avons prévu une installation via *pip*. Ceci permet à un utilisateur de simplement entrer :

```
$ pip install DocCreatorPackage
```

Nous avons réalisé un premier paquet de test que nous avons envoyé sur *PyPi.org* (*test.pypi.org*).

Pour ce faire nous avons utilisé le système *wheel* qui permet de gérer les différentes distributions d'un paquet. Le fichier */DocCreatorPackage/setup.py* permet d'effectuer toutes les configurations nécessaires.

Dans l'état actuel du paquet, seule la version Ubuntu 16.04 est disponible en ligne, car le système de *wheel*, qui permet de construire des distributions selon différents critères, dont le système d'exploitation, ne permet pas de différencier les différentes distributions Linux, qui sont regroupées sous *Operating System :: POSIX :: Linux*. D'autre part, l'utilisation de cette bibliothèque nécessite que l'utilisateur possède déjà toutes les dépendances nécessairement préalablement à l'installation du paquet.

Pour résoudre ces deux problèmes, il faudrait inclure dans le paquet toutes les dépendances nécessaires, ce qui permettrait d'éviter de devoir créer une distribution par système d'exploitation compatible et qui dispenserait l'utilisateur la recherche et l'installation des dépendances.

4.3.5 Portabilité

4.3.5.1 Version de Python

A la compilation, le *CMakeLists.txt* du wrapper va détecter automatiquement la version par défaut de Python installée sur le système et générer le wrapper adapté. Si l'on veut générer le wrapper pour une autre version de Python que celle par défaut, comme par exemple pour Python 3+ lorsque la version par défaut du système est Python 2.7, il faut remplacer dans le code du *CMakeLists.txt* les lignes :

```
execute_process(COMMAND python -c "from distutils ...")
execute_process(COMMAND python -c "import distutils ...")
```

par :

```
execute_process(COMMAND python3 -c "from distutils ...")
execute_process(COMMAND python3 -c "import distutils ...")
```

4.3.5.2 Système d'exploitation

Le processus de compilation, de génération du wrapper et les tests ont été testés (machines physiques et/ou virtuelles) sur les systèmes suivant :

- Ubuntu 16.04
- Ubuntu 18.04
- Ubuntu 18.10
- Debian 9
- Mac OS 10.13
- Mac OS 10.14

D'autre part, nous avons fait en sorte que notre code soit compatible avec les systèmes Windows avec des instructions encadrées par des *ifdef* `_WIN32` telles que :

```
#ifdef _WIN32
    /* Instruction spécifique a Windows : commande, chemin, etc
       ... */
#else
    /* ... */
#endif
```

ou en utilisant la gestion des instructions spécifiques au système d'exploitation de Python (module `os`).

Nous n'avons pas réalisé d'essais sur Windows, mais le code d'origine de DocCreator étant compatible avec Windows 10 et Python étant très portable, le projet devrait théoriquement fonctionner sous Windows 10 également.

Partie 5

Analyse du fonctionnement

5.1 Fonctionnement

5.1.1 Dégradation d'image

5.1.1.1 Utilisation

La bibliothèque Python de dégradation d'image s'utilise en important le fichier intermédiaire *DocCreator.py* ainsi qu'un module pour charger les images (exemple, *OpenCV*) :

```
import DocCreator
import cv2
```

On doit donc disposer des fichiers *DocCreator.py* (fichier intermédiaire Python), *Degradations.py* (fichier Python généré par SWIG contenant les fonctions de dégradation) ainsi que *_Degradations.so* (contenant les fonctions C++ compilées, qui seront appelées par le wrapper). On prendra pour référence dans l'exemple l'image non dégradée suivante :

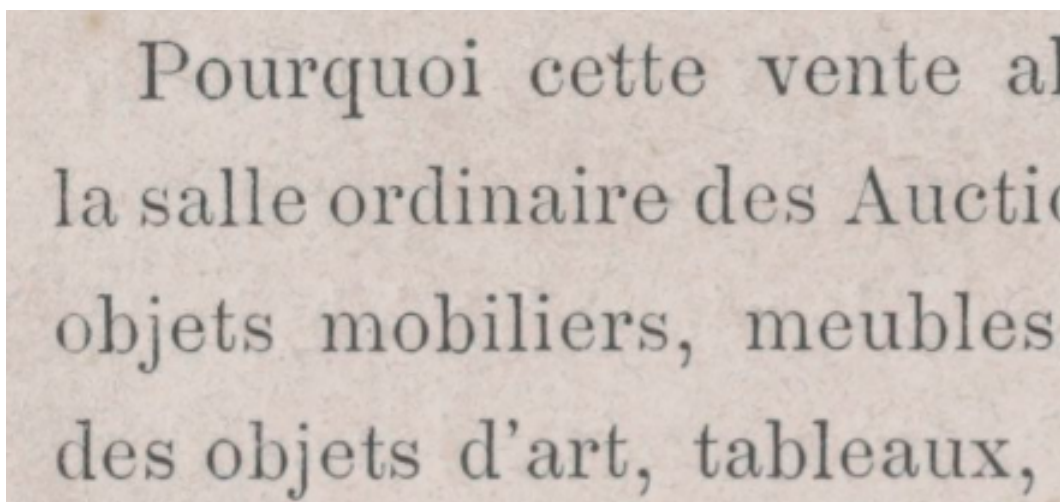


FIGURE 5.1 – Image de référence qui sera dégradée dans l'exemple

On prendra pour exemple la dégradation *GrayscaleCharsDegradationModel* (les appels aux fonctions de dégradation suivent le même modèle). Pour plus de lisibilité, le nom des arguments a été mis dans l' exemple lors de l'appel de la fonction (comme le permettent les *keyword args* de SWIG) mais ils ne sont pas obligatoires (les arguments doivent sinon être passés dans l'ordre). *grayscaleChars* peut être utilisée sans arguments, la fonction appliquera une dégradation standard avec des valeurs par défaut fixées. Une utilisation dans un script Python se fera de la manière suivante :

```
img = cv2.imread("inImage")
outImg = DocCreator.grayscaleChars(imgIn=img)
cv2.imwrite("outImage")
```

qui donne (*GrayscaleCharsDegradationModel* travaille en niveaux de gris, d'où le changement de couleur) :

