

Ecole Polytechnique de l'Université François Rabelais de Tours

Département Informatique

64 avenue Jean Portalis

37200 Tours, France

Tél. +33 (0)2 47 36 14 14

polytech.univ-tours.fr

Projet Recherche & Développement
2018-2019

Mapping vidéo 3D

Tuteurs académiques

Gilles VENTURINI

Barthélemy SERRES

Étudiant

Clément GRODECOEUR (DI5)



Liste des intervenants

Nom	Email	Qualité
Clément GRODECOEUR	clement.grodecoeur@etu.univ-tours.fr	Étudiant DI5
Gilles VENTURINI	gilles.venturini@univ-tours.fr	Tuteur académique, Département Informatique
Barthélemy SERRES	barthelemy.serres@univ-tours.fr	Tuteur académique, Département Informatique



Avertissement

Ce document a été rédigé par Clément Grodecoeur surnommé l'auteur.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Gilles Venturini et Barthélemy Serres surnommés les tuteurs académiques.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable des tuteurs académiques et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



Pour citer ce document

Clément Grodecoeur, *Mapping vidéo 3D*, Projet Recherche & Développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

```
@mastersthesis{
  author={Grodecoeur, Clément},
  title={Mapping vidéo 3D},
  type={Projet Recherche \& Développement},
  school={Ecole Polytechnique de l'Université François Rabelais de Tours},
  address={Tours, France},
  year={2018-2019}
}
```

Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	v
Liste des codes sources	vii
1 Introduction	1
1 Contexte de la réalisation	1
2 Objectifs.....	2
3 Bases méthodologiques.....	2
4 Hypothèses	2
2 Description générale	3
1 Environnement	3
2 Utilisateurs.....	3
3 Fonctionnalités du système	4
4 Structure générale du système.....	5
3 État de l'art	7
1 Utilisation du mapping vidéo	7
2 Logiciels existants.....	7
2.1 MadMapper.....	7

2.2	MWM	9
3	Technologies	9
3.1	Projection mapping	9
3.2	UV mapping.....	10
3.3	Texture mapping	11
3.4	Calibration	11
4	Analyse et conception	13
1	Analyse logicielle	13
1.1	Interface	13
1.2	Gestion des données.....	13
1.3	Moteur de rendu.....	14
2	Conception	14
2.1	Interface	15
2.2	Gestion des données.....	16
2.3	Moteur de rendu.....	16
2.4	Conception générale.....	16
3	Changements au semestre 10.....	16
5	Mise en œuvre	20
1	Outils utilisés.....	20
2	Implémentation	20
2.1	Modèle	20
2.1.1	Projecteurs.....	20
2.1.2	Écrans.....	21
2.1.3	Objets 3D.....	22
2.1.4	Contenus	22
2.2	Contrôleur	22
2.2.1	Interaction entre interface et modèle.....	22
2.2.2	Interaction entre projecteurs et écrans	23
2.2.3	Gestion des projets	23
2.3	Moteur de rendu.....	24
2.3.1	Boucle de rendu	24
2.3.2	GLSL	25
2.4	Interface	25
2.5	Tâches non développées	25
3	Difficultés rencontrées	25
3.1	Cinder	26
3.2	Vidéo.....	26
4	Qualité	27

6	Bilan et conclusion	28
1	Bilan du semestre 9	28
1.1	Tâches réalisées	28
1.2	Tâches en retard.....	28
1.3	Planification du semestre 10	28
2	Bilan du semestre 10.....	29
2.1	Tâches réalisées	29
2.2	Tâches en retard.....	29
2.3	Qualité du code	29
	Annexes	30
A	Planification	31
1	Découpage en tâches.....	31
1.1	Prise en main du sujet	31
1.2	État de l'art	31
1.3	Rédaction des spécifications.....	31
1.4	Modélisation.....	32
1.5	Rédaction du rapport (S9).....	32
1.6	Préparation de la soutenance (S9).....	32
1.7	Développement du moteur de rendu.....	32
1.8	Développement du système de calibration	32
1.9	Développement de l'interface	32
1.10	Rédaction du rapport (S10)	32
1.11	Préparation de la soutenance (S10).....	33
2	Diagrammes de Gantt.....	33
2.1	Semestre 9.....	33
2.2	Semestre 10	33
2.3	Semestre 10 rectifié.....	33
B	Spécifications fonctionnelles	35
1	Projection de contenu	35
2	Calibration	35
3	Projecteurs multiples	36
4	Configuration.....	36
C	Spécifications non fonctionnelles	37
1	Contraintes de développement et de conception.....	37
2	Contraintes de fonctionnement et d'exploitation.....	37
2.1	Performances	37
2.2	Ergonomie	37

D Document d'installation	38
1 Bibliothèques	38
2 Environnement de développement	38
3 Exécution	38
E Document d'utilisation	39
1 Lancement du programme.....	39
2 Terminologie.....	39
3 Utilisation du programme	40
3.1 Projet	40
3.2 Objet et contenu.....	40
3.3 Projecteurs et écrans	41
3.4 Propriétés du projecteur	42
4 Format des fichiers de projet.....	42
F Cahier du développeur	44
1 Code du projet	44
1.1 Diagramme de classes	44
1.2 Description des classes	44
1.2.1 WindowApp	44
1.2.2 Controller.....	45
1.2.3 Projector	45
1.2.4 Screen	45
1.2.5 Content	45
1.3 Fonctionnement du programme.....	45
2 Fichiers d'entrée / sortie	45
3 Structure du projet.....	46
G Cahier de tests	47
1 Tests unitaires	47
1.1 Modèle	47
1.2 Contrôleur	47
2 Tests fonctionnels	48
2.1 Contrôleur	48
2.2 Vue	48
3 Données de test	49
Comptes rendus hebdomadaires	52
Webographie	55
Bibliographie	56

Table des figures

1 Introduction

1	Œuvres de Laurence Dreano	1
---	---------------------------------	---

2 Description générale

1	Cas d'utilisation	4
2	Composants	6

3 État de l'art

1	MadMapper	8
2	MWM	8
3	UV mapping	10
4	Texture mapping	11
5	Calibration	12

4 Analyse et conception

1	Interface graphique	15
2	Diagramme de classes : interface	15
3	Diagramme de classes : données	17
4	Diagramme de classes : rendu	18
5	Diagramme de classes complet	18
6	Diagramme de classes revu au semestre 10	19

5 Mise en œuvre

1	Groupement de projecteurs sur un écran	21
---	--	----

2	Aperçu de l'interface.....	26
A Planification		
1	Planification du semestre 9	33
2	Planification du semestre 10.....	34
3	Planification rectifiée du semestre 10.....	34
E Document d'utilisation		
1	Différence entre écrans et projecteurs	40
2	Interface graphique du programme	41
F Cahier du développeur		
1	Diagramme de classes	44
G Cahier de tests		
1	Tests unitaires du modèle	47
2	Tests unitaires du contrôleur	48
3	Tests fonctionnels du contrôleur	48
4	Tests fonctionnels de la vue	48



Liste des codes sources

5.1	Exemple de fichier de projet.....	23
E.1	Exemple de fichier de projet.....	42
G.1	Un projecteur et un écran	49
G.2	Trois projecteurs et un écran.....	49
G.3	Deux projecteurs et deux écrans	50
G.4	Objet et contenu.....	51

1

Introduction

Ce document constitue les spécifications ainsi que le rapport du Projet de Recherche et Développement au sujet du Mapping Vidéo 3D, proposé par Gilles Venturini et Barthélemy Serres et réalisé par Clément Grodecœur à destination de Laurence Dreano.



Figure 1 – *Œuvres de Laurence Dreano*

1 Contexte de la réalisation

Ce projet s'inscrit dans le cadre des travaux de Laurence Dreano ; ses œuvres sont des sculptures de tailles, formes, couleurs et matériaux variés. Pour de prochaines œuvres, de nouveaux concepts ont été imaginés, parmi lesquels un orgue lumineux interactif, une visualisation d'œuvres en réalité augmentée et une projection de poésie sur une sculpture. C'est sur ce dernier concept que seront basés les travaux concernant ce projet. De nombreuses œuvres d'art exploitent des technologies de projection : il s'agit cependant la plupart du temps de projections sur des surfaces

planes, ou bien sur des volumes en exploitant les déformations de l'image projetée. Afin que le projet soit mené à bien, on comptera différents intervenants :

Cliente	Laurence DREANO
MOA	Gilles VENTURINI Barthélemy SERRES
MOE	Clément GRODECŒUR

2 Objectifs

Dans le but de créer une œuvre d'art d'un genre nouveau, nous désirons projeter une poésie animée sur une sculpture. Le texte devra suivre un chemin arbitraire au long des courbes de l'objet.

La projection sur des volumes est différente de la projection sur des surfaces planes : il faut notamment tenir compte des déformations subies par l'image. De plus, il sera nécessaire d'utiliser plusieurs projecteurs, afin de couvrir la plus grande surface possible autour de l'objet. Ensuite, le texte projeté devra rester lisible, tout en conservant un attrait artistique. Enfin, la solution devra pouvoir fonctionner sur une machine de puissance raisonnable.

Afin de projeter des images sur des volumes, il est nécessaire de disposer d'un fichier d'objet 3D représentant l'objet physique, d'une texture à mapper sur l'objet, ainsi que des informations pour le mapping (correspondances entre la texture et l'objet).

Il existe un certain nombre de logiciels permettant ce genre de projection ; cependant, ils ont généralement un coût élevé, ne possèdent pas toujours les fonctionnalités que nous attendons, ou sont parfois dédiés à des usages plus spécifiques. L'objectif de ce projet est alors de réaliser une solution adaptée à ces problématiques.

3 Bases méthodologiques

Afin de mener le projet au mieux, nous gèrerons son évolution à l'aide de la méthode agile, ce qui permettra chaque semaine de produire quelque chose afin d'avoir des retours réguliers et ainsi de s'approcher le plus possible des besoins et des attentes du client. Les tâches et le temps qui leur sera alloué sont décrites à la [Section 1](#) (Annexe A).

La gestion des versions et l'hébergement du code seront pris en charge par la plateforme GitLab. Grâce à cela, nous serons en mesure de garder une trace de tous les changements apportés au code. Cette plateforme propose également des outils d'intégration continue qui permettront d'automatiser les tests.

GitLab propose également un système de gestion de projet, comportant notamment un outil d'organisation de tâches sous forme de cartes à la manière de *Kanbans*. Cela nous permettra d'organiser facilement le projet et de garder à l'esprit les tâches en cours ainsi que celles qui seront prévues plus tard.

4 Hypothèses

Pour ce projet, on suppose que l'on aura à notre disposition un objet 3D représentant la sculpture et contenant des coordonnées UV. On suppose également que le contenu à projeter (vidéo du texte animé) est fourni.

On suppose que Qt et OpenGL seront adéquats pour ce projet. Dans le cas contraire, on se penchera sur d'autres technologies telles qu'OpenFrameworks.

2

Description générale

1 Environnement

Ce projet ne requiert aucun environnement particulier. Nous mettrons en œuvre le langage C++ pour profiter de sa performance et du grand nombre de bibliothèques existantes.

L'interface graphique sera réalisée avec le framework Qt, en raison de sa compatibilité avec de nombreux systèmes et de sa simplicité d'utilisation. Le rendu 3D, quant à lui, sera effectué par OpenGL (grâce à un wrapper inclus dans Qt).

Le système devra utiliser les ressources (processeur, GPU, mémoire) intelligemment pour pouvoir fonctionner sur des machines de puissance raisonnable.

2 Utilisateurs

Le programme final devra être utilisable sans nécessiter de connaissances poussées en informatique. Cependant, la mise en place de l'œuvre et du modèle 3D ainsi que calibration du système nécessiteront des compétences basiques. Le logiciel sera simple d'utilisation, intuitif et ergonomique. La production des fichiers d'objets 3D, des images à projeter ainsi que le mapping des textures sur l'objet restera cependant à la charge de l'utilisateur.

	Connaissances informatiques	Connaissances métier	Utilisation
Artiste	Non	Oui	Définition des éléments tels que l'objet 3D et le contenu à projeter ; calibration du système
Autres utilisateurs (ex : gardien de musée)	Non	Non	Lancement du rendu

3 Fonctionnalités du système

Un diagramme de cas d'utilisation présente les fonctionnalités principales du système.

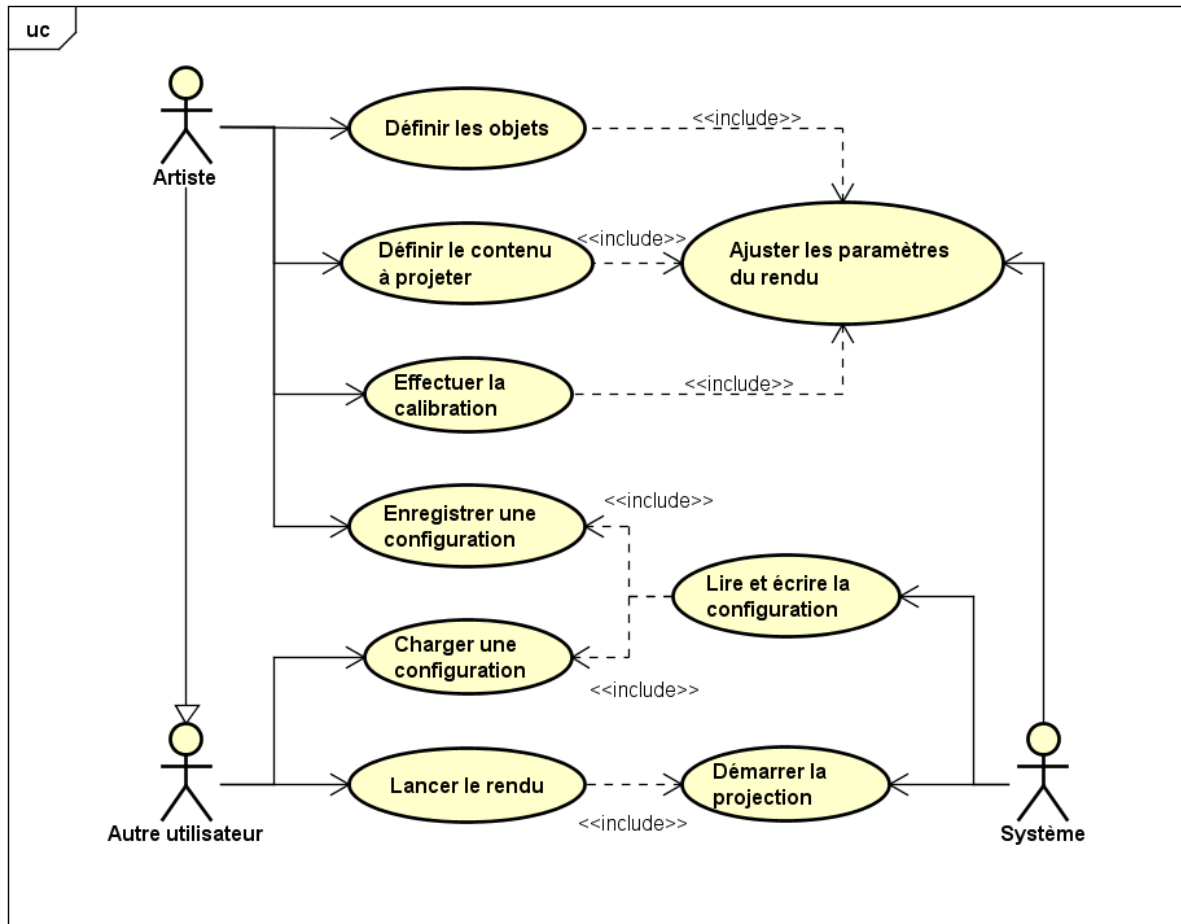


Figure 1 – Diagramme des cas d'utilisation

Les utilisateurs correspondent ici à ceux décrits à la [Section 2](#).

Un utilisateur normal (par exemple, un gardien de musée qui devrait lancer le programme tous les matins) peut effectuer certaines actions de base :

- **Charger un fichier de configuration.** Celui-ci contient les informations nécessaires à la projection : objet 3D, contenu à projeter (image ou vidéo), ainsi que les données de calibration. De cette façon, il n'est pas nécessaire de reproduire tout le travail à chaque lancement du programme.
- **Lancer le rendu 3D.**

L'artiste qui désire projeter un contenu sur un objet peut effectuer des actions plus particulières :

- **Définir les objets 3D.** Il s'agit de fichiers 3D représentant les objets physiques sur lesquels on effectuera la projection.
- **Définir le contenu à projeter.** Il peut s'agir :
 - d'images ;
 - de vidéos.
- **Calibrer le système,** en associant les points remarquables de l'objet 3D à leurs équivalents physiques à travers la projection.

