Université de Montréal

Eyes Of Darwin : une fenêtre ouverte sur l'évolution du logiciel

par

Julien Tantéri

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des Arts et des Sciences
en vue de l'obtention du grade de Maître ès sciences
en informatique

Décembre, 2009

Université de Montréal
Faculté des Arts et des Sciences

Cette mémoire intitulé :

Eyes Of Darwin : une fenêtre ouverte sur l'évolution du logiciel

présenté par :
Julien Tantéri

a été évaluée par un jury composé des personnes suivantes :

Stefan Monnier, président-rapporteur
Yann-Gaël Guéhéneuc, directeur de recherche
Sébastien Roy, membre du jury

RÉSUMÉ

De nos jours, les logiciels doivent continuellement évoluer et intégrer toujours plus de fonctionnalités pour ne pas devenir obsolètes. C'est pourquoi, la maintenance représente plus de 60% du coût d'un logiciel. Pour réduire les coûts de programmation, les fonctionnalités sont programmées plus rapidement, ce qui induit inévitablement une baisse de qualité. Comprendre l'évolution du logiciel est donc devenu nécessaire pour garantir un bon niveau de qualité et retarder le dépérissement du code. En analysant à la fois les données sur l'évolution du code contenues dans un système de gestion de versions et les données quantitatives que nous pouvons déduire du code, nous sommes en mesure de mieux comprendre l'évolution du logiciel. Cependant, la quantité de données générées par une telle analyse est trop importante pour être étudiées manuellement et les méthodes d'analyses automatiques sont peu précises.

Dans ce mémoire, nous proposons d'analyser ces données avec une méthode semi-automatique : la visualisation. *Eyes Of Darwin*, notre système de visualisation en 3D, utilise une métaphore avec des quartiers et des bâtiments d'une ville pour visualiser toute l'évolution du logiciel sur une seule vue. De plus, il intègre un système de réduction de l'occlusion qui transforme l'écran de l'utilisateur en une fenêtre ouverte sur la scène en 3D qu'il affiche. Pour finir, ce mémoire présente une étude exploratoire qui valide notre approche.

Mots-clés : visualisation du logiciel, évolution du logiciel, qualité, métrique, occlusion, stéréoscopie.

ABSTRACT

Software must continuously evolve and integrate more functionalities to remain useful. Consequently, more than 60% of a software system's cost is related to maintenance. To reduce this cost, programming must performed faster, witch leads to a decrease of the system code's quality. Therefore, understanding software evolution is becoming a necessity to prevent code decay and to increase the system life span. To ease software understanding, we perform a cross analysis of the historical data extracted from a version control system, with quantitative data that we obtain from the source code. However, the significant amount of data generated by this kind of analysis makes it necessary to have tools to support the maintainer's analysis. First, tools help because examining them manually is impossible. Second, they help because automatics methods are not accurate enough.

We present a new semiautomatic approach to help analysis. Our 3D visualization system, *Eyes Of Darwin*, uses a cityscape metaphor to show software's evolution on a single view. It integrates an occlusion reduction system, witch turns the screen to an open window on the 3D world. We conclude, with an exploratory study in order to validate our approach.

Keywords : software visualization, software evolution, quality, metric, occlusion, stereoscopy.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES FORMULES

LISTE DES SIGLES

2D	Deux Dimensions
3D	Trois Dimensions
AP	Anti-pattern
BS	Bad Smell
CS	Cityscape
CVS	Concurrent Versions System
DP	Design Pattern
EOD	Eyes Of Darwin
IR	Infrarouge
SGV	Système de Gestion de Version
VV	Variable Visuelle

À mes parents...

...et à Costa

REMERCIEMENTS

Tout d'abord, je tiens à remercier Yann-Gaël Guéhéneuc, pour m'avoir donné l'opportunité d'étudier sous sa direction. Je voudrais plus particulièrement le remercier pour sa bonne humeur quotidienne, son aide précieuse tout au long de mon projet et pour m'avoir ouvert l'esprit sur notre domaine, tout en me laissant forger mon propre sens critique.

Je souhaite également remercier Sébastien Roy, pour son enseignement transmis avec passion, qui a suscité chez moi un grand intérêt et dont une partie des notions acquises se retrouvent dans ce mémoire.

Je voudrais également exprimer ma gratitude à l'ensemble de l'équipe *Ptidej*. Plus particulièrement, à Foutse Khomh et Stéphane Vaucher, pour tous nos débats reliés de près ou de loin à mes travaux.

De plus, je remercie les sujets ayant accepté de participer à l'expérience présentée dans ce mémoire.

Je tiens à adresser une pensée toute particulière à Élisabeth, pour sa patience, son soutien inconditionnel et son aide inestimable tout au long de mes études.

Enfin, je souhaite exprimer ma plus profonde gratitude à mes parents, Lélian et Raymond, et à mon oncle Costa. Leurs encouragements et leur support durant toutes mes études, aussi bien sur le plan académique, financier, que moral compte beaucoup à mes yeux.

CHAPITRE 1.

INTRODUCTION

L'évolution rapide de l'informatique a induit de profonds changements dans les pratiques de programmation. Il y a peu de temps encore, les logiciels étaient développés pour répondre aux besoins d'un seul client. Les ressources en calcul et en espace étaient rares et chères, l'optimisation était donc la problématique principale. Aujourd'hui, la puissance sans cesse croissante des nouveaux ordinateurs rend cette problématique obsolète dans bien des domaines. En effet, les ressources, aussi bien en calcul qu'en espace, sont disponibles en grande quantité et à bas prix, ce qui permet de proposer des logiciels intégrant toujours plus de fonctionnalités, dans des domaines de plus en plus variés.

Cependant, implémenter autant de fonctionnalités demande un grand effort de programmation. Pour réduire cet effort, les logiciels sont distribués à grande échelle, ce qui change les règles de la concurrence. En effet, il ne suffit plus de gagner un marché pour développer un produit et s'assurer de plusieurs années de maintenance. Pour garder ses utilisateurs et en séduire de nouveaux, les logiciels doivent continuellement évoluer, pour proposer toujours plus de fonctionnalités, avant la concurrence et à moindre coût. La problématique d'optimisation des ressources a donc peu à peu laissé la place à une problématique de réduction de l'effort de programmation. Même si les langages de dernière génération (p. ex. : Java, .Net) permettent de programmer plus vite, il faut inévitablement accepter une baisse de la qualité pour répondre à cette logique de productivité.

Il y a longtemps que la communauté du génie logiciel a pris conscience de cette baisse de la qualité. Par exemple, des études montrent que près de 60% [1] du coût d'un logiciel est lié à sa maintenance. D'autres études ont mis en évidence que la modularité du code décline au fil des changements, alors que l'effort pour

apporter ces modifications augmente [2]. Ce dernier point est d'ailleurs parfaitement énoncé dans les lois de Lehman :

> *« As a program is evolved its complexity increases unless work is done to maintain or reduce it. »* [3]

De nombreuses méthodes sont disponibles pour évaluer la qualité d'un logiciel. Parmi les plus connus, nous retrouvons les métriques logicielles [4-6], les patrons de conception [7], les mauvaises odeurs [8] ou encore les défauts de conception [9]. Cependant, ces méthodes permettent uniquement l'analyse de l'état d'un système, à un instant donné. Comprendre l'évolution du logiciel est donc devenu nécessaire pour garantir un bon niveau de qualité afin de retarder le dépérissement du code et de prolonger la vie du logiciel.

Les systèmes de gestion de versions [10-18], offrent une bonne source de données temporelles, puisqu'ils conservent les changements du code. Il est donc possible d'extraire ces données pour établir une série de clichés qui représentent l'état du code à différents instants de son évolution. Puis, pour chacun de ces clichés, d'évaluer leurs qualités à l'aide par exemple des métriques. Ainsi, l'analyse des changements de la qualité (i.e., des valeurs des métriques) de ces clichés nous permet d'appréhender l'évolution de la qualité du logiciel. Il faut cependant tenir compte de la quantité de données importante qu'une telle analyse génère. Étudier ces données manuellement pour en tirer des observations globales est trop fastidieux, voire impossible et les méthodes automatiques manquent de précision [19], car les modèles de qualités qu'elles utilisent ne sont pas encore assez complets.

C'est pourquoi une partie de la communauté se dirige vers la visualisation, qui est une solution semi-automatique. Elle met en avant les aspects importants du logiciel, tout en laissant le soin à l'utilisateur de réaliser le processus analytique. Depuis le début des années 90, de nombreux systèmes de visualisation (SV) sont

apparus, d'abord en deux dimensions (2D), puis plus récemment en trois dimensions (3D). Les SV en 2D ont l'avantage de pouvoir réutiliser des concepts connus comme les diagrammes, mais leur capacité d'affichage est limitée. L'ajout d'une troisième dimension donne aux SV une capacité d'affichage plus importante. Pourtant, aucun SV en 2D ou 3D n'a la capacité de visualiser efficacement toute l'évolution d'un logiciel sur une seule vue unique, car la quantité de données à afficher est trop importante. En effet, par rapport à un SV du logiciel sur une version unique, un SV de l'évolution du logiciel doit multiplier la quantité de données à afficher par le nombre de versions visualisées.

Nous pensons qu'un SV de l'évolution du logiciel efficace doit permettre à ses utilisateurs de naviguer librement en même temps à travers les entités du code (p.ex. les classes) et leur évolution. C'est pourquoi notre problématique principale est de trouver un concept de visualisation qui permet d'afficher toute l'évolution d'un logiciel sur une vue unique.

Parmi les SV en 3D se trouvent les « métaphores de villes » (*Cityscape*, CS), qui modélisent le logiciel en suivant une métaphore avec les bâtiments et les quartiers d'une ville. Les CS ont deux avantages majeurs : (1) leur structure en entités et zones géographiques se prête bien à la visualisation du logiciel et (2) ils offrent une grande capacité d'affichage. De plus, nous pensons qu'il est possible d'accroître cette capacité en utilisant le côté des bâtiments pour afficher de l'information. Cependant, la plupart des CS n'utilisent pas le côté des bâtiments parce que la 3D est soumise à des problèmes d'occlusion. En effet, comme les bâtiments sont proches, leur base est souvent cachée par le sommet des autres bâtiments qui sont devant eux. Donc, si des informations sont affichées sur le coté des bâtiments, il est probable qu'une partie d'entre elles soient occultées, obligeant l'utilisateur à changer de point de vue pour les consulter. Or, le clavier et la souris sont deux instruments en 2D peu précis qui rendent les petits déplacements en 3D

délicats et qui obligent l'utilisateur à arrêter l'analyse des données pour contrôler le déplacement.

Thèse

Il est possible d'utiliser les données sur l'évolution du code contenues dans un système de gestion de versions pour obtenir des données qualitatives caractérisants les différentes périodes de cette évolution. Puis, de visualiser ces données qualitatives à l'aide d'un CS dont les bâtiments sont associés aux entités du code et qui décrivent l'évolution de ces derniers.

Notre première contribution est de proposer un SV de l'évolution du logiciel sur une seule vue. Ce SV est un CS dont les bâtiments réutilisent le concept de la pyramide des âges. Chaque pyramide représente toute l'évolution de l'entité à laquelle elle est associée et chaque tranche de la pyramide représente une période de l'évolution de cette l'entité (p. ex. la classe ou l'interface). Cependant, cette approche impose d'afficher de l'information sur le côté des bâtiments, puisque ces derniers sont découpés en tranches empilées verticalement. C'est pourquoi, notre deuxième contribution est de doter notre CS d'un module qui transforme l'écran en une fenêtre ouverte sur la scène qu'il affiche. Ainsi, pour régler les problèmes d'occlusion, l'utilisateur peut simplement déplacer sa tête par rapport à l'écran, ce qui est un réflexe et donc qui ne perturbe pas le processus analytique. Pour finir, ce mémoire présente une étude exploratoire comme début de preuve de l'efficacité de notre SV.

Ce mémoire est découpé en quatre chapitres. Le premier chapitre présente un état de l'art des travaux connexes aux nôtres. Le deuxième chapitre détaille notre SV. Le troisième chapitre aborde l'étude exploratoire que nous avons réalisée. Enfin, le quatrième chapitre est la conclusion de ce mémoire.

CHAPITRE 2.

ÉTAT DE L'ART

Cette section aborde les travaux connexes à nos travaux et se divise en quatre parties. La première partie traite des systèmes de gestion de version, car ils sont notre source de données temporelles. La deuxième partie aborde la qualité du code source et plus particulièrement les techniques qui permettent de l'évaluer, puisque ces techniques sont notre source de données qualitative. La troisième partie se concentre sur la visualisation, le sujet y est abordé sous plusieurs aspects : les principes à respecter, les SV existants, leurs objectifs et leurs atouts. Enfin, nous terminons par une discussion sur les travaux présentés.

2.1. Les systèmes de gestion de versions

Il existe de nombreux systèmes de gestion de versions (SGV), il est toute fois possible de les séparer en trois grands groupes :

- Les systèmes locaux.
- Les systèmes centralisés, qui ont profité de la généralisation des réseaux locaux pour se développer.
- Les systèmes décentralisés, dont l'apparition a largement été favorisée par le développement de l'internet.

Les systèmes locaux permettent de garder l'historique des modifications d'un système de fichiers localement à un ordinateur. Il s'agit des premiers systèmes de gestion de versions. Le plus vieux est *Source Code Control System* (SCCS) [17], il a été développé en 1972 par Rochkind. En 1985, *Revision Control System* (RCS) [18] fait son apparition, proposant une solution libre de droits qui apporte des améliorations comme la gestion des fichiers binaires.

Les systèmes de gestion de versions décentralisés n'utilisent pas de serveur central. Chaque usager dispose d'une copie locale de l'historique des fichiers. Les mises à jour se font directement de pair-à-pair (*i.e., peer-to-peer*) entre les usagers. Le système décentralisé le plus connu est Git [14] car il sert au développement du noyau *Linux*.

Enfin, les systèmes centralisés sont basés sur une architecture client/serveur. Le premier SGV centralisé, *Concurrent Versions System* (CVS) [12] est apparu en 1986. Ce système libre de droits est rapidement devenu la norme dans les universités, pour les développements libres et même dans une partie de l'industrie. Malgré une modification en 1989 [10], qui permet notamment de gérer les fichiers binaires, CVS est loin d'être parfait et laisse plusieurs indéterminations sur l'historique des changements :

- Les mises à jour ne sont pas atomiques : il n'est pas possible de savoir quels fichiers ont été mis à jour en même temps.
- Pas de suivi des déplacements et changements de nom : CVS ne garde aucune trace des fichiers et dossiers renommés ou déplacés.
- Gestion des branches trop libre :
    - Les utilisateurs peuvent choisir quels fichiers ajouter à une branche, et quand les ajouter. Les fichiers qui constituent l'espace de travail d'un utilisateur peuvent donc provenir de plusieurs branches et peuvent changer de branches à tout moment. Il est donc très difficile de retracer l'historique d'un fichier à travers les branches.
    - La fusion des fichiers d'une branche A vers une branche B correspond à copier la révision la plus récente présente sur la branche A vers la branche B. La branche fusionnée ne cesse jamais d'exister et CVS ne garde aucune trace de cette copie. Il n'est donc pas possible de savoir quand un fichier d'une branche a été fusionné, ni si une branche est toujours utilisée.

Ce constat a favorisé l'apparition de plusieurs systèmes qui cherchent à apporter des améliorations et à combler les faiblesses de CVS, parmi lesquels : *Microsoft Visual SourceSafe* (VSS) [15], *Perforce* [16] et plus récemment *Rational ClearCase* d'*IBM* [13]. Cependant, l'utilisation de ces systèmes propriétaires reste limitée à certaines entreprises.

Subversion (SVN) [11] est un SVG centralisé libre de droits qui est de plus en plus utilisé. Il se base sur le même modèle de données que CVS, mais propose une nouvelle implémentation qui apporte des améliorations. Avec SVN : (1) les mises à jour sont atomiques, (2) il est possible de déplacer et renommer des fichiers ou des dossiers sans perte d'historique, et enfin (3) chaque branche contient une copie de tous les fichiers d'un module.

De nos jours, la majorité des projets libres accessibles en ligne le sont sur des serveurs SVN ou CVS.

2.2. Qualité du code source

La communauté du génie logiciel s'intéresse depuis longtemps à la qualité du code source. Les premiers travaux sur ce sujet cherchent à mettre en place des modèles théoriques qui garantissent un bon niveau de qualité. Par exemple, dès la fin des années 60, les articles sur les GOTO de Knuth [20] et Dijkstra [21] animent le débat sur la lisibilité du code. Plus récemment, les patrons de conception (Design Patterns, DP) élaborés par Gamma *et al.* [7], proposent un ensemble de bonnes solutions à des problèmes récurrents de conception logicielle. Cependant, pour connaître l'impact de ces techniques sur la qualité, il est nécessaire de pouvoir évaluer cette dernière.

Les métriques logicielles sont des techniques qui cherchent à mesurer les propriétés du code. Il existe deux grands types de métriques : les métriques

dynamiques, qui sont calculées lors de l'exécution du code et les métriques statiques qui sont calculées directement sur le code source ou sur un modèle de celui-ci.

D'une façon générale, les métriques dynamiques s'intéressent aux aspects opérationnels du code. En effet, considérer des aspects comme les capacités de la machine, la fréquence ou le contenu des messages permet de faire un calcul qui tient compte du contexte d'exécution. Cependant, les métriques dynamiques ne permettent pas de dégager des observations globales sur le code, car leur validité se limite aux exécutions sur lesquelles elles ont été calculées. Ainsi, pour que leur résultat soit significatif, il est important que ces métriques soient calculées dans un contexte d'exécution réaliste. Or, les équipes de recherche ou de développement n'ont généralement pas accès aux données des utilisateurs. Et si c'est le cas, leur quantité, leur hétérogénéité et leurs évolutions futures empêchent de dégager un jeu des données typique. Même si des essais prometteurs [22] ont été réalisés, les métriques dynamiques restent difficiles à mettre en œuvre pour l'étude de la qualité du code. Elles sont toutefois couramment utilisées pour optimiser dynamiquement un programme à un site d'exploitation, ce qui augmente considérablement l'efficacité du programme.

Les métriques statiques se concentrent sur les aspects structurels du code. Elles ne tiennent pas compte du contexte d'exécution, mais elles permettent de dégager des observations globales valables pour toutes les exécutions. À l'époque de la programmation procédurale, la pertinence de l'évaluation de la complexité du code proposée par McCabe [6] contribua fortement à la popularité des métriques statiques. En orienté objet, il est important de souligner la suite de métriques de Chidamber et Kemerer [4, 5], qui est l'une des plus connues.

L'utilité des métriques en tant qu'indicateur de qualité a déjà été montrée [23-25]. Cependant, les études qui montrent l'efficacité des métriques ont toutes été

réalisées pour observer la qualité du code terminé. Or, si l'étude à posteriori du code permet de prouver que la qualité influence les valeurs des métriques, rien ne garantit que lors des phases de programmation, prendre en compte les valeurs des métriques puisse influencer la qualité du code. D'ailleurs, une étude réalisée par notre laboratoire avec un partenaire industriel tend à montrer ce point. Cette entreprise avait essayé d'utiliser les métriques comme critère de qualité en rejetant automatiquement un code qui ne respectait pas certains seuils. La réaction de beaucoup de développeurs a été de faire des changements dont le seul but était de respecter les seuils, perdant de vue l'objectif d'écrire un code de qualité. Par exemple, si une méthode était trop longue, elle pouvait être scindée en deux sans aucune logique. Finalement, la hausse de la qualité escomptée s'est traduite par une baisse. Cette étude nous montre à quel point les métriques sont sensibles au modèle de qualité dans lequel elles sont calculées et pourquoi il est difficile de les utiliser seules pour garantir du code de qualité.

C'est pourquoi, plusieurs techniques d'analyse basées sur un modèle de qualité sont apparues. Parmi elles se trouvent les mauvaises odeurs (*Bad smells/code smells,* BS) [8], qui sont des indicateurs de problèmes de conception ou d'implémentation. Les patrons de conception (*Design Patterns*, DP) sont également utilisés comme indicateurs de bonne qualité, puisqu'ils sont de bonnes solutions à des problèmes de conception récurrents et qu'ils facilitent la communication entre développeurs. À l'inverse, les défauts de conception (*Anti-patterns,* AP) [9] sont de mauvaises solutions architecturales, ils permettent donc de mettre en lumière des ensembles de classe dont la qualité architecturale est trop faible. De nombreux outils ont été développés pour détecter automatiquement les BS [26], les DP [27] ou les AP [28]. À l'heure actuelle, ces outils introduisent beaucoup de faux positifs.

2.3. Visualisation

> *« Software visualisation is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration. »* Knight et Munro [29]

Nous soutenons cette définition selon laquelle, le rôle principal d'un SV est de faciliter le plus possible le processus analytique réalisé par un utilisateur. Il est donc clair que pour être efficace, un SV doit tenir compte des capacités cognitives et physiques du système visuel humain. C'est pourquoi cette section commence par définir quelques principes visuels, puis présente les SV existants.

2.3.1. Définition de principes visuels

L'objectif de cette partie n'est pas de faire un état de l'art sur les capacités cognitives et visuelles des usagers, mais de définir certains principes auxquels nous nous référerons tout au long de ce mémoire.

2.3.1.1. La mémoire de travail

La mémoire de travail [30] est un modèle fonctionnel de la mémoire à court terme, qui permet de retenir temporairement les informations nécessaires à un processus cognitif. La mémoire de travail permet notamment de traiter les stimuli sensoriels, pour les transformer en information de plus haut niveau. Ces informations peuvent ensuite être conservées dans la mémoire de travail pour faire partie d'un processus cognitif itératif ou être transférées à la mémoire à long terme. Les capacités de la mémoire de travail sont très limitées, puisqu'elle permet de retenir seulement 7 éléments, à plus ou moins deux près, pendant 15 à 30 secondes [31]. Par ailleurs, plus la mémoire de travail est surchargée, moins le processus cognitif qu'elle réalise est efficace.

Si nous appliquons ce concept à la visualisation, nous en déduisons que :

1. Un système de visualisation doit permettre d'afficher un maximum de données simultanément, pour minimiser l'occupation de la mémoire de travail et ainsi optimiser le processus analytique.
2. Un système de visualisation qui ne permet pas d'afficher toutes les informations sous un même point de vue ne doit pas demander à ses utilisateurs de retenir plus de 7 informations simultanément. Sinon, la mémoire sature et l'utilisateur doit revenir sur les points de vues déjà observés pour se rappeler des informations oubliées.

2.3.1.2. Les variables visuelles

Une variable visuelle est une modification que l'on peut apporter à un objet graphique pour encoder de l'information. Selon Bertin [32], il existe 7 types de variables visuelles :

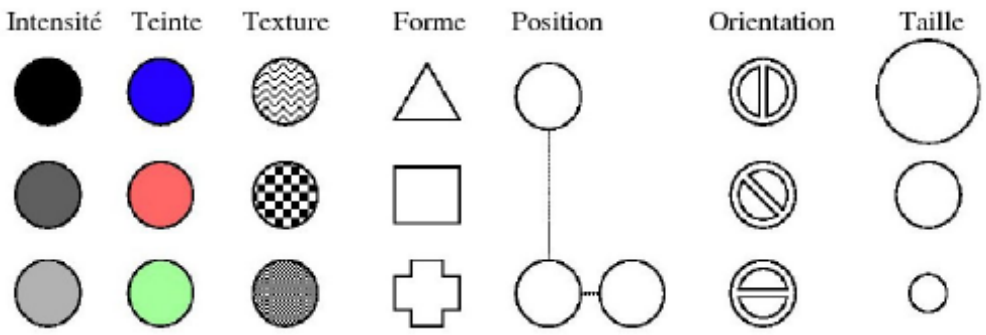

Figure 2.1 – Les 7 variables visuelles [33].

Chacune de ces variables a des caractéristiques différentes (cf. tableau 2.1). On dit d'une variable qu'elle est :

- Sélective, s'il est possible de fixer son attention sur une valeur en faisant abstraction des autres valeurs et des autres variables.
- Associative, si les valeurs de cette variable n'influencent pas la perception des autres variables.
- Quantitative, si l'observation d'une valeur de cette variable permet d'en définir la quantité, sans la comparer à d'autres valeurs.
- Ordonnée, s'il est possible de donner un ordre logique à ses valeurs. Autrement dit, s'il est possible de quantifier une de ses valeurs en la comparant aux autres.

	Intensité	Teinte	Texture	Forme	Position	Orientation	Taille
Sélective	✓	✓	✓	×	✓	✓	✓
Associative	×	✓	✓	✓	✓	✓	×
Quantitative	×	×	×	×	✓	×	✓
Ordonnée	✓	×	✓	×	✓	×	✓

Tableau 2.1 – Caractéristiques des 7 variables visuelles.

Pour un même objet, chaque variable visuelle peut être combinée avec les autres, ce qui permet d'ajouter de l'information ou de renforcer celle qui est déjà affichée. Cependant, comme chaque variable visuelle représente un aspect de l'objet sur lequel elle est affichée, il n'est pas possible de combiner une variable visuelle à elle-même sans gêner la perception des valeurs individuelles de cette variable.