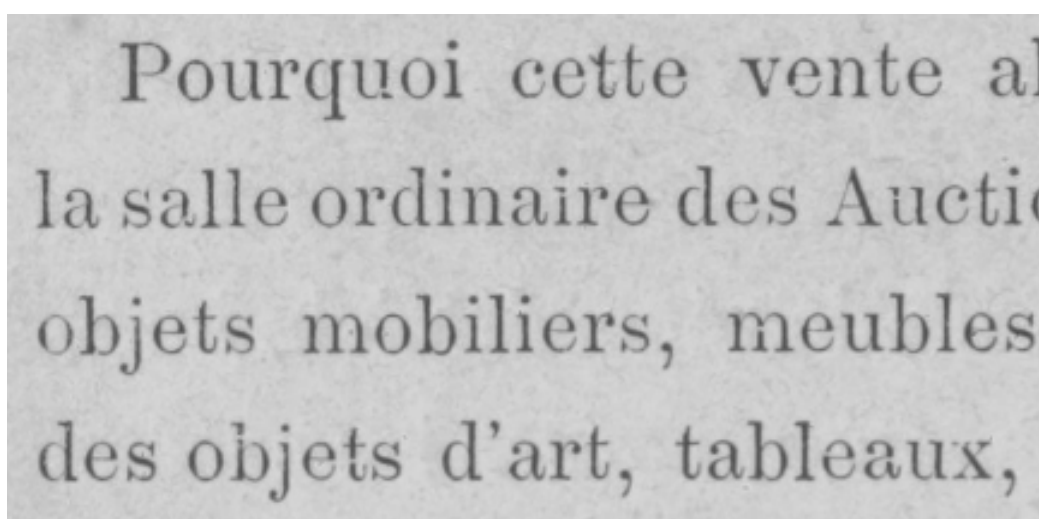


FIGURE 5.2 – Image dégradée avec les paramètres de *grayscaleChars* par défaut

Cet exemple permet de se rendre compte de la simplicité d'utilisation de la bibliothèque Python : il suffit d'ouvrir une image, d'appeler la fonction de dégradation et on a une autre image dégradée. L'utilisateur n'a que peu de code à écrire dans un script et n'aura donc aucune difficulté à automatiser la procédure de dégradation de grand nombre d'image, comme pour une utilisation en *data augmentation*.

5.1.1.2 Limites & Perspectives

Durant le développement, nous avons trouvé une solution alternative permettant de retourner une *cv::Mat* depuis C++, puis de convertir cet objet en *np array* via une *typemap* SWIG (écrite nous-mêmes) avant de le retourner en Python. Cette solution aurait pu permettre d'éviter la présence des fichiers intermédiaires, cependant cette solution s'est révélée désastreuse d'un point de vue mémoire. En effet, nous avons pu constater à l'aide de l'outil *Valgrind*, que les données étaient allouées côté C++ sans aucun moyen de les désallouer

depuis Python. L'ajout de méthodes supplémentaires de désallocation ne nous a pas paru être un choix judicieux du point de vue du confort et de la facilité d'utilisation de la bibliothèque pour l'utilisateur (l'utilisateur Python sera habitué au *Garbage Collector* de Python). Bien que les fichiers intermédiaires puissent sembler plus lourds car ils nécessitent l'écriture d'un peu de code supplémentaire, ils ne présentent aucun désavantage du point de vue des performances et de l'expérience utilisateur.

La dégradation *Distortion3DModel* n'a pas du tout été implémentée dans la version actuelle du projet. Pour compléter la bibliothèque Python DocCreator, il serait possible de partir des bases présentées en annexe C.

5.1.2 Génération de document

5.1.2.1 Code C++ refactoré

5.1.2.1.1 Utilisation Les fonctions de génération de document sont maintenant présentes dans `/framework/src/RandomDocument`. La classe *DocumentGeneration* ne comporte que deux méthodes publiques : le constructeur et *generate()*. Pour utiliser la génération, il suffit donc d'appeler le constructeur, de définir les différents arguments publics (le font, le background, le dossier contenant les textes, le dossier de sortie, les marges haut, bas, gauche et droite, ainsi que le line spacing), puis d'appeler *generate()*. Les autres arguments seront calculés puis utilisés par les méthodes privées.

5.1.2.1.2 Limites & Perspectives Les fonctions de génération sur lesquelles nous nous sommes basés ne sont pas les plus récentes et à jour du code de DocCreator. Pour une meilleure maintenabilité et une meilleure cohérence, il serait envisageable à l'avenir d'adapter certaines fonctions en se basant sur `/software/DocCreator/src/Document/DocumentController.cpp`.

5.1.2.2 Wrapper Python

5.1.2.2.1 Utilisation Les fonctions de génération de document de la bibliothèque Python s'utilisent comme suit :

```
import DocCreator

params = {
    'outputFolderPath' : './outputDir',
    'backgroundList' : ['montaigne-03.png'],
    'fontList' : ['JulesVerne'],
}

DocCreator.create(params)
```

Dans cet exemple, on utilise la première fonction de génération, qui prend aléatoirement un background et un font dans leurs listes respectives, et comme on a pas précisé de fichier texte en entrée, on génère un "Lorem Ipsum". Le début du document généré est :