- Ceci fait, l'artiste aura la possibilité d'**enregistrer la configuration** pour pouvoir la recharger plus tard.
- **Lancer le rendu 3D.**

Le système lui-même devra répondre à un certain nombre de critères :

- **Projection de contenu**
Le système doit être capable de projeter un contenu varié selon plusieurs conditions :
 - **Projection sur des surfaces en relief**
Le système effectuera des projections sur des surfaces arbitraires, possiblement avec des reliefs importants et variés. Les images projetées devront donc être adaptées et déformées de façon adéquate afin d'obtenir le résultat attendu.
 - **Projection de texte lisible**
Le but final étant de projeter de la poésie sur les sculptures, le système devra être capable de projeter un texte.
 - **Projection de texte animé**
Dans un but artistique, on recherche une animation dans les textes projetés : le système devra pouvoir s'acquitter de cette tâche.
- **Calibration du système**
Les images projetées devront se superposer au mieux possible à l'objet physique. Dans ce but, le système devra proposer une option de calibration : en positionnant manuellement des points remarquables sur la projection, il sera possible de faire correspondre l'objet et la projection.
- **Utilisation de plusieurs projecteurs**
Si l'on souhaite pouvoir se déplacer autour de l'objet, il sera nécessaire d'utiliser plusieurs projecteurs disposés sous plusieurs angles de l'objet. Le système devra être capable de piloter ces projecteurs, et le résultat devra être propre et net (d'où l'importance de la calibration citée précédemment).
- **Simplicité**
Le système devra conserver une certaine simplicité, pour permettre son utilisation par une personne non nécessairement expérimentée, ainsi que pour pouvoir exploiter des machines de puissance modeste pour la tâche de projection.
- **Configuration**
Le système devra être capable d'enregistrer les paramètres de configuration (constitués de l'objet 3D, du contenu, ainsi que des informations de calibration) et de les restaurer.

4 Structure générale du système

Afin de représenter au mieux la structure générale du système, nous avons réalisé un diagramme de composants (**Figure 2**).

Le projet sera composé de plusieurs éléments principaux :

- **Interface** : il s'agit de l'interface homme-machine du système.
- **Moteur de rendu** : ce composant a pour but de générer et afficher le rendu de l'objet 3D et de son contenu.
- **Gestion des données** : cet élément s'occupe de la gestion des objets 3D, des contenus à projeter et des données de calibration. Le chargement et l'enregistrement des données ainsi que l'étape de calibration sont pris en charge par ce composant.

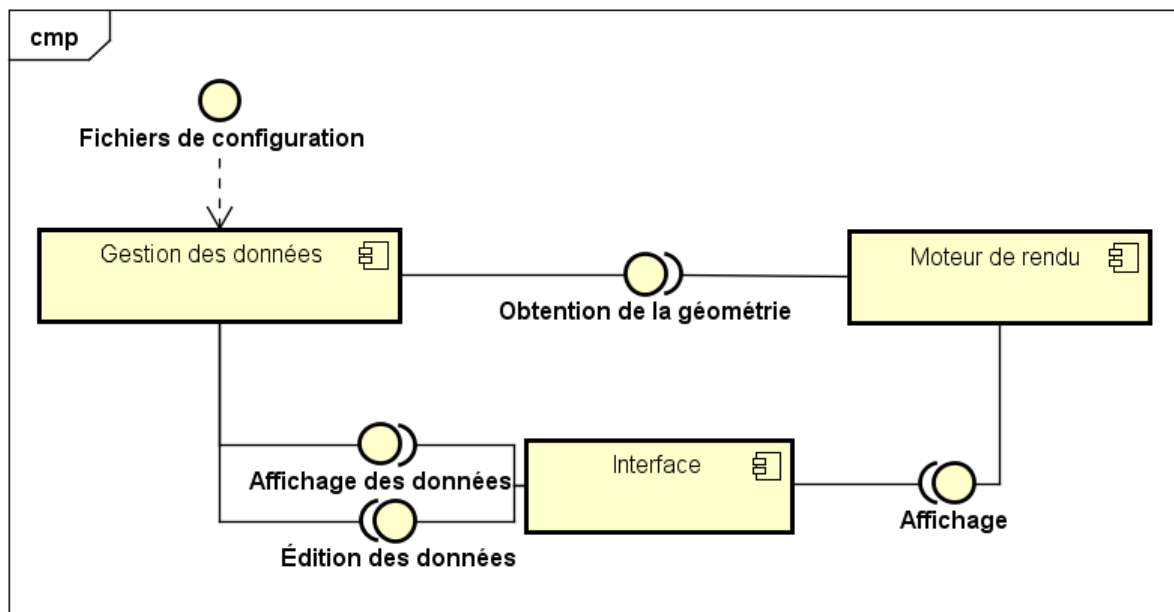


Figure 2 – Diagramme de composants

3

État de l'art

La projection d'images sur des surfaces en relief est une technique relativement répandue, dans des domaines variés. Pour cela, il existe de nombreux outils ayant chacun leurs spécificités.

1 Utilisation du mapping vidéo

Les techniques de mapping vidéo sont utilisées dans de nombreux domaines, notamment artistiques.

Certains VJs (video-jockeys, par analogie aux DJs) choisissent de projeter leurs performances sur des structures complexes, composées de plans, de volumes géométriques ou d'objets. On associe alors par exemple une vidéo ou une animation différente pour chaque surface ; en jouant sur la distorsion de l'image, on peut ainsi donner l'illusion que chaque surface se comporte comme un écran à part entière.

Des façades de bâtiments jouent également le rôle d'écrans géants dans le cadre de performances artistiques, comme par exemple la Fête des Lumières à Lyon où des animations sont projetées sur la cathédrale Saint-Jean, notamment. Dans ce cas, on peut profiter des reliefs des bâtiments et les mettre en valeur, ou au contraire les compenser grâce à la projection ; il est également possible de donner l'illusion d'un relief différent en jouant sur les ombres et sur l'éclairage.

2 Logiciels existants

Pour permettre aux artistes de réaliser leurs performances, il existe un nombre important de logiciels de mapping vidéo [WWW1]. Certains sont plus riches en fonctionnalités, d'autres sont plus adaptés à des utilisations spécifiques. Un certain nombre d'entre eux fonctionneraient tout à fait dans le cadre de ce projet ; nous détaillerons ici les plus adaptés, à savoir **MadMapper** et **MWM**.

2.1 MadMapper

MadMapper (Figure 1) est au départ destiné aux video-jockeys qui désirent projeter du contenu sur des surfaces arbitraires. Il est possible d'afficher des données très diverses :

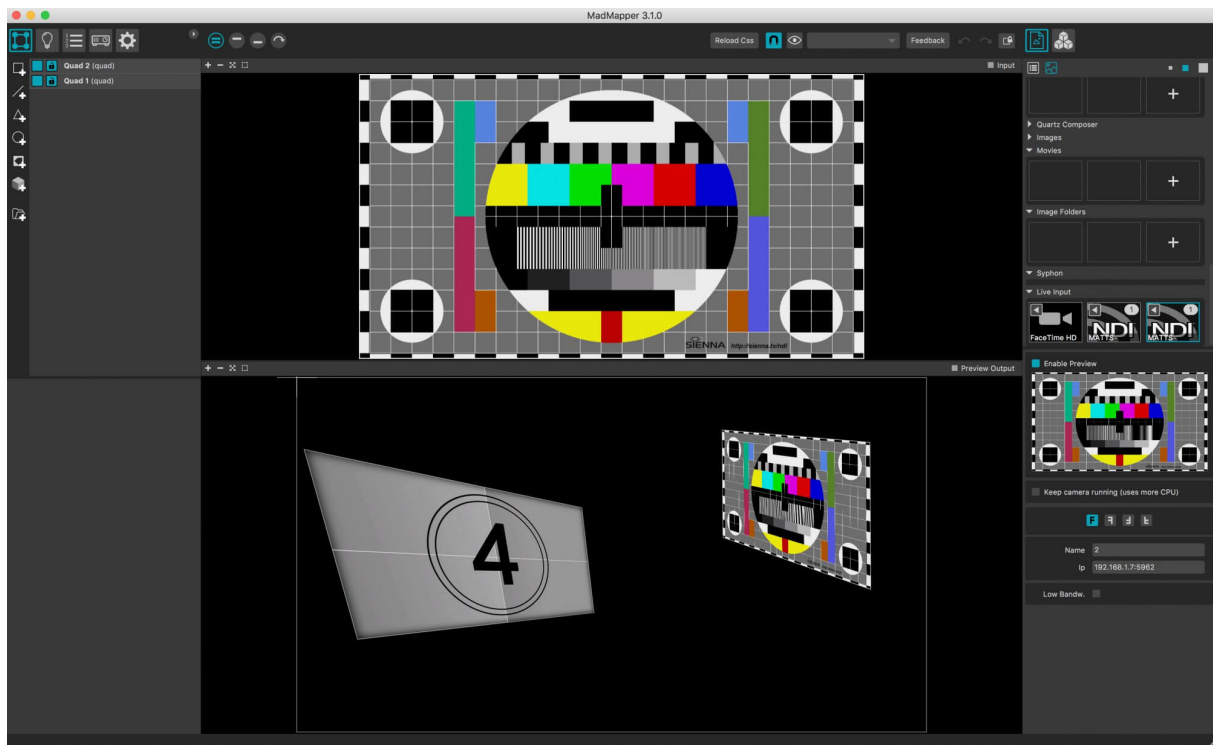


Figure 1 – Aperçu du logiciel MadMapper

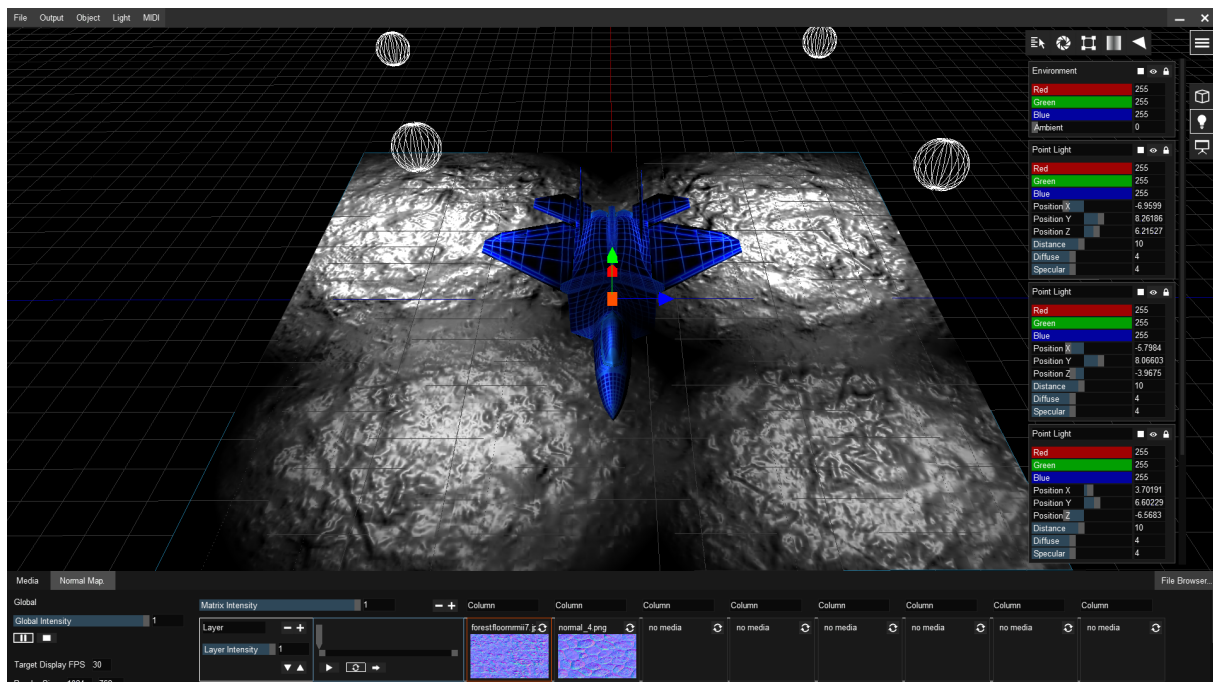


Figure 2 – Aperçu du logiciel MWM

- Images fixes (photos, arrière-plans, textures...);
- Vidéos, qui peuvent être lues en boucle;
- Flux vidéo, en provenance d'une caméra ou d'une autre application;
- Animations générées à la volée (dégradés, particules, bruit aléatoire...);
- Visualisations (onde sonore, VU-mètre) de flux audio.

Une fois sélectionnées, ces images doivent être déformées avant la projection afin de compenser les reliefs des surfaces sur lesquelles on voudra les afficher. Pour cela, il est possible de mapper les images de deux façons.

- Tout d'abord, on peut simplement associer l'image à un plan, pour ensuite régler manuellement l'emplacement de la surface sur la scène physique : dans le cas d'objets simples comportant peu de faces, cela évite d'avoir à sa disposition un objet 3D représentant l'objet physique.
- Deuxièmement, il est possible de mapper l'image sur un objet 3D (au format `.obj`) en utilisant des coordonnées de texture (*UV mapping*) : cela permet d'utiliser des formes bien plus complexes. Le passage de la représentation 3D à la scène physique s'effectue en associant des points remarquables (sommets de l'objet 3D, par exemple) à leur homologue sur l'objet physique.

L'utilisation d'une grande surface ou la nécessité de couvrir plusieurs angles d'un objet nécessitent d'utiliser plusieurs projecteurs. MadMapper permet ceci : en définissant un angle de vue par projecteur, on peut ainsi avoir plusieurs projections qui couvrent un même objet.

L'inconvénient majeur de MadMapper est très certainement son prix élevé : bien qu'étant très complet et offrant un bon rapport qualité-prix, il n'est pas à la portée de toutes les bourses.

2.2 MWM

MWM (Multi-Window Mapper, [Figure 2](#)) est un outil avant tout dédié à la projection sur des objets complexes avec plusieurs projecteurs simultanément. De façon comparable à MadMapper, il permet de projeter du contenu varié : images fixes, vidéos, ou bien flux de provenances diverses. Le mapping peut s'effectuer à la main en associant à chaque face la partie de la texture qui lui correspond. Il est également possible (et même conseillé lorsque les objets ont une géométrie complexe) d'utiliser un mapping UV.

La différence entre MadMapper et MWM se caractérise notamment par la possibilité pour ce dernier d'afficher un rendu dynamiquement lors de l'édition, ce qui permet d'avoir une idée précise du résultat sans avoir besoin de tester réellement. Il dispose également d'un système d'éclairage avancé qui permet de rendre les scènes extrêmement réalistes. L'utilisation massive du GPU rend le programme très performant.

La force de MWM réside dans sa capacité à utiliser plusieurs projecteurs simultanément sur un même objet. Les bords des zones de projection sont prises en compte, et une compensation est effectuée pour que les séparations ne soient pas visibles. Ainsi, l'illusion créée par les projections de ce logiciel est parfaite.

3 Technologies

Les outils de projection vidéo reposent sur des technologies existantes et éprouvées, parmi lesquelles on compte le projection mapping lui-même, mais également l'UV mapping, le texture mapping ainsi que des techniques de calibration.

3.1 Projection mapping

La projection sur des surfaces en 3D est réductible à un simple rendu, affiché directement sur la surface avec un projecteur. En effet, on peut comparer le rendu, où l'on dispose d'une scène comportant une caméra et un objet, à la projection où l'on dispose d'un objet et d'un projecteur. Si les deux scènes (la scène réelle et la scène virtuelle) sont agencées de la même façon (autrement

dit, si les objets des deux scènes ont la même position relative à la caméra et au projecteur) alors l'objet projeté se superpose à l'objet physique.

Grâce à cela, on peut projeter une texture sur n'importe quelle surface, qu'elle soit plane ou non : il suffit d'avoir à sa disposition un objet 3D représentant la surface (ou, si la surface est simple – par exemple de simples plans peu nombreux – de pouvoir la modéliser directement) et de savoir où afficher la texture (dans le cas d'un objet complexe, on utilise le plus souvent des coordonnées UV) pour la projeter sur l'objet physique.

3.2 UV mapping

Lorsque l'on souhaite associer une texture à un objet 3D, on ne peut pas se contenter de la projeter linéairement sur ce dernier : l'irrégularité des différents polygones de l'objet entraînerait une forte déformation sur la texture. Pour compenser cela, on associe à chaque sommet de l'objet les coordonnées correspondantes de la texture. Ainsi, une petite partie de la texture est affectée à chaque polygone de l'objet : chacune de ces parties se retrouve exactement où elle devrait être. Afin d'appliquer correctement la texture sur l'objet, il est nécessaire de disposer des coordonnées de texture pour chaque sommet (**Figure 3**). Pour les obtenir, on "déplie" l'objet pour avoir toutes les faces sur le même plan, généralement connectées entre elles dans la mesure du possible. Ensuite, on associe des coordonnées de texture individuellement à chaque face, de façon à ce que la texture soit adéquatement positionnée sur l'objet. Ce processus semble aisé pour des objets ayant une géométrie simple, mais devient fastidieux dès que le nombre de faces devient important. [\[WWW3\]](#)

Il est possible de mapper des textures différentes avec les mêmes coordonnées UV. Cela permet par exemple d'avoir des détails différents sans changer de modèle, ou dans notre cas, d'avoir une texture animée sur notre modèle.