2.3.1.3. La métaphore visuelle

La métaphore visuelle associe un concept d'un domaine source vers une forme visuelle d'un domaine cible. La communauté scientifique [33-36] s'accorde généralement pour dire que la métaphore visuelle est une bonne solution pour faciliter la compréhension d'un concept, car elle réutilise les connaissances et l'expérience acquise dans d'autres domaines. Cependant, plusieurs auteurs [34, 36] insistent sur l'importance du choix de la métaphore, qui doit suggérer des points d'associations naturels entre le domaine source et le domaine cible. De plus, Tufte [36] précise que la forme visuelle du domaine cible impose un certain nombre de contraintes qu'il est possible de simplifier dans sa forme métaphorique, mais que cette simplification ne doit pas enlever des détails ancrés dans l'imaginaire collectif qui permettent de reconnaître la métaphore.

2.3.1.4. Simplicité

« La simplicité, c'est la sophistication suprême » Léonard de Vinci

La simplicité joue un rôle central dans toute représentation graphique efficace [33, 34, 36], y compris en visualisation. Cependant, simplifier ne signifie pas retirer de l'information. Nielsen [34] précise que l'idée est de présenter exactement l'information dont l'utilisateur a besoin, ni plus, ni moins, au moment et à l'endroit où l'utilisateur la cherchera.

Selon Mullet et Sano [33], un design simple (1) est appréhendé assez rapidement pour immédiatement supporter une analyse approfondit, (2) est plus facilement reconnu que son équivalent sophistiqué, (3) a un plus grand impact sur l'utilisateur et (4) favorise les actions naturelles, ce qui réduit l'utilisation de la mémoire.

2.3.1.5. Principes de la Gestalt

La Gestalt est une école de pensée de la psychologie de la forme, dont le fondement est que l'interprétation des formes est le résultat d'un processus cognitif structuré qui répond à des principes de perception innés et identiques chez tous les humains [37]. La Gestalt a notamment établi six principes majeurs [33] qui régissent le processus de perception des formes.

Principe de proximité

Les éléments proches sont naturellement regroupés. La figure 2.2 est interprétée comme 4 colonnes de 4 éléments, plutôt que 4 rangées de 4 éléments.

Figure 2.2 – Principe de proximité [33].

Principe de similarité

Les éléments similaires sont naturellement regroupés. La figure 2.3 est interprétée comme 4 rangées de 4 éléments, plutôt que 4 colonnes de 4 éléments.

Figure 2.3 – Principe de similarité [33].

Principe de continuité

L'œil préfère les formes continues. La figure 2.4 sera perçue comme 2 lignes qui s'entrecroisent, plutôt que 2 formes angulaires qui s'accotent.

Figure 2.4 – Principe de continuité [33].

Principe de fermeture

L'humain cherche à voir des formes complètes, même si des contours sont manquants. La figure 2.5 sera perçue comme un triangle au-dessus de 3 disques, plutôt que comme 3 disques incomplets.

Figure 2.5 – Principe de fermeture [33].

Principe des aires

Lorsque deux figures sont superposées, la plus petite des deux sera perçue comme étant au premier plan. La figure 2.6 sera perçue comme un carré au-dessus d'un rectangle, plutôt qu'un rectangle troué.

Figure 2.6 – Principe des aires [33].

Principe de la symétrie

L'œil groupe les figures symétriques. La figure 2.7 sera perçue comme deux losanges superposés, plutôt que trois objets.

Figure 2.7 – Principe de la symétrie [33].

2.3.1.6. Minimiser les problèmes d'occlusion.

L'occlusion survient lorsque toute ou une partie d'un objet est cachée par un autre objet ou par lui-même. Pour un SV, l'occlusion devient un problème si une partie des informations visualisées est cachée. C'est pourquoi il est important de mettre en place des mesures qui minimisent les problèmes d'occlusion.

1. Les objets devraient présenter les mêmes informations, quelque soit le point d'observation. Ainsi, un objet ne peut pas générer des problèmes d'occlusion sur lui-même.
2. Les objets devraient tous avoir des proportions relativement semblables, car les gros objets cachent facilement les plus petits qui sont derrière eux.
3. Les objets devraient être répartis sur un plan. Ainsi, il est possible d'adopter un point de vue aérien depuis lequel les objets ne s'occultent pas ou peu.
4. Les objets devraient être répartis régulièrement, de façon à offrir des points de vue qui maximisent l'utilisation de l'espace d'affichage et donc qui réduisent l'occlusion.

2.3.1.7. La saturation visuelle

La saturation ou surcharge visuelle survient lorsque la densité d'information visuelle est trop grande pour que le cerveau prenne en compte tous les détails (cf. figure 2.8). Ainsi, une scène qui présente une faible quantité d'information peut être saturée, si ces informations sont réparties dans l'espace d'affichage en petits groupes denses.

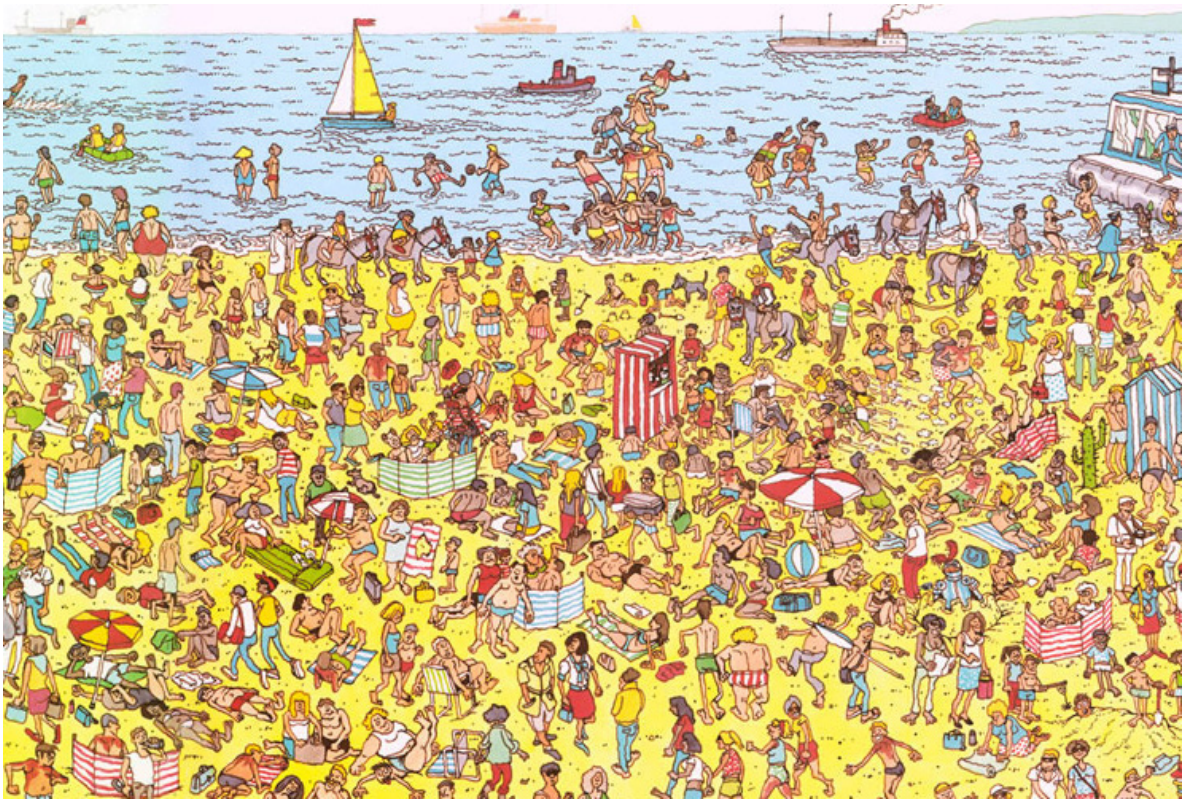

Figure 2.8 – Exemple de saturation visuelle, extrait de « Où est Charlie? » [38].

Les effets principaux de la saturation visuelle sont : (1) une augmentation du temps d'analyse de chaque information, (2) des informations sont ignorées, (3) des informations sont mal interprétées et (4) des informations déjà observées sont oubliées.

Des expériences [39, 40] montrent que ces effets sont en partie dus à l'interférence des images trop proches. En effet, pour considérer toutes les

informations d'une scène saturée, le cerveau fait des simplifications en regroupant les éléments les plus proches. La série de livres pour enfants « Où est Charlie? » [38] est un bon exemple de saturation visuelle

2.3.2. Quelques travaux sur la visualisation

La visualisation du logiciel est un domaine très vaste qui ne se limite pas à l'étude de la qualité du code. Parmi les SV dédiés à la qualité, la majeure partie sont des SV de métriques. Si ces derniers sont un bon moyen d'analyser la qualité du code, ils ne permettent pas de comprendre les facteurs qui ont influencé cette qualité. C'est pour comprendre ces facteurs que les SV de l'évolution des métriques sont apparus.

L'idée de visualiser l'évolution des métriques est assez nouvelle, ce qui explique que pour l'instant peu de systèmes existent. Par ailleurs, la différence entre ces systèmes et les SV de métriques est l'ajout de la dimension temporelle. Ce qui fait des SV de métriques une bonne base pour la visualisation de leur évolution.

Dans cette section, nous commencerons par faire un tour d'horizon des SV du logiciel, puis nous aborderons plus en détail les SV de métriques et de leur évolution.

2.3.2.1. Les systèmes de visualisation du logiciel

En 1992, Eick *et al.* proposent l'un des premiers SV du logiciel : *Seesoft* [41]. Ce système représente les lignes de code par des segments de couleur. La largeur de chaque segment est définie par la longueur de la ligne (*i.e.,* le nombre de caractères de chaque ligne). Puisqu'une longue ligne prendra plus d'espace qu'une courte, *Seesoft* fait l'hypothèse que l'importance d'une ligne est directement liée à sa taille. De son côté, la hauteur des segments est définie par le ratio entre le nombre de lignes à afficher et la hauteur (en pixel) de l'espace d'affichage. Enfin, la couleur

des segments permet de représenter les informations extraites d'un serveur CVS, comme l'auteur, l'age ou la fréquence des changements d'une ligne de code. Fort de sa popularité, *Seesoft* a inspiré de nombreux travaux, parmi lesquels figurent les trois prochains SV présentés.

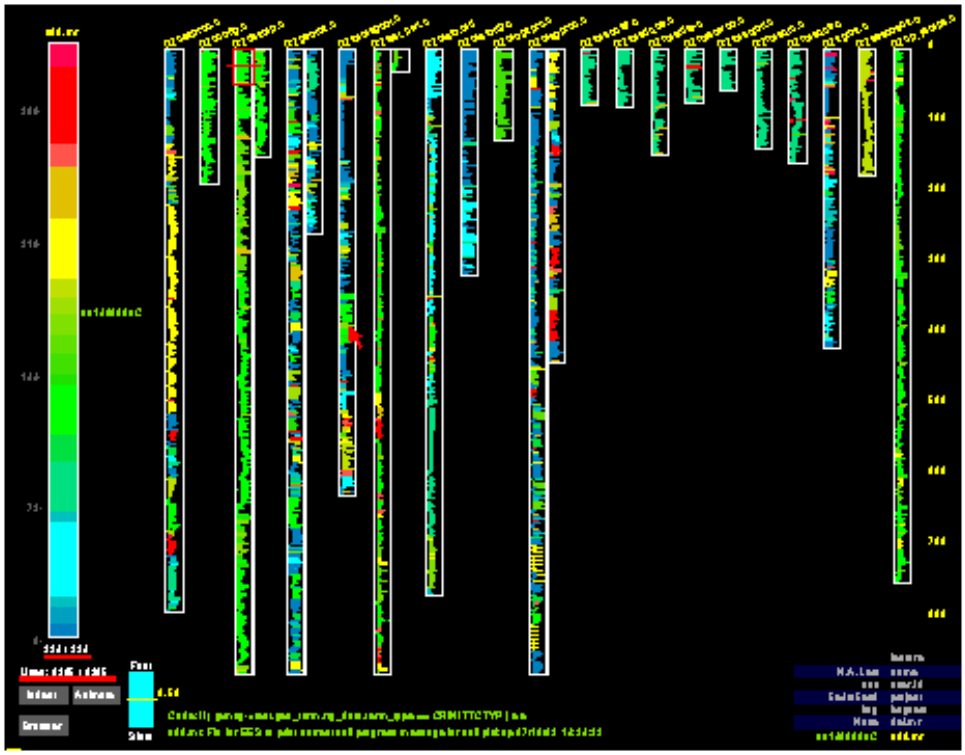

Figure 2.9 – Vue principale de *Seesoft* [41].

White Coast [42] est un SV en 3D qui permet de visualiser les informations contenues dans les journaux d'un référentiel CVS. Les fichiers sont représentés par des boites (*i.e.,* des parallélépipèdes rectangles) réparties dans un espace d'affichage cubique (cf. figure 2.10). Cette répartition trop libre a l'inconvénient de favoriser les problèmes d'occlusion (cf. section 2.3.1.6, p. 15). En contrepartie, *White Coast* à la capacité d'afficher quatre variables simultanément pour chaque fichier. Les variables sont associées sur la hauteur, la longueur, la largeur et la couleur des boites.

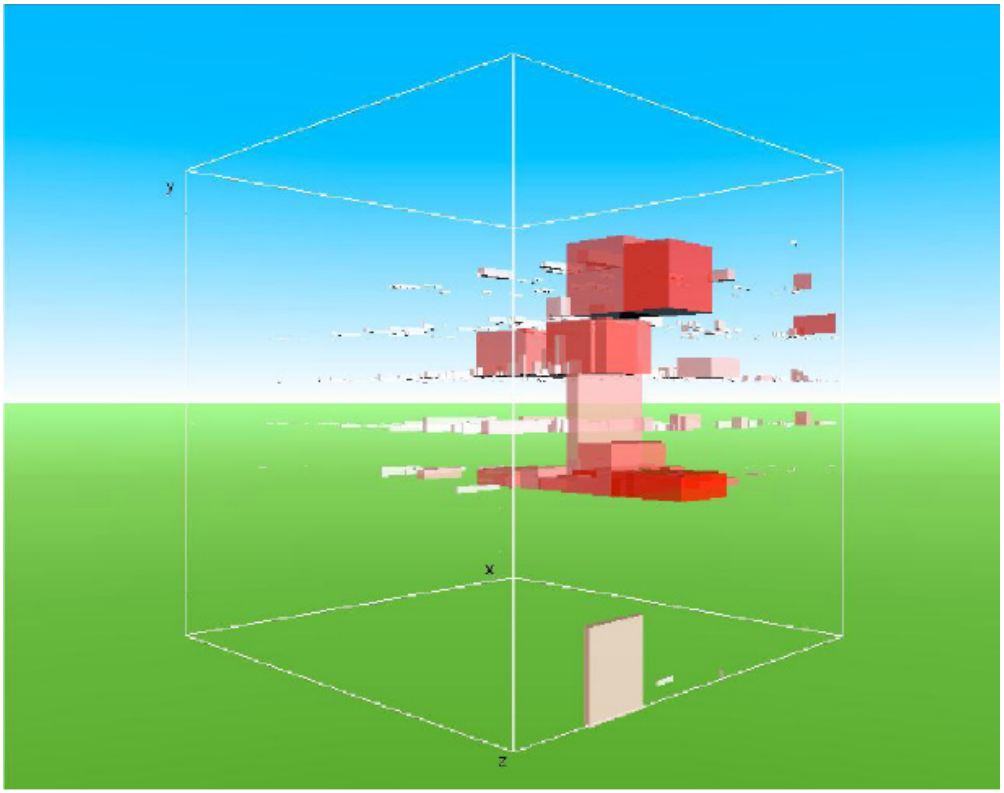

Figure 2.10 – *White Cost* [42] : vue de l'activité des fichiers du projet *Azureus*.

Softchange [43] permet de croiser les informations issues d'un CVS et d'un serveur de suivi des bogues *Bugzilla*. Ce croisement est intéressant, car il permet à la fois de comprendre les changements du code et les raisons qui ont motivé ces changements. Une fois croisées, les informations sont visualisées en 2D à travers une interface web, sous forme de diagrammes.

sv3D (*Source Viewer* 3D) [44] représente les lignes de code d'un programme sur un diagramme battons en 3D. Grâce à un affichage dense (cf. figure 2.11), ce SV permet d'afficher un grand nombre d'entités, mais les problèmes d'occlusion sont importants. Pour minimiser ces problèmes, sv3D propose un mécanisme de filtre qui rend les entités de moindres importances transparentes

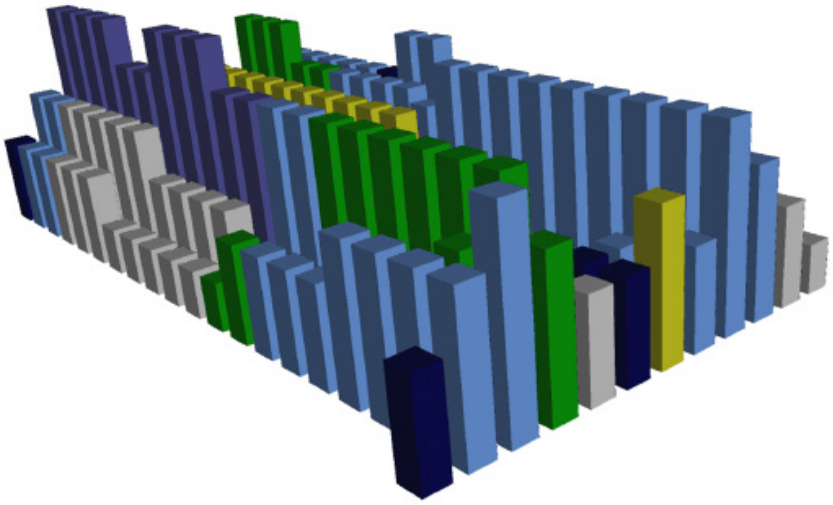

Figure 2.11 – Lignes de code d'un fichier visualisées avec sv3D [44].

Figure 2.12 – Exemple d'arrangement de vues avec *Advizor* [45].

Certains SV proposent de diviser l'affichage en plusieurs vues pour présenter plus d'informations. Chaque vue peut accueillir un principe de visualisation différent pour détailler les informations déjà affichées ou pour présenter de

nouvelles informations. Les vues sont interdépendantes, car par exemple, une sélection dans l'une des vues peut entraîner la mise à jour des autres. La plupart des SV multi-vues sont en 2D, c'est le cas de *Advizor* [45] (cf. figure 2.12) qui propose à l'utilisateur de choisir de visualiser l'information sous forme de feuilles de données, de graphes, de matrices ou de différents types de diagrammes. *Advizor* est dédié à la compréhension des changements du code et de l'architecture, mais il existe d'autres systèmes multi-vues, comme *Augur* [46] (cf. figure 2.13) qui est destiné à faciliter la communication et le travail collaboratif.

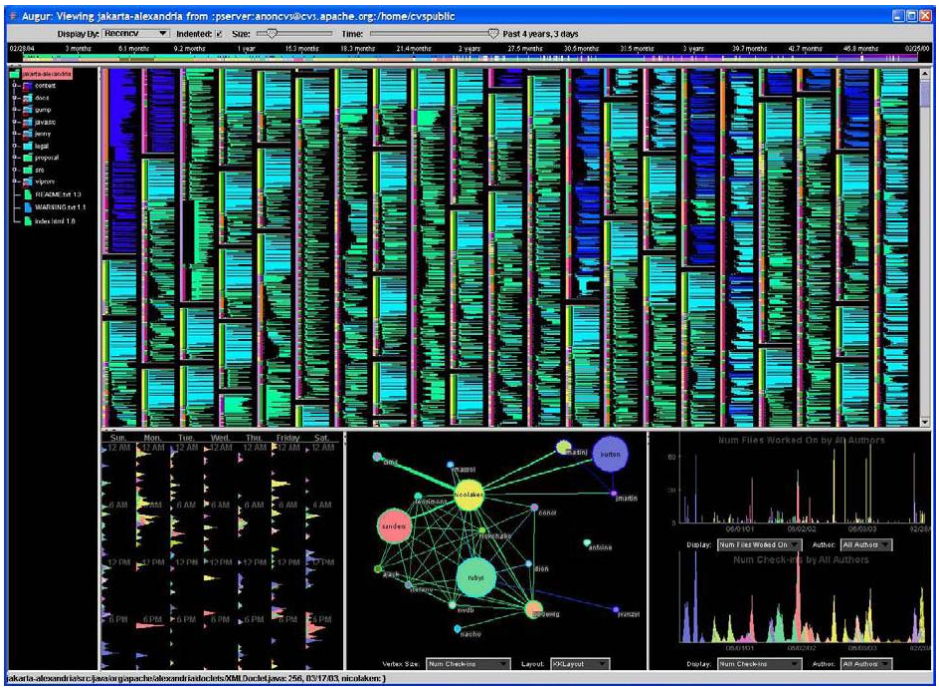

Figure 2.13 – Exemple d'arrangement de vues avec *Augur* [45].

Panas *et al.* [47] proposent également un SV destiné à faciliter la communication et le travail collaboratif. Ce CS, très réaliste, utilise des bâtiments comme des usines, des grattes-ciel ou des maisons, pour représenter respectivement des classes fabrique (*factory*), des objets gestionnaires (*manager*) ou de simples classes. Si une classe contient une erreur (*i.e.,* un *bug*), le bâtiment qui la représente prend feu. Si une classe nécessite des fonctionnalités supplémentaires, le nom de l'architecte responsable de distribuer le travail apparaît dessus. La figure

2.14 montre que les auteurs poussent la métaphore très loin, allant même jusqu'à représenter les lampadaires dans les rues, ce qui contrevient au principe de simplicité (cf. section 2.3.1.4, p. 13).

Figure 2.14 – Visualisation avec le CS proposé par Pansa *et al.* [47].

2.3.2.2. Les systèmes de visualisation de métriques

Les analyses de qualité à base de métriques nécessitent d'observer plusieurs valeurs de métriques différentes sur un grand nombre d'entités. Pour être efficace, un SV de métriques doit donc permettre d'afficher à la fois suffisamment de métriques et d'entités, ce qui est loin d'être trivial.

L'algorithme du *treemap* [48] permet de représenter un arbre sur un plan, tout en maximisant l'utilisation de l'espace d'affichage. Le *Voronoi treemap* [49] est un SV 2D basé sur cet algorithme. Contrairement à un *treemap* traditionnel [48], les nœuds et les feuilles ne sont pas des rectangles (cf. figure 2.15), mais des polygones aléatoires créés par des tesselations de Voronoi. Les polygones ainsi générés ont un aspect ratio hauteur/largeur relativement balancé, ce qui permet de mieux les regrouper dans l'espace qu'avec un *treemap* traditionnel. Pour permettre la lecture de la hiérarchie de l'arbre, l'épaisseur des bordures des polygones est calculée en fonction de leur niveau d'imbrication. Malheureusement, ce SV permet

de visualiser seulement deux métriques simultanément sur chaque feuille de l'arbre (*i.e.,* les polygones de plus bas niveau). Ces métriques sont représentées par la superficie et la couleur des polygones.

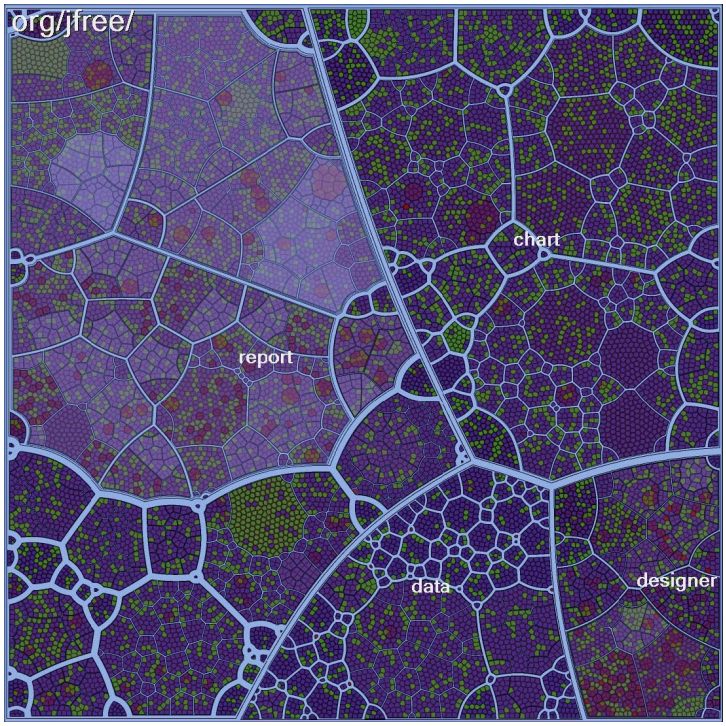

Figure 2.15 – Visualisation avec le *Voronoi treemap* [48].