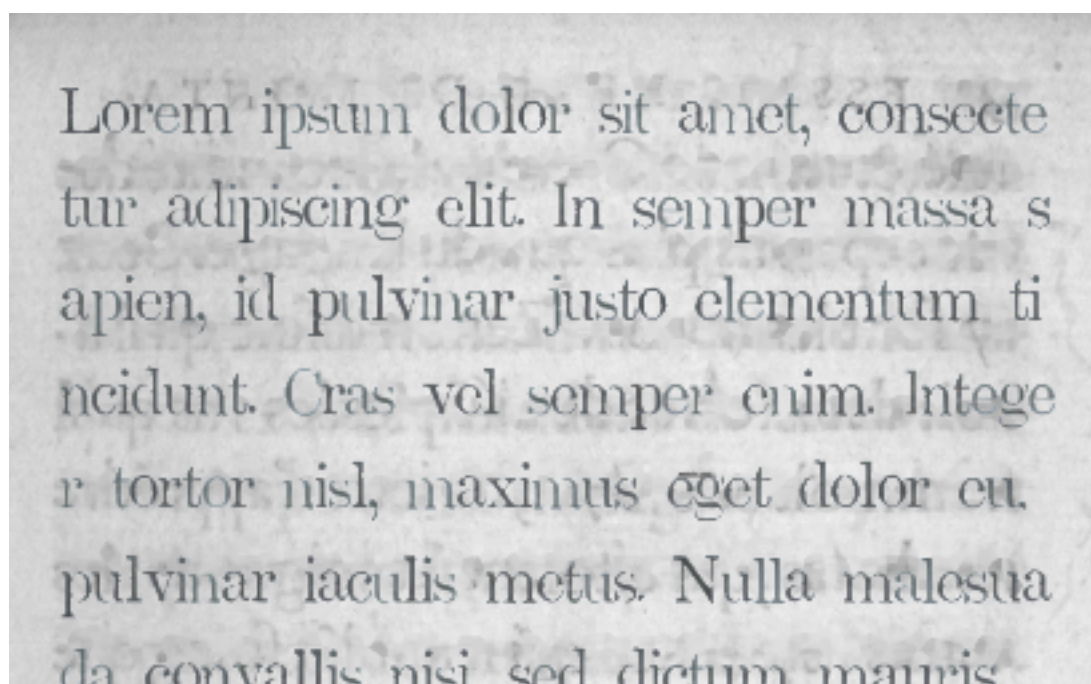


FIGURE 5.3 – Exemple de document généré par le code montré précédemment

De la même manière que pour les dégradations, l'utilisation des fonctions de génération de la bibliothèque est très simple, étant donné que tout le travail préliminaire est effectué dans les fichiers intermédiaires du wrapper.

5.1.2.2.2 Limites & Perspectives Notre génération de document présente 2 bugs connus (d'autres bugs sont connus, comme le fait que les polices ne se mettent pas à l'échelle, mais ce sont des défauts qui apparaissent déjà dans la version de DocCreator d'origine), et nous avons malheureusement manqué de temps pour les résoudre :

Marge du bas La marge du bas d'un document ne fonctionne pas comme attendu : la position Y de notre fonction qui place les caractères est erronée. En effet, lorsque l'on ajoute un caractère à cette position Y comme position verticale, le caractère n'est pas réellement à l'endroit prévu. Les lignes n'atteignent pas le bas du document, car la valeur de la position Y a atteint la valeur limite (valeur limite qui est cependant cohérente avec la taille de l'image, même si le rendu, lui, n'est pas correct).

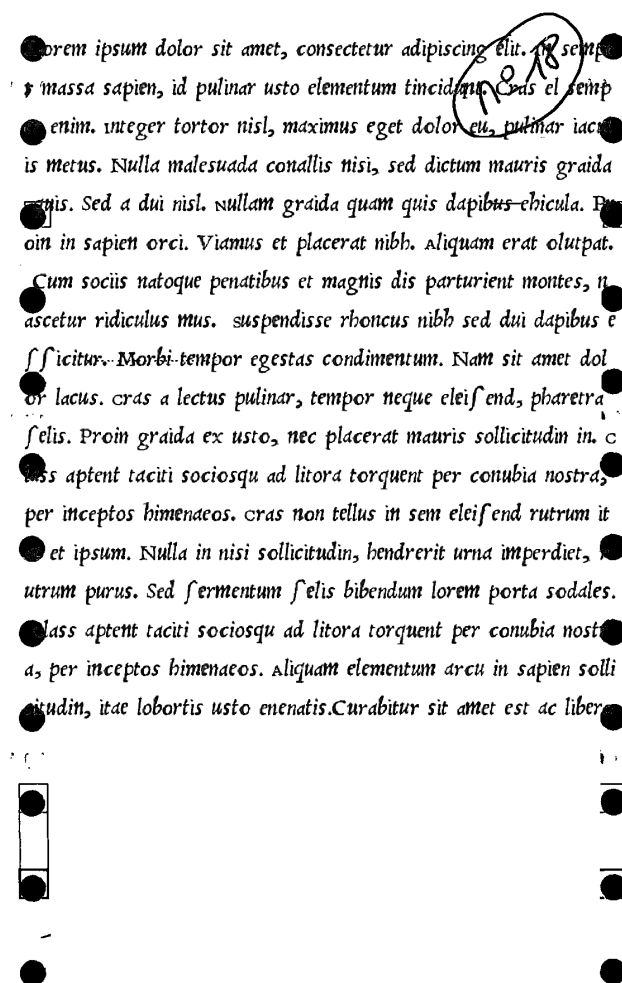


FIGURE 5.4 – L'espace en bas de l'image n'est pas rempli et crée une second page

De plus, si l'on ajoute un caractère en-dehors de l'image, à la position verticale *image-Height* + 200, il apparaît en-dessous des lignes précédemment écrites sur le document, qui devraient aller jusqu'en bas de la page.

Caractères spéciaux Les caractères spéciaux ne sont pas supportés, si ils sont présents dans le fichier texte donné en entrée, ils ne seront tout simplement pas affichés sur l'image de sortie. Une perspective d'amélioration serait d'utiliser la méthode *addString()* du fichier *DocumentController.cpp* évoqué précédemment.

5.1.3 Interface en ligne de commande

5.1.3.1 Langage & Utilisation

La CLI reconnaît un langage, ce qui signifie que son utilisation nécessite d'adopter des règles. Dans notre cas, ces règles sont :

- Les répertoires d'entrée et de sortie doivent toujours être précédés par "r :".
- Le premier paramètre des dégradations, l'image à modifier, n'est jamais donnée dans la liste des arguments des dégradations, elle sera prise automatiquement dans le dossier d'entrée.
- Les noms des dégradations ainsi que les paramètres sont les mêmes que pour le wrapper Python (voir *DocCreator.py*), excepté les tableaux (notamment de couleur), qui doivent être passés en "r:g:b", au lieu de "[r:g:b]"

La CLI permet d'utiliser toutes les fonctions de dégradation de la bibliothèque Python Doc-Creator, ainsi que les fonctions de génération.

5.1.3.1.1 Cycle de dégradation Il est possible d'enchaîner plusieurs dégradations en un seul appel de la CLI, il s'agit d'un cycle de dégradation. Par exemple, l'appel

```
$ python cli.py "r:imageDirIn/=>grayscaleChars(5)=>
shadowBinding(3,100)=>r:imageDirOut/"
```

prendra toutes les images du dossier */imageDirIn*; leur appliquera la dégradation *grayscaleChars*, puis, sur les images nouvellement dégradées, appliquera la dégradation *shadowBinding*. Les images dégradées (2 fois par 2 dégradations différentes donc) seront ensuite placées dans le dossier */imageDirOut*.