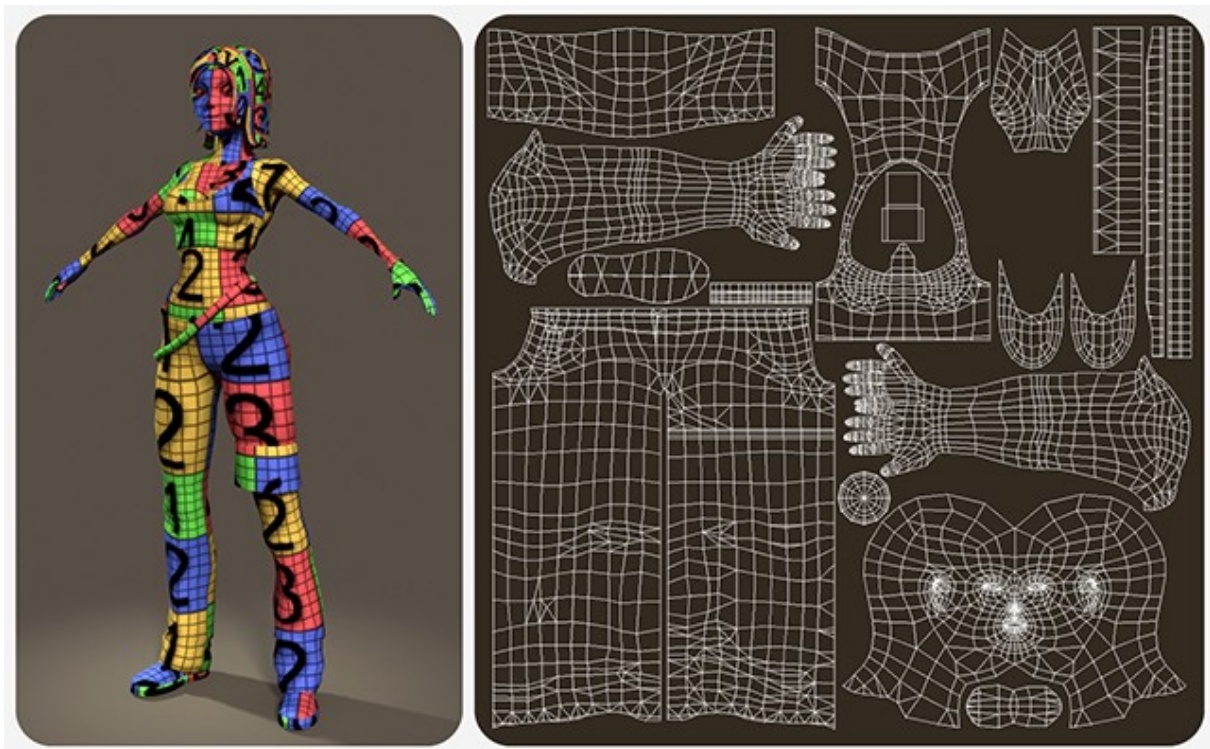


Figure 3 – Le "dépliage" de l'objet 3D permet de plaquer correctement une texture sur celui-ci. Cependant, plus l'objet est complexe et plus cette tâche devient fastidieuse.

3.3 Texture mapping

L'affichage de textures sur des polygones doit prendre en compte certaines contraintes. En effet, la projection d'un triangle texturé sur un écran peut induire des distorsions dues à la perspective : avec certains algorithmes, la texture peut sembler brisée (**Figure 4**). C'est notamment le cas du mapping affine : il s'agit d'une interpolation linéaire des coordonnées de texture entre chaque sommet. Bien que très simple et suffisant dans le cas où les polygones à afficher sont perpendiculaires à l'angle de la caméra, il montre vite ses faiblesses dans les autres cas. Pour remédier à cela et obtenir une projection correcte, il faut prendre en compte la distance des sommets par rapport à la caméra. [WWW2]

Aux débuts de la 3D, la question de l'algorithme à choisir était importante, car un mapping plus précis avait un plus grand impact sur les performances. Il fallait alors faire un compromis entre performances et qualité. Cependant, de nos jours ce travail est effectué en masse par les cartes graphiques directement : nous n'avons donc plus à nous en soucier.



Figure 4 – Les algorithmes existants de texture mapping peuvent donner des résultats différents. Le matériel moderne permet de conserver une perspective correcte sans souci de performances.

3.4 Calibration

La plus grande difficulté lors de la projection sur un objet physique est l'étape de calibration. En effet, pour avoir un rendu correct, l'image projetée et l'objet doivent être superposés le plus précisément possible. Pour ce faire, plusieurs choix sont possibles.

Le premier est simple à mettre en œuvre du côté du système, mais fastidieux en pratique. Il s'agit de reporter manuellement les positions et les angles des objets et des projecteurs, pour reproduire l'agencement des éléments dans la scène 3D. Cela nécessite notamment de tout devoir refaire au moindre changement, même minime. Il faut également disposer des mesures précises des positions et des angles, ce qui peut être délicat à obtenir.

Une deuxième méthode, utilisée notamment par MadMapper, consiste en la définition de points "remarquables" sur l'objet 3D, tels qu'un angle saillant ou un point reconnaissable de l'objet. On vient ensuite faire correspondre ces points à leurs équivalents de l'objet physique grâce à la projection et au placement manuel de points lumineux (**Figure 5**). Une fois l'on a obtenu de cette manière la position de plusieurs points (non coplanaires), il est possible de déterminer l'orientation de l'objet par rapport au projecteur. Il suffit ensuite de positionner la caméra de la même façon dans la scène virtuelle : ainsi, la projection se superpose parfaitement à l'objet. [1] Enfin, certains systèmes disposent d'une caméra et / ou de capteurs de profondeur, dans le but

de déterminer automatiquement la position de l'objet sur lequel on veut effectuer la projection. Bien que beaucoup plus complexe à mettre en œuvre (cela implique notamment de mettre en place un système de reconnaissance de formes), cette méthode offre de multiples avantages : il n'y a donc pas de nécessité de connaître ou de mesurer la position des différents éléments. De plus, le déplacement d'objets n'a aucune conséquence. Des applications peuvent détecter en temps réel les déplacements et ainsi projeter une image sur un objet en mouvement permanent.

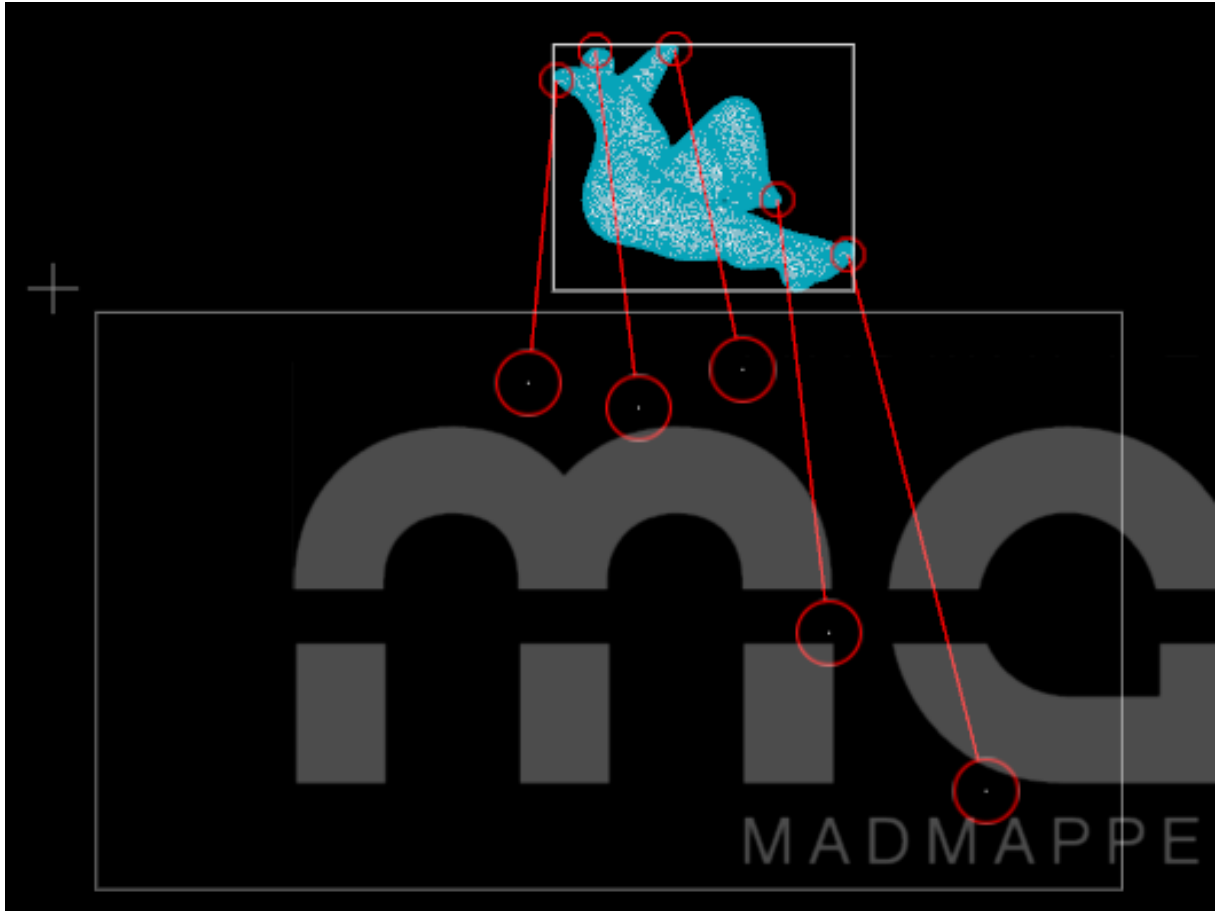


Figure 5 – *La calibration avec MadMapper s'effectue en plaçant manuellement des points remarquables d'un objet 3D dans la zone de projection, pour que ceux-ci correspondent à leurs équivalents sur l'objet réel.*

4

Analyse et conception

Ce chapitre concerne l'analyse logicielle et la conception du projet. Nous allons nous intéresser aux différents éléments le constituant, puis détailler la façon dont ces derniers seront réalisés.

1 Analyse logicielle

Pour mener le projet à bien, nous allons commencer par définir quelles solutions adopter pour chacun des éléments constituant le projet. La structure du projet ([Section 4](#) (Chapitre 2)) définit trois éléments principaux : interface, gestion des données et moteur de rendu.

1.1 Interface

L'interface homme-machine est l'élément avec lequel l'utilisateur final interagit. Il est donc important que cet élément soit ergonomique et simple d'utilisation, tout en permettant un usage efficace.

Afin de mettre cela en œuvre, nous avons décidé d'utiliser Qt. Il s'agit d'un framework écrit en C++ permettant de créer des interfaces graphiques simplement. De plus, Qt fournit un grand nombre de fonctionnalités annexes et génériques, telles que la lecture de fichiers XML ou la gestion du réseau. Ensuite, ce framework dispose d'une vaste communauté et d'une documentation fiable et détaillée : le développement en est donc simplifié. Comme on le verra en [1.3](#), Qt contient également une interface avec OpenGL. Grâce à tout cela, il nous sera possible d'utiliser cette seule bibliothèque pour nos besoins sans avoir recours à d'autres bibliothèques extérieures ; de cette façon, nous sommes certains que les fonctionnalités sont prévues pour fonctionner ensemble, tout en limitant la complexité et la taille du projet.

1.2 Gestion des données

Le programme final devra pouvoir gérer un grand nombre de données de types différents : objets 3D, images, vidéos et fichiers de configuration. Il est donc important de définir quels formats seront acceptés et traités par le système.

Les objets 3D existent sous un grand nombre de formats. Parmi les formats non propriétaires les plus utilisés, deux ont retenu notre attention : le format `.stl` et le format `.obj`. Les deux

décrivent les objets en définissant des faces individuellement. Cependant, le format `.obj` est bien plus léger et permet également d'utiliser un même sommet sur plusieurs faces. En outre, ce format est un standard extrêmement répandu : ainsi, des outils de conversion existant pour de nombreux autres formats. C'est donc celui-ci que nous choisirons.

Les images sont, elles aussi, présentes sous beaucoup de formats différents. Cependant, Qt propose nativement un certain nombre de méthodes permettant de charger les formats les plus usuels : `.bmp`, `.png`, `.jpg`, etc. Cela sera largement suffisant dans notre cas : si l'on désire utiliser un autre format, il suffira de le convertir au préalable pour un format supporté par Qt.

De la même manière, il est possible de passer par Qt pour charger des vidéos. Là encore, les formats supportés sont variés : citons par exemple `.mp4` ou `.mov`. Il peut être important de noter que les fonctionnalités de chargement et de lecture des vidéos ne font pas partie directement de Qt, mais sont dépendantes du système ; cela implique que les formats supportés et les implémentations peuvent varier d'une machine à l'autre. Cependant, cela ne devrait pas poser de problèmes pour les formats les plus courants sur des systèmes récents.

Le choix du format pour les fichiers de configuration ne tient qu'à nous : ces fichiers n'auront pas pour but d'être lus par un programme externe. Nous choisirons un format libre, structuré et pouvant être lisible et éditable par un humain dans la mesure du possible. Deux choix principaux s'offrent à nous : les formats XML et JSON. Ces deux formats sont pris en charge par Qt, qui propose des fonctionnalités pour les lire, les écrire et les modifier de façon simple. Le format JSON a l'avantage d'être bien plus léger : c'est donc celui que nous choisirons.

1.3 Moteur de rendu

La partie la plus importante du projet consiste certainement en l'étape du rendu de l'objet 3D. Pour cela, plusieurs options sont possibles. La première serait d'utiliser une bibliothèque bas niveau telle qu'OpenGL ou DirectX ; la seconde consisterait en l'utilisation d'une bibliothèque de plus haut niveau.

L'avantage indéniable que possèdent, par nature, les bibliothèques haut niveau sont leur plus grande simplicité d'utilisation. Cela n'est cependant pas sans conséquences : ce qui est gagné en simplicité est souvent perdu en flexibilité.

Les bibliothèques bas niveau, quant à elles, sont certes plus complexes et demandent un apprentissage un peu plus poussé, mais permettent d'avoir un contrôle presque total sur les rendus effectués. Comme nous prévoyons de gérer des cas qui ne sont pas forcément très conventionnels (notamment l'utilisation d'une vidéo en tant que texture), la flexibilité est un critère majeur de sélection.

Deux bibliothèques bas niveau sont dignes d'intérêt : OpenGL et DirectX. Ce dernier étant développé et maintenu par Microsoft, il n'est donc utilisable que sous Windows de façon native ; ce n'est pas ce que nous recherchons. OpenGL est une alternative libre et open source à DirectX : tout en possédant les mêmes atouts, cette bibliothèque est disponible sur la plupart des systèmes. De plus, Qt dispose de fonctionnalités ainsi que de fonctions wrappers pour intégrer aisément OpenGL à une interface graphique. C'est donc cette bibliothèque que nous choisirons pour ce projet.

2 Conception

Une fois que nous avons déterminé les différentes technologies à utiliser, nous pouvons nous pencher sur la conception du système.

2.1 Interface

L'interface graphique est le premier – et le seul – élément que l'utilisateur peut voir et avec lequel il peut interagir. Il est donc important qu'elle soit ergonomique et simple d'utilisation.

Étant donné qu'il y a peu d'informations à gérer, on peut se permettre d'utiliser une seule fenêtre principale pour le corps de l'application (excluant donc le rendu 3D). Nous aurons donc à notre disposition deux éléments permettant de choisir respectivement le fichier d'objet 3D à utiliser et le fichier de contenu à projeter (image ou vidéo). Nous afficherons également un aperçu de l'objet et son contenu. Nous avons prévu de pouvoir visualiser et modifier les données de calibration, afin d'avoir un contrôle plus précis lors de cette étape.

Le rendu 3D s'effectuera dans une fenêtre séparée, pour profiter des fonctionnalités d'OpenGL. La [Figure 1](#) présente une maquette de l'interface graphique ainsi imaginée.

Le développement suivra le diagramme de classes défini par la [Figure 2](#).