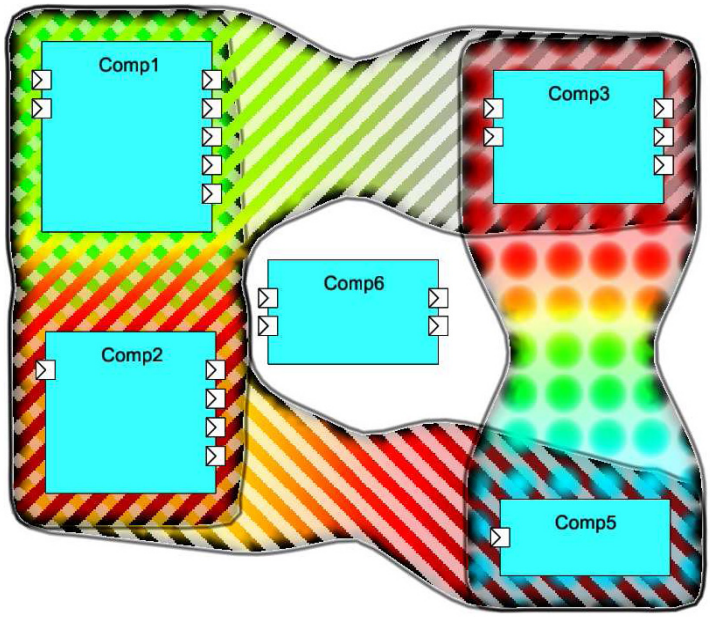

Figure 2.16 – Diagramme de composants vu avec le SV de Byelas et Telea [50].

Plusieurs travaux [50-53] enrichissent des diagrammes UML, en particulier des diagrammes de classes. Comme les développeurs utilisent régulièrement UML, ce principe à l'avantage d'être rapide à prendre en main et de s'intégrer facilement aux méthodes de travail existantes. Cependant, il ne permet pas d'ajouter beaucoup d'informations, car il faut éviter la surcharge visuelle (cf. section 2.3.1.7, p. 16) et les diagrammes UML sont déjà très chargés.

Byelas et Telea [50] visualisent en 2D des métriques de qualité sur des diagrammes de classes et de composants. L'idée est d'ajouter des textures en arrière-plan des éléments du diagramme (cf. figure 2.16) pour représenter les métriques. Les couleurs des textures permettent de représenter la valeur des métriques. Grâce à cinq textures différentes, ce SV permet d'observer cinq métriques simultanément. Cependant, la superposition de textures pose un sérieux problème de lisibilité (cf. section 2.3.1.2, p. 11). En effet, d'une part l'utilisateur doit faire un effort non négligeable pour différencier leurs couleurs. Et d'autre part, la superposition de textures génère de nouvelles textures. La figure 2.16 montre l'exemple de la superposition de deux textures rayées selon des diagonales opposées, qui génèrent une texture en damier.

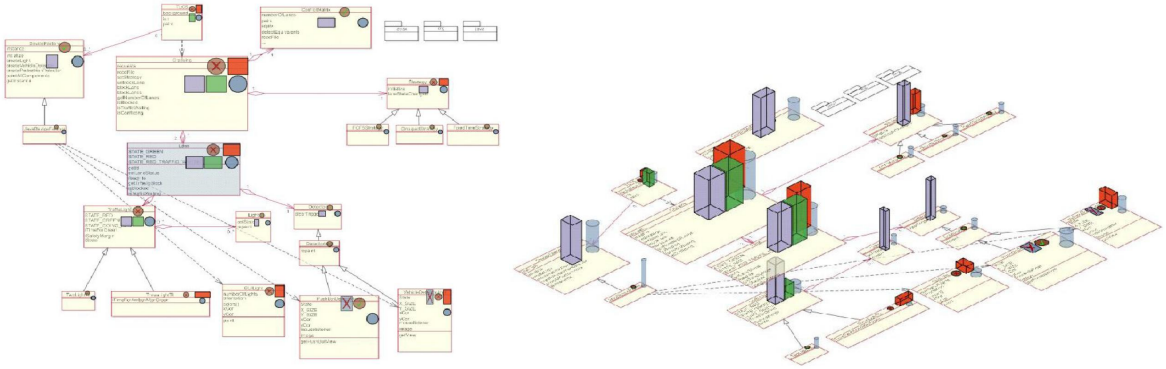

Figure 2.17 – *MetricView* [53] en 2D (à gauche) et en 3D (à droite).

MetricView [53] est un système à la fois en 2D et en 3D, qui visualise des métriques sur des diagrammes classes. La vue en 2D (cf. figure 2.17 gauche) permet

d'explorer le diagramme librement et de choisir quelles métriques visualisées sur chaque classe. Les métriques sont ajoutées à l'intérieur des classes du diagramme et sont représentées par des cercles ou des rectangles de couleurs différentes. Pour observer la valeur des métriques, il faut passer en 3D (cf. figure 2.17 droite). Les cercles deviennent alors des cylindres et les rectangles des parallélépipèdes rectangles dont la hauteur représente la valeur de la métrique. Cette approche est intéressante, car l'ajout d'une troisième dimension pour les valeurs de métriques permet de limiter la surcharge du diagramme.

Il existe d'autres SV qui enrichissent des diagrammes UML, mais qui ne sont pas destinés à la visualisation de métriques. Par exemple, Lanza et Ducasse [51] ajoutent des informations structurelles sur des diagrammes de classes pour faciliter la compréhension du code. McNair *et al.* [52] visualisent les informations contenues dans les journaux d'un référentiel CVS sur des diagrammes de classes.

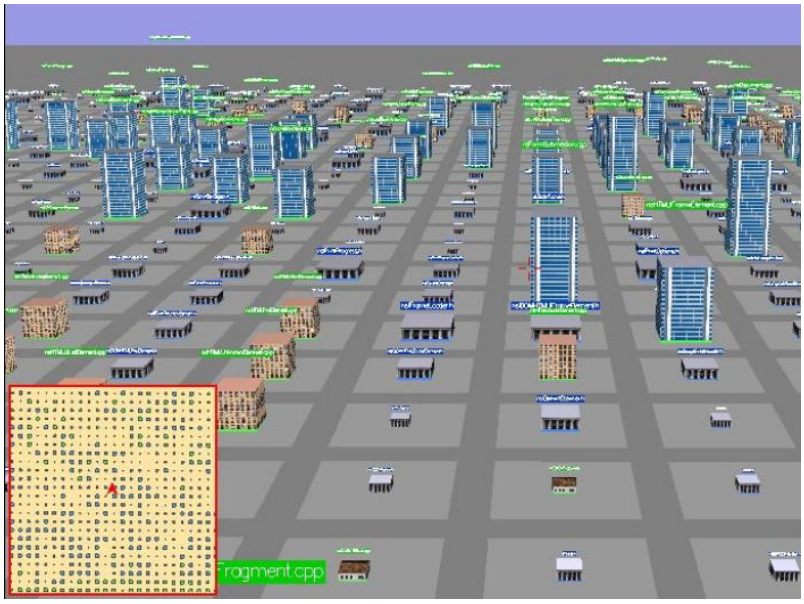

Figure 2.18 – Visualisation avec *EvoSpace* [54].

EvoSpace [54] est un CS qui permet de visualiser des métriques et d'explorer l'architecture d'un système. Les classes sont représentées par les bâtiments, qui peuvent être des maisons, des immeubles résidentiels ou des gratte-ciels selon leur

nombre de lignes de code. La hauteur et la teinte des bâtiments peuvent varier pour représenter deux métriques. Il est possible de visiter l'intérieur des bâtiments pour voir les attributs et les méthodes de la classe qu'il représente. Enfin, l'utilisateur peut interagir avec les bâtiments pour afficher, à la demande, leurs liens avec les autres classes du système.

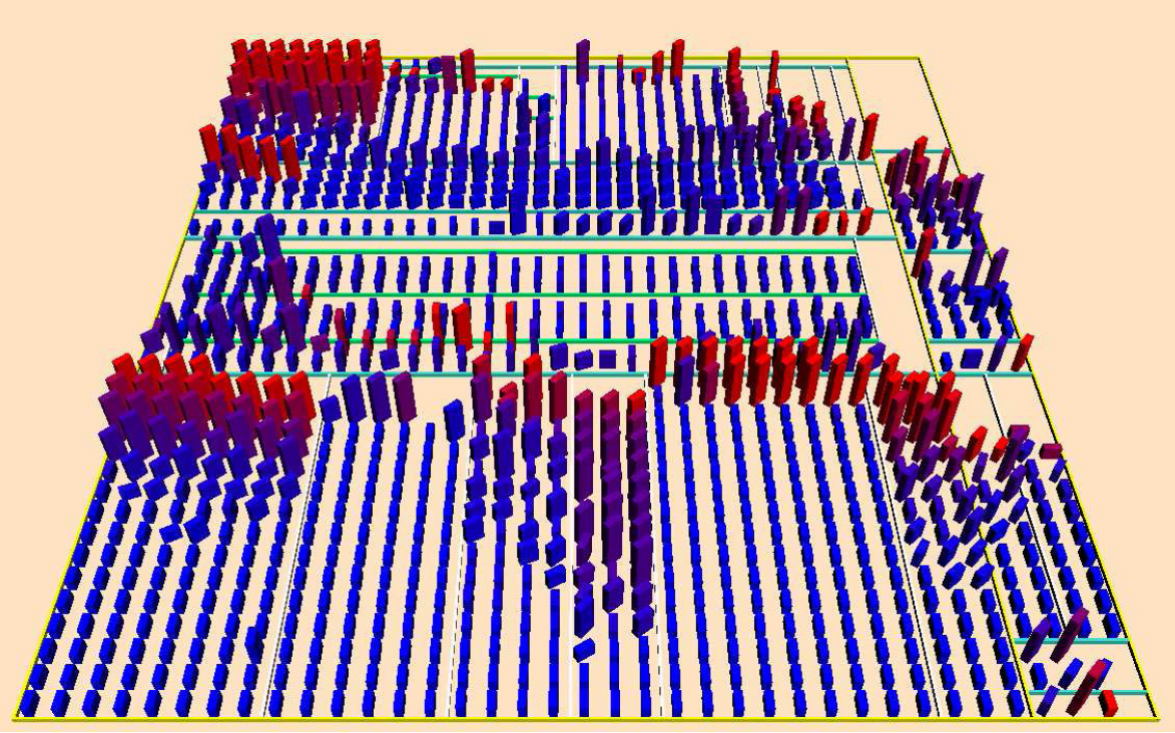

Figure 2.19 – Visualisation avec *Verso* [55].

Notre SV s'inspire de *Verso* [55], car nous pensons que c'est l'un des SV dédiés aux métriques les plus prometteurs. Les bâtiments de ce CS, qui représentent les classes, sont de simples parallélépipèdes rectangles dont la hauteur, la couleur et l'orientation peuvent varier pour représenter des valeurs de métriques. Ce graphisme simple et épuré a l'avantage de présenter une information claire et sans bruit (cf. section 2.3.1.4, p. 13). Par ailleurs, les bâtiments sont tous inclus dans un volume limite (*bounding box*), ce qui (1) évite les disproportions et (2) permet de les aligner sur une grille régulière. Comme nous l'avons vu à la section 2.3.1.6, ces deux mesures permettent de réduire les problèmes d'occlusion.

Verso utilise l'organisation géographique des bâtiments pour représenter la structure des paquetages et de leurs classes. La grille d'affichage est divisée en cellules par un *treemap* et les bâtiments (*i.e.,* les classes) sont placés dans la cellule qui correspond à leur paquetage (*i.e., package*). Cette nouvelle façon d'organiser la géographie d'un CS est intéressante, car elle permet d'explorer la hiérarchie des paquetages et de leurs classes, tout en optimisant l'espace d'affichage. En effet, *Verso* permet d'afficher plus de 5000 classes sans surcharge visuelle. Cependant, pour garantir l'alignement des cellules du *treemap* sur la grille, deux modifications ont été apportées à l'algorithme original [48]. D'une part, les divisions effectuées par l'algorithme doivent être discrétisées suivant la grille. Lors de sa division, il faut donc arrondir supérieurement la taille des cellules, pour garantir qu'elles contiennent tous leurs descendants. C'est pourquoi la superficie d'une cellule de ce *treemap* ne représente pas le nombre d'entités dont elle est l'ascendant, mais en est une borne supérieure. D'autre part, si plusieurs cellules parentes ont une frontière commune, seule la frontière de plus haut niveau est représentée. Cette mesure permet de garantir que l'épaisseur des frontières entre les cellules soit fixe et donc qu'elle n'induira pas de décalage sur la grille. Cependant, cette simplification implique que l'utilisateur ne peut pas déduire le niveau d'imbrication d'une cellule en observant seulement ses frontières, car il doit élargir sa vue afin de compter ses parents. Nous pensons que cette perte d'information est la source de problèmes de lisibilité, car elle contrevient au principe de simplicité (cf. section 2.3.1.4, p. 13).

Le CS proposé par Wettel et Lanza [56] s'inspire de Verso, mais apporte plusieurs modifications qui changent son principe de visualisation. Comme Verso, ce CS utilise des bâtiments pour représenter les classes. Les bâtiments sont également de simples parallélépipèdes rectangles à section carré (cf. : figure 2.20), mais dont les trois variables visuelles sont (1) la largeur de la section, (2) la hauteur et (3) la couleur des boites. En utilisant la largeur de la section et la hauteur comme variables visuelles, les auteurs permettent aux bâtiments d'être

disproportionnés. Par exemple, un bâtiment peut être large et plat, alors qu'un autre est haut et étroit. Si cette idée permet à l'utilisateur de combiner rapidement deux valeurs de métriques, elle pose des problèmes d'occlusion (cf. section 2.3.1.6, p. 15), car un gros bâtiment peut facilement en cacher de plus petits. De plus, comme les bâtiments n'ont pas la même section, ils ne peuvent pas être alignés sur une grille, ce qui favorise également l'occlusion.

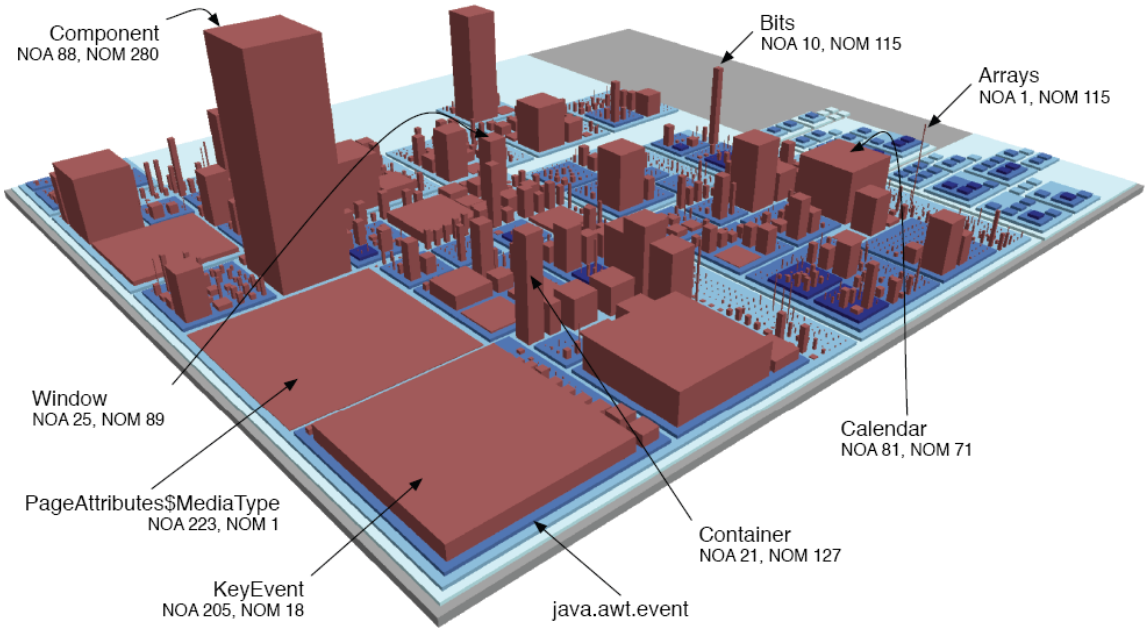

Figure 2.20 – Visualisation avec le CS de Wettel et Lanza [56].

Pour représenter la structure des paquetages et de leurs classes, les auteurs réutilisent l'idée du *treemap*, introduite par Verso. Par contre, ce *treemap* a l'avantage de représenter les bordures de toutes ses cellules, ce qui facilite la lecture de leur niveau d'imbrication. De plus, comme les bâtiments ne sont pas alignés sur une grille, les auteurs ont pu mettre en œuvre une version légèrement modifiée de l'algorithme du *squarified treemap* [57]. Cet algorithme crée des cellules dont le ratio hauteur/largeur est mieux balancé qu'avec l'algorithme classique [48], ce qui a l'avantage d'éliminer le problème de longues bandes [48, 55, 57] générées par les nœuds de haut niveau avec peu de descendants.

2.3.2.3. Les systèmes de visualisation de l'évolution des métriques

La capacité d'affichage est clairement la problématique principale des SV de l'évolution des métriques, car l'ajout de la dimension temporelle multiplie la quantité de données à afficher par le nombre d'intervalles de cette dimension.

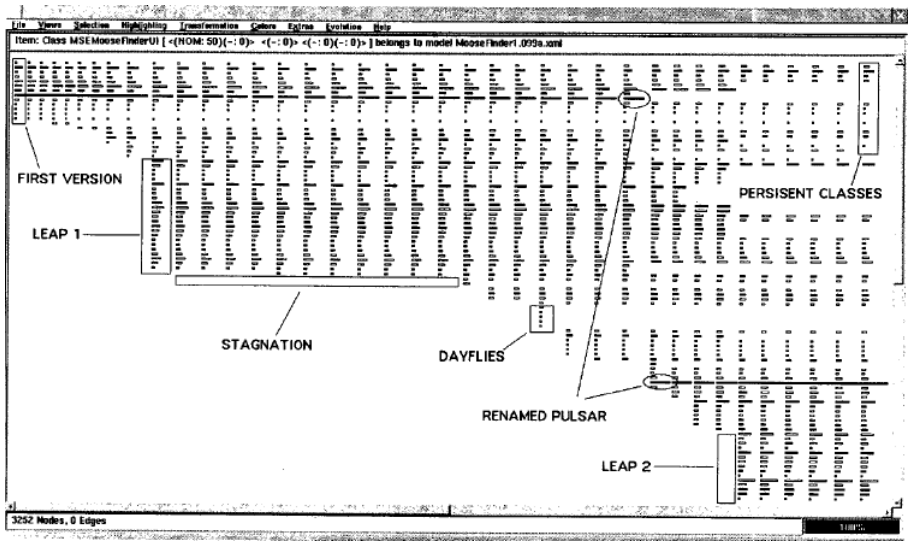

Figure 2.21 – Visualisation avec *Evolution Matrix* [58].

Evolution Matrix [58] est le premier SV entièrement dédié à la visualisation de l'évolution des métriques, il a été proposé en 2001 par Lanza. Ce SV en 2D propose une vue matricielle (cf. figure 2.21) contenant des rectangles dont la longueur et la largeur peuvent varier pour représenter des valeurs de métriques. Les lignes de la matrice représentent les fichiers et les colonnes représentent leurs versions. Comme beaucoup de SV en 2D, *Evolution Matrix* ne permet pas d'afficher beaucoup d'entités, car l'affichage est rapidement saturé (cf. section 2.3.1.7, p. 16).

Le SV 2D proposé par Pinzger *et al.* [59] (cf. figure 2.22) utilise des graphes dont les nœuds sont des diagrammes en toile d'araignée. Cette approche est intéressante, car les diagrammes en toile d'araignée permettent d'afficher l'évolution d'une dizaine de métriques et sont efficaces pour comparer leurs valeurs. Cependant, les graphes ne permettent pas d'afficher beaucoup d'éléments, car ils ont une structure très riche et les arcs occupent une grande partie de l'espace

d'affichage. Par ailleurs, la position des nœuds est faite pour favoriser la lisibilité du graphe et non pour maximiser l'utilisation de l'espace d'affichage, car (1) il faut minimiser le nombre d'arcs qui se croisent, (2) les noeuds voisins dans la structure du graphe devraient être proches dans sa représentation graphique (cf. Principe de proximité, p.14) et (3) il faut suffisamment écarter les noeuds pour que l'affichage des arcs ne soit pas trop dense, c'est-à-dire, pour éviter la saturation visuelle (cf. section 2.3.1.7, p. 16).

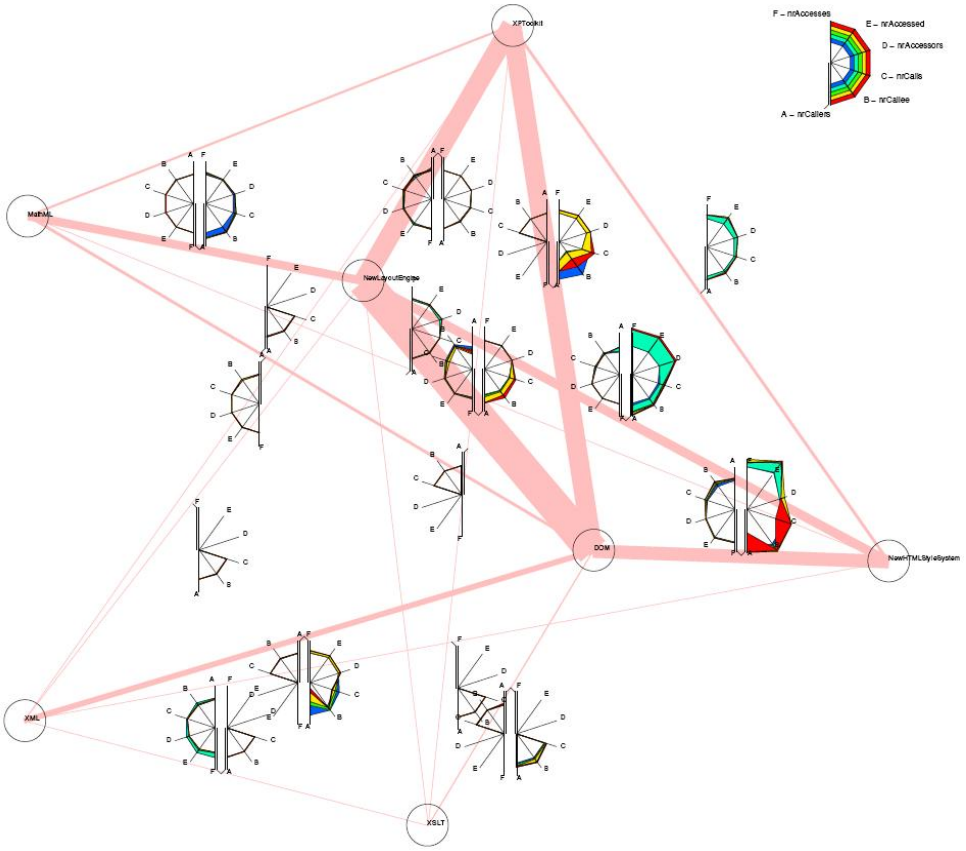

Figure 2.22 – Exemple du SV de Pinzger *et al.* [59].

Verso [55] est un SV avant tout dédié à la visualisation des métriques sur une seule version d'un logiciel, mais il propose deux solutions pour visualiser l'évolution des métriques. La première consiste à préparer un *treemap*, pour chaque version du logiciel, puis à les visualiser côte à côte. Nous pensons que cette solution n'est pas satisfaisante, car pour suivre l'évolution d'une classe, l'utilisateur doit

mettre en correspondance la position du bâtiment qui la représente dans chaque *treemap* (*i.e.,* dans chaque version). Par ailleurs, comme le constate l'auteur de Verso, cette solution ne permet pas d'afficher suffisamment d'entités, ni de versions.

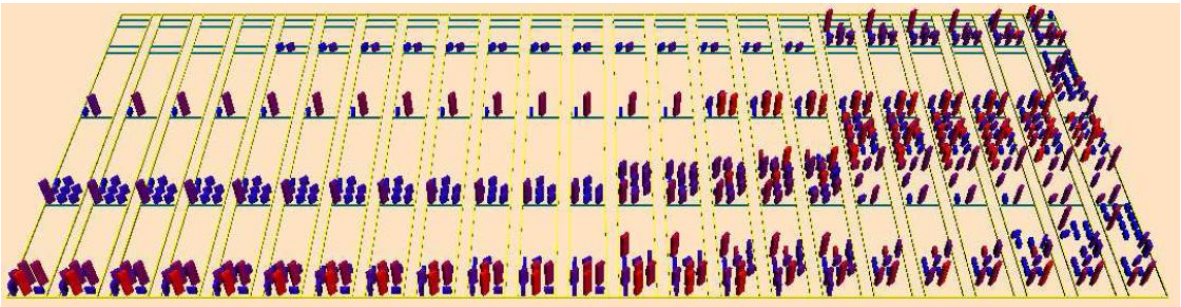

Figure 2.23 – Visualisation de l'évolution des métriques avec *Verso* [55].

La deuxième solution consiste à utiliser des animations pour afficher séquentiellement les versions du logiciel à visualiser. Les animations permettent (1) de mettre en évidence les variations entre les deux versions et (2) de mettre en correspondance les bâtiments. Comme les versions sont affichées séquentiellement, l'utilisateur doit faire appel à sa mémoire de travail pour retenir les informations utiles au processus analytique. Or, comme nous l'avons vue à la section 2.3.1.1, la mémoire de travail permet de stocker 5 à 9 informations pendant une durée d'environ 15 secondes, ce qui est largement insuffisant pour l'observation d'une scène de plusieurs centaines d'entités. Par ailleurs, l'animation ajoute des informations sur le changement qui viennent aggraver la surcharge de cette mémoire. Enfin, pour observer deux versions éloignées l'utilisateur doit traverser toutes les versions intermédiaires, ce qui pose des problèmes de navigation et consomme du temps sur la mémoire de travail.