5.1.3.1.2 Fichier de configuration Pour simplifier le passage d'arguments à la CLI, sauvegarder des appels ou encore les enchaîner, il est possible d'utiliser un fichier externe contenant les expressions à passer à la CLI. Le langage utilisé par ces expressions est le même que pour une utilisation en ligne de commande. Un exemple de fichier de configuration peut donc être :

```
create(bg-1.png:montaigne-03.png,JulesVerne:manto,Loremlpsum,2)
=>r:testGeneration
r:testGeneration/=>grayscaleChars(50)=>r:testGrayscaleChars/
r:data/imagesIn/=>BlurFilter(0,11)=>r:testBlurFilter/
r:data/imagesIn/=>bleedThrough(data/images/p01.png,50)=>r:
testBleedThrough
```

Dans cet exemple on commence par générer des documents que l'on place dans le dossier *testGeneration*, ensuite, ce dossier est utilisé comme répertoire d'entrée pour l'application de la dégradation *GrayscaleChars*. Les deux dernières lignes sont juste des dégradations classiques depuis un dossier */data/imagesIn*.

5.1.3.2 Limites & Perspectives

La CLI a pour but de permettre à un utilisateur non expérimenté en Python de pouvoir utiliser les fonctions de DocCreator en ligne de commande. De ce fait, la CLI permet d'utiliser toutes les fonctions de dégradation et de génération de la bibliothèque Python. L'utilisation du langage peut nécessiter la lecture de la documentation, mais offre un bon compromis entre fonctionnalités proposées (enchaînement des dégradations, fichiers de configuration) et simplicité d'utilisation (à part les quelques règles, il ne suffit de connaître que le nom des fonctions et leur arguments). De ce fait, le plus gros inconvénient de notre CLI repose sur son manque de puissance. En effet, les images sont placées dans des buffers entre deux traitements, ce stockage peut poser problème, notamment lors de la génération d'image, si l'on souhaite générer un (très) grand nombre de documents, étant donné que les arrières-plans peuvent être volumineux, il y a possibilité de dépasser la mémoire (ce problème n'est pas présent dans les fonctions de la bibliothèque). Une amélioration possible serait de repenser la gestion des cycles de la CLI pour traiter les images dans un cycle une par une au lieu de toutes en même temps.

5.1.4 Installation via *pip*

5.1.4.1 Utilisation

Le paquet pip se trouve actuellement sur un serveur de test, son installation diffère donc quelque peu des autres bibliothèques installables via pip :

```
$ python -m pip install --index-url https://test.pypi.org/
simple/ DocCreatorPackage
```

Le fichier README.md de */DocCreatorPackage* donne les instructions pour mettre à jour ce package, en donnant notamment les identifiants pour accéder au compte *test.pypi.org* associé.

A terme, l'installation via pip pourra être effectuée très simplement avec la commande :

```
$ pip install DocCreator
```

Une fois installée, et si toutes les dépendances sont présentes, la bibliothèque peut être importée très simplement dans un script Python via :

```
import DocCreator
```

De même, la CLI peut être utilisée en tapant :

```
$ python -m DocCreator.cli [args]
```

Avec ces exemples, on peut affirmer que l'on remplit bien le besoin de la simplicité, en rendant l'installation très basique pour l'utilisateur.

5.1.4.2 Limites & Perspectives

Cependant, comme évoqué précédemment, la version actuelle de notre paquet est incomplète et pas aussi souple que l'on voudrait, ne fonctionnant que sous Ubuntu 16.04 et à condition que toutes les dépendances soient installées dans les bonnes versions. Comme évoqué dans la partie 4.3.4 : implémentation sur l'installation via pip, il serait nécessaire d'inclure toutes les dépendances dans le paquet pour résoudre ces deux problèmes.

5.1.5 Utilisation en *data augmentation*

Le but de la bibliothèque Python est de pouvoir être utilisée pour réaliser de la *data augmentation* en Python. Précédemment nous avons mentionné d'autres bibliothèques permettant la dégradation d'images dans ce but, comme *ocrodeg*, *Augmentor* ou encore *imgaug*. Pour vérifier la compatibilité de notre travail, nous avons réalisé plusieurs essais, parmi ceux-là, en voici deux représentatifs :

- Génération de document puis dégradation avec *Augmentor* :

```
import Augmentor
import DocCreator

new_data_folder = './imgs'

params = {
    'outputFolderPath' : new_data_folder,
    'backgroundList' : ['bg-1.png'],
    'fontList' : ['BMTcomplete'],
}
DocCreator.create(params)

p = Augmentor.Pipeline(new_data_folder)
p.rotate270(probability=0.40)
p.crop_random(probability=1, percentage_area=0.75)
p.sample(3)
```

- Dégradation avec *imgaug* puis *DocCreator* :

```
from imgaug import augmenters as iaa
import DocCreator
import cv2

img = cv2.imread("p00.png")
seq = iaa.Sequential([
    iaa.Crop(px=(0, 16)),
    iaa.Fliplr(0.5),
    iaa.GaussianBlur(sigma=(0, 3.0))
```

```

])

image_aug = seq.augment_image(img)
image_aug = DocCreator.shadowBinding(image_aug, 3, 500)
cv2.imwrite("doccreator_imgaug.png")

```

Ces deux scripts fonctionnent comme attendu, les images générées par DocCreator ainsi que ses fonctions se mêlent sans problèmes avec les autres fonctions de ces autres bibliothèques, comme le souhaitent les clients.

5.2 Tests

Pour tester le bon fonctionnement de notre wrapper, nous avons réalisé différentes séries de tests, par dégradation, qui suivent le même principe. Les tests ont été effectués sur les systèmes suivants : Ubuntu 16.04, Ubuntu 18.04, Ubuntu 18.10, Debian 9, Mac OS 10.13 et Mac OS 10.14.

5.2.1 Détail

5.2.1.1 Wrapper

5.2.1.1.1 Fonctionnement Les différentes séries de tests sont regroupées par dégradation (dans des fonctions), dans deux fichiers : dans le fichier de test principal : `/tests/wrapper/python/testDegradationWrapper.py` (pour les fonctions de la bibliothèque Python) et dans `/tests/wrapper/python/testDegradationWrapper.cpp` (pour les fonctions d'origine en C++). Pour une dégradation donnée, on génère d'abord en Python une série d'images en fonction de différents arguments. Ces images sont sauvegardées dans un dossier, numérotées par ordre de génération. Ensuite, on fait appel à la même fonction de test de la dégradation, mais dans le fichier C++. Les images sont générées de la même manière, avec les mêmes arguments. Si toutes les images ont pu être correctement générées, on les compare une par une, pixel par pixel. Enfin, on fait le compte des comparaisons successives, et si une image n'est pas identique à sa paire, le test échoue. Les dégradations *GrayscaleCharsDegradationModel* et *PhantomCharacter* contiennent de l'aléatoire dans leur code d'origine. De ce fait, pour tester ces fonctions, il est nécessaire de compiler avec l'option `WITHOUT_RANDOM` :

```

$ cmake .. -DWITHOUT_RANDOM=ON
$ make

```

La *seed* aléatoire sera alors fixée pour ces deux dégradations et les tests seront effectués avec les autres (l'utilisation de `make test` sans avoir compilé de cette manière ne lancera pas les tests sur *GrayscaleCharsDegradationModel* et *PhantomCharacter*).