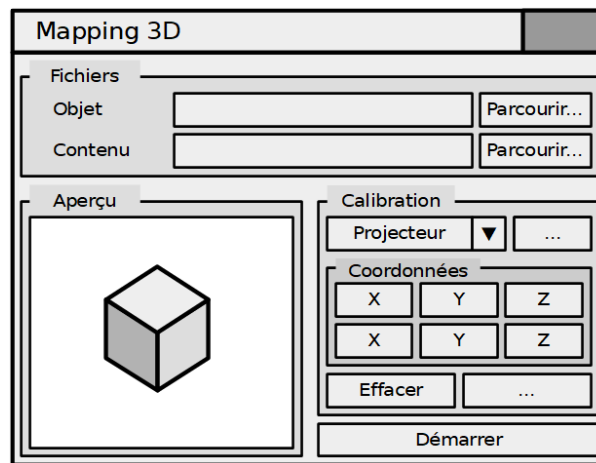


Figure 1 – Maquette d'interface graphique de l'application.

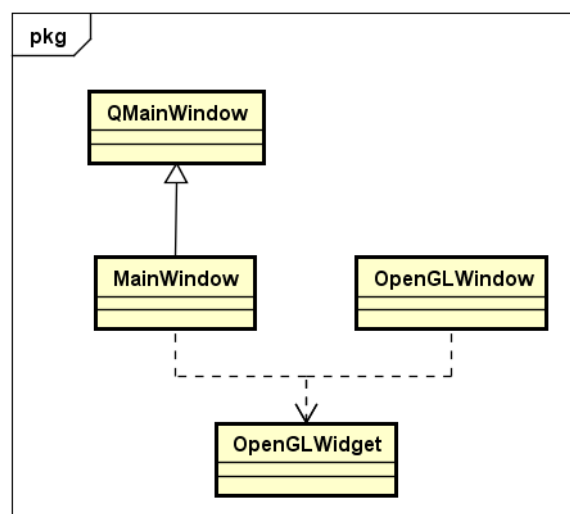


Figure 2 – Diagramme de classes de la partie interface.

2.2 Gestion des données

Qt n'inclut pas de moyen de lire des fichiers au format `.obj`. Il sera donc primordial de développer une fonction dans ce but. Heureusement, la structure de ces fichiers est très simple et ne demande que peu de code pour être interprétée.

On définit un objet 3D par les données suivantes :

- Chaque sommet est un vecteur de 3 flottants représentant chaque coordonnée ;
- L'ensemble des sommets est conservé dans une liste ;
- Chaque face est constituée d'un vecteur de 3 entiers ;
- Chacun de ces entiers représente l'indice du sommet dans la liste susnommée.

Il peut être intéressant de noter que cela correspond précisément à la façon dont sont enregistrés les objets au format `.obj`.

On accèdera aux coordonnées individuelles des points de chaque face grâce à une méthode `getCoord(face, vertex, dimension)`.

Les images et les vidéos seront gérées directement par Qt. Les objets ainsi créés seront stockés tels quels ; lorsque cela sera nécessaire, on extraira les informations brutes dont l'on aura besoin (par exemple, le tableau de pixels représentant une frame d'une vidéo) pour les envoyer à OpenGL.

Afin de pouvoir enregistrer et restaurer les paramètres de la projection, nous utiliserons des fichiers de configuration. Ceux-ci seront au format JSON pour les raisons expliquées en 1.2. Ce fichier contiendra les chemins menant à l'objet 3D et au contenu à projeter, ainsi que les informations de calibration (points remarquables de l'objet 3D et position des différents projecteurs par rapport à celui-ci).

Le développement suivra le diagramme de classes défini par la [Figure 3](#).

2.3 Moteur de rendu

Qt propose des éléments graphiques permettant d'utiliser directement les fonctionnalités d'OpenGL. Le rendu s'effectue comme dans n'importe quelle application OpenGL classique, mais l'image est affichée sur le widget concerné.

Le développement suivra le diagramme de classes défini par la [Figure 4](#).

2.4 Conception générale

Ces éléments sont réunis dans le diagramme de classes défini par la [Figure 5](#).

3 Changements au semestre 10

Au début du semestre 10, nous avons décidé d'utiliser le framework Cinder plutôt que le couple Qt / OpenGL (bien que Cinder utilise OpenGL, nous n'avons pas à l'utiliser "à part"). La raison de ce choix est la capacité de Cinder à produire des applications 3D plus simplement qu'en utilisant directement OpenGL, grâce à une interface simplifiée pour contrôler le rendu ainsi qu'un grand nombre de fonctionnalités pour gérer les objets 3D et les ressources graphiques. Ce choix apporte cependant certains inconvénients : le système d'interface de Cinder est bien plus

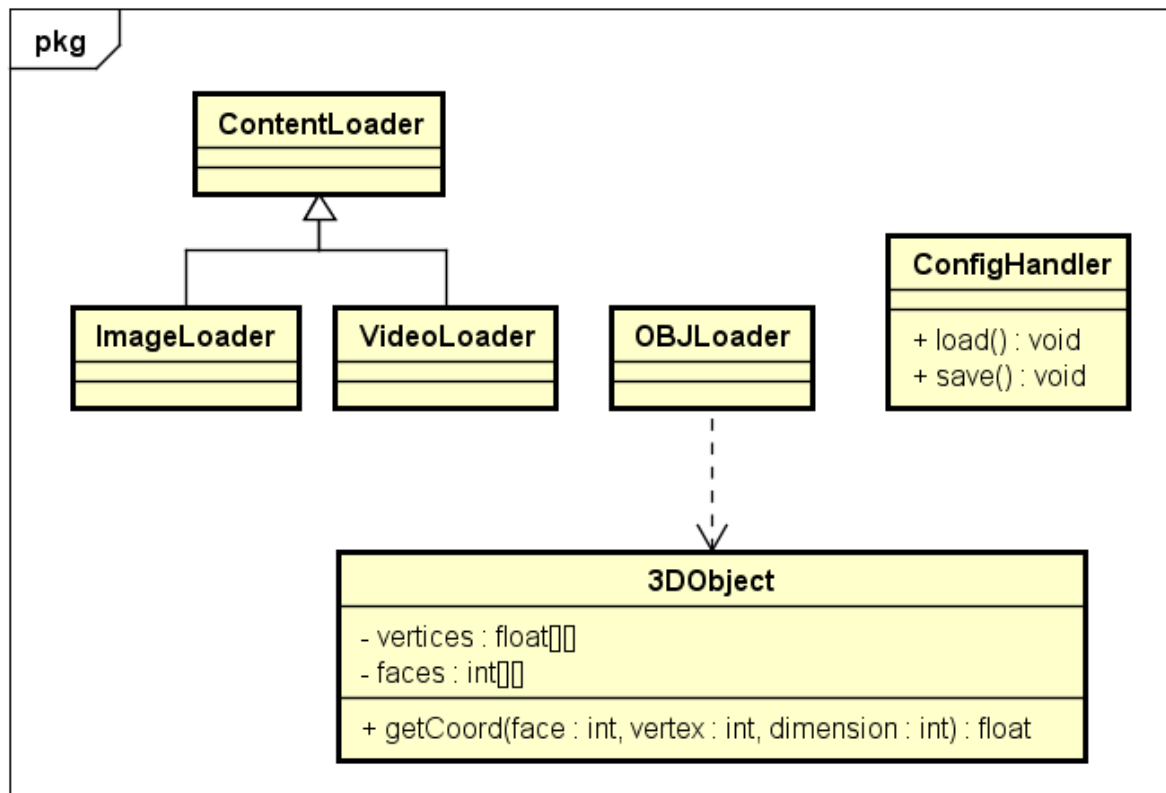


Figure 3 – Diagramme de classes de la partie données.

rudimentaire, et nous ne pouvons pas profiter d'une communauté aussi large que pour Qt ou OpenGL.

La **Figure 6** montre le diagramme de classes revu au semestre 10, tenant compte des changements apportés à la modélisation.

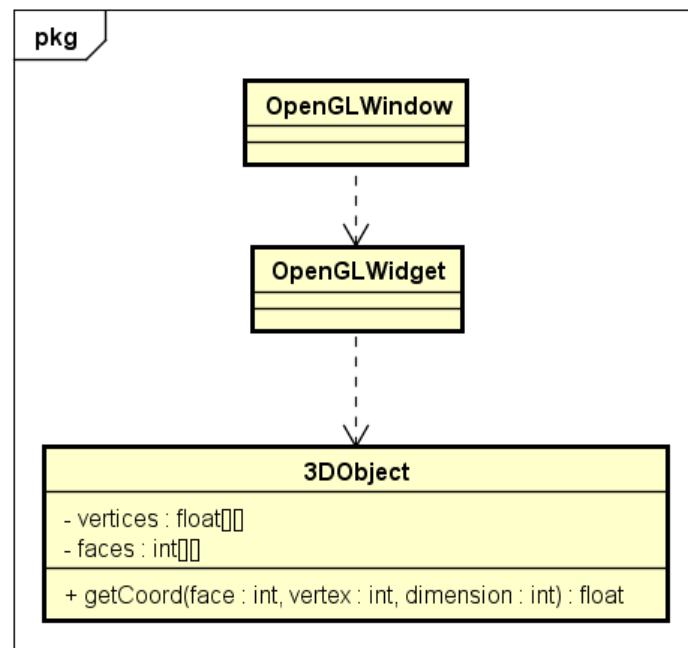


Figure 4 – Diagramme de classes de la partie rendu.

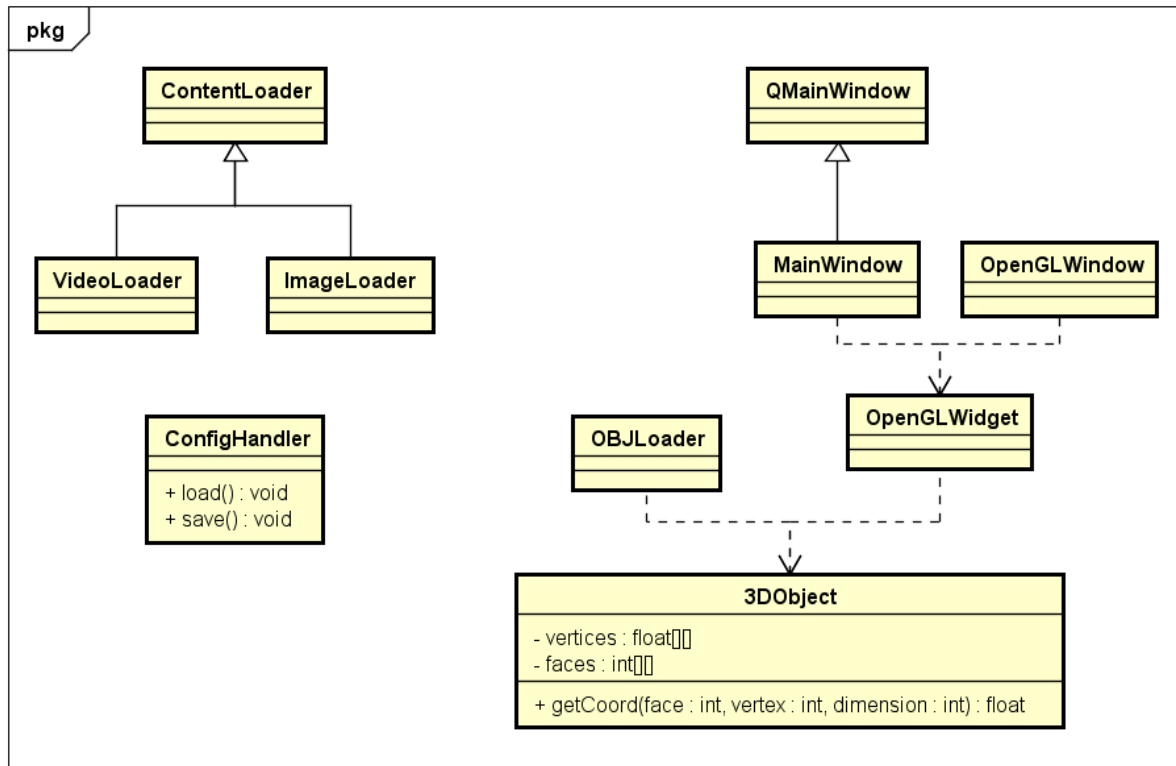


Figure 5 – Diagramme de classes complet.

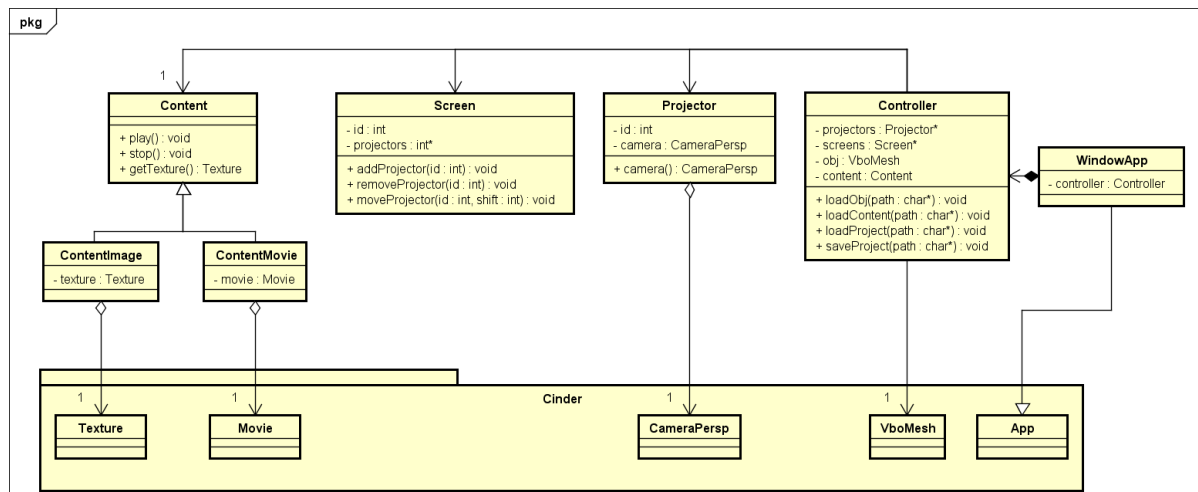


Figure 6 – Diagramme de classes complet, revu au semestre 10.

5

Mise en œuvre

Ce chapitre présente le travail qui a été effectué durant le semestre 10. Il détaille les outils utilisés pour le développement, les choix qui ont été faits tout au long du projet, ainsi que les particularités de l'implémentation. Enfin, nous couvrirons l'aspect qualité du projet.

1 Outils utilisés

Ce projet utilise le framework Cinder. Celui-ci offre un grand nombre de fonctionnalités dont certaines nous intéressent particulièrement : gestion de l'affichage et du rendu 3D, chargement et utilisation d'objets 3D, ainsi qu'une interface graphique. Ce framework est codé en C++ et utilise le runtime Visual C++ 2013 : nous l'avons donc utilisé pour développer notre application. Pour cela, nous nous sommes servis de l'environnement de développement Visual Studio.

2 Implémentation

Notre programme final devra présenter une interface graphique, tout en manipulant des données. Pour cela, l'utilisation du design pattern MVC est appropriée, car celui-ci permet de séparer les données de l'interface pour ainsi avoir un code plus modulaire et maintenable.

2.1 Modèle

Le modèle de notre application regroupera plusieurs éléments : la représentation des projecteurs de la scène, le groupement de différents projecteurs sur un même écran, ainsi que les objets 3D et les contenus à projeter.

2.1.1 Projecteurs

Pour pouvoir effectuer un rendu de l'objet 3D sous l'angle adéquat, nous devons représenter les projecteurs de la scène réelle. Pour cela, nous avons créé une classe **Projector** chargée de contenir la position et l'orientation d'un projecteur, ainsi que la caméra dont se servira OpenGL pour effectuer le rendu lui-même. Pour cela, Cinder propose une classe **CameraPersp** qui dispose

elle-même ce toutes ces propriétés : il est possible de lui donner une position, une orientation, ainsi que de changer d'autres paramètres tels que le champ de vision ou la distance d'affichage maximale, par exemple. La classe **Projector** se compose alors majoritairement de getters et de setters pour avoir accès à cette caméra.

Nous avons également ajouté un nom à notre projecteur, pour pouvoir le reconnaître plus facilement lorsqu'on en utilise plusieurs simultanément. Le nom en lui-même n'a autrement pas d'influence sur le fonctionnement du programme.

Nous avons enfin intégré un identifiant unique, ou ID, à chaque projecteur. Cela permet au programme de pouvoir différencier plusieurs projecteurs sans avoir à garder un pointeur sur chacun d'entre eux ; de plus, cela permet de lier les éléments entre eux lors de l'enregistrement d'un projet (voir [Section 2.2.3](#)).