Wettel et Lanza [60] proposent une extension au SV [56] présenté dans la section précédente, qui permet de visualiser l'évolution des métriques. Leur solution s'inspire de Verso et consiste à afficher les versions du logiciel à visualiser en

séquence, mais sans les séparer par des animations. Nous pensons que, comme pour *Verso*, cette solution se heurte aux limitations de la mémoire de travail.

2.4. Discutions sur l'état de l'art

Les SV en 2D peuvent réutiliser des concepts comme les graphes ou les diagrammes. C'est un avantage majeur, car ces concepts sont largement utilisés, notamment sur papier, ils sont donc facilement reconnus et compris. Cependant, en 2D la capacité d'affichage reste limitée. Par exemple, plusieurs systèmes [41, 49, 58] proposent au plus deux variables visuelles par entité, alors que d'autres [53, 58] ne permettent pas d'afficher un grand nombre d'entités simultanément. C'est pour cette raison que plusieurs systèmes [45, 46] proposent des vues multiples. L'avantage majeur est de permettre d'afficher l'information sous différents points de vue (*i.e.,* afficher de nouvelles informations) ou à différents niveaux de granularités (*i.e.,* détailler les informations affichées). Cependant, les SV multi-vues ont deux inconvénients majeurs. D'une part, l'utilisateur doit faire un effort d'apprentissage pour savoir quelles vues sont disponibles, à quoi elles servent et comment les interpréter. D'autre part, l'utilisateur doit faire un effort non négligeable pour mettre ces vues en correspondance, ce qui complique les changements de point de vue ou de niveau de granularité. Pour rendre cette mise en correspondance possible, les informations structurelles doivent être dupliquées. Ces informations dupliquées, ajoutées à la multiplication des vues génèrent souvent une surcharge visuelle.

En 3D l'espace d'affichage est beaucoup plus grand, il n'est donc pas nécessaire d'afficher plusieurs vues. Beaucoup de SV en 3D [42, 54-56, 60] permettent d'afficher trois variables visuelles ou plus, sur un grand nombre d'entités. La 3D offre également beaucoup de liberté pour imager de nouveaux concepts de visualisation, c'est pourquoi il faut faire attention à ne pas tomber dans certains pièges. Tout d'abord, il faut appliquer le principe de simplicité (cf.

section 2.3.1.4, p. 13) et bannir les graphismes trop réalistes [47]. D'une part, car les SV sont faits pour communiquer de l'information à ses utilisateurs, la communication ne doit donc pas être bruitée par des informations inutiles. D'autre part, parce que les SV sont destinés à des professionnels, ils n'ont pas besoin de graphismes attrayants comme un jeu vidéo. Les SV en 3D doivent également faire attention à l'éclairage, car l'intensité lumineuse baisse avec la distance ce qui modifie les couleurs. Or, l'éclairage des scènes 3D est une imitation simpliste de la réalité, ce qui rend sa mise en place délicate. Enfin, il faut particulièrement faire attention aux placements des objets, s'ils sont répartis trop librement dans l'espace, ils génèrent des problèmes d'occlusion.

Parmi les SV en 3D, nous choisissons de faire un CS, car nous pensons que ce concept laisse beaucoup de liberté, tout en imposant certaines contraintes intéressantes. Leur organisation sur un plan limite les problèmes d'occlusion. Les changements de point de vue et de niveau de granularité sont naturels, puisqu'il suffit de se déplacer ou de zoomer au-dessus d'un plan, de la même façon que l'on observerait un jeu d'échec. Mais surtout, les CS sont parmi les SV en 3D qui offrent la plus grande capacité d'affichage. Pourtant aucun CS n'arrive à visualiser efficacement l'évolution des métriques d'un système sur une image unique. Nous souhaitons donc trouver un nouveau concept de CS qui offre une plus grande capacité d'affichage. Pour ce faire, nous proposons de visualiser de l'information sur le côté des bâtiments. Or, la plupart des CS n'utilisent pas le coté des bâtiments parce que leur base est souvent occultée par le sommet des bâtiments environnants. Dans la réalité, nous réglons ces petits problèmes d'occlusion par un simple déplacement de la tête, mais comme un écran affiche une image en 2D, il n'est pas possible de faire la même chose. L'utilisateur doit donc changer de point de vue en utilisant le clavier et la souris, qui sont deux instruments en 2D peu précis. De plus, contrairement à un déplacement de la tête, les changements de point de vue à l'aide du clavier et de la souris ne sont pas des réflexes, ils obligent donc

l'utilisateur à arrêter, puis à reprendre le processus d'analyse des données, ce qui n'est pas acceptable.

Pour utiliser efficacement le côté des bâtiments, nous proposons d'étendre la 3D classique avec un système peu intrusif qui permet de localiser la tête de l'utilisateur par rapport à l'écran et ainsi d'afficher l'image correspondante à son point de vue. Autrement dit, ce système transforme un écran standard en une fenêtre ouverte sur le monde 3D qu'il affiche. Pour régler les problèmes d'occlusion, l'utilisateur aura simplement besoin de déplacer sa tête vis-à-vis de l'écran.

CHAPITRE 3.

EOD

Les SGV et les techniques d'analyse de la qualité du code évoluent rapidement. C'est pourquoi notre SV, *Eyes Of Darwin* (EOD), est un CS générique pour la visualisation de l'évolution du logiciel, qui permet d'afficher toutes les données respectant un format d'entrer défini. Ainsi, EOD associe la dimension verticale au temps. Ses bâtiments représentent l'évolution des entités du code et sont constitués par un empilement de sections représentant les versions de l'évolution du code pour cette entité (cf. figure 3.1). Chaque section permet de visualiser de 2 à 6 variables, qui peuvent être continues, discrètes ou aléatoires.

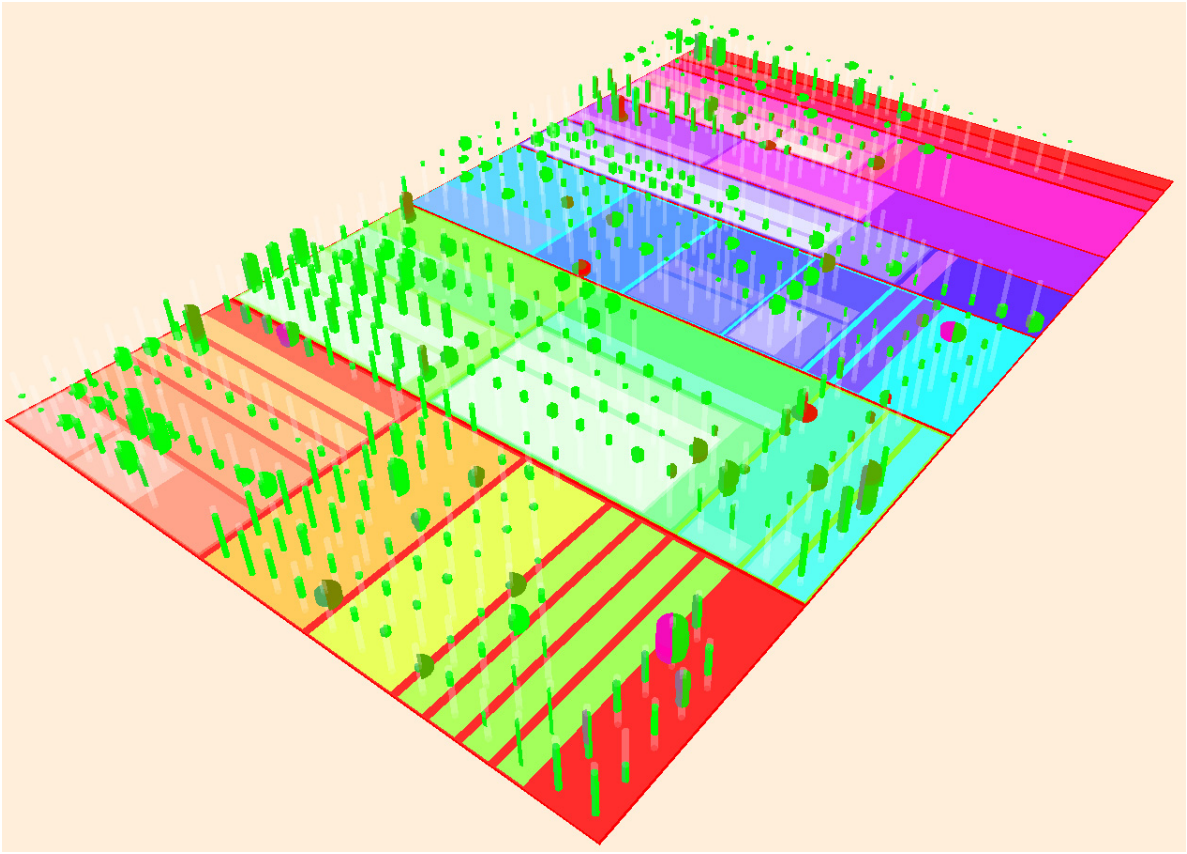

Figure 3.1 – EOD.

l'organisation géographique des bâtiments est faite avec un *treemap*, qui permet de représenter la structure du code. Enfin, les données d'entrée fournies à EOD définissent : (1) le niveau de granularité des entités représentées par les bâtiments (p. ex. modules, classes ou méthodes), (2) celui des versions représentées par leurs sections (p. ex. chaque mise à jour ou les révisions majeures), (4) les données visualisables et (5) la structure représentée par le *treemap*.

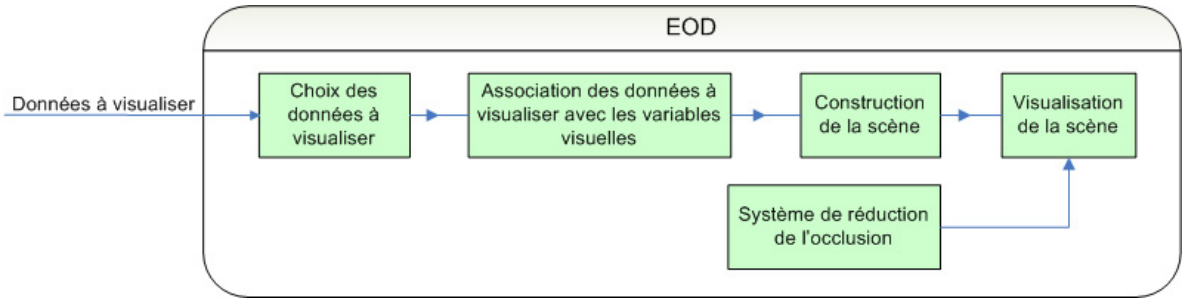

Figure 3.2 – Processus de visualisation d'EOD.

Le processus de visualisation d'EOD se définit en 5 étapes (cf. figure 3.2) :

1. Nous choisissons les données à visualiser parmi la liste des données disponibles et nous définissons éventuellement des transformations à appliquer à ces données, pour les transformer en données de plus haut niveau.

2. Nous associons les données à visualiser avec les variables visuelles (VV) et nous définissons la méthode de calcul des valeurs des VV en fonction des données.

3. Nous pouvons maintenant construire la scène à visualiser en fonction des données à visualiser et de leur association avec les VV.

4. La scène est alors affichée, avec l'éclairage et les caméras nécessaires.

5. Nous ajoutons à l'environnement de la scène notre système de réduction de l'occlusion.

Pour démarrer ce processus, nous devons fournir des données à EOD. C'est pourquoi, nous avons réalisé un outil capable de collecter des informations sur l'évolution d'un code donnée. Cet outil permet d'obtenir un modèle mémoire du contenu d'un SGV et utilise un processus en 4 étapes pour collecter des informations sur l'évolution du code qu'il contient. Nous définissons un cliché comme une unité temporelle, qui représente l'état du référentiel à un instant donné et qui est constitué de l'ensemble des versions des fichiers présents à cet instant. Tout d'abord, nous créons un cliché pour chaque groupe de fichiers mis à jour en même temps. Puis, nous utilisons la suite logicielle d'analyse de la qualité *Ptidej* [27], réalisée dans notre laboratoire, pour établir un méta-modèle du code contenu dans chaque cliché et calculer des métriques de classes sur ce méta modèle. Pour chaque cliché, nous sauvegardons la hiérarchie des paquetages jusqu'aux classes et interfaces. Enfin, pour chaque classe ou interface, dans chaque cliché, nous sauvegardons les informations qui la caractérise, comme par exemple leur niveau de visibilité, les méta-informations associées à leur fichier source dans le SGV et, bien sûr, les valeurs des métriques calculées dessus.

Par conséquent, dans ce mémoire, (1) les bâtiments d'EOD sont associés aux classes et aux interfaces, (2) les sections des bâtiments sont associées aux clichés, (3) les données visualisables sont les valeurs des métriques calculées, les informations qui caractérisent les classes et les interfaces, ainsi que les méta-informations provenant du fichier source sur le SGV, enfin (4) l'organisation géographique des bâtiments (*i.e.,* le *treemap*) permet de représenter la hiérarchie des paquetages.

Ce chapitre est organisé selon les étapes du processus de visualisation d'EOD. Nous y abordons successivement le choix de données à visualiser, le processus d'association des données à visualiser avec les VV, la construction de la scène, sa visualisation et enfin, le système de réduction de l'occlusion.

3.1. Choix des données à visualiser

Le choix des données à visualiser est important, car ce sont ces données qui supporteront le processus analytique des usagers. L'utilisateur peut les choisir directement parmi la liste des données disponibles. Cependant, (1) certaines données ne sont pas disponibles pour tous les types d'entité et (2) l'utilisateur peut désirer filtrer ces données, pour les transformer en données de plus haut niveau.

Dans cette partie, nous présentons les mécanismes mis en place pour traiter ces deux cas.

3.1.1. Données différentes selon le type des entités

La liste des données disponibles varie selon les types d'entités, car toutes les données n'ont pas de signification pour tous les types d'entités. Par exemple, le nombre des méthodes implémentées n'a pas de sens pour une interface. Si l'utilisateur choisit un de ces types de données, il devra fournir une valeur alternative du même type (*i.e.,* continue, discrète ou aléatoire). Cette valeur alternative peut être une constante ou être choisie parmi les autres données disponibles. Par exemple, le nombre de méthodes implémentées peut être remplacé par le nombre de méthodes déclarées, pour les interfaces.

3.1.2. Les filtres

Nous proposons un système de filtres qui permet d'effectuer des tris ou des calculs sur les données, pour les transformer en données de plus haut niveau. Nous définissons un filtre comme une opération qui reçoit en entrée une ou plusieurs sources de données d'un type défini et retourne en sortie les données transformées, d'un type également défini. Les sources de données fournies au filtre peuvent provenir des données d'entrée d'EOD ou d'un autre filtre. Ainsi, il est possible d'imbriquer les filtres pour faire des opérations complexes.

Dans ce mémoire, nous proposons une série de quatre filtres, qui effectuent des opérations basiques de calcul ou de tri de données. Ainsi, en les imbriquant, l'utilisateur a la possibilité de créer un grand nombre d'opérations de haut niveau.

Filtre boite de Tukey

Intention : observer une variable selon sa distribution en boite de Tukey [61] (*i.e.,* boite à moustaches).

Opération : groupe les valeurs d'une source de données, pour les discrétiser selon leur distribution en boite de Tukey. Les valeurs de la source de données peuvent être discrétisée en les groupant : (1) toutes ensembles, (2) par entité, (3) par cliché, (4) par paquetage et (5) par paquetage et cliché.

Données d'entrée :

- une source de données continues;
- la méthode de groupement des valeurs;
- les noms des groupes de sorti, si l'utilisateur souhaite modifier les valeurs par défaut.

Données de sortie : données discrètes, l'un des 6 groupes suivants : valeurs minimales, premier décile au premier quartile, premier quartile à la médiane, médiane au troisième quartile, troisième quartile au neuvième décile, valeurs maximales.

Exemple d'utilisation : observer la distribution du couplage entre les entités du code pour chaque cliché.

Filtre associatif

Intention : associer les valeurs d'une ou plusieurs sources de données, pour les grouper en données de plus haut niveau.

Opération : utilise une matrice de correspondance entre les valeurs d'entrées possibles, pour fournir une valeur de sortie.

Données d'entrée :

- une ou plusieurs sources de données discrètes;
- la matrice de correspondance entre les valeurs d'entrées et de sorties.

Données de sortie : données discrètes dont les valeurs possibles sont définies par la matrice de correspondance.

Exemple d'utilisation :

- mettre en évidence les valeurs extrêmes d'une boite de Tukey, en les associant selon l'un des groupes : valeur minimale, entre le premier et le troisième quartile, et valeur maximale;
- associer le type des entités et la propriété « est abstraite », pour séparer les entités en deux groupes : classe abstraite ou interface, et classe non abstraite.

Filtre arithmétique

Intention : effectuer des calculs sur les données d'entrée

Opération : applique une opération arithmétique (addition, soustraction, multiplication ou division) sur les données d'entrée.

Données d'entrée :

- Deux sources de données continues, dont l'une peut être une constante.
- L'opérateur arithmétique.

Données de sortie : données continues. Le résultat de l'opération arithmétique.

Exemple d'utilisation : compter le nombre de méthodes qui ne peuvent pas être polymorphiques, en faisant la différence entre le nombre total de méthodes et le nombre de méthodes susceptibles d'être polymorphiques.

Filtre de seuil

Intention : observer si une valeur dépasse un seuil ou pas, plutôt que la valeur elle-même.

Opération : compare les valeurs d'entrées avec un seuil, pour définir si elles le dépassent ou non. L'utilisateur peut choisir de grouper le cas d'égalité avec les valeurs inférieures ou les valeurs supérieures au seuil.

Données d'entrée :

- une source de données continues;
- le seuil;
- la méthode de gestion du cas d'égalité;
- le nom des groupes de sortie, si l'utilisateur souhaite modifier les valeurs par défaut.

Données de sortie : données discrètes. Selon la méthode de gestion du cas d'égalité, l'une des deux variables discrètes suivantes : « supérieur au seuil » et « inférieur ou égal au seuil », ou « supérieur ou égal au seuil » et « inférieur au seuil ».

Exemple d'utilisation : mettre en évidence les classes ou les interfaces qui ont plus de 8 méthodes.

3.2. Association des données avec les variables visuelles

EOD utilise deux types de VV (cf. section 2.3.1.2, p. 11) : la taille des objets et leur couleur. La taille permet de visualiser une variable, alors que la couleur permet de soit visualiser une variable, deux variables ou une variable par rapport à un seuil.

Toutes les VV d'EOD varient sur un intervalle de valeurs bornées. Pour cela, nous calculons leurs valeurs à partir de ratios compris entre 0 et 1. Le processus d'association entre les données et les VV se sépare donc en deux étapes distinctes : la normalisation des données et leur association avec les VV.

Dans cette partie, nous commençons par décrire le processus de normalisation des données. Puis, nous abordons l'association de ces données avec les VV.

3.2.1. Normalisation des données

Le processus de normalisation est un prétraitement qui transforme toute variable en une variable continue, normalisée entre 0 et 1. Il permet de définir la méthode de distribution des valeurs d'une variable à visualiser sur l'intervalle des valeurs de la VV.

Nous proposons quatre méthodes de normalisation. La première normalise les variables discrètes. La deuxième discrétise les variables aléatoires, puis les normalise. La troisième méthode normalise les variables continues. Enfin, la quatrième méthode normalise les variables continues par rapport à un seuil. Les trois premières méthodes dépendent seulement du type de la variable à normaliser. La quatrième méthode doit être calculée sur une variable continue et associée à la couleur. En effet, la couleur est la seule VV d'EOD dont les valeurs peuvent être calculées en fonction d'un seuil.

3.2.1.1. Normalisation des variables discrètes

La normalisation des variables discrètes revient à distribuer les valeurs possibles de cette variable entre 0 et 1. Pour cela, nous proposons deux mécanismes. Le premier est d'ordonner les valeurs de la variable puis de les distribuer linéairement entre 0 et 1. La figure 3.3 (gauche) montre l'exemple du niveau de visibilité des entités distribué linéairement entre 0 et 1.

Le deuxième mécanisme consiste simplement à distribuer les valeurs manuellement. Ainsi, l'utilisateur peut décider de faire une distribution non linéaire. Comme toutes les VV utilisées par EOD sont ordonnées (cf. section

2.3.1.2, p. 11), il est alors possible de mettre plus de contraste sur certaines valeurs, pour favoriser leur perception et le regroupement par similarité (cf. section 2.3.1.4, p. 13). La figure 3.3 (droite) montre l'exemple des valeurs de sortie du filtre en boite de Tukey, dont les valeurs extrêmes sont mises en évidence.

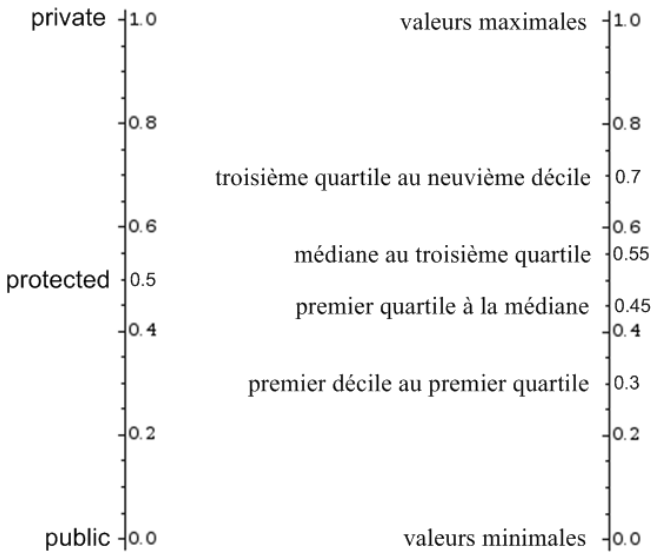

Figure 3.3 – Distribution linéaire (à gauche) et non linéaire (à droite) des valeurs d'une variable discrète.

3.2.1.2. Discrétisation et normalisation des variables aléatoires

Pour EOD, la différence entre une variable aléatoire et une variable discrète est que les variables discrètes sont accompagnées d'une liste des valeurs possibles. Le processus de discrétisation des variables aléatoires revient donc à établir une liste des valeurs possibles pour la variable. Pour cela, EOD fait une liste des valeurs distinctes de la variable aléatoire et laisse l'utilisateur compléter cette liste si des valeurs possibles sont absentes de la distribution. Les variables ainsi discrétisées sont ensuite distribuées entre 0 et 1, tel que décrit dans la section précédente (cf. section 3.2.1.1, p. 42).

3.2.1.3. Normalisation des variables continues

Nous normalisons les variables continues de la façon suivante : soit x une variable continue et f la fonction de normalisation.

$$\forall x \in \left[X_{-}; X_{+} \right]$$
$$f(x) = \frac{x - X_{-}}{X_{+} - X_{-}}$$

Formule 3.1 – Normalisation des variables continues.

Certaines variables présentent des bornes naturelles, par exemple un pourcentage. Dans ce cas, l'utilisateur peut saisir manuellement les bornes X_{-} et X_{+} Dans le cas contraire, nous proposons cinq méthodes pour calculer ces bornes automatiquement. La première méthode est simple : EOD parcourt toutes les valeurs de la variable pour trouver les extremums qui serviront de bornes pour la normalisation. La fonction de normalisation devient :

$$f(x) = \frac{x - Min(x)}{Max(x) - Min(x)}$$

Formule 3.2 – Normalisation globale des variables continues.

Les quatre autres méthodes séparent la variable à normaliser en sous-ensembles, pour mettre en avant leurs changements. Pour cela, nous cherchons les extremums des sous-ensembles et nous normalisons chaque valeur de la variable avec les extremums de son sous-ensemble.

Normalisation par entité

La normalisation par entité permet de s'attarder sur les changements internes d'une variable au sein d'une même entité.

Soit n entités visualisées et $x_i \mid i \in [0;n]$, l'ensemble des valeurs de x pour la i-ième entité. La fonction de normalisation devient :

$$\forall i \in [0;n]$$
$$f(x_i) = \frac{x_i - Min(x_i)}{Max(x_i) - Min(x_i)}$$

Formule 3.3 – Normalisation par entité des variables continues.

Normalisation par cliché

La normalisation par cliché permet de s'attarder sur les changements d'une variable au sein d'un même cliché.

Soit m clichés visualisés et $x_j \mid j \in [0;m]$, l'ensemble des valeurs de x pour le j-ième cliché. La fonction de normalisation devient :

$$\forall j \in [0;m]$$
$$f(x_j) = \frac{x_j - Min(x_j)}{Max(x_j) - Min(x_j)}$$

Formule 3.4 – Normalisation par cliché des variables continues.

Normalisation par paquetage

La normalisation par paquetage permet de s'attarder sur les changements d'une variable au sein d'un même paquetage.