5.2.1.1.2 Test de dégradation Les tests de dégradations suivent tous le même schéma : les fonctions sont appelées en Python, à chaque appel de fonction, on vérifie à l'aide d'un `"try : except :"` qu'il n'y a pas eu d'erreur. Toutes les étapes des tests sont vérifiées et des

interruptions avec message d'erreur sont générées si besoin. On appelle ensuite le programme de test en C++. Les images sont générées de la même manière en suivant les mêmes tableaux d'arguments. Les arguments sont généralement choisis pour tester les cas d'utilisation normale, les cas "extrêmes" (valeur min et max/élevée des arguments) et les arguments optionnels de certaines dégradations (le détail des arguments essayés par les tests est décrit en commentaire, directement dans le code). Les fonctions de tests retourneront 1 si l'exécution ne s'est pas déroulée normalement (un message d'erreur aura été envoyé) et 0 sinon. La fonction Python étudiera ce code de retour, si il est égal à 0, on lance la comparaison des images générées.

5.2.1.1.3 Fonctions intermédiaires Le fichier de test Python contient différentes fonctions de tests qui peuvent être réutilisées pour de futurs tests :

- *compareImage* : prend en entrée deux images et retourne 0 si elles sont identiques, 1 sinon
- *saveImg* : sauvegarde une image donnée dans un dossier donné, après l'avoir nommée en fonction de la dégradation correspondante et de son indice
- *exitError* : renvoie un message indiquant que le test a échoué en testant une dégradation donnée, avec un numéro donné (correspondant à l'indice de l'image testée)
- *compareDir* : compare les images de deux dossiers pour une dégradation donnée. Pour cela on trie d'abord le contenu des dossiers par ordre alphabétique (les images de test étant nommée selon : [Dégradation][Indice].png) puis on appelle *compare* sur toutes les images trouvées. Les messages de sortie sont adaptés en fonction du code de retour de *compare*, et du nombre d'images identiques (0, toutes, ou une partie)
- *callCFunc* : appelle le programme de test C++ à partir d'une liste d'arguments donnée
- *manageDir* : crée un dossier donné si il n'existe pas, le vide sinon

Le fichier de test C++ contient l'équivalent des fonctions *exitError* et *saveImg* pour de la factorisation de code.

5.2.1.2 CLI

Les tests de l'interface en ligne de commande, dans */tests/wrapper/python/cli/testCLI.py* fonctionnent sur le même principe, on génère une série d'image (dans l'état actuel de nos tests, 2) par dégradation, une première fois avec la CLI, une seconde fois avec les fonctions de la bibliothèque Python directement, puis on compare les images générées pixel par pixel. Comme les fonctions du wrapper sont gérées par les tests présentés précédemment, nous avons réduit le nombre d'appels de fonction dans ces tests. De même, nous n'appellons pas les fonctions de dégradation qui contiennent de l'aléatoire (elles seront toujours testées dans les tests de dégradation). De ce fait, chacune des 4 fonctions de tests pour la CLI génère 4 images au total (2 CLI et 2 wrapper). Ces tests ont également été intégrés au CMake.

5.2.2 Limites & Perspectives

Nous ne testons pas les fonctions de génération, que ce soit pour la bibliothèque Python ou la CLI. Nous ne testons pas non plus la fonctionnalité de prise en charge de fichier de configuration pour la CLI. Ce dernier cas aurait été très simple à tester, mais n'a pas été ajouté par manque de temps. En revanche, le test de la génération aurait demandé un peu plus de travail, en effet, la génération contient de l'aléatoire, il aurait donc fallu opérer de la même manière que pour les dégradations afin de pouvoir compiler sans aléatoire, mais les fonctions qui le gère pour la génération sont plus éparpillées et nombreuses dans le code.

5.2.3 Résultats

5.2.3.1 Lancement

L'exécution des tests a été intégrée dans la compilation. La déclaration des tests se fait dans le fichier `/tests/CMakeLists.txt`. Un test est défini par dégradation. Pour le wrapper, par défaut, les images seront générées dans des dossiers `/testpython/[degradation]` et `/testc/[degradation]`. Un test d'une dégradation échouera si une seule paire d'image ne passe pas le test de comparaison. Les tests ne passeront donc à 100% que si toutes les images de toutes les dégradations passent les vérifications de comparaison pixel par pixel. Pour exécuter ces tests, il faut se placer dans le repertoire de la compilation (`/build`) puis faire :

```
$ make test
```

ou bien :

```
$ ctest
```

L'exécution de ces commandes donnera le résultat des tests mais pas le détail des messages affichés durant l'exécution, pour les visualiser, ajouter l'option `-verbose` à `ctest` ou consulter les logs générés dans `/build/Testing/Temporary/LastTest.log`. Ce détail donnera également la possibilité de voir le temps pris par les différentes fonctions, en ainsi comparer les temps C++ et Python.

Si le programme a été compilé avec `cmake .. -DWITHOUT_RANDOM`, les deux dégradations contenant de l'aléatoire, `GrayscaleCharsDegradationModel` et `PhantomCharacter` seront testées par le CMake.

5.2.3.2 Conditions

Les paramètres des différents tests (précisés en commentaire dans le code) ont été choisis pour tester deux types d'utilisation :

- Une utilisation normale des fonctions, avec des paramètres standards ou des valeurs par défaut, pour une dégradation régulière de l'image.
- Une utilisation "extrême" des fonctions, en utilisant les valeurs minimum ou maximum, ou en l'absence de bornes, des valeurs très au-delà de la normale.

Les tests ont été exécutés tour à tour sur différentes machines et différents systèmes pour s'assurer la portabilité de nos fonctions. Les systèmes suivants ont été testés : Ubuntu 16.04, Ubuntu 18.04, Ubuntu 18.10, Mac OS 10.13, Mac OS 10.14, Debian 9.