2.1.2 Écrans

L'utilisation de plusieurs projecteurs soulève une question importante : comment faire dans le cas où l'ordinateur ne posséderait pas suffisamment de sorties vidéo pour piloter tous les projecteurs ? La solution à ce problème est l'utilisation de *splitters vidéo*. Il s'agit de cartes vidéo externes disposant de plusieurs sorties, et qui se connectent sur une sortie vidéo de l'ordinateur. Du point de vue de ce dernier, tous les écrans (ou, dans notre cas, projecteurs) connectés au splitter se présentent comme un seul et même écran, dont la largeur est celle de tous les écrans combinés. Cela nous permettra donc d'afficher une fenêtre en plein écran partie par partie sur les différents projecteurs. Ce comportement est représenté par la [Figure 1](#).

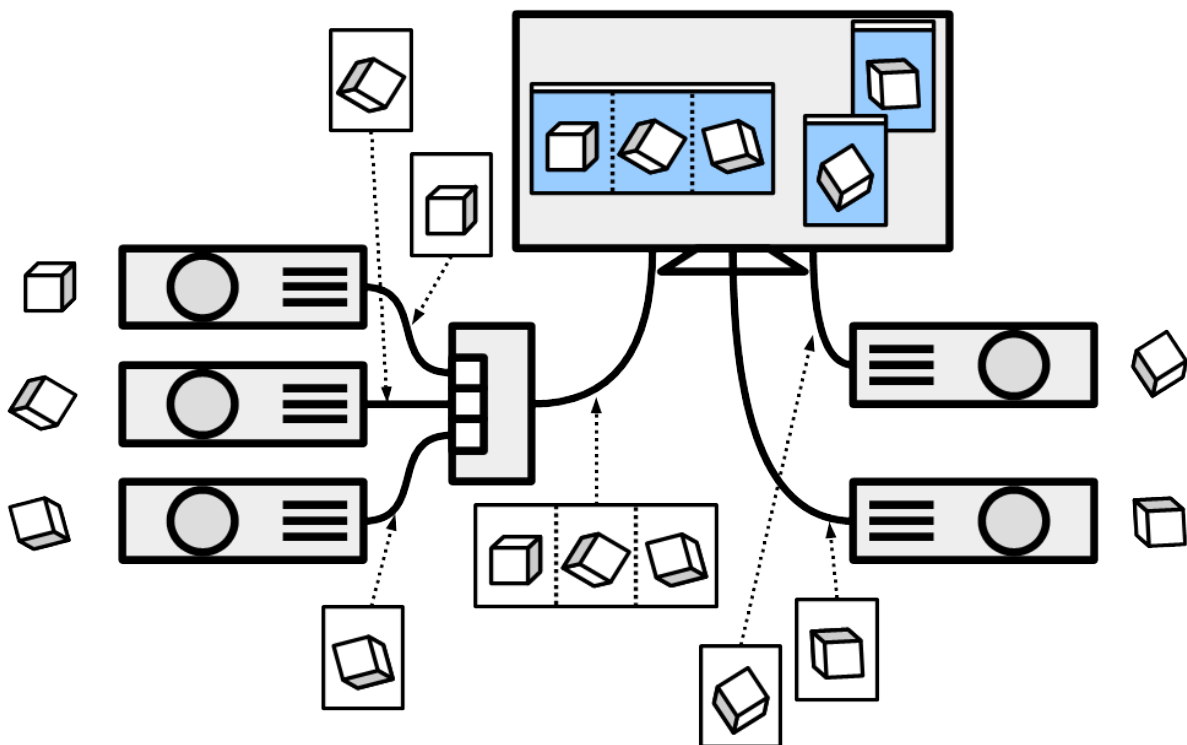


Figure 1 – Sur cet exemple, on considère un projet comportant 5 projecteurs. Comme l'on dispose uniquement de 3 sorties vidéo, on utilise un *splitter vidéo* qui nous permet de brancher 3 projecteurs sur la même sortie. Les deux autres sorties ont un projecteur branché directement dessus. On configure le programme pour afficher 3 projecteurs sur le même écran, c'est-à-dire dans la même fenêtre. Sur ce schéma, on peut voir l'image qui transite par chaque sortie vidéo.

Nous avons donc créé la classe **Screen**, représentant un groupe de **Projectors** à afficher sur le même écran. Celle-ci dispose de méthodes qui lui permettent d'y ajouter et d'en retirer des projecteurs, ainsi que d'en gérer l'organisation, afin de pouvoir changer l'ordre de ce qu'affichent les projecteurs sans avoir à modifier les branchements de l'installation.

À l'instar de la classe **Projector**, notre classe **Screen** possède un nom et un ID pour les mêmes raisons.

2.1.3 Objets 3D

Le rendu ne peut s'effectuer sans une représentation 3D de l'objet réel sur lequel on projettera notre contenu. Pour cela, Cinder propose une classe **VboMesh**. Celle-ci représente l'objet 3D utilisable directement par la carte graphique, avec la particularité d'être stocké dans la mémoire graphique plutôt que dans la mémoire centrale de la machine. Cela permet d'améliorer significativement les performances : étant donné que la géométrie de notre objet ne change pas dans le temps, il n'est pas nécessaire de le copier à chaque frame sur la mémoire graphique.

Comme nous n'utilisons qu'un seul objet 3D dans notre programme et que nous n'avons pas besoin d'informations additionnelles, on utilisera directement la classe **VboMesh** de Cinder plutôt que de créer une classe wrapper.

Les objets **VboMesh** disposent de méthodes pour charger des objets depuis des fichiers au format **.obj** : ce format est très répandu, ce qui rend l'utilisation de notre programme d'autant plus simple.

2.1.4 Contenus

Les contenus à projeter – images et vidéos – sont gérés respectivement par les classes **Texture** de Cinder, et **MovieGl** du bloc Cinder-FFmpeg. Afin d'obtenir une interface transparente dans un cas comme dans l'autre, nous avons créé une classe abstraite **Content** dont héritent **ContentImage** et **ContentVideo**, et contenant respectivement une image et une vidéo. Les deux possèdent des méthodes **play** et **pause** : dans le cas d'une vidéo, leur utilité est évidente et permet de démarrer et de stopper la vidéo ; dans le cas d'une image, ces méthodes activent ou non son affichage.

Ces deux classes disposent elles aussi de méthodes permettant le chargement de fichiers. Les formats d'images acceptés sont variés, la plupart des formats les plus courants sont reconnus (**.jpg**, **.png**, **.bmp...**). Concernant la vidéo, FFmpeg permet également de lire les formats plus courants (**.mp4**, **.mov**, **.avi...**).

2.2 Contrôleur

Le contrôleur est en quelque sorte le cœur de notre programme : il s'occupe de faire le lien entre l'interface et le modèle, dont il stocke les objets.

Les projecteurs et écrans sont stockés dans deux listes séparées. L'accès aux éléments s'effectue la plupart du temps en spécifiant l'index du projecteur ou de l'écran dans la liste.

2.2.1 Interaction entre interface et modèle

Le MVC permet une séparation totale entre l'interface et le modèle. Cela implique de devoir intégrer des méthodes au contrôleur pour pouvoir manipuler ce dernier au travers de l'interface. Pour cela, le contrôleur expose la plupart des méthodes des **Projectors** avec des méthodes wrappers acceptant l'index du projecteur dans la liste. Les **Screens** sont traités de la même façon. À cela s'ajoutent des méthodes de chargement des contenus et des objets 3D à partir de chemins

de fichiers.

L'ajout d'un nouveau projecteur crée un projecteur placé aux coordonnées (10;0;0) et orienté vers le centre du repère. Cela permet de voir directement l'objet 3D si celui-ci est centré et de taille raisonnable (autrement, il suffit de déplacer le projecteur pour qu'il se trouve au bon endroit). L'ajout d'un nouvel écran crée un écran vide, c'est-à-dire ne contenant pas de projecteur. Comme chaque écran correspond à une fenêtre de l'application, l'écran 0 est spécial : il est interdit de fermer sa fenêtre associée¹, et de le supprimer.

2.2.2 Interaction entre projecteurs et écrans

L'organisation des projecteurs sur les différents écrans est gérée par le contrôleur. Ainsi, ni le modèle ni la vue n'ont besoin de s'en occuper.

Le contrôleur offre donc différentes méthodes dans ce but : il est possible de connaître l'écran sur lequel un projecteur sera affiché et sa position sur cet écran. On peut également changer l'ordre d'affichage des projecteurs sur un écran, et également déplacer un projecteur d'un écran vers un autre.

2.2.3 Gestion des projets

Paramétrer une scène peut prendre du temps, particulièrement si l'on utilise plusieurs projecteurs. Heureusement, il n'est pas nécessaire de tout recommencer dès que l'on ferme l'application : il est possible de sauvegarder un projet pour l'ouvrir à nouveau plus tard.

Pour l'enregistrement des projets, nous avons choisi d'utiliser le format JSON. Il s'agit d'un format ouvert et très répandu. Les données sont textuelles et structurées, ce qui rend les fichiers lisibles et éditables par un être humain. Chaque fichier de projet contiendra la liste des projecteurs et toutes leurs caractéristiques (géométrie, nom et ID) ; la liste des écrans et leurs projecteurs associés, ainsi que leur nom et ID ; et enfin, le chemin vers l'objet 3D et le contenu. Comme l'interface est totalement à part, son état (projecteur actif, lecture en cours ou non...) n'est pas enregistré dans le fichier.

Voici un exemple de fichier de projet :

Code source 5.1 – Exemple de fichier de projet

```

1 {
2   "projectors" : [
3     {
4       "name": "Projecteur 1",
5       "id": 0,
6       "show": true,
7       "x": 10,
8       "y": 0,
9       "z": 0,
10      "qx": 0,
11      "qy": 1,
12      "qz": 0,
13      "qs": 0,
14      "fovX": 90,
15      "fovy": 45
16    },
17    {
18      "name": "Projecteur 2",

```

1. Il est interdit de fermer la fenêtre via le contrôleur. La fenêtre elle-même est normale, sa fermeture empêche seulement d'utiliser l'interface, et implique donc la plupart du temps de devoir fermer l'application.

```

19         "id": 0,
20         "show": true,
21         "x": 0,
22         "y": 10,
23         "z": 0,
24         "qx": 1,
25         "qy": 0,
26         "qz": 0,
27         "qs": 0,
28         "fovx": 90,
29         "fovy": 45
30     }
31 ],
32 "screens" : [
33     {
34         "id": 0,
35         "name": "Ecran",
36         "projectors": [
37             0
38         ]
39     },
40     {
41         "id": 1,
42         "name": "Ecran 2",
43         "projectors": [
44             1
45         ]
46     }
47 ],
48 "object": "",
49 "content": ""
50 }

```

2.3 Moteur de rendu

Cinder propose une interface directe avec OpenGL. Grâce à cela, nous sommes en mesure de créer des rendus de façon très souple, mais néanmoins puissante.

Le rendu 3D s'effectue dans chaque fenêtre de l'application. Chacune d'entre elles dispose de l'identifiant de l'écran correspondant, ainsi que le contexte OpenGL de la fenêtre.

2.3.1 Boucle de rendu

Pour chaque frame affichée, nous devons redessiner le contenu de chaque fenêtre. Pour cela, il est important de suivre les étapes dans un ordre précis :

- Récupération des données de la fenêtre ;
- Définition du viewport (zone de dessin d'OpenGL) pour utiliser la fenêtre entière ;
- Effacement de l'image (c'est-à-dire remplissage de la fenêtre en noir) ;
- Récupération de la texture ;
- Ensuite, pour chaque projecteur de l'écran :
 - Définition du viewport en fonction de la position du projecteur à afficher ;
 - Définition des matrices de projection en fonction de la géométrie du projecteur ;
 - Activation de la texture ;
 - Dessin.
- Pour la fenêtre principale, on dessine également l'interface au-dessus du rendu.

2.3.2 GLSL

Le rendu 3D est effectué par la carte graphique. De ce fait, nous devons nous charger nous-mêmes de l'algorithme de rendu. Comme nous utilisons OpenGL, cela s'effectue grâce au langage GLSL ; le programme obtenu est appelé *shader*.

Le rendu 3D se passe en deux étapes, chacune ayant son propre shader. La première, à laquelle est associé le *vertex shader*, est effectuée pour chaque sommet de l'objet 3D à rendre. Elle permet d'effectuer des transformations géométriques ; dans notre cas, de telles transformations n'ont pas d'intérêt, notre vertex shader se charge donc uniquement de transférer certaines de ses données au shader suivant : le *fragment shader*. Celui-ci s'occupe de l'affichage à proprement parler : en fonction des coordonnées de texture envoyés par le vertex shader et linéairement interpolées pour chaque pixel de l'image de sortie.

2.4 Interface

Pour pouvoir interagir avec la représentation de la scène 3D (gérer les projecteurs ou les écrans, ainsi que les objets 3D et les contenus), nous avons besoin d'une interface. Cinder propose une classe `InterfaceGl`, qui permet de créer des interfaces très simplement. Celles-ci sont plutôt rudimentaires, mais suffisent pour être utilisables.

Deux types de contrôles se distinguent dans leur fonctionnement :

- Les **boutons** permettent d'appeler une fonction dès que ceux-ci sont pressés. Nous les avons notamment utilisés pour l'ouverture et l'enregistrement de fichiers, afin d'ouvrir les boîtes de dialogue correspondant à ces actions.
- Les **paramètres** permettent de relier un contrôle de l'interface directement à une variable. Le type de variable utilisé définit le type de contrôle qui sera affiché (par exemple, un champ de texte pour une chaîne de caractères, un sélecteur de nombre pour un entier ou un flottant, ou même une flèche interactive dans le cas d'un vecteur). Il est également possible de définir une fonction qui sera appelée à chaque changement de la valeur du paramètre : nous avons utilisé ce principe pour actualiser en direct les éléments du modèle.

La [Figure 2](#) donne un aperçu en pratique de l'interface de l'application.

2.5 Tâches non développées

Par manque cruel de temps, la partie calibration n'a pas pu être implémentée comme nous l'avions initialement prévu. L'utilisateur peut toujours positionner manuellement les projecteurs de façon à s'approcher au mieux de l'agencement de la scène réelle.

Pour des raisons techniques, l'utilisation de contenus vidéo ne fonctionne pas. Il est cependant toujours possible de projeter des images statiques.

L'utilisation de plusieurs écrans n'est pas encore fonctionnelle. La façon dont OpenGL fonctionne implique de partager des ressources graphiques entre les différentes fenêtres, et cela semble produire des bugs qui provoquent l'arrêt du programme.

3 Difficultés rencontrées

Le développement de ce projet n'a pas été sans un certain nombre de rebondissements.

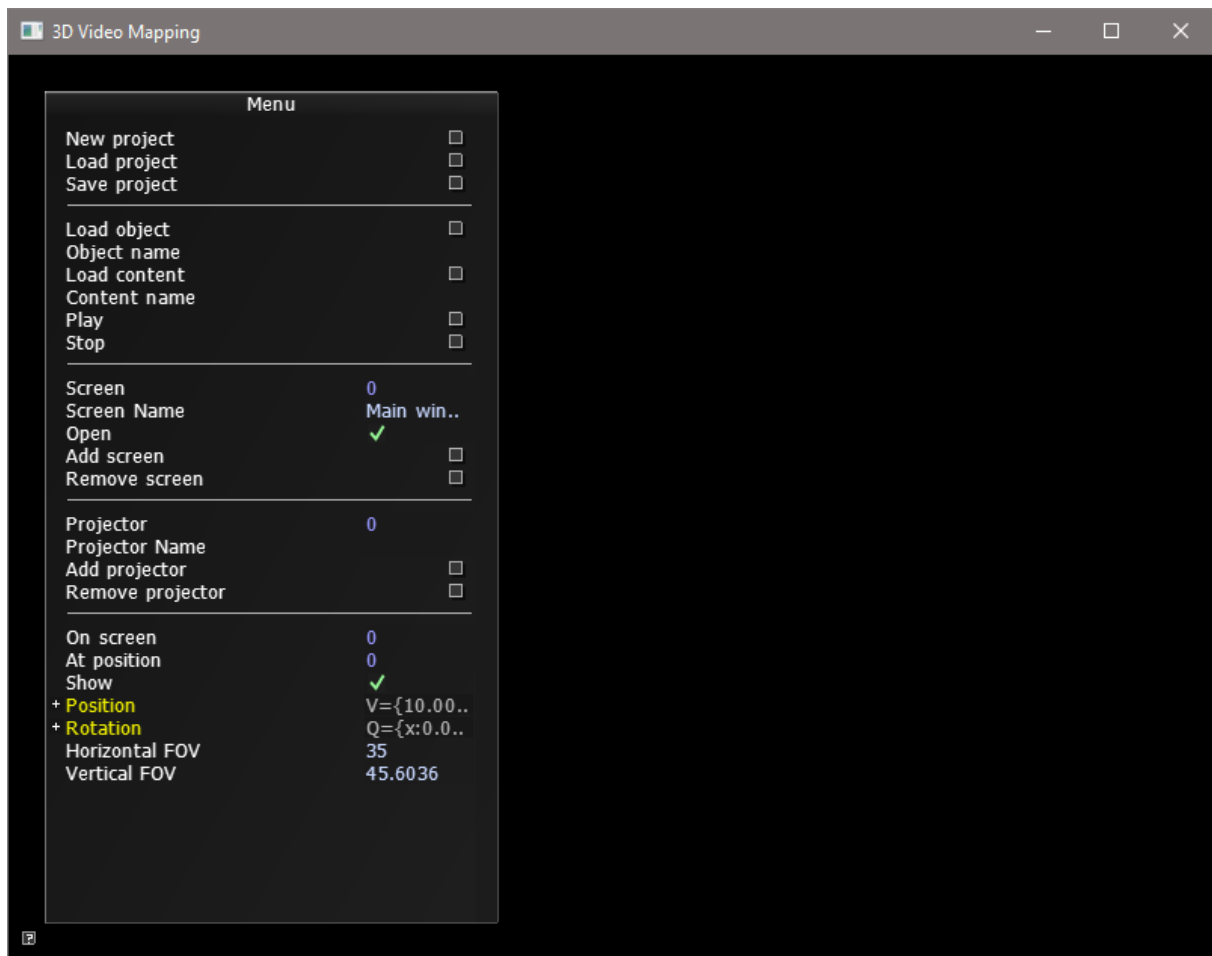


Figure 2 – Aperçu de l'interface de l'application.