Soit p clichés visualisés et $x_k \mid k \in [0;p]$, l'ensemble des valeurs de x pour le k-ième paquetage. La fonction de normalisation devient :

$$\forall k \in [0;p]$$
$$f(x_k) = \frac{x_k - Min(x_k)}{Max(x_k) - Min(x_k)}$$

Formule 3.5 – Normalisation par paquetage des variables continues.

Normalisation par paquetage et cliché

La normalisation par paquetage et cliché permet de s'attarder sur les changements d'une variable au sein d'un même paquetage et d'un même cliché.

Soit $x_{kj} \mid k \in [0; p], j \in [0; m]$, l'ensemble des valeurs de x pour le k-ième paquetage et le j-ième cliché. La fonction de normalisation devient :

$$\forall k \in [0; p]$$
$$\forall j \in [0; m]$$
$$f(x_{kj}) = \frac{x_{kj} - Min(x_{kj})}{Max(x_{kj}) - Min(x_{kj})}$$

Formule 3.6 – Normalisation par paquetage et cliché des variables continues.

3.2.1.4. Normalisation des variables continues avec un seuil

EOD permet de visualiser la distribution d'une variable par rapport à un seuil. Dans ce cas, les valeurs de la variable doivent être normalisées par rapport à ses extremums et à son seuil.

Nous normalisons les variables avec seuil de la façon suivante : soit x une variable continue, s un seuil et g la fonction de normalisation. Nous avons:

$$\forall s \in ¡$$
$$\forall x \in \left\{ \left[X\_; X\_+ \right] \mid (s - X\_ = X\_+ - s) \right\}$$
$$g(x,s) = \begin{cases} \dfrac{x - X\_}{2(s - X\_)}, \forall x \in \left[X\_; s \right] \\ \dfrac{x - s}{2(X\_+ - s)} + \dfrac{1}{2}, \forall x \in \left] s; X\_+ \right] \end{cases}$$

Formule 3.7 – Normalisation des variables continues avec un seuil.

La première étape de normalisation consiste à trouver des bornes pour x. De la même façon que pour la normalisation d'une variable continue sans seuil (cf. section précédente), l'utilisateur peut fournir ces bornes ou les calculer avec l'une des cinq méthodes automatiques (*i.e.,* globale, par entité, par cliché, par paquetage ou par paquetage et cliché).

La deuxième étape consiste à trouver le seuil. Il peut être fourni manuellement ou calculé. Nous proposons de le déterminer en calculant soit la moyenne ou soit la médiane. Comme pour les bornes, nous proposons soit de déterminer un seuil global, en le calculant sur l'ensemble des valeurs de la variable, soit de séparer la variable à normaliser en sous-groupes et de calculer un seuil pour chaque sous-groupe. Ainsi, il est possible de calculer le seuil, indépendamment des bornes, par entité, par cliché, par paquetage ou par paquetage et entité.

Nous disposons maintenant, pour chaque valeur de la variable, d'un seuil s et de bornes X_-^* et X_+^*, tel que $\forall x \in \left[X_-^*; X_+^*\right]$. La troisième étape consiste à s'assurer que les valeurs inférieures et supérieures au seuil seront normalisées selon la même échelle. C'est pourquoi, pour chacun de ces couples seuil/bornes, nous recalculons les bornes de façon à ramener le seuil au milieu de celles-ci, tel que :

$$
\begin{aligned}
&\forall s \in ¡ \\
&\forall x \in \left[X_-^*; X_+^*\right] \\
&\left[X_-; X_+\right] = \begin{cases} \left[X_-^*; 2s - X_-^*\right], & \text{si } s - X_- > X_+ - s \\ \left[2s - X_+^*; X_+^*\right], & \text{sinon} \end{cases} \\
&\Leftrightarrow \left[X_-^*; X_+^*\right] \subseteq \left\{ \left[X_-; X_+\right] \middle| (s - X_-; X_+ - s) \right\}
\end{aligned}
$$

Formule 3.8 – Rééquilibrage des bornes pour la normalisation des variables continues avec seuil.

Enfin, à l'aide de la fonction g, nous normalisons chaque valeur de x avec son seuil s et ses bornes $X_{\min}$ et $X_{\max}$.

3.2.2. Association des données normalisées avec les variables visuelles

Nous disposons maintenant de variables (ou données) normalisées, que nous pouvons associer avec les VV. Toutes les variables peuvent être associées à l'un des deux types de VV d'EOD. Cependant, il faut être prudent lors des choix d'associations, car la taille et la couleur ont des caractéristiques différentes.

La taille des objets permet de visualiser une variable. Elle doit être associée en priorité aux variables continues, car c'est la seule VV quantitative. Par ailleurs, l'utilisateur doit garder à l'esprit que les objets de grandes tailles seront perçus plus facilement que les petits, car la taille n'est pas une VV associative. Donc, lorsque l'utilisateur cherche à repérer les valeurs faibles d'une variable, il est préférable d'utiliser un filtre arithmétique, pour associer la taille avec l'inverse de cette variable. Par exemple, si l'utilisateur cherche des classes faiblement cohésives, il peut associer la taille des objets avec l'inverse de la métrique de cohésion. Il est ainsi possible de mettre en avant les entités qui ont un problème de cohésion (*i.e,* qui sont faiblement cohésives), plutôt que l'inverse.

La couleur permet de visualiser une ou deux variables. Elle peut également être associée aux variables continues, mais l'utilisateur devra estimer ses valeurs par comparaison, car c'est une VV ordonnée. Si la couleur est associée à une seule variable continue, nous proposons de visualiser les valeurs de cette variable par rapport à un seuil. Autrement dit, nous proposons de visualiser dans quelles proportions les valeurs d'une variable dépassent ou sont en dessous d'un seuil. Par ailleurs, nous calculons la couleur des objets à partir de couleurs de base qui représentent les points extrêmes du domaine d'entrée. Le choix des couleurs de base ne doit pas être fait au hasard, car certaines couleurs ont une signification sous-jacente. Par exemple, le rouge suppose généralement un problème, alors que le vert laisse supposé l'inverse. Donc, si la couleur est associée à la métrique de cohésion, il

est préférable d'associer les valeurs fortes avec le vert et les valeurs faibles au rouge, plutôt que l'inverse.

Nous allons maintenant décrire les méthodes de calcul des valeurs des VV à partir des variables normalisées.

3.2.2.1. La taille

Le calcul de la taille des objets graphiques revient simplement à utiliser les valeurs de la variable normalisée x_{norm}, comme un ratio d'accroissement entre $t_{\min}$ et $t_{\max}$, respectivement, la taille minimale et maximale des objets. La figure 3.4 montre la variation des tailles en fonction de x_{norm}, si $t_{\min} = 20$ et $t_{\max} = 120$.

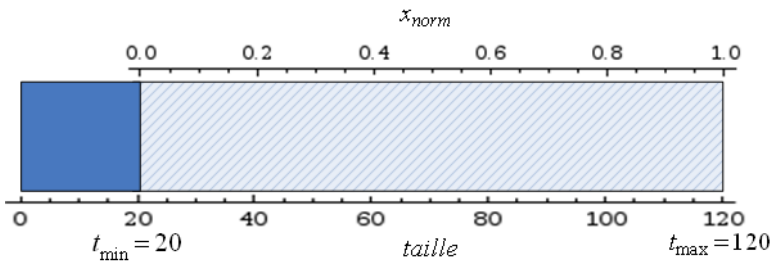

Figure 3.4 – Espace de variation de la taille.

Nous calculons la taille des objets avec la fonction t, tel que :

$$t(x_{norm}) = x_{norm} * t_{\max} + t_{\min}$$
$$\Rightarrow \forall x_{norm} \in [0;1], t(x_{norm}) \in [t_{\min}; t_{\max}]$$

Formule 3.9 – Fonction de calcul de la taille.

La figure 3.5 montre l'exemple de la taille obtenue, pour $x_{norm} = 0,7$, si $t_{\min} = 20$ et $t_{\max} = 120$.

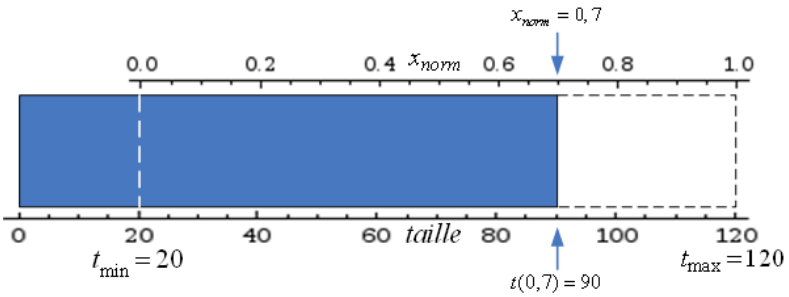

Figure 3.5 – Calcul de la taille d'un objet.

3.2.2.2. La couleur

EOD utilise des couleurs additives à composantes rouge, verte et bleue, qui sont des entiers compris entre 0 et 255. Cependant, les fonctions que nous utilisons pour les calculer sont définies sur l'ensemble des réels. Donc, nous arrondissons les valeurs de ces composantes à l'entier le plus proche, après chaque calcul.

Dans ce mémoire, nous notons une couleur c comme un triplet $\{r, v, b\}$ de ses trois composantes rouge, verte et bleue; tel que r, v et $b \in [0;255]$.

Couleur calculée avec deux couleurs de base

Cette méthode de calcul permet d'associer la couleur à une variable normalisée : x_{norm}. Nous utilisons deux couleurs de base : c_- et c_+. La couleur c_- représente les valeurs minimales de x_{norm} et la couleur c_+ représente ses valeurs maximales. La couleur des objets est ensuite calculée en effectuant un balancement linéaire entre c_- et c_+, proportionnel aux valeurs de x_{norm}. La figure 3.6 montre un exemple de l'association du domaine de la variable x_{norm}, avec le domaine de sortie défini par les couleurs c_- et c_+, respectivement verte et rouge.

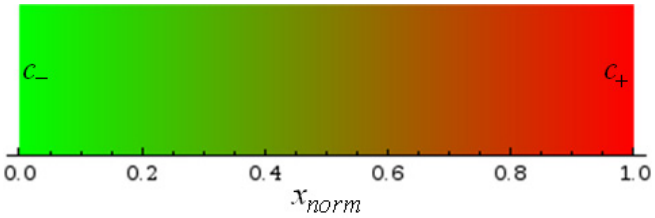

Figure 3.6 – Association d'une variable avec deus couleurs.

Chaque composante de la couleur de sortie c est calculée indépendamment avec la fonction f, qui prend en paramètre x_{norm} et les composante cp_- et cp_+, provenant respectivement des couleurs c_- et c_+. Nous avons :

$$f(x_{norm}, cp_-, cp_+) = x_{norm} * (cp_+ - cp_-) + cp_-$$

Formule 3.10 – Fonction de calcul des composantes d'une couleur.

Tel que :

$$\forall x_{norm} \in [0;1]$$
$$\forall c_- = \{r_-, v_-, b_-\} \mid r_-, v_-, b_- \in [0;255]$$
$$\forall c_+ = \{r_+, v_+, b_+\} \mid r_+, v_+, b_+ \in [0;255]$$

$$c = \{f(x_{norm}, r_-, r_+); f(x_{norm}, v_-, v_+); f(x_{norm}, b_-, b_+)\}$$

Formule 3.11 – Calcul de la couleur à partir d'une variable.

La figure 3.7 montre l'exemple des couleurs calculées à partir de quatre valeurs de x_{norm}.

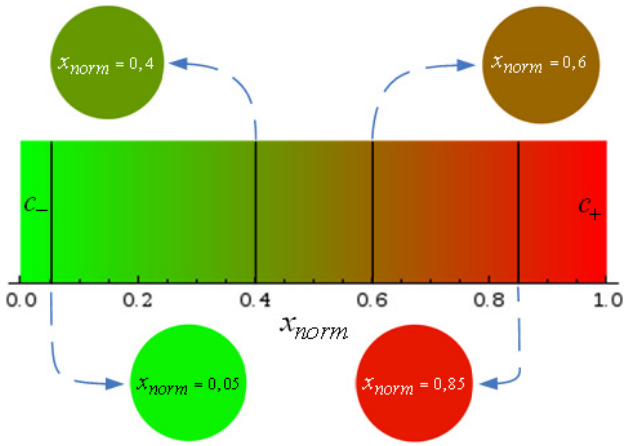

Figure 3.7 – Calculs de la couleur à partir d'une variable.

Couleur calculée avec quatre couleurs de base

Cette méthode de calcul de la couleur permet de visualiser deux variables normalisées x_{norm} et y_{norm}. Nous utilisons quatre couleurs de base : c_{--}, c_{+-}, c_{-+} et c_{++}. Ces quatre couleurs permettent d'associer le domaine d'entrée et de sortie. La couleur c_{--} représente les valeurs minimales de x_{norm} et y_{norm}. La couleur c_{+-} représente les valeurs maximales de x_{norm} et minimales de y_{norm}. La couleur c_{-+} représente les valeurs minimales de x_{norm} et maximales de y_{norm}. Et enfin, la couleur c_{++} représente les valeurs maximales de x_{norm} et y_{norm}. La figure 3.8 montre un exemple d'association des variables x_{norm} et y_{norm}, avec les couleurs c_{--}, c_{+-}, c_{-+} et c_{++}, respectivement blanche, bleue, verte et noire.

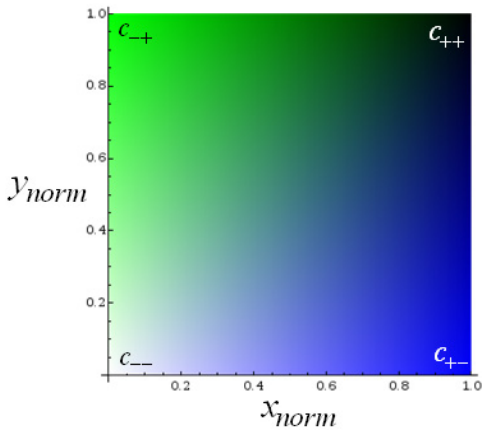

Figure 3.8 – Association d'une variable avec quatre couleurs.

Ensuite, nous calculons chaque composante de la couleur de sortie c avec la fonction f (cf. Couleur calculée avec deux couleurs de base, p. 50), en effectuant un balancement linéaire en deux temps. Le premier balancement permet de trouver les bornes c_{y-} et c_{y+} de l'intervalle de couleurs, sur lequel les valeurs de y_{norm} peuvent variées, pour une valeur de x_{norm} donnée.

Nous avons :

$$\forall x_{norm} \in [0;1]$$
$$\forall c_{--} = \{r_{--}, v_{--}, b_{--}\} \mid r_{--}, v_{--}, b_{--} \in [0;255]$$
$$\forall c_{+-} = \{r_{+-}, v_{+-}, b_{+-}\} \mid r_{+-}, v_{+-}, b_{+-} \in [0;255]$$
$$\forall c_{-+} = \{r_{-+}, v_{-+}, b_{-+}\} \mid r_{-+}, v_{-+}, b_{-+} \in [0;255]$$
$$\forall c_{++} = \{r_{++}, v_{++}, b_{++}\} \mid r_{++}, v_{++}, b_{++} \in [0;255]$$

$$c_{y-} = \{f(x_{norm}, r_{--}, r_{+-}); f(x_{norm}, v_{--}, v_{+-}); f(x_{norm}, b_{--}, b_{+-})\}$$
$$c_{y+} = \{f(x_{norm}, r_{-+}, r_{--}); f(x_{norm}, v_{-+}, v_{--}); f(x_{norm}, b_{-+}, b_{--})\}$$

Formule 3.12 – Variation d'une couleur pour une valeur de x_{norm} donnée.

La figure 3.9 montre un exemple de l'intervalle de couleurs obtenu pour $x_{norm} = 0,7$.

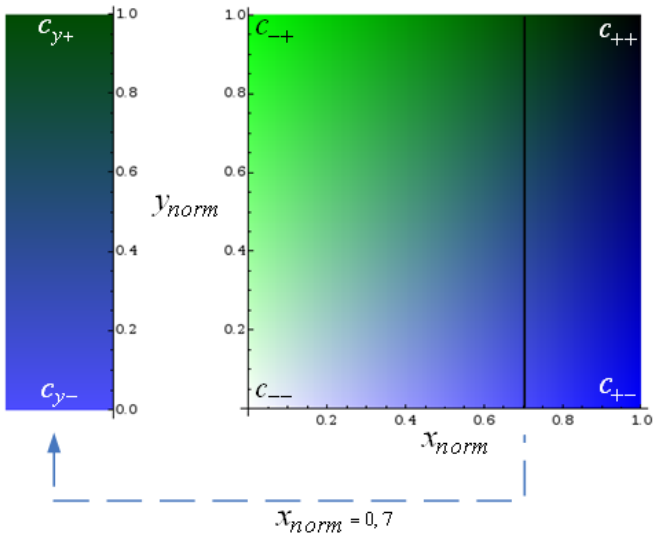

Figure 3.9 – Calcul de l'intervalle de variation de la couleur pour une valeur de x_{norm} donnée.

Nous pouvons maintenant calculer la couleur c en effectuant un balancement linéaire entre c_{y-} et c_{y+}, proportionnel à y_{norm}. Nous avons :

$$\forall y_{norm} \in [0;1]$$
$$\forall c_{y-} = \{r_{y-}, v_{y-}, b_{y-}\} \mid r_{y-}, v_{y-}, b_{y-} \in [0;255]$$
$$\forall c_{y+} = \{r_{y+}, v_{y+}, b_{y+}\} \mid r_{y+}, v_{y+}, b_{y+} \in [0;255]$$

$$c = \{f(y_{norm}, r_{y-}, r_{y+}); f(y_{norm}, v_{y-}, v_{y+}); f(y_{norm}, b_{y-}, b_{y+})\}$$

Figure 3.10 – Calcul de la couleur à partir de l'intervalle de variation obtenu pour une valeur de x_{norm} donnée.

La figure 3.11 montre la couleur c obtenu pour $x_{norm} = 0,8$ et $y_{norm} = 0,4$.

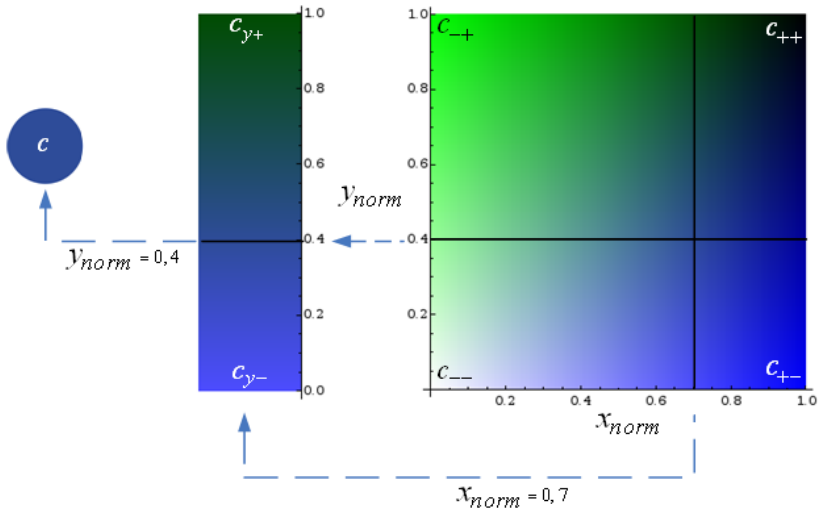

Figure 3.11 – Calcul d'une couleur à partir de deux variables.

Couleur calculée avec trois couleurs de base

Cette méthode de calcul de la couleur permet de visualiser une variable continue, normalisée par rapport à un seuil. Cependant, elle peut être utilisée avec toutes les variables normalisées.

Pour calculer la couleur c d'un objet en fonction de la variable x_{norm}, nous utilisons trois couleurs de base : c_-, c_{ct} et c_+. Les couleurs c_- et c_+ sont associées

respectivement au minimum et au maximum de x_{norm}. La couleur c_{ct} est associée au centre de l'intervalle de x_{norm}, c'est-à-dire 0,5. La figure 3.12 montre un exemple d'association de la variable x_{norm}, avec les couleurs c_-, c_{ct} et c_+, respectivement verte, blanche et rouge.

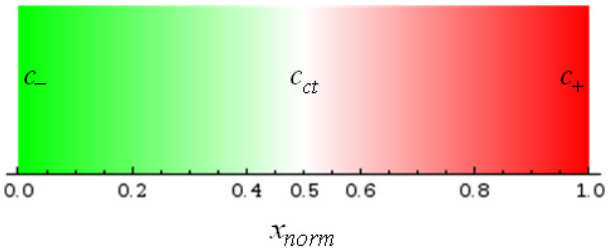

Figure 3.12 – Association d'une couleur avec une variable et un seuil.

Nous calculons ensuite les valeurs de chaque composante de c, avec la fonction g, qui fait un balancement linéaire entre les couleurs c_-, c_{ct} et c_+, proportionnel à x_{norm}. Nous avons :

$$g(x_{norm}, cp_-, cp_{ct}, cp_+) = \begin{cases} 2x_{norm} * (cp_{ct} - cp_-) + cp_-, \text{ si } x_{norm} < 0,5 \\ 2 * (x_{norm} - 0,5) * (cp_+ - cp_{ct}) + cp_{ct}, \text{ sinon} \end{cases}$$

Formule 3.13 – Fonction de calcul des composantes d'une couleur à partir d'une variable et d'un seuil.

tel que :

$$\forall x_{norm} \in [0;1]$$
$$\forall c_- = \{r_-, v_-, b_-\} \mid r_-, v_-, b_- \in [0;255]$$
$$\forall c_{ct} = \{r_{ct}, v_{ct}, b_{ct}\} \mid r_{ct}, v_{ct}, b_{ct} \in [0;255]$$
$$\forall c_+ = \{r_+, v_+, b_+\} \mid r_+, v_+, b_+ \in [0;255]$$

$$c = \{g(y_{norm}, r_-, r_{ct}, r_+); g(y_{norm}, v_-, v_{ct}, v_+); g(y_{norm}, b_-, b_{ct}, b_+)\}$$

Formule 3.14 – Calcul de la couleur à partir d'une variable et d'un seuil.

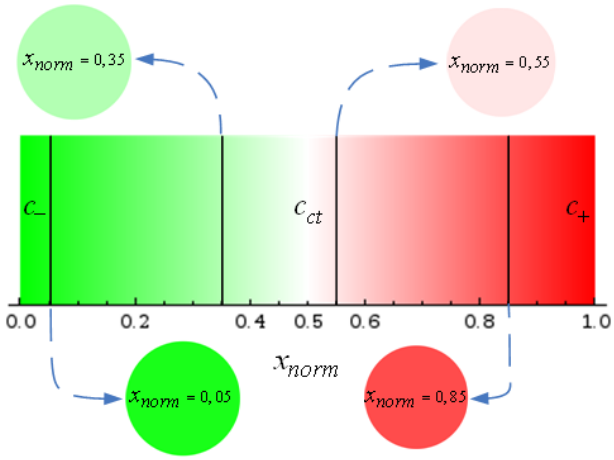

Figure 3.13 – Calculs de la couleur à partir d'une variable et d'un seuil.

La figure 3.13 montre un exemple des couleurs obtenues à partir de quatre valeurs de x_{norm}. Plus la couleur est verte plus les valeurs de x_{norm} sont en dessous du seuil. Plus la couleur s'éclaircit, plus les valeurs de x_{norm} s'approchent du seuil et plus les couleurs sont rouges, plus les valeurs du x_{norm} sont au dessus du seuil.

3.3. Construction de la scène

Nous connaissons maintenant les données à visualiser et leur association avec les VV, nous pouvons donc construire la scène. C'est à dire, construire les bâtiments avec les valeurs des VV qu'ils doivent afficher et les répartir avec un *treemap*, afin de visualiser leur structure.