5.2.3.3 Résultats & Temps d'exécution

Sur les systèmes évoqués ci-dessus, les tests sont tous passés, cela garantit une utilisation normale des fonctions de dégradation, directement via la bibliothèque ou via la CLI (les fonctions de génération n'ont pas été testées par manque de temps, car elles contenaient de l'aléatoire plus long à retirer que pour les fonctions de dégradation). Pour les dégradations : les temps d'exécution (en secondes) ont donné les valeurs suivantes (sur une machine virtuelle Ubuntu 18.10, Processeur Intel i5 4 coeurs @3.3GHz, 6208Mo de RAM allouée) :

GrayscaleChars	C++	Python	ShadowBinding	C++	Python
1	0,41	0,45	1	1,27	1,28
2	0,41	0,47	2	1,33	1,22
3	0,42	0,44	3	1,41	1,32
4	0,46	0,49	4	1,39	1,33
5	0,41	0,45	5	1,48	1,2
Moyenne	0,422	0,46	Moyenne	1,376	1,27

HoleDegradation	C++	Python	BleedThrough	C++	Python
1	0,93	0,86	1	3,43	2,83
2	0,83	0,76	2	2,96	2,58
3	1,08	0,87	3	2,91	2,66
4	1,02	0,94	4	2,98	2,75
5	1,08	0,8	5	2,95	2,96
Moyenne	0,988	0,846	Moyenne	3,046	2,756

PhantomCharacter	C++	Python	BlurFilter	C++	Python
1	1,31	1,22	1	1,65	1,22
2	1,31	1,18	2	1,67	1,65
3	1,36	1,44	3	1,65	1,65
4	1,36	1,38	4	1,76	1,57
5	1,29	1,47	5	1,67	1,67
Moyenne	1,326	1,338	Moyenne	1,68	1,552

Les temps des tests ne sont pas à comparer entre deux dégradations car le contenu des fonctions est différent et le nombre d'appel par dégradation varie dans nos tests.

Une des problématiques qui s'est imposée durant le développement du wrapper Python a été la différence de temps d'exécution par rapport au code original. La toute première version de notre wrapper était de l'ordre de 6 fois plus lente. Dans la version actuelle, on peut voir que ce problème n'apparaît plus, le temps Python étant même légèrement inférieur au temps C++ (nous n'avons pas étudié ce phénomène en détail mais étant donné que les fonctions appellent le code C++, et que nous avons réalisé certains essais sans mesurer le temps d'ouverture et de sauvegarde des images, nous ignorons d'où peut provenir ce phénomène). Il n'y aura donc pas de perte de temps sur l'utilisation des fonctions de DocCreator. D'autre part, comme elles sont utilisables dans un script, on réalise bien l'objectif de rendre les démarches de dégradations et de génération de document automatisables dans le but de gagner du temps par rapport à une utilisation avec interface graphique.

Partie 6

Conclusion

Si la mise en oeuvre de notre projet ne présentait pas de complexité algorithmique majeure, il nous a demandé de manipuler de nombreuses technologies différentes : C++, Python, SWIG, NumPy, pip, Lex-Yacc, etc ... ce qui nous a parfois pris du temps à bien intégrer. Nous pensons avoir répondu correctement aux besoins, et cela notamment grâce à des contacts réguliers avec les clients, même si certains besoins auraient pû être développés un peu plus. Nous pensons d'ailleurs qu'il nous aurait été possible de résoudre la plupart de ces points délicats assez rapidement, avec un peu plus de temps.

Le résultat de notre travail est la génération d'une bibliothèque Python à la compilation de DocCreator. Cette bibliothèque contient 6 des 7 fonctions de dégradations d'origine, ainsi que la génération de document. Concernant la facilité d'utilisation, l'interface en ligne de commande permet à un utilisateur sans connaissances particulières en Python de pouvoir appliquer ces fonctions. Bien que l'installation via paquet pip ne soit pas totalement finalisée, l'intégration dans un script est simple et la simplification des démarches d'utilisation a été soignée. Cette simplicité ainsi que la portabilité du projet répond à la demande principale des clients : disposer d'une bibliothèque Python de dégradation et de génération d'image, utilisable par le plus grand nombre, qui puisse permettre des exercices de *data augmentation* pour le Machine Learning.

Bibliographie

- [1] Nicholas Journet, Muriel Visani, Boris Mansencal, Kieu Van-Cuong, and Antoine Billy. Doccreator : A new software for creating synthetic ground-truthed document images. 2017. <http://doc-creator.labri.fr/>.
- [2] François Chollet et al. Keras. <https://keras.io>, 2015.
- [3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow : Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [4] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.
- [5] Thomas Breuel. ocodeg : a small python library implementing document image degradation for data augmentation for handwriting recognition and ocr applications., 2018. <https://github.com/aleju/imgaug>.
- [6] kmader. Augmentor : an image augmentation library in python for machine learning, 2016. <https://github.com/mdbloice/Augmentor>.
- [7] Alexander B. Jung. imgaug : This python library helps you with augmenting images for your machine learning projects. it converts a set of input images into a new, much larger set of slightly altered images. <https://github.com/aleju/imgaug>, 2018. [Online; accessed 21-Mar-2018].
- [8] beazley and wsfulton. Swig : A code generator for connecting c/c++ with other programming languages, 2008. <http://www.swig.org/> <https://sourceforge.net/projects/swig/>.
- [9] Steven J. Bethard. argparse : Parser for command-line options, arguments and sub-commands, 2010. <https://docs.python.org/2/library/argparse.html?highlight=argparse#module-argparse>.
- [10] Silicon Graphics. Opengl : The industry's foundation for high performance graphics, 1992. <https://www.opengl.org/>.

- [11] Khronos Group. Egl : an interface between khronos rendering apis such as opengl es or openvg and the underlying native platform window system., 2003. <https://www.khronos.org/egl>.
- [12] NumPy Developers. numpy.i : a swig interface file for easier type conversions, 2005. <https://github.com/numpy/numpy/blob/master/tools/swig/numpy.i>.
- [13] Messmer Peter and Fogal Tom. Nvidia developer blog : Linking opengl for server-side rendering, egl eye : Opengl visualization without an x server, 2016, 2017. <https://devblogs.nvidia.com/linking-opengl-server-side-rendering/>, <https://devblogs.nvidia.com/egl-eye-opengl-visualization-without-x-server>.