3.1 Cinder

Au début du développement, l'installation de Cinder était assez fastidieuse. Ayant déjà installé Visual Studio 2017 auparavant, j'ai tenté d'utiliser Cinder avec celui-ci. Cependant, les bibliothèques statiques fournies sur le site de Cinder étaient compilées avec Visual C++ 2013; les compilateurs de Microsoft n'étant pas compatibles d'une version à l'autre, j'ai tenté de compiler la bibliothèque moi-même à partir des sources. Cela fonctionnait, mais impliquait des efforts trop importants comparés à la simple installation de Visual C++ 2013.

Dans un tout autre registre, il se trouve que Cinder est une bibliothèque qui n'est pas couramment utilisée; elle ne dispose pas d'une communauté aussi importante que d'autres bibliothèques plus connues. L'inconvénient à cela est qu'il est bien moins aisé de trouver réponse à une question lorsque l'on rencontre un problème. De plus, la documentation de Cinder est incomplète et en partie obsolète, ce qui en limite l'utilité.

3.2 Vidéo

Au départ, nous avions prévu d'utiliser le module QuickTime fourni avec Cinder pour la gestion de la vidéo. Ce que nous n'avions pas prévu, c'est que ce module ne fonctionnerait pas avec les versions récentes de Windows, car la bibliothèque QuickTime n'est plus maintenue depuis plusieurs années. Nous avons donc cherché des alternatives à ce module; le créateur de Cinder a

développé un module pour Cinder utilisant FFmpeg pour la gestion de la vidéo. Malheureusement, ce module est encore expérimental, et ne fonctionne pas correctement. À l'heure actuelle, les contenus vidéo ne fonctionnent pas ; peut-être qu'une mise à jour ou une autre bibliothèque permettront à l'avenir d'obtenir un résultat fonctionnel.

4 Qualité

Prévoir la maintenabilité du code et assurer son utilisation par d'autres développeurs implique de prévoir une certaine qualité de code. Pour cela, nous avons effectué plusieurs démarches.

Tout d'abord, l'intégralité des méthodes et des classes est documentée dans le code. L'utilitaire Doxygen permet d'utiliser ces commentaires pour générer une documentation sous plusieurs formats (notamment HTML et PDF). Celle-ci contient la description des méthode, de leurs argument et des données retournées, ainsi que les interactions entre les différentes classes du projet.

Ensuite, le code du projet a été validé par un ensemble de tests. Des tests unitaires sur les méthodes des classes du modèle et du contrôleur ont permis de s'assurer que ces méthodes se comportent comme elles le devraient. De plus, des tests fonctionnels réguliers ont montré que le programme effectue correctement les tâches qui lui sont demandées.

Un ensemble de documents à destination des développeurs et des utilisateurs est disponible en annexe à la fin de ce rapport.

6

Bilan et conclusion

Ce projet s'étend sur les deux derniers semestres du cursus ingénieur à Polytech Tours, et de ce fait se trouve divisé en deux parties correspondantes. La première comprend la phase de spécifications, ainsi que l'analyse du projet et sa conception. La seconde représente la mise en œuvre des solutions imaginées.

1 Bilan du semestre 9

1.1 Tâches réalisées

Au cours de ce semestre, nous avons pu réaliser les tâches suivantes :

- Prise en main du projet
- Analyse du besoin
- Élaboration des spécifications
- Modélisation
- Écriture du rapport

1.2 Tâches en retard

Aucun retard n'est à déplorer ce semestre : au contraire, les essais de développement avec OpenGL constituent une bonne base de départ pour le semestre 10.

1.3 Planification du semestre 10

Le semestre 10 sera principalement constitué du développement de l'application. Pour cela, nous répartirons le travail en plusieurs tâches :

- Développement de l'interface
- Développement du moteur de rendu
- Développement du système de calibration
- Rédaction du rapport du semestre 10

2 Bilan du semestre 10

2.1 Tâches réalisées

Au cours de ce semestre, nous avons pu réaliser les tâches suivantes :

- Développement du moteur de rendu
- Développement de l'interface
- Développement de la logique de l'application
- Rédaction de la documentation
- Développement de tests
- Rédaction du rapport

2.2 Tâches en retard

Des imprévus techniques ont fait accumuler un certain retard au projet. Tout d'abord, le système de calibration a été abandonné, faute de temps pour le réaliser. Ensuite, des problèmes de compatibilité avec Cinder et le non-fonctionnement des modules gérant la vidéo nous ont fait perdre beaucoup de temps, ce qui a fortement limité l'avancement du projet. De ce fait, les contenus vidéo ne sont pas pris en charge. Par manque de temps suite à cela, un bug empêchant l'utilisation de plusieurs fenêtres n'a pas pu être corrigé. Cependant, le projet reste fonctionnel et utilisable en tant que tel.

2.3 Qualité du code

La maintenabilité du code et sa réutilisation sont garanties par un soin important apporté à la qualité du code. Des tests unitaires montrent que les méthodes ont le comportement qu'elles doivent avoir ; la documentation permet de connaître l'utilité et le fonctionnement de tous les éléments du projet ; une série de documents permet aux développeurs et aux utilisateurs de s'imprégner du projet et de l'utiliser simplement ; enfin, le code lui-même est propre et lisible. Les documents en question sont disponibles dans les annexes.

Annexes

A

Planification

La planification s'effectuera majoritairement avec le système de gestion de projet de GitLab. Celui-ci permet, à travers un système de cartes inspiré par les *Kanbans*, de garder une trace des tâches en cours, des tâches prévues et des tâches terminées.

Bien que différente dans sa forme, le fond est très similaire à un diagramme de Gantt : on a effectivement accès à la description de la tâche, le temps passé sur la tâche et la date due. GitLab permet également de définir des catégories ou d'assigner une tâche à une personne en particulier.

1 Découpage en tâches

1.1 Prise en main du sujet

- **Description**

Cette tâche représente la période de prise en main du sujet : découverte des objectifs, interprétation du problème, pistes à explorer.

- **Estimation**

4 jours.

1.2 État de l'art

- **Description**

Cette tâche représente le processus d'étude des logiciels existants et de leurs technologies associées.

- **Estimation**

6 jours.

1.3 Rédaction des spécifications

- **Description**

Cette tâche représente la rédaction du cahier de spécifications.

- **Estimation**

4 jours.

1.4 Modélisation

- **Description**
Cette tâche représente la modélisation et la conception du projet.
- **Estimation**
4 jours.

1.5 Rédaction du rapport (S9)

- **Description**
Cette tâche représente la rédaction du rapport de projet du semestre 9.
- **Estimation**
?

1.6 Préparation de la soutenance (S9)

- **Description**
Cette tâche représente la préparation pour la soutenance de projet du semestre 9.
- **Estimation**
?

1.7 Développement du moteur de rendu

- **Description**
Cette tâche représente le développement du moteur de rendu de l'application.
- **Estimation**
8 jours.

1.8 Développement du système de calibration

- **Description**
Cette tâche représente le développement du système de calibration de l'application.
- **Estimation**
12 jours.

1.9 Développement de l'interface

- **Description**
Cette tâche représente le développement de l'interface graphique du logiciel.
- **Estimation**
6 jours.

1.10 Rédaction du rapport (S10)

- **Description**
Cette tâche représente la rédaction du rapport de projet du semestre 10.
- **Estimation**
?

1.11 Préparation de la soutenance (S10)

— Description

Cette tâche représente la préparation pour la soutenance de projet du semestre 10.

— Estimation

?

2 Diagrammes de Gantt

2.1 Semestre 9

La **Figure 1** montre la planification effectuée au semestre 9.

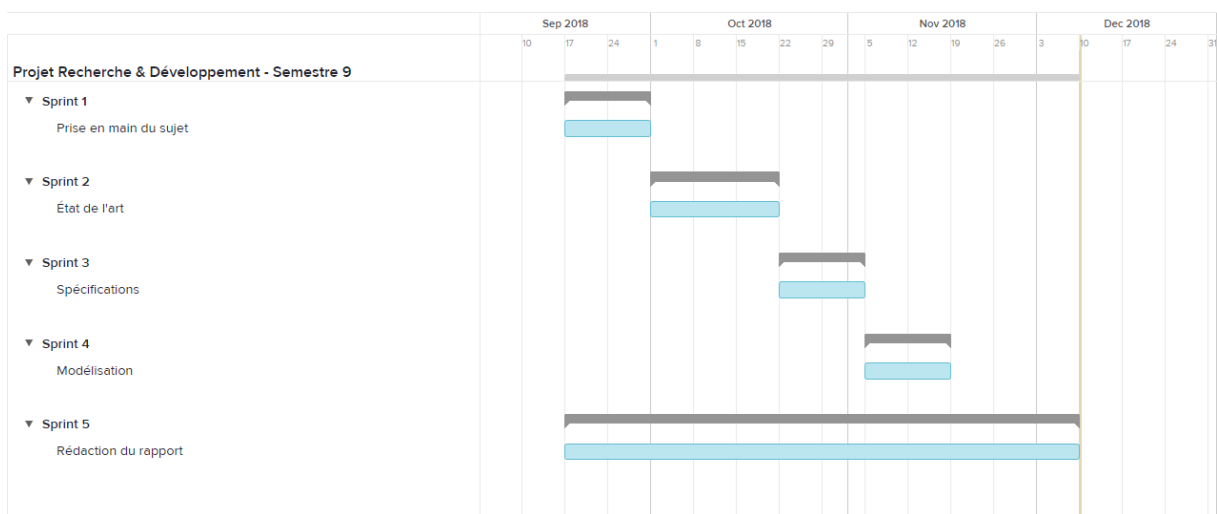


Figure 1 – Planification du semestre 9

2.2 Semestre 10

La **Figure 2** montre la planification prévue au semestre 10.

2.3 Semestre 10 rectifié

La **Figure 3** montre la planification effectuée au semestre 10, après modification. Des changements importants dans les tâches à effectuer et les technologies utilisées ont impliqué une révision de la planification.

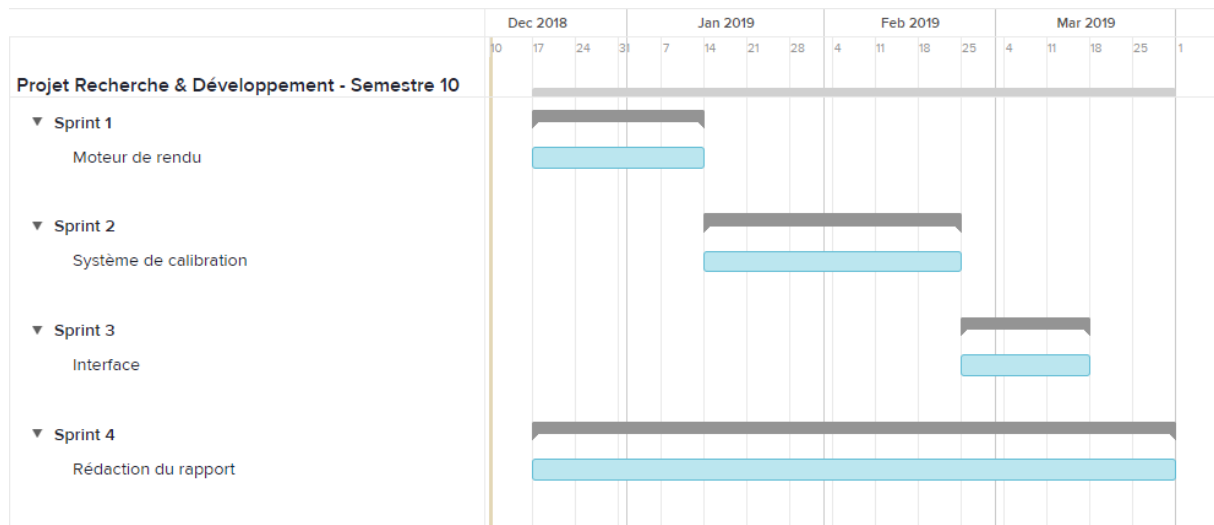


Figure 2 – Planification du semestre 10

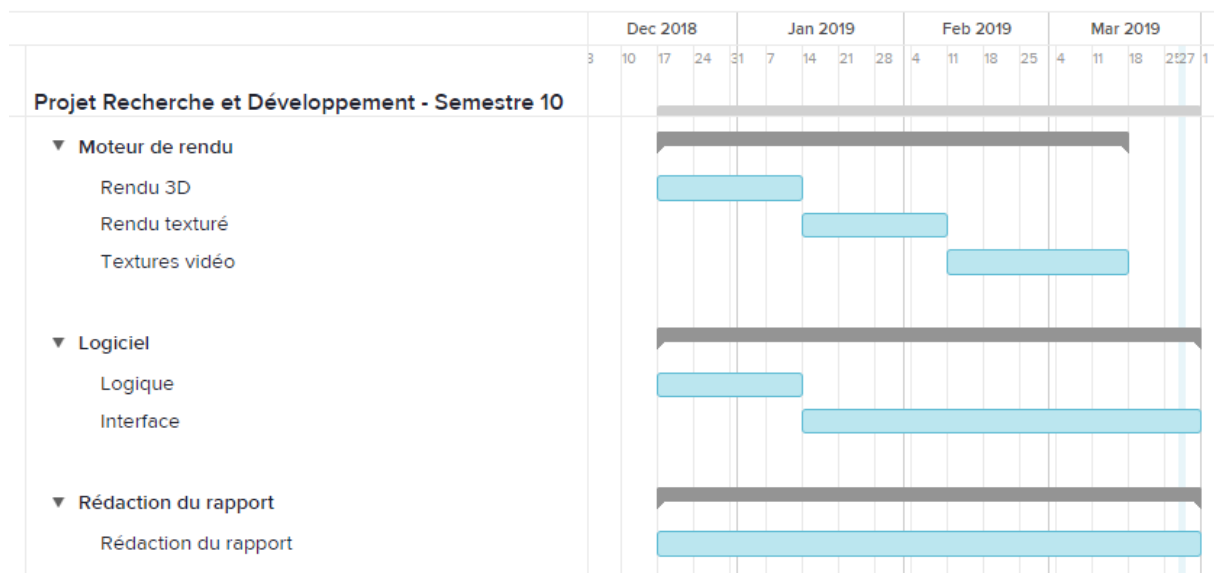


Figure 3 – Planification rectifiée du semestre 10

B

Spécifications fonctionnelles

Au sein de ce projet, on distingue 4 fonctionnalités principales.

1 Projection de contenu

- **Identification**

Projection de contenu (images, textes) sur une surface en relief.

- **Priorité**

Primordiale.

- **Description**

Le système doit projeter des données graphiques (images, textes, vidéos) sur des sculptures, qui peuvent être simplement considérées comme des surfaces non planes (en relief). Pour cela, les images sont préalablement déformées afin de correspondre à l'objet physique une fois projetées.

La qualité de la projection sera suffisante pour qu'un texte soit correctement lisible, même avec les différentes déformations qui lui seront appliquées.

On aura la possibilité de déformer et projeter une vidéo, ce qui permet d'obtenir facilement un texte animé sur la surface de l'objet.

- **Entrées**

Les données à projeter (texte, image, vidéo) ainsi que les objets 3D représentant les surfaces.

- **Sorties**

L'image adéquatement déformée à envoyer directement aux projecteurs.

2 Calibration

- **Identification**

Calibration de la projection.

- **Priorité**

Basse.