3.3.1. Les bâtiments

Tous les bâtiments d'EOD (cf. figure 3.14) sont inclus dans le même volume limite (*i.e., bounding box*) à base carrée. Ainsi, nous pouvons les aligner sur une grille régulière et nous évitons les disproportions, ce qui réduit les problèmes d'occlusion (cf. section 2.3.1.6, p. 15). Les bâtiments réutilisent le concept de la pyramide des âges porté en 3D. La dimension verticale représente le temps (dates les plus récentes en haut). Les bâtiments (*i.e.,* les classes et interfaces) sont

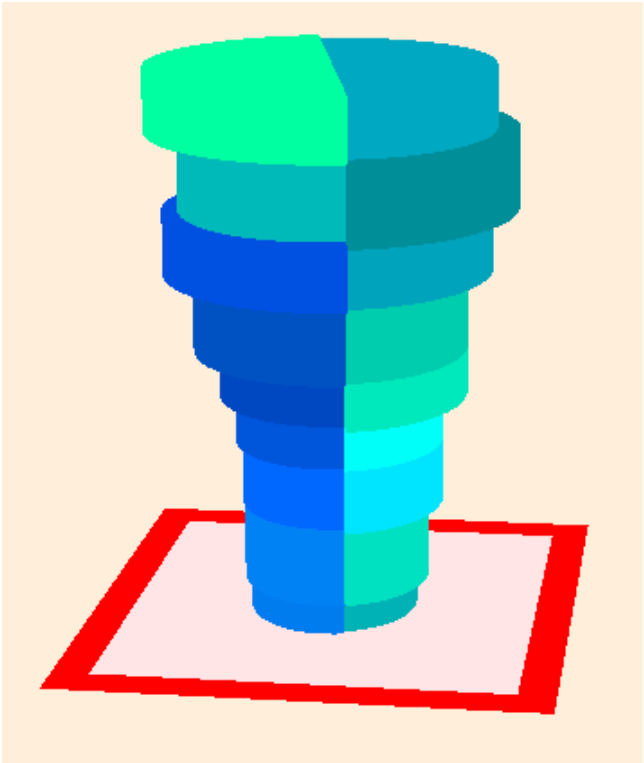

Figure 3.14 – Un bâtiment d'EOD.

constitués par un empilement de sections représentant les différentes périodes de l'évolution du code (*i.e.,* les clichés). Chaque section est constituée par deux demi-cylindres, mis dos à dos. Les demi-cylindres offrent chacun deux variables visuelles : leur rayon (*i.e.,* leur taille) et leur couleur. Il est possible d'associer la même variable, aux deux rayons ou aux deux couleurs. Comme la couleur permet de visualiser une ou deux variables et la taille une seule, chaque section peut visualiser de 2 à 6 variables. Enfin, le rayon des demi-cylindres est borné entre 10% et 50% de la largeur des cellules des bâtiments. Cette mesure permet de garantir qu'aucun bâtiment ne dépasse les limites de sa cellule et qu'aucun demi-cylindre d'une section ne disparaisse parce que son rayon est nul, car il ne serait plus possible d'observer sa couleur.

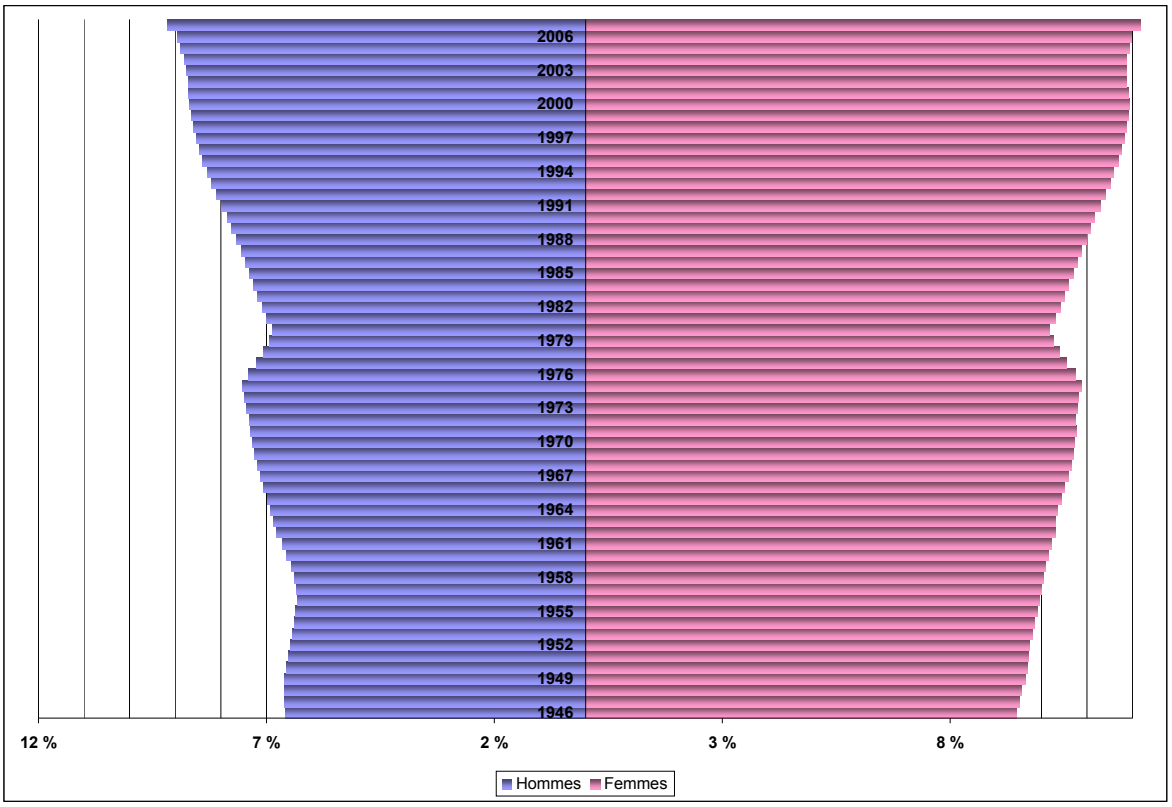

Figure 3.15 – proportion des français de plus de 60 ans, réparti par sexe, entre 1946 et 2007.

Généralement, les pyramides des âges sont utilisées pour observer la répartition des aspects d'une entité selon un axe chronologique, comme par exemple, la répartition d'une population par sexe et par tranche d'âge à un instant donné. Nous avons donc légèrement modifié le concept original. Cependant, ces modifications ne changent pas les avantages de la pyramide des âges, qui permet toujours de naviguer à la fois à travers le temps et les variables visualisées de chaque côté de la pyramide. Prenons un exemple avec la figure 3.15, qui utilise notre version modifiée de la pyramide des âges pour visualiser l'évolution de la proportion des Français de plus de 60 ans, répartie par sexe, entre 1946 et 2007. Nous pouvons faire des comparaisons selon l'axe du temps. Par exemple, on remarque facilement la chute de la population des 60 ans et plus aux alentours de 1980, due aux deux guerres mondiales. Nous pouvons également comparer les

aspects d'une période. Par exemple, en 1980, les femmes avaient une espérance de vie plus grande que les hommes, puisqu'elles étaient plus nombreuses. Enfin, il est aussi possible de faire des comparaisons à la fois sur les aspects représentés et sur l'axe temporel. Par exemple, la forme échancrée vers le haut de la pyramide montre le vieillissement de la population française. De plus, globalement, les femmes ont une espérance de vie plus grande que les hommes.

3.3.1.1. Point d'observation et lecture des informations

Les bâtiments d'EOD tournent autour de leur axe vertical, pour faire face en permanence à la caméra. Plus précisément, les bâtiments tournent sur eux-mêmes pour faire face à la tête de l'utilisateur, car notre système de résolution de l'occlusion permet de localiser cette dernière. Ainsi, l'utilisateur peut lire et comparer les informations visualisées sur les bâtiments en tout temps, quel que soit son point d'observation.

L'utilisation de la pyramide des âges en 3D, cylindres, fonctionne bien si l'utilisateur observe la pyramide dans le prolongement du plan de séparation de ses deux moitiés. C'est à dire, si l'utilisateur observe la pyramide de face. Cependant, si la pyramide est observée avec une légère rotation, la ligne de séparation des deux moitiés se déplace, ne laissant plus apparaître la même échelle de chaque côté (cf. figure 3.16). Par exemple, la figure 3.16 montre une section d'une pyramide, dont les deux moitiés sont identiques, observée de face et avec une rotation. À gauche, la section est observée de face, l'utilisateur peut remarquer que les demi-sections sont proportionnelles et peut estimer leur taille (*i.e.,* leur rayon). À droite, la section est observée avec une légère rotation, l'échelle à gauche et à droite n'est plus la même, il est alors impossible de comparer les demi-sections ou d'estimer leurs tailles.

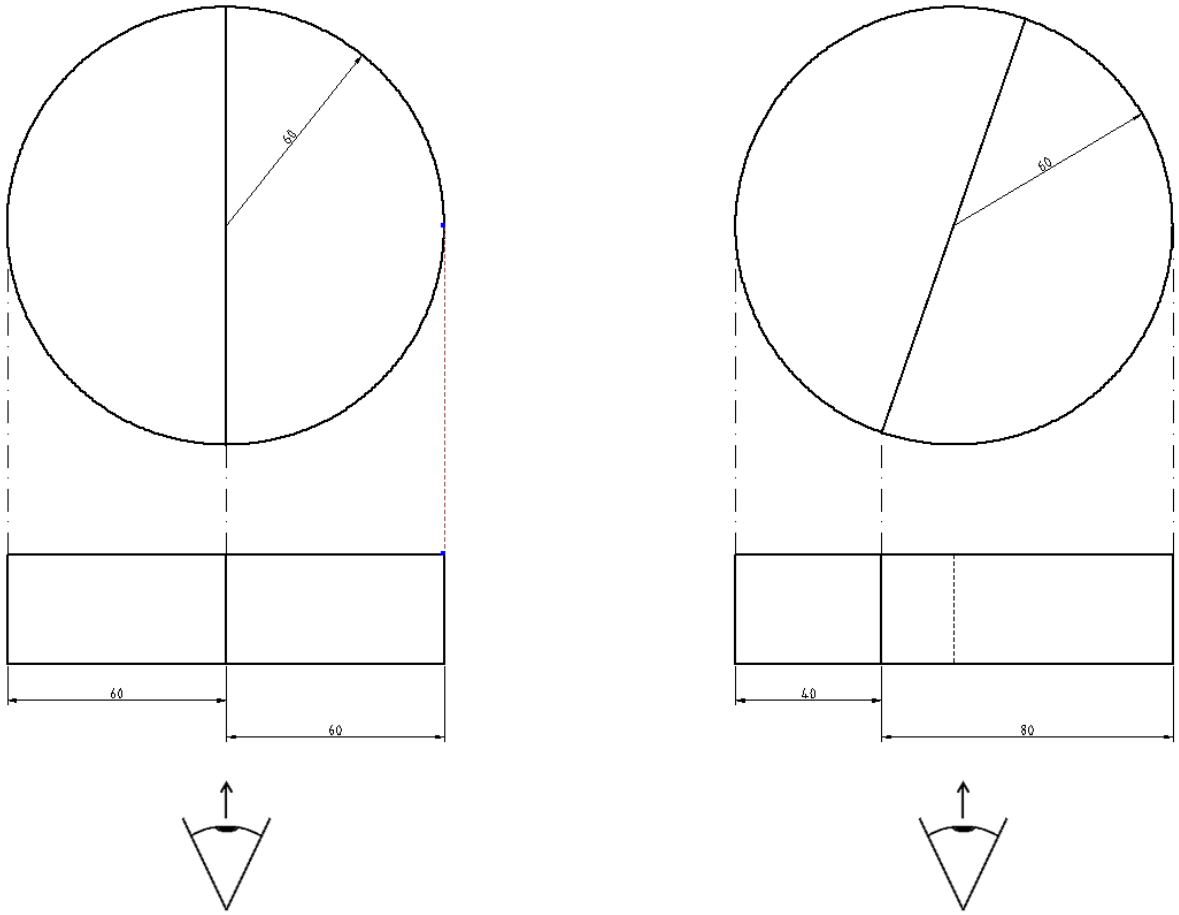

Figure 3.16 – Section d'une pyramide observée de face et avec une rotation.

3.3.1.2. Classes crées ou détruites pendant la période visualisée

Une classe ou une interface peut être créée et/ou détruite pendant la période de l'évolution visualisée et donc absente d'une partie des clichés. Dans ce cas, les sections du bâtiment associées à ces clichés ne peuvent pas être représentées. Nous les remplaçons par un cylindre semi-transparent, de rayon égal au rayon minimal des sections, qui sert de guide (cf. figure 3.17). Ainsi, tous les bâtiments touchent le sol, ce qui règle le problème de correspondance métaphorique et de positionnement au sol. De plus, tous les bâtiments se terminent à la même hauteur, ce qui permet d'estimer à quel cliché ils s'arrêtent.

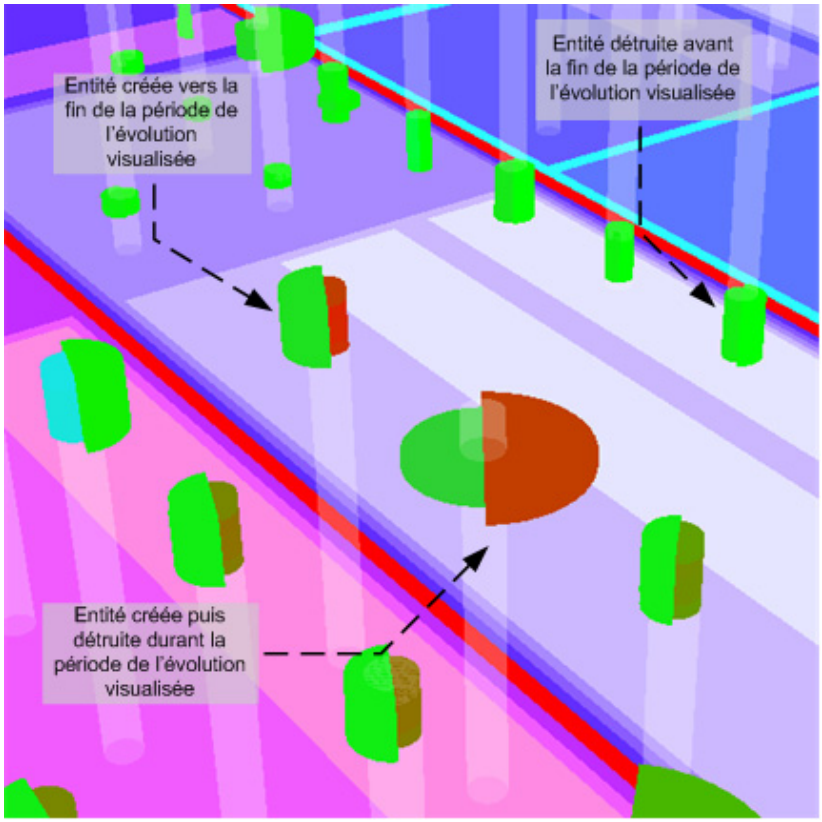

Figure 3.17 – Entités créées et/ou détruite durant la période visualisée.

Afficher un bâtiment qui flotte dans les airs et/ou se termine avant les autres poserait plusieurs problèmes, car dans la réalité, les bâtiments ne flottent pas dans les airs et pour qu'une métaphore soit reconnaissable, il ne faut pas enlever les détails ancrés dans l'imaginaire collectif qui permettent de la comprendre (cf. section 2.3.1.3, p. 13). De plus, pour connaître la position d'un bâtiment qui flotte dans les airs, c'est-à-dire, connaître le paquetage dans lequel se trouve la classe, l'utilisateur doit le projeter mentalement sur le sol, ce qui est une opération délicate et imprécise. Par ailleurs, la fin de la période visualisée n'est pas matérialisée, contrairement au début de cette période qui est matérialisée par le sol. Or, l'absence de point de repère complique l'estimation de la taille d'un bâtiment, soit l'estimation du cliché à partir duquel la classe ne fait plus partie du système.

3.3.1.3. Hauteur des sections et unité de temps

Certaines sections des bâtiments paraissent plus épaisses que d'autres. En réalité, c'est une illusion créée par l'enchaînement d'une série de sections identiques. En effet, nous avons choisi le cliché comme unité de temps et comme les sections des bâtiments sont associées aux clichés, leur hauteur est fixe. Par exemple, sur la figure 3.18, montre deux bâtiments ayant une série de sept sections identiques.

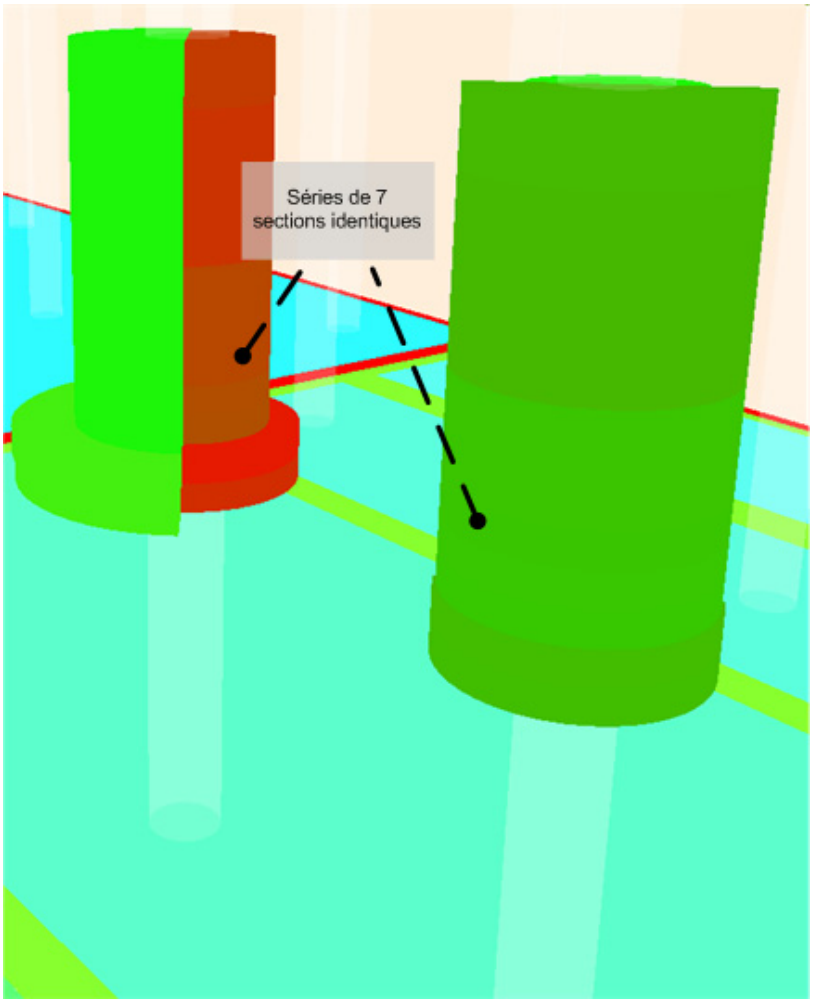

Figure 3.18 – Bâtiments présentant une série des sept sections identiques.

Nous acceptons cette illusion, car elle permet d'estimer le nombre de clichés séparant deux modifications du code qui affectent les aspects visualisés. Autrement dit, elle permet d'estimer la fréquence des modifications d'une classe ou d'une interface par rapport à l'ensemble des modifications du code. Or, ce sont bien les modifications des aspects visualisés qui nous intéressent, car pour une analyse donnée, seuls les changements des aspects du code prenant part à cette analyse (*i.e.,* les aspects visualisés) présentent un intérêt.

Par ailleurs, nous avons choisi le cliché comme unité de temps, car le développement d'un projet est irrégulier, les périodes de forte et faible activité s'enchaînent. Nous pourrions faire des sections proportionnelles à la durée de la période couverte par les clichés, mais dans ce cas, les sections représentants les périodes où la fréquence des modifications est faible seraient plus épaisses que celles où elle est forte. Comme la taille n'est pas une VV associative (cf. section 2.3.1.2, p. 11), les sections les plus épaisses seront perçues plus facilement, leur donnant une plus grande importance. Or, un cliché représente la mise à jour d'un groupe de fichiers sur le SGV, son importance ne varie donc pas selon la période qu'il couvre, mais selon plusieurs critères, dont notamment, l'ampleur des modifications qu'il apporte au code et la tâche d'analyse à réaliser.

3.3.2. Organisation géographique

Nous utilisons l'organisation géographique des bâtiments pour visualiser la structure des entités du code auxquelles ils sont associés. Cette structure est un graphe complexe comprenant des liens de différentes natures, nous devons donc la simplifier. D'une part, parce qu'il faut éviter de saturer l'espace d'affichage (cf. section 2.3.1.7, p. 16) avec des informations structurelles. D'autre part, parce que tous les liens de ce graphe ne sont pas nécessaires pour une analyse donnée et selon le principe de simplicité (cf. section 2.3.1.4, p. 13), nous ne devons pas afficher d'informations inutiles. C'est pourquoi, nous choisissons de visualiser cette

structure avec un *treemap*, c'est à dire, un arbre. Tout d'abord, parce que cette méthode a déjà été mise en place et validée par Langelier [55]; ensuite, parce que nous souhaitons visualiser un grand nombre d'entités, qui seront réparties sur une grille régulière pour minimiser les problèmes d'occlusion (cf. section 2.3.1.6, p. 15). Pour cela, nous devons éviter de représenter un graphe autre qu'un arbre, car leur mise en forme répond à des contraintes de lisibilité plutôt que de maximisation de l'utilisation de l'espace d'affichage. Il faut : (1) limiter le nombre d'arcs qui se croisent, (2) rapprocher géographiquement les nœuds voisins dans le graphe et (3) écarter suffisamment les nœuds pour que l'affichage des arcs ne soit pas trop dense, c'est-à-dire, pour éviter la saturation visuelle.

À notre connaissance, aucun algorithme de *treemap* permettant de placer des nœuds de tailles fixes sur une grille n'a été détaillé dans la littérature. C'est pourquoi, cette partie commence par détailler l'algorithme de *treemap* que nous avons conçu avant d'aborder sa mise en forme graphique.

3.3.2.1. L'algorithme

L'objectif d'un *treemap* est de visualiser un arbre sur un plan, en maximisant l'utilisation de l'espace d'affichage. L'algorithme original [48], proposé par Shneiderman, a été conçu pour visualiser l'utilisation de l'espace disque par un système de fichiers. Le principe de l'algorithme est le suivant :

1. Une première cellule rectangulaire, qui occupe tout l'espace d'affichage, est créée pour représenter le noeud racine.
2. Cette cellule est découpée en sections horizontales ou verticales, qui représentent les cellules des fichiers et des dossiers contenus à la racine. Chaque section a une épaisseur proportionnelle à la taille du fichier ou du dossier qu'elle représente.

3. Pour chaque nouvelle cellule qui représente un dossier, le point 2 est répété récursivement, en alternant le sens de découpage.

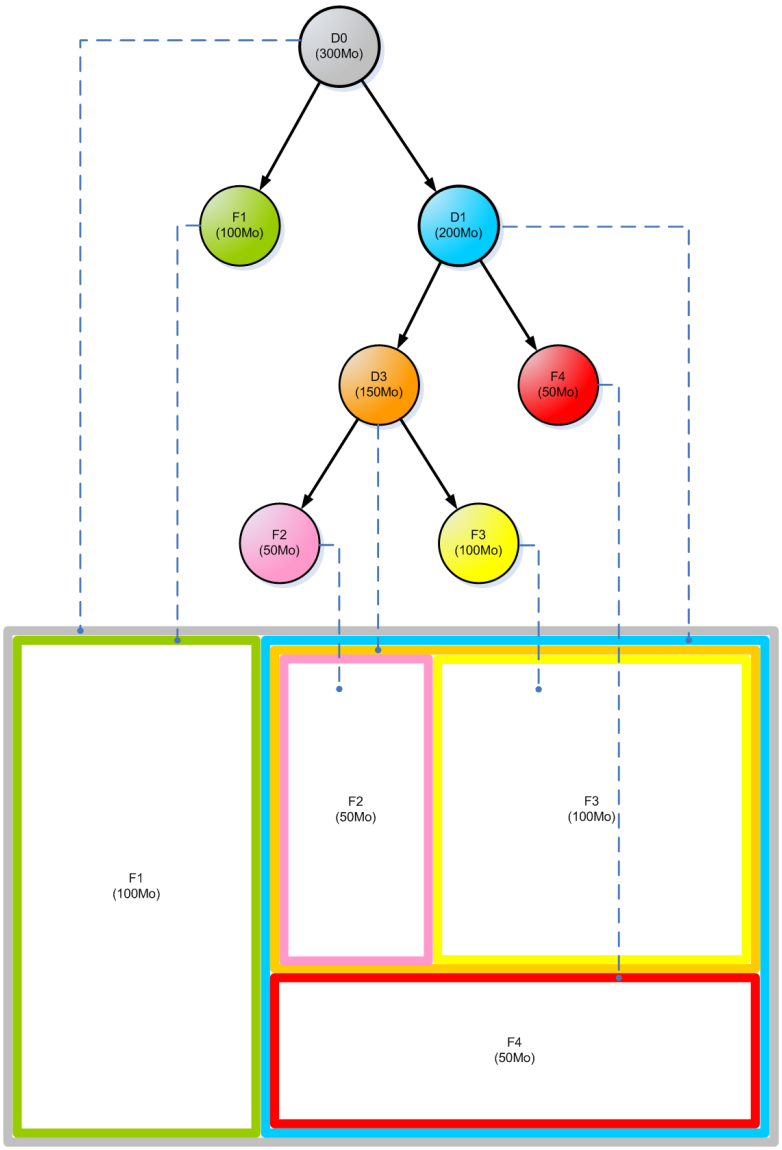

Figure 3.19 – Algorithme original du *treemap* [48].

La figure 3.19 présente un exemple simple de cet algorithme. Le noeud D0 est le dossier racine, il est représenté par le rectangle gris. Ce dossier contient deux fils : le fichier F1 de 100Mo et le dossier D1 de 200Mo. Le rectangle gris est donc découpé en deux sections verticales proportionnelles à la taille de F1 et de D1. La section verte qui représente le fichier F1 est donc deux fois moins épaisse que la

section bleue qui représente le dossier D1. Puis la section bleue est découpée horizontalement, pour créer les cellules du dossier D3 (en orange) et du fichier F4 (en rouge). Enfin, la section du dossier D3 est découpée verticalement pour représenter les cellules des fichiers F2 et F3.