Annexe A

Informations pratiques

- Lien de la page du sujet : http://dept-info.labri.fr/~narbel/PdP/Subjects18-19/pdp_biblio_mansencaljournal.html
- Lien du dépôt Git (Savane) : <https://services.emi.u-bordeaux.fr/projet/git/pdp-bfhl>

Annexe B

Compilation & Dépendances

B.1 Dépendances

Cette partie liste les différentes dépendances nécessaires aux différentes composantes du projet.

B.1.1 DocCreator

Les dépendances d'origine de DocCreator sont toujours nécessaires à la compilation :

- libopencv-dev
- qtbase5-dev
- qtdeclarative5-dev
- libqt5xmlpatterns5-dev
- qt5-default
- cmake

Les dépendances suivantes sont optionnelles (les modules seront ajoutés à la compilation) mais leur installation préalable accélère grandement la vitesse de compilation :

- tesseract-ocr
- tesseractocr-fra
- libtesseract-dev
- libleptonica-dev

B.1.2 Dépendances ajoutées

Notre projet nécessite quelques dépendances supplémentaires pour fonctionner correctement :

B.1.2.1 Bibliothèque Python

La bibliothèque est constituée via le wrapper SWIG, en Python, les dépendances suivantes sont donc maintenant nécessaires :

- python
- python2.7-dev (ou python3-dev)
- numpy
- swig

D'autre part, pour que les instructions SWIG puissent être intégrées dans un *CMakeLists.txt*, il faut que la version de CMake soit 3.8+.

B.1.2.2 Interface en ligne de commande

La partie interface en ligne de commande utilise la bibliothèque Python wrappée, ainsi que le module *Python Lex-Yacc* (ply), qui doit donc être installé préalablement. La CLI lit et sauvegarde des images à l'aide du module opencv-python, qui doit donc être installé.

B.1.2.3 Tests

Comme la CLI, les tests lisent et sauvegardent des images à l'aide du module opencv-python, qui doit donc être installé pour que les tests s'exécutent sans erreur.

B.2 Compilation

La génération du wrapper Python se fait à la compilation de DocCreator. Sur Linux et Mac il est possible d'utiliser le script que nous avons prévu à cet effet :

```
$ ./compile-project.sh <MyInstallationPrefix>
```

Les fichiers du wrapper seront générés dans le dossier *wrapper* du dossier de compilation (*build*).

B.3 Installation

Le processus d'installation est optionnel, il s'agit juste de copier les fichiers nécessaires à l'utilisation de DocCreator dans un repertoire choisi.

B.3.1 A la compilation

Le script *compile-project.sh* installe DocCreator dans le dossier <MyInstallationPrefix> spécifié. Les fichiers du wrapper Python et de l'interface en ligne de commande seront copiés dans */<MyInstallationPrefix>/libpython*

B.3.2 Installation via *pip*

La bibliothèque Python et l'interface en ligne de commande peuvent être installées via *pip*. Cependant à ce jour, seule la version Ubuntu 16.04 est sur le dépôt *pip*.

Annexe C

Pistes sur *Distortion3DModel*

Bien que nous n'ayons pas implémenté comme prévu la dégradation *Distortion3DModel* (que ce soit son refactoring ou le wrapper), nous avons cherché des pistes qui pourraient permettre à l'avenir son ajout à la bibliothèque Python. Les différents points sont les suivants :

C.1 Refactoring

Le refactoring des fichiers de la dégradation. En effet, avant de commencer le wrapping ou tout autre manipulation, il aurait été nécessaire de déplacer les 1765 fichiers de *Distortion3DModel* (répartis en 148 dossiers ou sous-dossiers) du répertoire */software* vers le répertoire */framework*, tout en s'assurant que la dégradation fonctionne toujours aussi bien pour la bibliothèque Python et la version originale de DocCreator.

C.2 OpenGL ES

Le passage vers OpenGL ES. Pour pouvoir utiliser des fonctions OpenGL dans un environnement sans fenêtre, il est nécessaire d'utiliser un outil tel qu'EGL (voir ci-dessous). Cependant, EGL ne fonctionne qu'avec la version "légère" (et plus portable) d'OpenGL, à savoir OpenGL ES. Il aurait donc fallu étudier le code de DocCreator pour évaluer la possibilité de remplacer les fonctions OpenGL, si besoin, pour restreindre la dépendance à OpenGL ES.

C.3 EGL

EGL permet de prendre en charge différents contextes graphiques. Dans notre cas, il aurait permis d'exécuter des fonctions OpenGL sans fenêtre, permettant ainsi à un utilisateur travaillant sans interface graphique de tout de même pouvoir utiliser cette dégradation d'image. Pour cela, il est nécessaire de déclarer un nouveau contexte avant l'appel des fonctions OpenGL :

```

// 1. Initialisation d'EGL
EGLDisplay eglDpy = eglGetDisplay(EGL_DEFAULT_DISPLAY);

EGLint major, minor;

eglInitialize(eglDpy, &major, &minor);

// 2. Selection de la configuration voulue
EGLint numConfigs;
EGLConfig eglCfg;
eglChooseConfig(eglDpy, configAttribs, &eglCfg, 1, &numConfigs);

// 3. Creation de la surface
EGLSurface eglSurf = eglCreatePbufferSurface(eglDpy, eglCfg,
    pBufferAttribs);

// 4. Lien vers l'API
eglBindAPI(EGL_OPENGL_API);

// 5. Creation d'un nouveau contexte
EGLContext eglCtx = eglCreateContext(eglDpy, eglCfg,
    EGL_NO_CONTEXT, NULL);
eglMakeCurrent(eglDpy, eglSurf, eglSurf, eglCtx);

/* Appel des fonctions OpenGL
    ...
*/

// 6. Destruction du contexte OpenGL (en fin de programme)
eglTerminate(eglDpy);

```

Attention, nous n'avons pas écrit ce code et il ne nous appartient pas, il provient en intégralité du NVIDIA Developer Blog[13] (liens en bas de page¹ et dans la bibliographie). Cependant, il pourra servir de base à l'avenir pour réaliser le wrapper de *Distortion3DModel*. D'autre part, il est intéressant de noter que EGL a un support CMake², ce qui facilitera grandement son intégration au projet.