- **Description**

Il est difficile de disposer "à la main" un objet 3D dans une scène pour que celui-ci corresponde parfaitement à l'objet physique une fois projeté. La calibration permet de faire correspondre manuellement des points remarquables (angles saillants ou détails précis, par

exemple) de la vue 3D avec l'objet, pour ensuite calculer automatiquement la position réelle grâce à ces informations.

- **Entrées**

Un objet 3D représentant la surface et les coordonnées des points remarquables, une fois placés.

- **Sorties**

La position rectifiée de l'objet 3D et de la caméra.

3 Projecteurs multiples

- **Identification**

Utilisation de plusieurs projecteurs.

- **Priorité**

Modérée.

- **Description**

Pour que le spectateur puisse se déplacer autour de l'objet et observer les projections sous plusieurs angles, on utilisera plusieurs projecteurs simultanément. On considère l'utilisation d'un module d'expansion afin de connecter plusieurs projecteurs sur la même sortie vidéo, afin de se comporter comme un seul et même écran.

- **Entrées**

Un rendu 3D par angle à projeter.

- **Sorties**

Une image combinant les différents rendus, à envoyer sur les projecteurs.

4 Configuration

- **Identification**

Utilisation de fichiers de configuration.

- **Priorité**

Primordiale.

- **Description**

Il peut être fastidieux de paramétrer et de calibrer correctement une scène. La mise en place de fichiers de configuration permet de sauvegarder et de restaurer toutes les informations de la scène : objets, contenus, données de calibration.

- **Sauvegarde**

- **Entrées**

Les informations à sauvegarder.

- **Sorties**

Le fichier de configuration.

- **Chargement**

- **Entrées**

Un fichier de configuration.

- **Sorties**

Les informations restaurées.

C

Spécifications non fonctionnelles

1 Contraintes de développement et de conception

- Développement :
 - Langage C++
 - Bibliothèque Cinder
- Systèmes cibles : Windows, Linux, Mac

2 Contraintes de fonctionnement et d'exploitation

2.1 Performances

Le projet nécessite d'obtenir des performances suffisantes pour que l'affichage soit fluide lors des animations (notamment avec du texte défilant pour garder une certaine lisibilité). L'utilisation de plusieurs projecteurs ne doit pas influencer de façon significative sur la fluidité. De plus, on suppose que le programme fonctionnera potentiellement sur des machines de puissance modeste, il sera nécessaire d'en tenir compte lors du développement.

2.2 Ergonomie

Le logiciel devra rester ergonomique et simple d'utilisation, pour permettre à des utilisateurs non expérimentés de s'en servir sans nécessiter de connaissances particulières.

D

Document d'installation

Ce document récapitule les étapes requises pour mettre en place les outils nécessaires au projet.

1 Bibliothèques

Le projet utilise la bibliothèque Cinder, disponible à l'adresse <https://libcinder.org/download>. La version à télécharger ici est la 0.9.1 pour Visual C++ 2013 (il est important d'utiliser cette version pour garantir la compatibilité entre les différents outils du projet). Une fois téléchargée, l'archive peut être extraite à l'endroit de son choix.

La gestion de la vidéo est prise en charge par l'extension Cinder-FFmpeg, disponible sur GitHub : <https://github.com/paulhox/Cinder-FFmpeg>. Les sources devront être ajoutées au projet directement.

Le projet devra être configuré de manière à avoir accès aux headers des deux bibliothèques. Les bibliothèques statiques (fichiers `.lib`) de Cinder ainsi que de Cinder-FFmpeg devront être ajoutées au projet.

2 Environnement de développement

La bibliothèque Cinder impose l'utilisation de Visual C++ 2013 pour la compilation du projet. Il est recommandé de se servir également de Visual Studio 2013 pour assurer une compatibilité maximale. Dans le cas où l'on souhaiterait utiliser un autre environnement de développement, il faut s'assurer de disposer du même compilateur (Microsoft Build Tools v120) ; cependant, cela n'a pas été testé et est susceptible de ne pas fonctionner correctement.

3 Exécution

Le projet s'installe simplement. L'exécutable principal doit se trouver à côté du dossier `resources` ; ce dernier contient les données nécessaires au bon fonctionnement du programme.

E

Document d'utilisation

1 Lancement du programme

Le programme peut être démarré de deux façons.

- **Lancement direct** : en exécutant directement le programme, on accède à un projet vide sur l'interface habituelle.
- **Ligne de commande** : en exécutant le programme depuis une ligne de commande, il est alors possible de spécifier le projet à ouvrir. La syntaxe de la commande est la suivante :
`projector.exe <chemin/du/fichier/projet.json>`

2 Terminologie

Les divers éléments mis en œuvre par ce programme sont mentionnés par des termes précis :

- Les **objets 3D** sont la représentation des objets réels sur lesquels l'on voudra projeter un contenu. Il s'agit de fichiers au format `.obj`.
- Ce que l'on appelle **contenu** représente tout ce que l'on voudra projeter sur l'objet. Il s'agit ici d'images (sous les formats les plus courants : `.png`, `.jpg...`) ou de vidéos (aux formats `.mp4` et `.mov`).
- La représentation de la scène par le programme est composée de **projecteurs** : chacun d'entre eux représentera un projecteur de notre scène réelle. Il dispose d'informations géométriques telles que sa taille, son orientation ou son champ de vision.
- La notion d'**écran** est certainement la plus particulière à notre programme. En effet, piloter plusieurs projecteurs n'est pas toujours possible pour tous les ordinateurs. Heureusement, des solutions existent, permettant de relier plusieurs projecteurs à une même sortie vidéo, et se comportant pour l'ordinateur comme un seul et même écran de résolution plus élevée. Ainsi, du côté du programme, l'on pourra associer plusieurs projecteurs à un même écran, afin de pouvoir piloter efficacement plusieurs projecteurs sur la même sortie vidéo. La **Figure 1** récapitule ces explications.

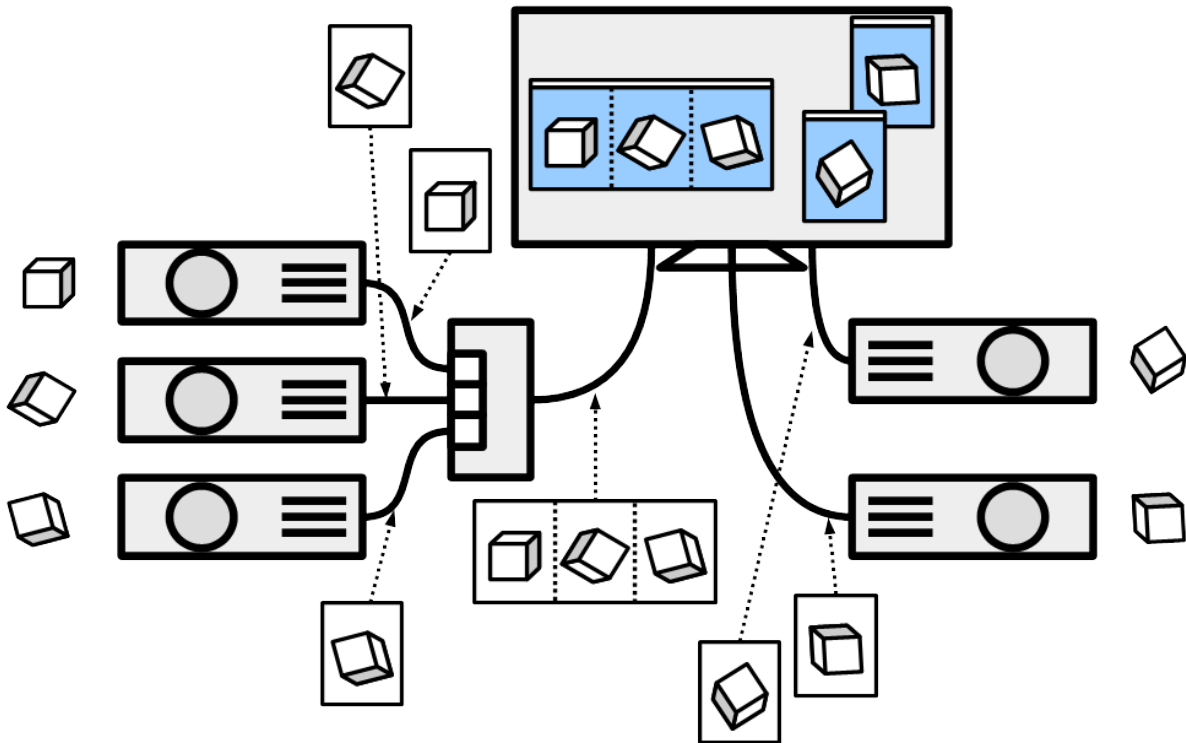


Figure 1 – Sur cet exemple, on considère un projet comportant 5 projecteurs. Comme l'on dispose uniquement de 3 sorties vidéo, on utilise un splitter vidéo qui nous permet de brancher 3 projecteurs sur la même sortie. Les deux autres sorties ont un projecteur branché directement dessus. On configure le programme pour afficher 3 projecteurs sur le même écran, c'est-à-dire dans la même fenêtre. Sur ce schéma, on peut voir l'image qui transite par chaque sortie vidéo.

3 Utilisation du programme

Une fois le programme lancé, une interface s'affiche, permettant à l'utilisateur d'effectuer un certain nombre d'actions. Celles-ci sont regroupées en plusieurs sections. L'interface est illustrée par la [Figure 2](#).

3.1 Projet

Cette section regroupe les actions relatives aux fichiers de projet : création d'un nouveau projet, sauvegarde et chargement au format JSON (les spécifications du format sont détaillées en [Section 4](#)). La création d'un nouveau projet efface tous les changements effectués sur le projet en cours et place le programme dans un état basique : pas d'objet ni de contenu chargés, un seul projecteur, ainsi que l'écran par défaut. Le chargement d'un projet existant, comme on pourrait s'y attendre, efface également tous les changements du projet en cours et remplace le programme dans l'état du projet sauvegardé.

3.2 Objet et contenu

Cette section contient les contrôles permettant de charger des fichiers d'objet 3D, ainsi que les contenus à projeter. Lorsque l'un de ces éléments est chargé, le nom du fichier s'affiche, ce qui

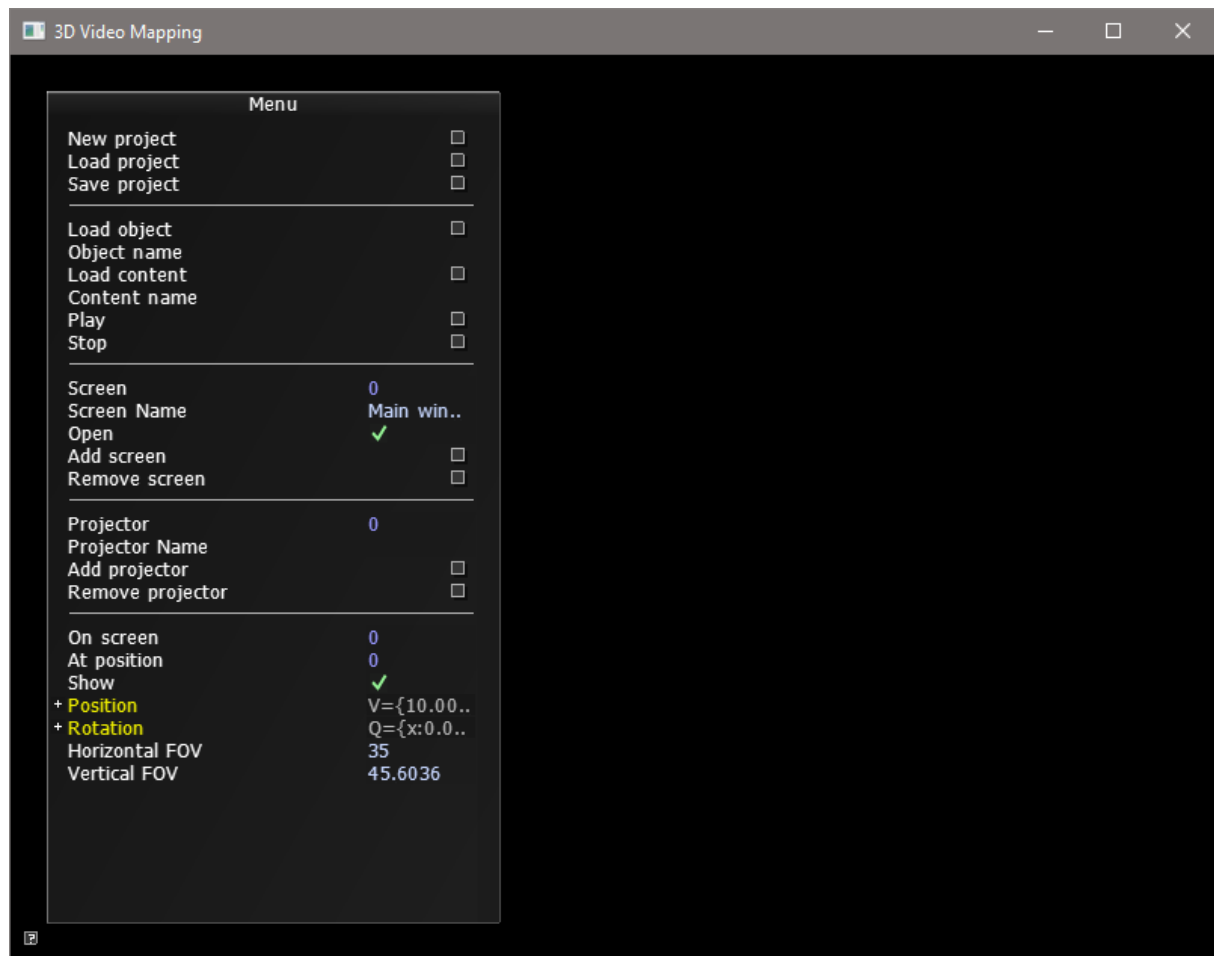


Figure 2 – Interface graphique du programme

permet d'identifier facilement les données sur lesquelles l'utilisateur travaille.

Les vidéos et les images ne s'affichent pas immédiatement. Pour cela, l'utilisateur doit cliquer sur le bouton **Play** : pour une vidéo, cette action démarre la lecture ; dans le cas d'une image, l'action déclenche seulement son affichage. De façon logique, le bouton **Stop** permet de stopper le contenu. Lorsque le contenu est stoppé, une image de test est projetée à la place du contenu : cela a pour avantage de pouvoir visualiser la projection dans le cas où le contenu serait partiellement ou entièrement noir.

3.3 Projecteurs et écrans

Le programme peut gérer plusieurs projecteurs, ainsi que plusieurs écrans (voir [Section 2](#)). Les projecteurs et les écrans disposent de contrôles similaires :

- Un indicateur de projecteur / d'écran actif. Cet indicateur définit le numéro du projecteur / de l'écran auquel se réfèrent les autres informations. Pour changer d'écran ou de projecteur actif, il suffit d'augmenter ou de diminuer ce nombre à l'aide des boutons + et -, ou bien de spécifier directement le numéro du projecteur ou de l'écran à utiliser.
- Le nom du projecteur / de l'écran. Celui-ci est modifiable pour pouvoir se repérer facilement.
- Deux boutons permettant respectivement d'ajouter un écran / un projecteur, et de supprimer l'écran / le projecteur actif.

Comme les écrans sont représentés chacun par une fenêtre, un contrôle supplémentaire nommé

Open permet d'ouvrir et de fermer chaque fenêtre. L'écran 0 représente la fenêtre principale du programme, et par conséquent ne peut être ni fermé, ni supprimé.

3.4 Propriétés du projecteur

Chaque projecteur peut être contrôlé indépendamment. La position du projecteur ainsi que son orientation sont modifiables respectivement grâce aux contrôles **Position** et **Rotation**. Les champs de vision verticaux et horizontaux sont également disponibles. Un simple bouton permet d'activer ou non le projecteur. Enfin, il est possible de choisir l'écran sur lequel le projecteur sera actif, ainsi que sa position sur cet écran (voir [Figure 1](#)).

4 Format des fichiers de projet

Le programme enregistre ses projets au format JSON. Il s'agit d'un format léger, très répandu, lisible et modifiable aisément à la main par un humain. Voici un exemple de fichier de projet :

Code source E.1 – Exemple de fichier de projet

```

1 {
2   "projectors" : [
3     {
4       "name": "Projecteur 1",
5       "id": 0,
6       "show": true,
7       "x": 10,
8       "y": 20,
9       "z": 30,
10      "qx": 5,
11      "qy": 4,
12      "qz": 3,
13      "qs": 2,
14      "fov_x": 90,
15      "fov_y": 90
16    },
17    ...
18  ],
19  "screens" : [
20    {
21      "id": 0,
22      "name": "The screen",
23      "projectors": [
24        1, 0
25      ]
26    },
27    ...
28  ],
29  "object": "C:\\path\\to\\object.obj",
30  "content": "C:\\path\\to\\content.png"
31 }
```

Détaillons la structure d'un de ces fichiers :

- **projectors** contient la liste de tous les projecteurs du projet. Chaque projecteur possède plusieurs attributs :
 - **id** représente l'identifiant du projecteur, utilisé en interne par le programme ainsi que dans ces fichiers de projet.