L'algorithme que nous utilisons est différent de l'algorithme original. Tout d'abord, parce que les feuilles de notre arbre, qui sont les bâtiments, ont un poids unitaire. Donc, le poids d'un noeud est égal au nombre de feuilles qui sont ses descendantes. Ensuite, parce que nous sommes en 3D, la taille du plan du *treemap* n'a donc pas de limites puisqu'il est toujours possible de zoomer. Par contre, le ratio hauteur/longueur du plan doit être le plus proche possible de celui de l'écran, pour permettre d'utiliser tout l'espace d'affichage en zoomant, c'est-à-dire, en le cadrant. Enfin, parce que les bâtiments d'EOD sont alignés sur une grille régulière et occupent tous le même espace carré au sol, la taille et la position des cellules doivent donc être discrétisées selon cette grille. De plus, lors de cette discrétisation, il faut arrondir supérieurement la taille des cellules, pour garantir que tous ses bâtiments contiennent à l'intérieur. Ainsi, le nombre de bâtiments d'une cellule n'est pas égal à sa superficie, mais en est une borne inférieure. C'est pourquoi il est impossible de déterminer d'avance les dimensions du plan de notre *treemap*. Par contre, nous savons que la superficie de ce plan est au moins égale au nombre de bâtiments qu'il contient.

Lors du placement des cellules et des bâtiments, notre algorithme cherche à atteindre deux objectifs. Le premier est de minimiser le plus possible la perte d'espace dans ses cellules : d'une part, pour que la superficie d'une cellule soit le plus proche possible du nombre de bâtiments qu'elle contient; d'autre part, pour minimiser la taille du plan d'affichage de notre *treemap*. En effet, plus le plan est grand, plus il faut reculer par rapport à celui-ci pour le voir en entier et donc, plus les bâtiments sont petits sur l'écran. Or, la taille n'est pas une VV associative, donc

plus un objet graphique est petit, plus les informations qu'il visualise sont difficiles à distinguer.

Le deuxième objectif de notre algorithme est de créer un plan d'affichage dont le ratio est le plus proche possible de celui de l'écran. Toutefois, ce dernier objectif entre en conflit avec la minimisation de la perte d'espace dans les cellules, car il est possible que le plan qui minimise au mieux la perte d'espace ait un ratio différent de celui de l'écran. C'est pourquoi nous décidons de faire un compromis entre ces deux objectifs, en autorisant le ratio de notre plan à varier sur une plage de valeurs. Plus précisément, pour un écran de largeur l et de hauteur h, tel que $l > h$, le ratio du plan peut varier entre $\left[l/h - 1/3; l/h\right]$.

Dans cette section, nous ne traitons pas le cas où $l < h$, car le *treemap* d'un écran de taille $\{h,l\}$ est équivalent au *treemap* d'un écran de taille $\{l,h\}$, à une rotation de 90 degrés près.

Avant d'entrer dans les détails, voici le principe général de notre algorithme. Tout d'abord, nous définissons les dimensions minimales du plan d'affichage. Puis, nous essayons de placer les cellules et leurs bâtiments dans ce plan, en commençant par le découper verticalement. Tant que l'espace est insuffisant pour faire le placement, nous agrandissons le plan et nous réessayons. Enfin, si l'utilisateur choisit la version longue de l'algorithme, nous essayons de trouver une meilleure solution, avec le même processus, mais en commençant par découper le plan d'affichage horizontalement.

Dimensions du plan d'affichage

Les dimensions du plan de notre *treemap* sont également les dimensions de la cellule racine. Nous les calculons la première fois de façon à ce que la superficie

de ce plan soit minimale pour contenir tous les n bâtiments et que son ratio r soit proche de celui du ratio minimal autorisé.

Soit $Ceil(x)$ la fonction qui arrondit la valeur de x à l'entier supérieur ou égal. Soit un écran de largeur l et de hauteur h. Et soit i et j, respectivement la largeur et la hauteur du plan de notre *treemap*. Nous avons :

$$r = \frac{l}{h} - \frac{1}{3}$$

$$\begin{cases} \frac{i'}{j'} = \frac{l}{h} - \frac{1}{3} \\ i'.j' = n \end{cases} \Leftrightarrow \begin{cases} i' = \sqrt{n \times \frac{l}{h} - \frac{1}{3}} \\ j' = \frac{n}{i'} \end{cases}$$

Formule 3.15 – Calcul de la taille minimale du plan en valeurs continues.

d'où :

$$i = Ceil(i')$$
$$j = Ceil\left(\frac{n}{i}\right)$$

Formule 3.16 – Calcul de la taille minimale du plan en valeurs discrètes.

Si les dimensions de ce plan sont insuffisantes pour contenir toutes les cellules et les bâtiments de notre *treemap*, nous l'agrandissons progressivement, selon l'ordre croissant de la superficie totale perdue. Autrement dit, pour n bâtiments et un plan de taille $\{i, j\}$, nous cherchons à minimiser $i \times j - n$.

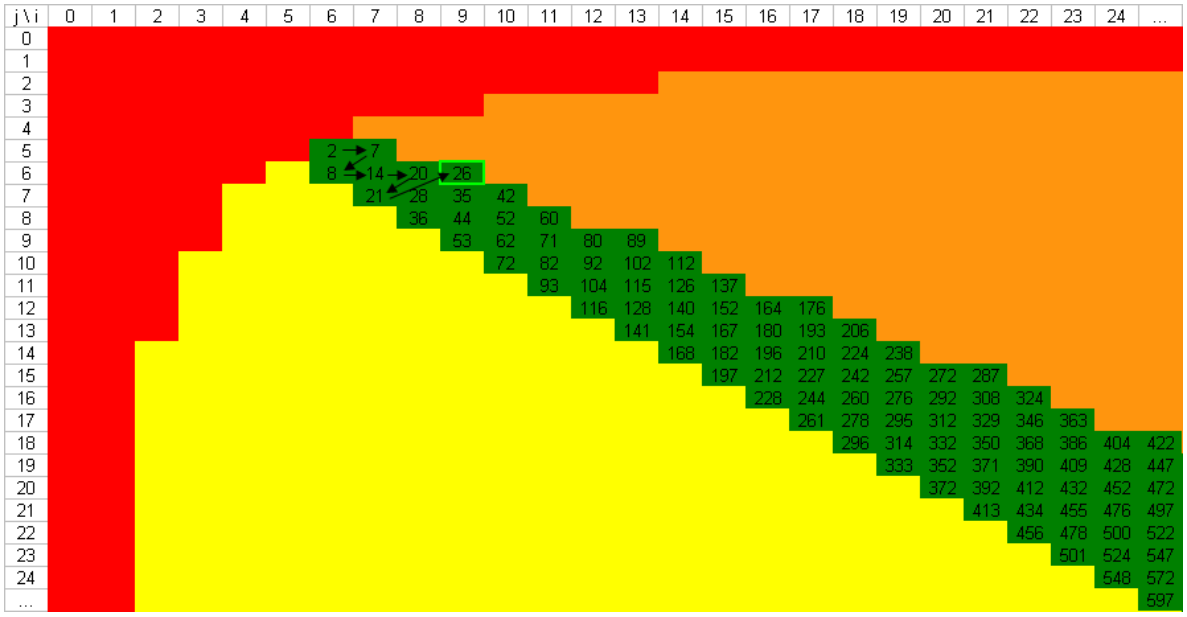

Figure 3.20 – Parcours de l'espace des dimensions du plan pour un treemap de 28 bâtiments.

La figure 3.20 montre un exemple du parcours de l'espace des dimensions du plan, pour un *treemap* de 28 bâtiments, dont la taille $\{i, j\}$ optimale est $\{9,6\}$ et un écran de ratio 4/3. L'espace rouge représente les dimensions dont la superficie est insuffisante pour contenir tous les bâtiments. L'espace orange représente les dimensions dont le ratio dépasse 4/3 (nous acceptons une itération qui dégrade ce ratio). L'espace jaune représente les dimensions dont le ratio est inférieur à 1, c'est-à-dire, dont le ratio est inférieur au minimum autorisé. Enfin, l'espace en vert représente les dimensions autorisées pour notre plan d'affichage et indique pour chacune de ces dimensions la superficie perdue (*i.e.*, $i \times j - 28$). Nous commençons par la cellule de taille $\{i, j\}$ la plus petite, c'est-à-dire la cellule de taille $\{6,5\}$. Puis nous traversons l'espace des dimensions possible selon l'ordre croissant de l'espace perdu (indiqué par les flèches), jusqu'à trouver les premières dimensions qui permettent de placer toutes les cellules et les bâtiments de notre *treemap*. Dans cet exemple, il s'agit de la cellule de taille $\{9,6\}$, qui fait perdre une superficie de 26.

Placer les cellules et les bâtiments dans le plan d'affichage

Nous plaçons les cellules et les bâtiments de notre *treemap* à l'aide de la fonction placer (ci-dessous). Cette fonction place récursivement les descendants d'un nœud dans la cellule qui lui est attribuée.

```
  placer(Nœud : n,
         Hauteur de la cellule de n : h,
         Longueur de la cellule de n : l,
         Sens de découpage : s) :
1:    Classe les fils de n par ordre croissant du nombre de bâtiments qui
      sont leurs descendants.
2:    Pour chaque nœud n_fils, fils de n {
3:          Définir les dimensions de la cellule de n_fils, en découpant
            une section de la cellule de n, dans le sens s.
4:          Si il ne reste pas suffisamment de place dans la cellule de n
            pour contenir la section de n_fils {
5:                Retourner « espace insuffisant »
            } Sinon {
7:                Appeler récursivement la fonction placer sur n_fils,
                  avec les dimension sa section et en alternant le sens
                  de découpage.
8:                Si la fonction « placer » retourne « espace
                  insuffisant »
9:                      Agrandir la cellule de n_fils et retourner en 4
            }
      }
10:   Pour chaque feuille f_fille, filles de n {
11:         Placer f_fille dans la première case disponible de la cellule
            de n
12:         Si toutes les cases sont pleines, retourner « espace
            insuffisant »
      }
13:   Retourner « espace suffisant »
```

La ligne 1 de la fonction permet que les cellules filles de chaque cellule soient placées par ordre de taille décroissante. Ainsi, il est facile de distinguer si une cellule contient plus de bâtiments que ses voisines, même si elle contient des espaces vides.

En ligne 2, nous définissons la taille de la cellule de n_fils, de façon à ce que sa superficie soit minimale pour contenir les bâtiments qui sont descendants de cette cellule.

En ligne 9, nous agrandissons les cellules dans le sens opposé au sens de découpage. Autrement dit, si nous découpons des sections verticales, nous agrandissons leur largeur et si nous découpons des sections horizontales, nous agrandissons leur hauteur.

Cette fonction de placement est fortement inspirée de celle de Shneiderman [48]. Les deux différences notables sont le trie des nœuds par ordre décroissant (ligne 1) et l'agrandissement d'une cellule en cas d'espace insuffisant (lignes 4, 5, 8 et 9).

Premier sens de découpage

La première itération de notre algorithme cherche les dimensions minimales du plan d'affichage, en commençant par le découper verticalement, car ce sens de découpage est le plus susceptible de minimiser la perte d'espace dans les cellules. En effet, avant de placer les cellules de notre *treemap*, nous n'avons aucune information sur leurs dimensions. Nous considérons donc toutes les dimensions possibles comme étant équiprobables. Par contre, nous savons que le plan d'affichage de largeur l et de hauteur h, a un ratio supérieur à 1, c'est-à-dire que $l \geq h$. De plus, nous savons que la cellule racine a les mêmes dimensions que le plan. Lors du découpage d'une cellule, dans le pire des cas, une section sera élargie pour un seul bâtiment. Par conséquent, si nous découpons une section de la cellule racine verticalement, nous pouvons perdre une superficie maximale de $h-1$ et si nous la découpons horizontalement, nous pouvons perdre une superficie maximale de $l-1$. Comme, $l-1 \geq h-1$, il est donc préférable de découper cette cellule dans le sens vertical.

Ce raisonnement est une heuristique, car il tient seulement compte du premier niveau de nœuds. Cependant, cette heuristique semble bien fonctionner, car le découpage avec départ vertical s'est avéré le meilleur pour la majorité des *treemaps* que nous avons testés. Néanmoins, nous proposons aussi une version longue de notre algorithme, qui va essayer de trouver une meilleure solution en commençant le découpage horizontalement.

3.3.2.2. Mise en forme

La mise en forme de notre *treemap* doit faciliter la navigation verticale et horizontale de l'arbre. C'est pour cette raison que nous colorons les cellules par rapport à leur position dans l'arbre et que nous représentons les bordures des cellules de façon à ce que leur niveau d'imbrication soit lisible.

Couleurs des cellules

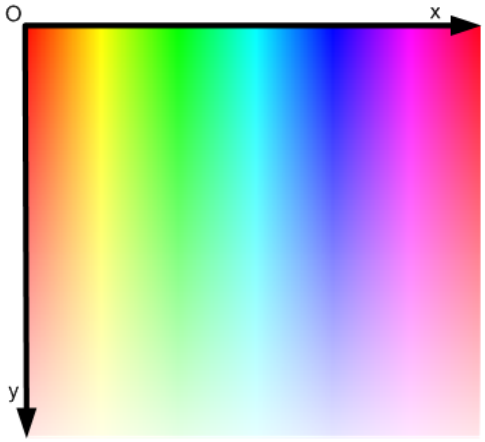

Figure 3.21 – Variations des couleurs des cellules.

Nous choisissons de colorier les cellules représentant des nœuds voisins avec des couleurs proches, tout en éclaircissant la couleur selon la profondeur des nœuds. Pour cela, nous utilisons l'intervalle des couleurs représenté en figure 3.21. Les couleurs de cet intervalle varient le long de l'axe Ox selon les couleurs de l'arc-en-ciel et le long de l'axe Oy selon leur niveau de saturation (*i.e.,* leur clarté). Le

niveau de saturation des couleurs peut varier de 100% à 10%, car en dessous de cette limite, les couleurs deviennent trop claires pour en distinguer la teinte.

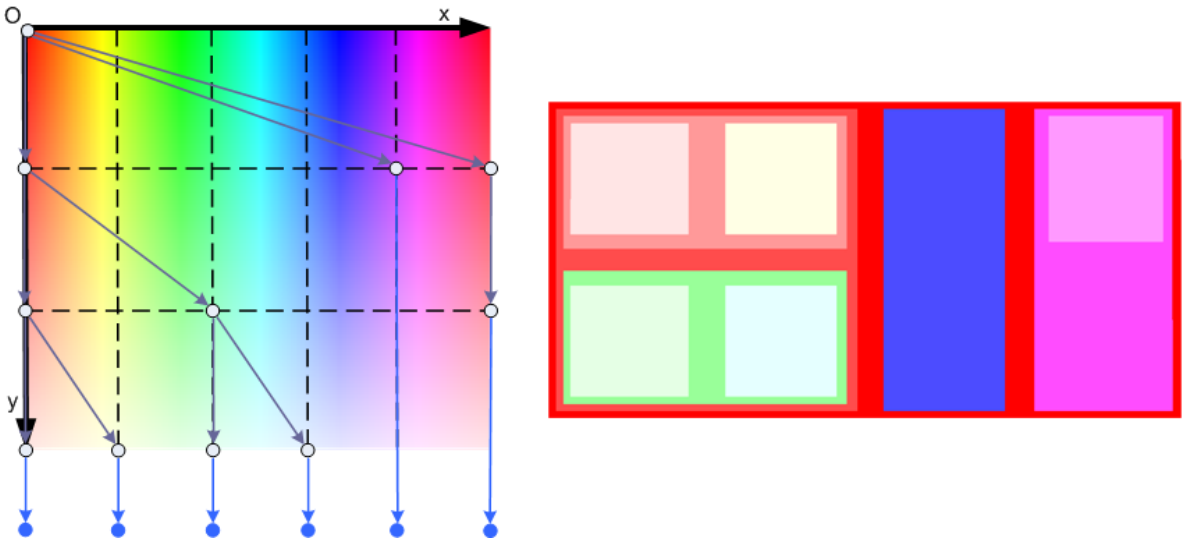

Figure 3.22 – Exemple du calcul des couleurs des cellules d'un *treemap*.

Pour déterminer la couleur des cellules d'un *treemap* (cf. figure 3.22), nous discrétisons l'axe Oy selon le nombre de niveaux de l'arbre et l'axe Ox selon le nombre de feuilles (*i.e.,* de bâtiments) de l'arbre. Puis, nous attribuons à chaque niveau de l'arbre un niveau de saturation des couleurs sur l'axe Oy. Enfin, les nœuds de chaque niveau de l'arbre sont distribués sur l'axe Ox, proportionnellement au nombre de feuilles total de ses voisins de gauche.

La figure 3.22 montre un exemple du calcul des couleurs pour un arbre à quatre niveaux, dont le dernier niveau de nœud de chaque branche contient un bâtiment (représenter en bleu et non affiché sur le *treemap*). Cet arbre contient peu de nœuds et de feuille, c'est pourquoi les couleurs de nœuds voisins paraissent éloignées. Cependant, ce n'est pas le cas avec un arbre plus grand (cf. figure 3.23).

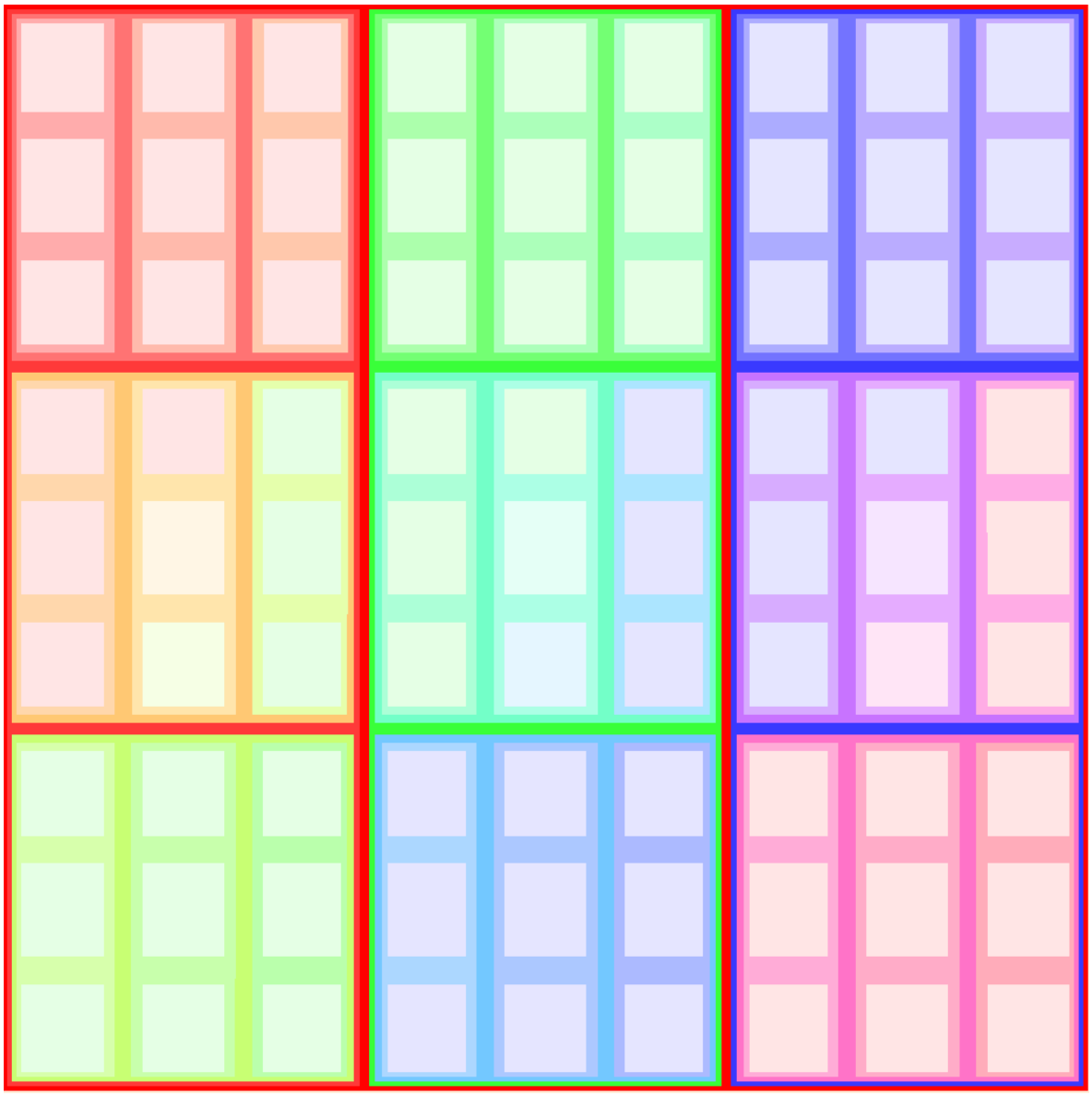

Figure 3.23 – Autre exemple du calcul des couleurs des cellules d'un *treemap*.

Bordures des cellules

Nous choisissons d'afficher les bordures de toutes les cellules, pour que l'utilisateur n'ait pas besoin d'élargir sa vue pour observer le niveau d'imbrication. Cependant, les bordures ne doivent pas induire de décalage sur la grille du *treemap*. C'est pourquoi, nous décidons de séparer les cases de la grille par une zone morte

d'épaisseur fixe, qui permet de représenter les bordures. Cette zone morte est découpée selon le nombre de cellules ayant une bordure commune (cf. figure 2.20).

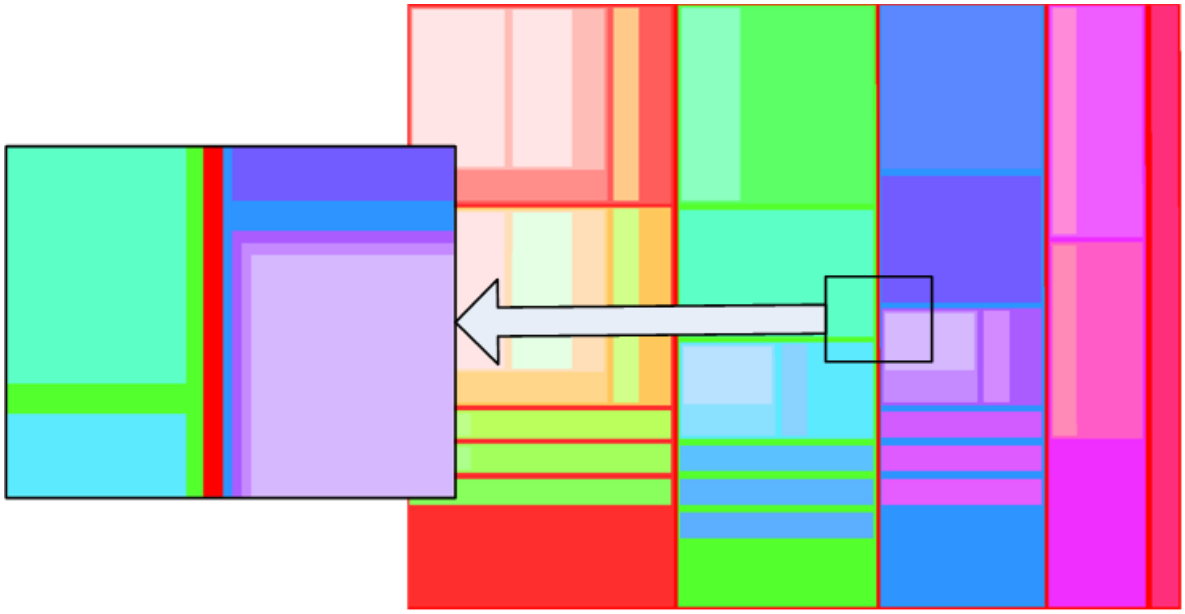

Figure 3.24 – Exemple des bordures des cellules d'un *treemap*.

3.4. Visualisation de la scène

Nous pouvons maintenant visualiser la scène que nous venons de construire. Pour cela, nous devons l'éclairer et y ajouter une caméra qui fera le rendu de l'image.

3.4.1. Éclairage

La lumière est délicate à mettre en place dans les SV en 3D : d'une part, parce que l'éclairage des scènes 3D est une imitation simpliste de la réalité; d'autre part, parce que la lumière modifie la couleur d'un objet selon sa distance de la source lumineuse et son angle d'éclairage. Donc, si deux objets ne sont pas à la même distance d'une source de lumière ou éclairés avec le même angle, leur couleur sera modifiée différemment, ce qui peut biaiser leur comparaison.

Nous choisissons d'utiliser un éclairage ambiant, c'est-à-dire d'utiliser une lumière qui vient de toutes les directions à la fois, avec la même intensité. L'avantage de ce type d'éclairage est de toujours modifier la couleur des objets de la même façon, ce qui permet de comparer la couleur des objets quelque soit leur position. Cependant, son inconvénient est de ne pas mettre en valeur les reliefs.

3.4.2. Caméra

La caméra d'EOD se décompose en deux parties interchangeables : le pied de caméra qui définit ses mouvements et l'objectif qui définit la façon dont la scène se projette sur l'écran.