1. <https://devblogs.nvidia.com/egl-eye-opengl-visualization-without-x-server>

2. <https://devblogs.nvidia.com/linking-opengl-server-side-rendering/>

Annexe D

Organisation

D.1 Diagrammes de Gantt

Voici une comparaison entre la planification initiale de notre projet, et le déroulement réel du projet durant le semestre :

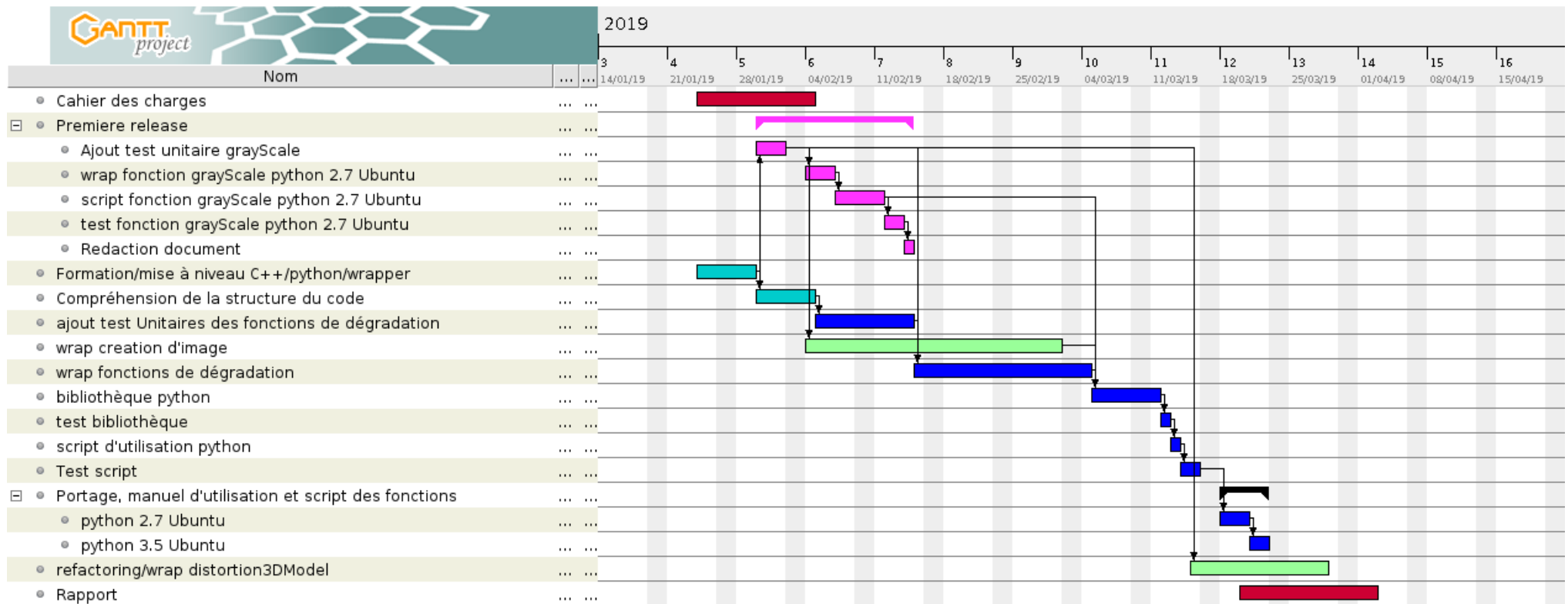


FIGURE D.1 – Diagramme de Gantt tel que prévu lors de la première analyse des besoins

qui est devenu (la portabilité ne figure plus sur le diagramme car c'est une tâche que nous avons géré au fur et à mesure de notre implémentation des différentes fonctionnalités et non pas comme un tâche à part) :

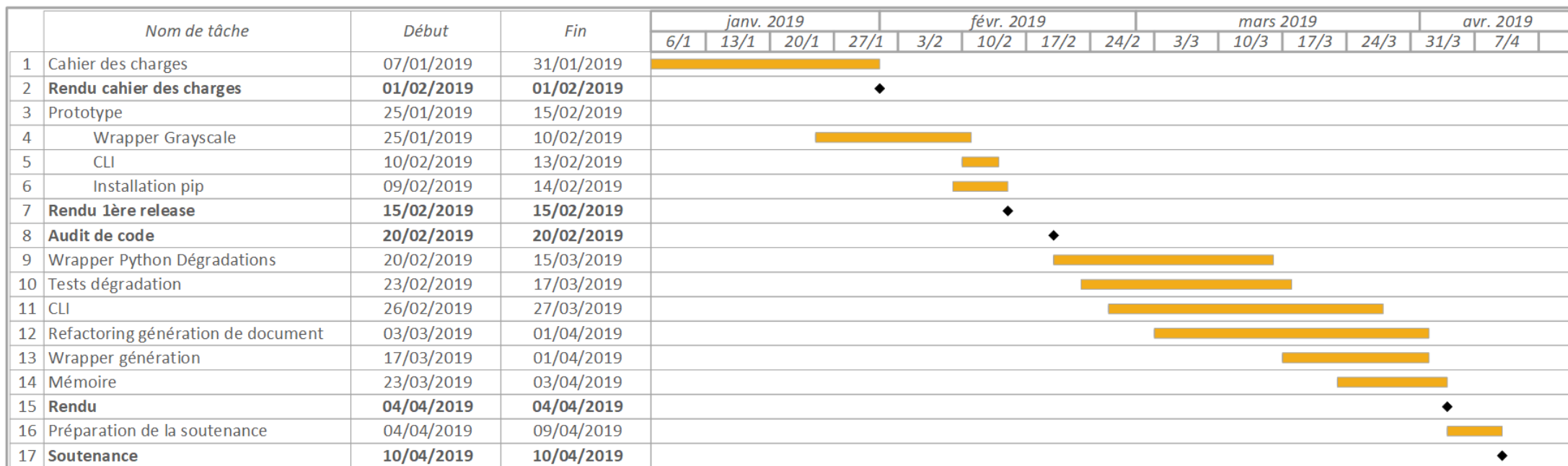


FIGURE D.2 – Diagramme de Gantt de l'organisation durant le semestre

Le constat principal que l'on puisse faire sur les différences présentées est que la plupart des tâches ont pris plus de temps que prévu, mais qu'elles ont également été commencées en parallèle, et plus tôt. D'autre part, l'évolution de l'analyse des besoins fait que le deuxième diagramme de Gantt différencie mieux les tâches effectuées.

D.2 Outils utilisés

Durant notre projet, nous avons utilisé (principalement) deux outils pour travailler en équipe :

- Git : pour le versionning des fichiers
- Trello : pour l'application de la méthode de travail *Kanban*