- **name** représente son nom.
- **show** vaut *true* si le projecteur est activé, *false* sinon.
- **x**, **y** et **z** correspondent à la position du projecteur.
- **qx**, **qy**, **qz** et **qs** correspondent à son orientation.
- **fovx** et **fovy** représentent le champ de vision du projecteur.
- **screens** contient la liste de tous les écrans du projet. Comme pour les projecteurs, chaque écran possède plusieurs caractéristiques :
 - **id** représente l'identifiant de l'écran.
 - **name** représente son nom.
 - **projectors** contient la liste des identifiants des projecteurs affichés sur cet écran. Cette liste est ordonnée (dans l'exemple du dessus, le projecteur d'identifiant 1 sera affiché à gauche).
- **object** contient le chemin du fichier d'objet 3D.
- **content** contient le chemin du fichier de contenu.

Il est à noter que les fichiers doivent être cohérents : les ID des projecteurs doivent tous être uniques et chacun doit être présent sur un et un seul écran. Si un fichier n'est pas bien construit, il ne sera pas chargé.

1 Code du projet

Ce projet a été pensé pour être développé de façon maintenable. Pour cela, le code a été découpé en différentes classes, chacune ayant une fonctionnalité précise.

1.1 Diagramme de classes

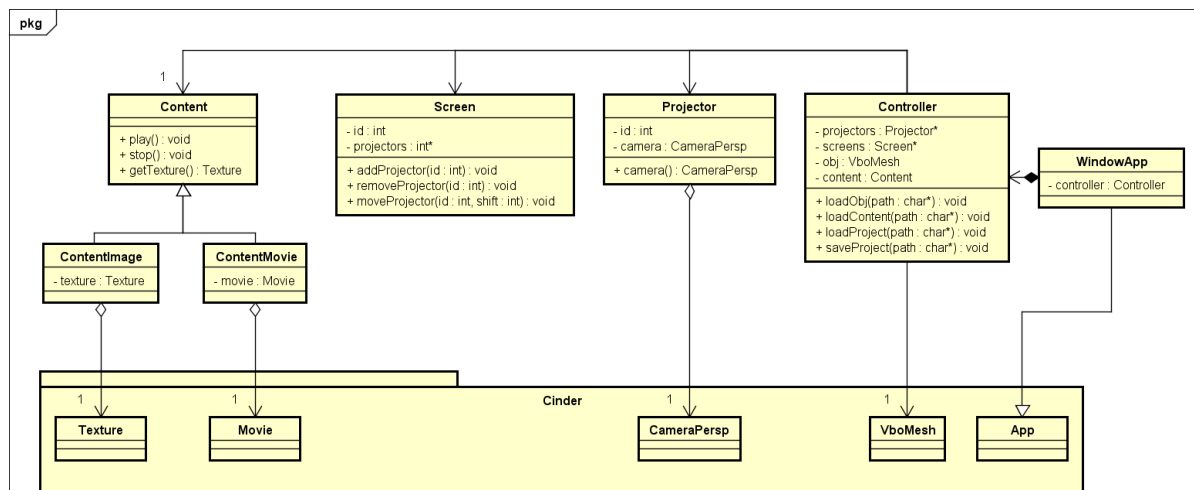


Figure 1 – Diagramme de classes

1.2 Description des classes

1.2.1 WindowApp

Cette classe hérite de la classe **App** de Cinder. Elle contient tout la logique de l'interface graphique : gestion des différentes fenêtres, rendu 3D et interface utilisateur. Des callbacks relient les contrôles

de cette interface aux méthodes du contrôleur de l'application.

1.2.2 Controller

Cette classe est le cœur du programme. Elle gère l'intégralité des éléments de la projection (projecteurs, écrans, objet 3D et contenu) et propose de méthodes pour accéder aux propriétés de ces éléments. Elle s'occupe également de la gestion des projets (chargement et enregistrement au format JSON). La classe **WindowApp** interagit au travers de callbacks reliés aux événements de l'interface.

1.2.3 Projector

La classe **Projector** définit un projecteur. Elle combine les informations géométriques du projecteur de notre scène (position, rotation...) à une caméra OpenGL qui permet d'effectuer le rendu. Elle contient également un identifiant unique, utilisé en interne pour différencier les différents projecteurs, ainsi qu'un nom, utilisé par l'interface pour que l'utilisateur puisse l'identifier facilement.

1.2.4 Screen

La classe **Screen** représente une fenêtre qui pourra accueillir plusieurs projecteurs. Cela permet, à l'aide d'un splitter vidéo, d'envoyer plusieurs images à plusieurs projecteurs à travers une unique sortie vidéo. Elle contient l'ordre de tous les projecteurs qui lui sont associés, afin d'afficher correctement chaque rendu sur le projecteur adéquat.

1.2.5 Content

Cette classe abstraite représente le contenu qui sera projeté. Ses classes filles, **ContentImage** et **ContentVideo**, permettent respectivement d'afficher une image ou une vidéo à travers une même interface.

1.3 Fonctionnement du programme

Le projet suit un design pattern MVC. En effet, l'interface graphique est totalement séparée des données, auxquelles elle accède au travers d'un contrôleur.

Le modèle est constitué des **Projectors**, **Screens**, **Contents** ainsi que des objets 3D (représentés par une classe de **Cinder**, **VboMesh**). Chacun comporte des méthodes permettant d'accéder à leurs attributs. Le contrôleur contient les instances des classes du modèles correspondant à la représentation de la scène. L'interface graphique, elle, dispose de contrôles pour interagir avec le contrôleur, ainsi que d'une zone de dessin pour afficher le rendu à projeter.

2 Fichiers d'entrée / sortie

Le programme utilise des fichiers JSON pour enregistrer et charger des projets. Ces fichiers contiennent la description complète de tous les **Projectors** et les **Screens** du projet, ainsi que les objets 3D et les contenus. Le **Controller** se charge de lire et écrire ces fichiers.

Pour une description détaillée, voir [Section 4](#) (Annexe E).

3 Structure du projet

Le projet est organisé en plusieurs dossiers :

```
Application ..... Racine du projet
├── Application ..... Sources du programme
│   ├── Debug ..... Fichiers de build
│   └── resources ..... Fichiers nécessaires au projet
├── Debug ..... Fichiers de l'exécutable
├── doc ..... Documentation
│   ├── html ..... Documentation html
│   └── latex ..... Documentation latex
```

Un projet de qualité utilise du code validé par des tests. On distingue les tests unitaires, chargés de valider une fonction de façon indépendante, et les tests fonctionnels, qui permettent de vérifier qu'un ensemble de fonctions se comportent correctement dans leur ensemble.

1 Tests unitaires

Nos tests unitaires permettent de s'assurer dans la mesure du possible que chaque méthode effectue correctement le travail qui lui est demandé.

1.1 Modèle

Le modèle est l'élément qui se prête le mieux aux tests unitaires. En effet, les classes interagissent peu les unes avec les autres, il est donc plus aisé de tester chaque méthode indépendamment.

Élément	Contenu	Statut
Projector	Constructeurs	✓
Projector	Getters / Setters	✓
Screen	Constructeurs	✓
Screen	Getters / Setters	✓

Tests unitaires du modèle

Figure 1 – Tests unitaires du modèle

1.2 Contrôleur

Bien que toutes ne s'y prêtent pas, certaines méthodes du contrôleur sont testables unitairement.

Élément	Contenu	Statut
Controller	Gestion des Projectors	✓
Controller	Gestion des Screens	✓
Controller	Interaction entre Projectors et Screens	✓

Tests unitaires du contrôleur

Figure 2 – *Tests unitaires du contrôleur*

2 Tests fonctionnels

Certaines méthodes sont difficiles, voire impossible à couvrir avec des tests unitaires, notamment lorsqu'elles touchent à l'affichage ou à des fichiers.

2.1 Contrôleur

Les méthodes de lecture et d'écriture de fichiers du contrôleur ont été testées de façon fonctionnelle, en utilisant des données de tests spécialement préparées pour couvrir un maximum de cas.

Élément	Contenu	Statut
Controller	Chargement de projets	✓
Controller	Enregistrement de projets	✓
Controller	Chargement d'objets	✓
Controller	Chargement de vidéos	✓
Controller	Chargement d'images	✓

Tests fonctionnels du contrôleur

Figure 3 – *Tests fonctionnels du contrôleur*

2.2 Vue

Les méthodes d'affichage sont difficiles à évaluer avec des tests unitaires, mais il est facile de se rendre compte visuellement de leur correction.

Élément	Contenu	Statut
WindowApp	Callbacks	✓
WindowApp	Rendu 3D	✓
WindowApp	Multi-projecteur	✓
WindowApp	Multi-écran	✗

Tests fonctionnels de la vue

Figure 4 – *Tests fonctionnels de la vue*

Le test du multi-écran ne passe pas, car le programme ne fonctionne pas avec plusieurs fenêtres différentes. Cela est dû à la façon dont OpenGL partage ses contextes entre les fenêtres ; cela constitue un des axes d'améliorations pour ce projet.

3 Données de test

Pour tester le chargement de fichiers de projet, et également pour tester un maximum de configurations différentes, nous avons utilisé plusieurs fichiers JSON.

Code source G.1 – *Un projecteur et un écran*

```

1 {
2   "projectors" : [
3     {
4       "name": "Projecteur 1",
5       "id": 0,
6       "show": true,
7       "x": 10,
8       "y": 0,
9       "z": 0,
10      "qx": 0,
11      "qy": 1,
12      "qz": 0,
13      "qs": 0,
14      "fovx": 90,
15      "fovy": 45
16    }
17  ],
18  "screens" : [
19    {
20      "id": 0,
21      "name": "Ecran",
22      "projectors": [
23        0
24      ]
25    }
26  ],
27  "object": "",
28  "content": ""
29 }
```

Code source G.2 – *Trois projecteurs et un écran*

```

1 {
2   "projectors" : [
3     {
4       "name": "Projecteur 1",
5       "id": 0,
6       "show": true,
7       "x": 10,
8       "y": 0,
9       "z": 0,
10      "qx": 0,
11      "qy": 1,
12      "qz": 0,
13      "qs": 0,
14      "fovx": 90,
15      "fovy": 45
16    },
17    {
18      "name": "Projecteur 2",
19      "id": 0,
20      "show": true,
21      "x": 0,
```

```

22         "y": 10,
23         "z": 0,
24         "qx": 1,
25         "qy": 0,
26         "qz": 0,
27         "qs": 0,
28         "fov_x": 90,
29         "fov_y": 45
30     },
31     {
32         "name": "Projecteur 2",
33         "id": 0,
34         "show": true,
35         "x": 10,
36         "y": 10,
37         "z": 0,
38         "qx": 1,
39         "qy": 1,
40         "qz": 0,
41         "qs": 0,
42         "fov_x": 90,
43         "fov_y": 45
44     }
45 ],
46 "screens" : [
47     {
48         "id": 0,
49         "name": "Ecran",
50         "projectors": [
51             0, 1, 2
52         ]
53     }
54 ],
55 "object": "",
56 "content": ""
57 }

```

Code source G.3 – Deux projecteurs et deux écrans

```

1 {
2     "projectors" : [
3         {
4             "name": "Projecteur 1",
5             "id": 0,
6             "show": true,
7             "x": 10,
8             "y": 0,
9             "z": 0,
10            "qx": 0,
11            "qy": 1,
12            "qz": 0,
13            "qs": 0,
14            "fov_x": 90,
15            "fov_y": 45
16        },
17        {
18            "name": "Projecteur 2",
19            "id": 0,
20            "show": true,
21            "x": 0,
22            "y": 10,

```

```

23         "z": 0,
24         "qx": 1,
25         "qy": 0,
26         "qz": 0,
27         "qs": 0,
28         "fovx": 90,
29         "fovy": 45
30     }
31 ],
32 "screens" : [
33     {
34         "id": 0,
35         "name": "Ecran",
36         "projectors": [
37             0
38         ]
39     },
40     {
41         "id": 1,
42         "name": "Ecran 2",
43         "projectors": [
44             1
45         ]
46     }
47 ],
48 "object": "",
49 "content": ""
50 }

```

Code source G.4 – *Objet et contenu*

```

1 {
2     "projectors" : [
3         {
4             "name": "Projecteur 1",
5             "id": 0,
6             "show": true,
7             "x": 10,
8             "y": 0,
9             "z": 0,
10            "qx": 0,
11            "qy": 1,
12            "qz": 0,
13            "qs": 0,
14            "fovx": 90,
15            "fovy": 45
16        }
17    ],
18    "screens" : [
19        {
20            "id": 0,
21            "name": "Ecran",
22            "projectors": [
23                0
24            ]
25        }
26    ],
27    "object": "C:\\path\\to\\object.obj",
28    "content": "C:\\path\\to\\content.png"
29 }

```



Comptes rendus hebdomadaires

Compte rendu n°1 du 23/09/2018

Prise en main du sujet
Lectures sur le mapping vidéo
Recherche de logiciels existants

Compte rendu n°2 du 30/09/2018

Essais avec MadMapper
Veille technologique
Rencontre avec la cliente

Compte rendu n°3 du 07/10/2018

Début de la rédaction des spécifications
Recherches sur le mapping

Compte rendu n°4 du 14/10/2018

Rédaction du rapport
Recherches sur OpenGL

Compte rendu n°5 du 21/10/2018

Ébauche d'interface
Essais avec Qt et OpenGL

Compte rendu n°6 du 28/10/2018

Rédaction du rapport
Essais avec OpenGL

Compte rendu n°7 du 11/11/2018

Rédaction du rapport
Essais avec OpenGL

Compte rendu n°8 du 18/11/2018

Rédaction du rapport
Essais avec OpenGL
Modélisation

Compte rendu n°9 du 25/11/2018

Rédaction du rapport
Modélisation

Compte rendu n°10 du 2/12/2018

Rédaction du rapport
Préparation de la soutenance

Compte rendu n°11 du 9/12/2018

Étude d'un exemple Cinder (PmSimulator)

Compte rendu n°12 du 16/12/2018

Étude de Cinder

Compte rendu n°13 du 23/12/2018

Étude de Cinder

Compte rendu n°14 du 13/01/2019

Prise en main de Cinder
Installation et mise en route

Compte rendu n°15 du 20/01/2019

Prise en main de Cinder
Compilation du simulateur

Compte rendu n°16 du 27/01/2019

Prise en main de Cinder
Base du moteur de rendu

Compte rendu n°17 du 3/02/2019

Révision de l'architecture du projet

Compte rendu n°18 du 10/02/2019

Tentatives d'implémentation de la vidéo

Compte rendu n°19 du 24/02/2019

Développement du moteur de rendu
Tentatives d'implémentation de la vidéo

Compte rendu n°20 du 3/03/2019

Développement de l'application

Compte rendu n°21 du 10/03/2019

Développement de l'interface

Compte rendu n°22 du 17/03/2019

Rédaction de la documentation

Développement des tests

Compte rendu n°23 du 24/03/2019

Rédaction du rapport

Préparation de la soutenance

Compte rendu n°24 du 31/03/2019

Rédaction du rapport



Webographie

- [WWW1] PROJECTION-MAPPING.ORG. *Projection mapping software*. 2018. URL : <http://projection-mapping.org/software/>.
- [WWW2] WIKIPEDIA. *Texture mapping - Rasterization algorithms*. 2018. URL : https://en.wikipedia.org/wiki/Texture_mapping#Rasterisation_algorithms.
- [WWW3] WIKIPEDIA. *UV Mapping*. 2018. URL : https://en.wikipedia.org/wiki/UV_mapping.



Bibliographie

- [1] V. V. TRUSHKOV et V. M. KHACHUMOV. « Determining the object orientation in a 3D space ». In : *Optoelectronics, Instrumentation and Data Processing* 44.3 (juin 2008), p. 245-248. ISSN : 1934-7944. URL : <https://doi.org/10.3103/S8756699008030084>.

Mapping vidéo 3D

Résumé

Ce projet consiste en le développement d'une application permettant la projection d'images et de vidéos sur des objets 3D en utilisant plusieurs projecteurs grâce à des techniques de mapping vidéo. Le projet utilise le framework Cinder et est écrit en C++.

Mots-clés

Mapping vidéo, C++, Cinder

Abstract

This project deals with developing an application that is able to project images and videos on 3D objects, using multiple projectors, thanks to projection mapping techniques. The project uses the Cinder framework and is written in C++.

Keywords

Projection mapping, C++, Cinder

Tuteurs académiques

Gilles VENTURINI
Barthélemy SERRES

Étudiant

Clément GRODECOEUR (DI5)