3.4.2.1. Pied de la caméra

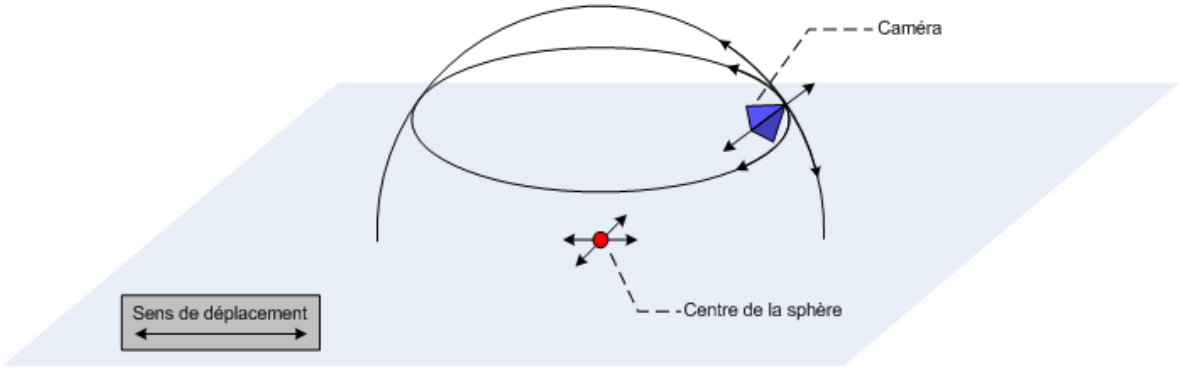

Figure 3.25 – Mouvements de la caméra d'EOD.

Notre caméra se déplace autour d'une demi-sphère et vise toujours son centre. La demi-sphère est placée au-dessus de la scène et son centre est aligné sur le plan d'affichage (*i.e.,* du *treemap*). L'idée est de simplifier les déplacements de la caméra en l'empêchant de viser autre chose que le plan sur lequel les entités sont affichées.

La caméra peut être déplacée selon cinq degrés de liberté (cf. figure 3.25). Deux degrés de liberté permettent de déplacer le centre de la demi-sphère sur le plan d'affichage, pour viser une autre partie de la scène. Les trois autres degrés de

liberté permettent d'observer le point central sous un autre angle de vue. Plus précisément, ils permettent de tourner horizontalement ou verticalement sur la demi-sphère et de réduire ou agrandir son rayon afin de zoomer.

3.4.2.2. Objectif de la caméra

L'objectif de notre caméra est celui d'une caméra perspective de 45 degrés (cf. figure 3.26). Cet angle de vue correspond à l'angle de vue moyen de l'œil humain. La profondeur de champ minimale est alignée avec le plan image, le centre optique est face au centre du plan image et l'axe optique est perpendiculaire au plan image. Cependant, les paramètres de cet objectif peuvent être modifiés par notre système de réduction de l'occlusion, que nous détaillons dans la section suivante.

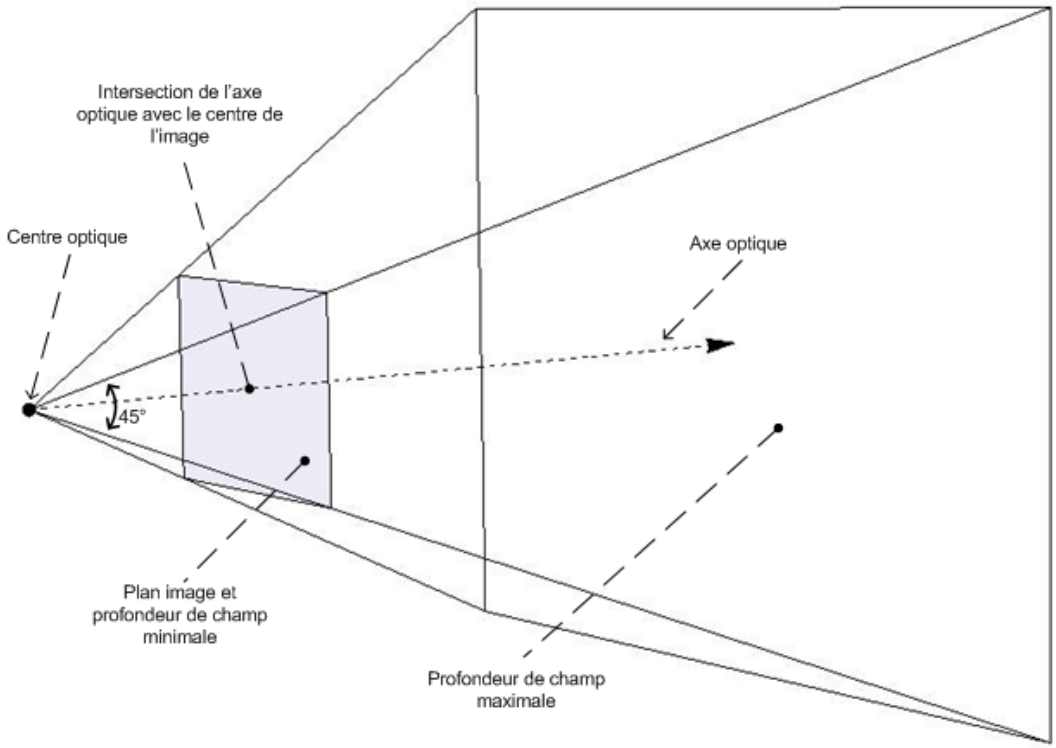

Figure 3.26 – Objectif de la caméra d'EOD.

3.4.3. Réduction des problèmes d'occlusion

Notre système de réduction de l'occlusion s'appuie sur les travaux de Chung Lee [62]. Il transforme l'écran de l'utilisateur en une fenêtre ouverte sur la scène qu'il affiche.

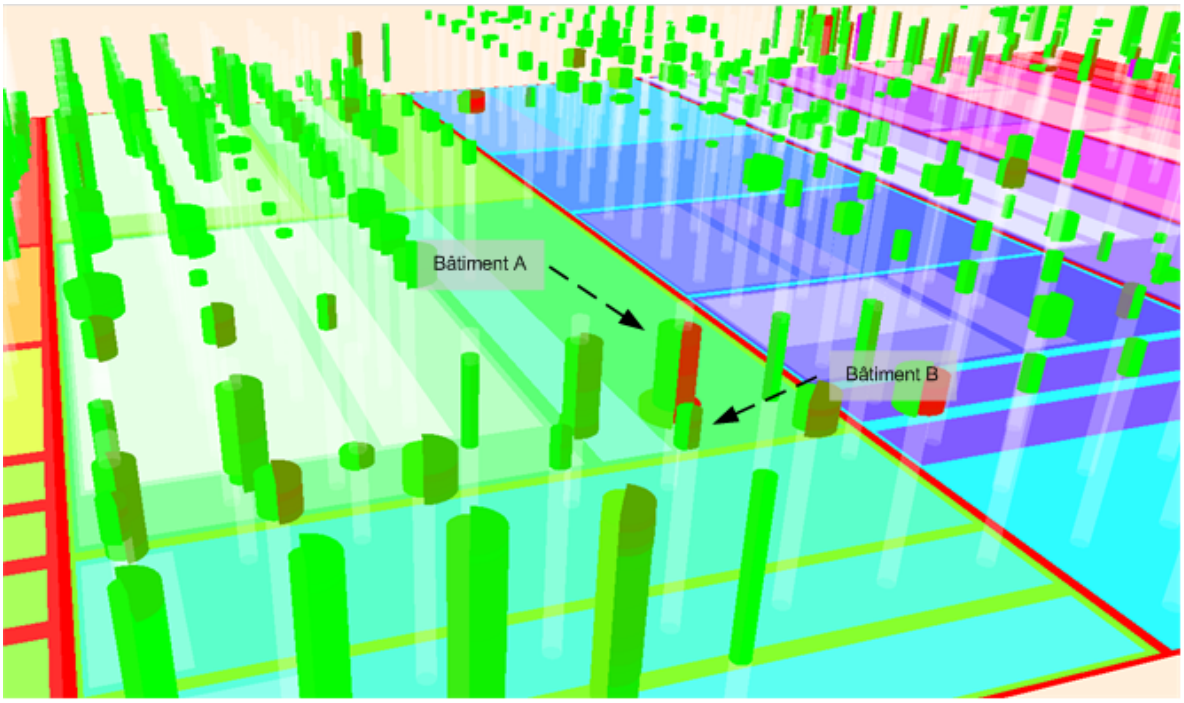

Figure 3.27 – Scène avec problèmes d'occlusion. L'usager est face à l'écran.

Nous dotons EOD de ce système, car notre scène est soumise à des problèmes d'occlusion. En effet, la proximité des bâtiments implique qu'ils peuvent en partie s'occulter entre eux (cf. figure 3.27). Comme nous affichons de l'information sur toute la hauteur des bâtiments, l'utilisateur peut être obligé de changer de point de vue pour observer les parties cachées. Or, les déplacements avec le clavier et la souris sont peu précis. De plus, ils demandent à l'utilisateur d'arrêter puis de reprendre son processus analytique pour contrôler ses mouvements. Avec notre système, l'utilisateur peut résoudre les problèmes d'occlusion (*i.e.,* changer de point de vue), en déplaçant simplement sa tête par

rapport à l'écran, ce qui est un réflexe et donc, qui ne l'oblige pas à arrêter son processus analytique.

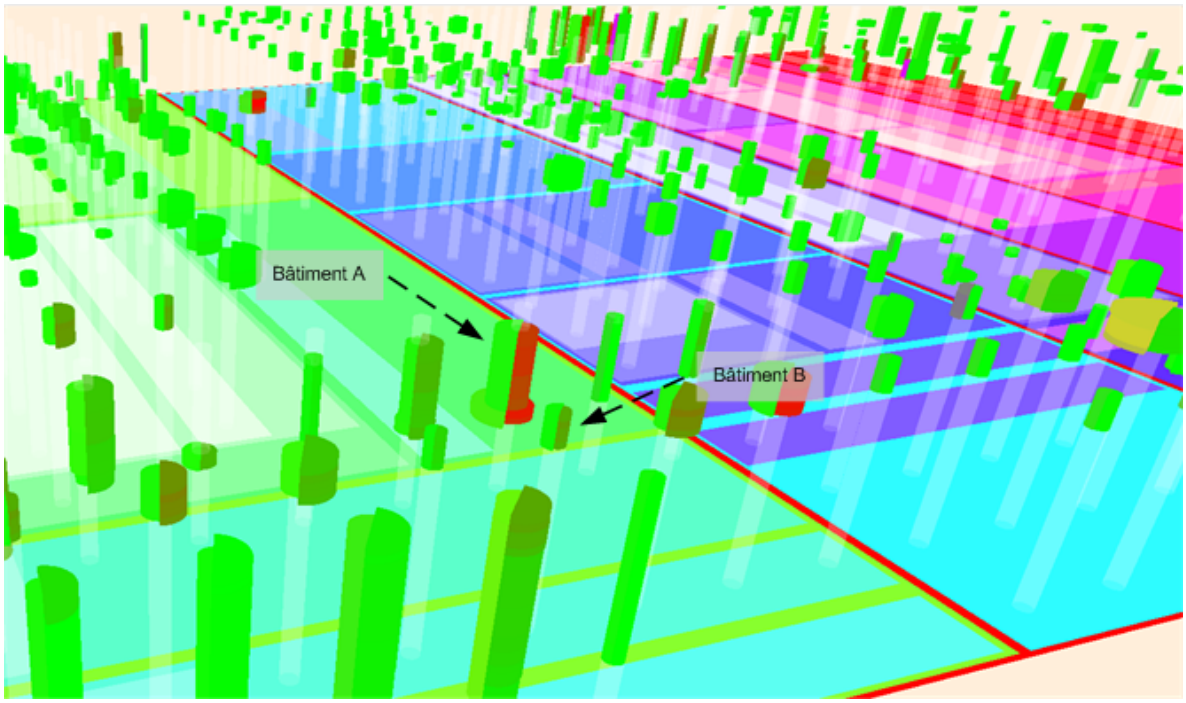

Figure 3.28 – Scène sans problème d'occlusion. L'usager a déplacé sa tête sur la gauche.

Les figures 3.27 et 3.28 montrent un exemple de l'utilisation de notre système. Sur la figure 3.27, le bâtiment B occulte le bâtiment A. L'utilisateur déplace alors sa tête sur la droite (cf. figure 3.28) pour changer de point de vue. Les deux bâtiments sont maintenant visibles, ce qui permet d'observer toutes les valeurs de variables qu'ils affichent.

Notre système reconstruit la vue que l'utilisateur aurait s'il regardait à travers une fenêtre en localisant sa tête par rapport à l'écran. Cette localisation est faite à partir de deux caméras infrarouges (IR) et d'un point IR placé sur la tête de l'utilisateur. Notre système est peu intrusif, car même si l'impression de regarder par une fenêtre ne peut être perçue que par l'utilisateur, il peut communiquer avec les personnes qui sont autour de lui et ces personnes peuvent voir la scène affichée

sur l'écran. De plus, notre système impose simplement à l'utilisateur de placer un point IR sur sa tête.

Dans cette partie, nous abordons la reconstruction de la vue de l'utilisateur, avant de détailler la localisation de sa tête.

3.4.3.1. Reconstruction de la vue de l'utilisateur

Notre système reproduit la vue de l'utilisateur en modifiant les paramètres de l'objectif de la caméra (cf. figure 3.29). Pour cela, nous associons, à un facteur d'échelle près, les dimensions du plan image avec celles de l'écran de l'ordinateur et la position du centre optique de la caméra avec la position de la tête de l'utilisateur par rapport à l'écran.

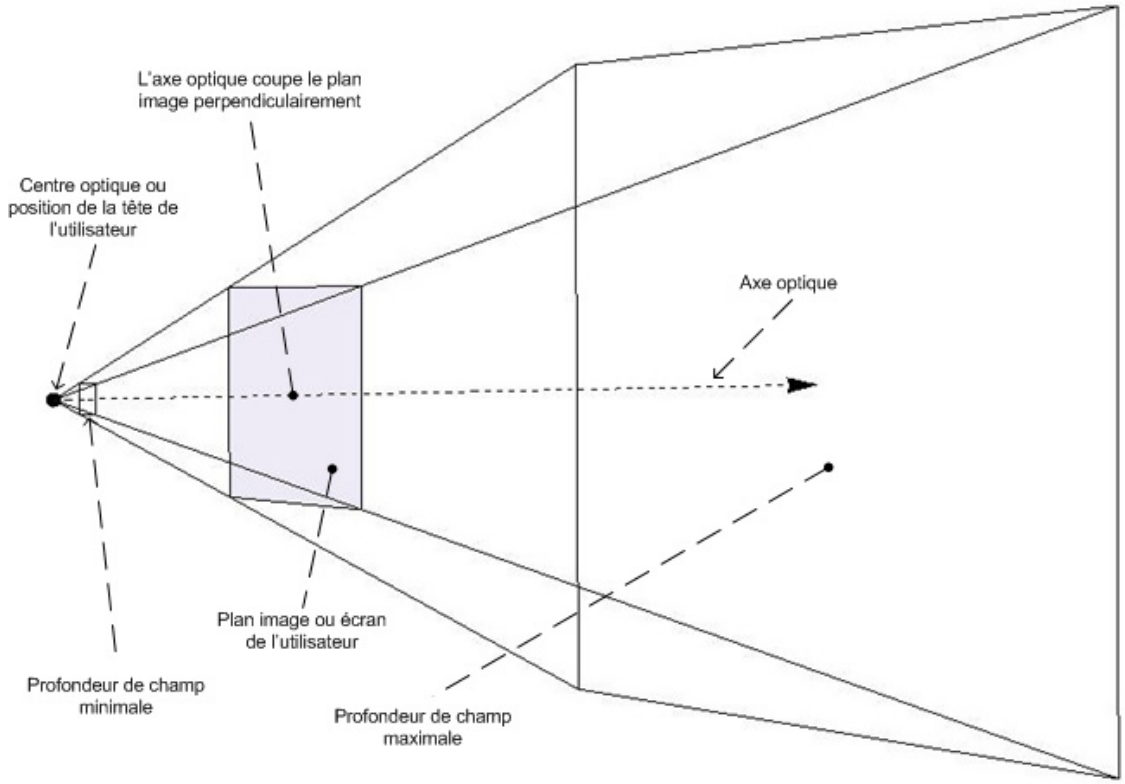

Figure 3.29 – Modifications de l'objectif de la caméra.

L'axe optique de la caméra est toujours perpendiculaire au plan image, ce qui est une heuristique, car en réalité la direction de cet axe devrait être celle du regard. Cependant, nous connaissons seulement la position de la tête de l'utilisateur, nous devions donc trouver une simplification. Les deux choix les plus logiques étaient de considérer que lorsque l'utilisateur déplace sa tête (1) son regard reste toujours face à l'écran ou (2) son regard vise toujours le centre de l'écran. Ces deux heuristiques sont relativement équivalentes, dans la mesure où elles induisent toutes les deux une erreur et que la taille de l'écran et l'amplitude des mouvements de l'utilisateur ne sont pas suffisamment grandes pour que la différence soit significative. Nous avons donc choisi la solution (1) qui est la plus facile à mettre en place.

Enfin, nous choisissons de reculer la profondeur de champ minimale en arrière du plan image. Ainsi, nous pouvons afficher les objets qui se trouvent entre l'écran et l'utilisateur, ce qui donne l'impression que la scène peut sortir de l'écran. Dans la réalité, il est possible de voir un objet qui traverse une fenêtre. Ce choix est donc réaliste, ce qui est un avantage, car plus les mouvements de la tête de l'utilisateur seront reproduits de façon réaliste, plus notre système suggèrera une interaction naturelle.

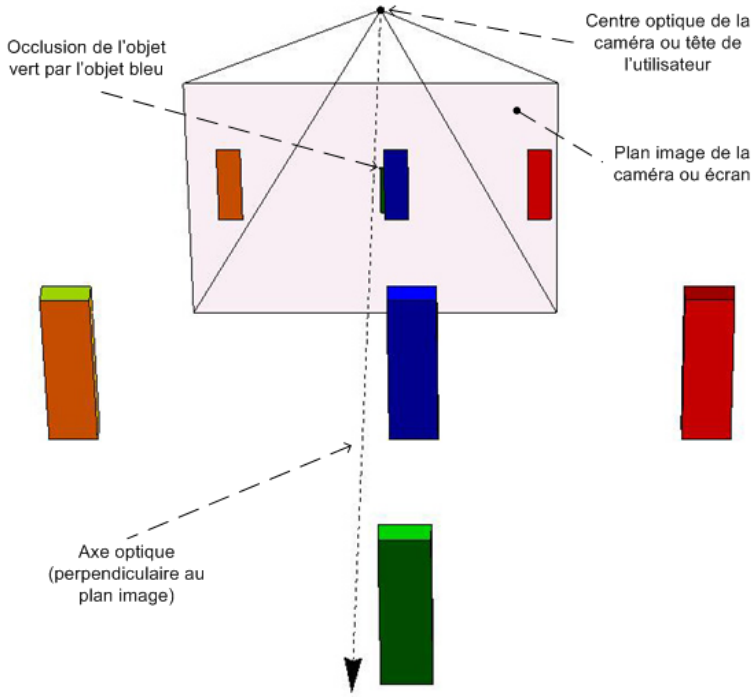

Figure 3.30 – Image observée par la caméra lorsque l'utilisateur est face à l'écran.

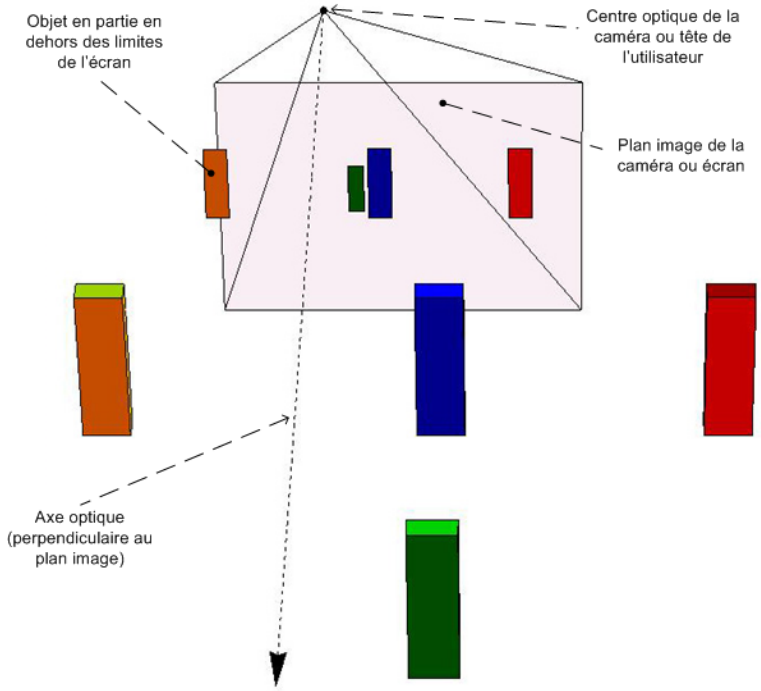

Figure 3.31 – Image observée par la caméra lorsque l'utilisateur est sur la droite de l'écran.

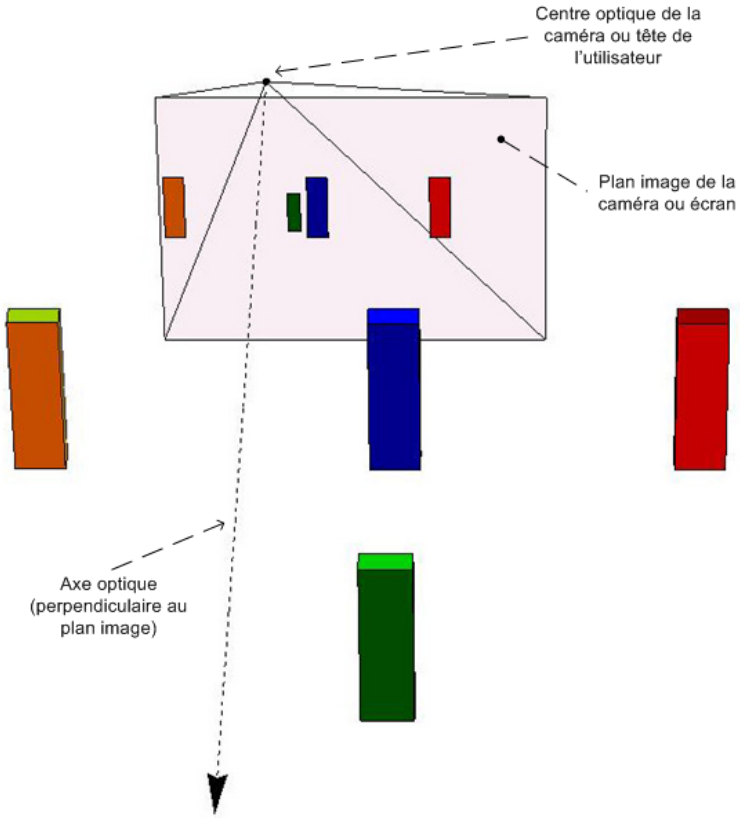

Figure 3.32 – Image observée par la caméra lorsque l'utilisateur s'est avancé vers l'écran.

Prenons un exemple pour illustrer l'impact des modifications de l'objectif que nous venons de décrire. Les figures 3.30 à 3.32 montrent comment l'image observée par notre caméra se modifie selon la position de la tête de l'utilisateur. Sur la figure 3.30, la tête de l'utilisateur est centrée par rapport à l'écran. Le plan image de la caméra montre que l'objet bleu occulte l'objet vert. L'utilisateur déplace donc sa tête vers la droite (cf. figure 3.31), ce qui permet d'observer l'objet vert, mais l'objet jaune est maintenant en partie en-dehors de l'image. Pour observer toute la scène, l'utilisateur a simplement besoin de s'approcher de l'écran (cf. figure 3.32), ce qui a pour conséquence d'élargir l'angle de vue. Tous les objets sont maintenant visibles, mais ils sont plus petits sur l'écran.

3.4.3.2. Localisation de la tête de l'utilisateur

La tête de l'utilisateur est localisée à l'aide d'une caméra stéréoscopique. Une caméra stéréoscopique est un système composé d'au moins deux caméras, qui permet de trianguler des points dans l'espace. Notre caméra stéréoscopique utilise des caméras IR qui observent un point IR placé sur la tête de l'utilisateur.

Principe de fonctionnement

Dans cette partie nous détaillons seulement le principe de fonctionnement d'une caméra stéréoscopique, car les détails mathématiques sont connus [63].

Pour trianguler un point de l'espace, nous avons besoin des paramètres des caméras qui l'observent. Les paramètres d'une caméra sont un modèle mathématique qui définit où un point de l'espace se projette dans l'image de la caméra.

Ce modèle est une matrice M, qui permet de projeter un point P dans l'image de la caméra aux coordonnées p, tel que :

$$p \propto MP$$

Formule 3.17 – Projection d'un point dans l'image d'une camera.

Par extension, nous pouvons utiliser la matrice M et la position p, pour déterminer la droite de l'espace sur laquelle se trouve P. Donc, avec deux caméras, nous pouvons déterminer deux droites distinctes qui passent par P et en calculant l'intersection de ces deux droites nous pouvons déterminer la position de P (cf. figure 3.33).

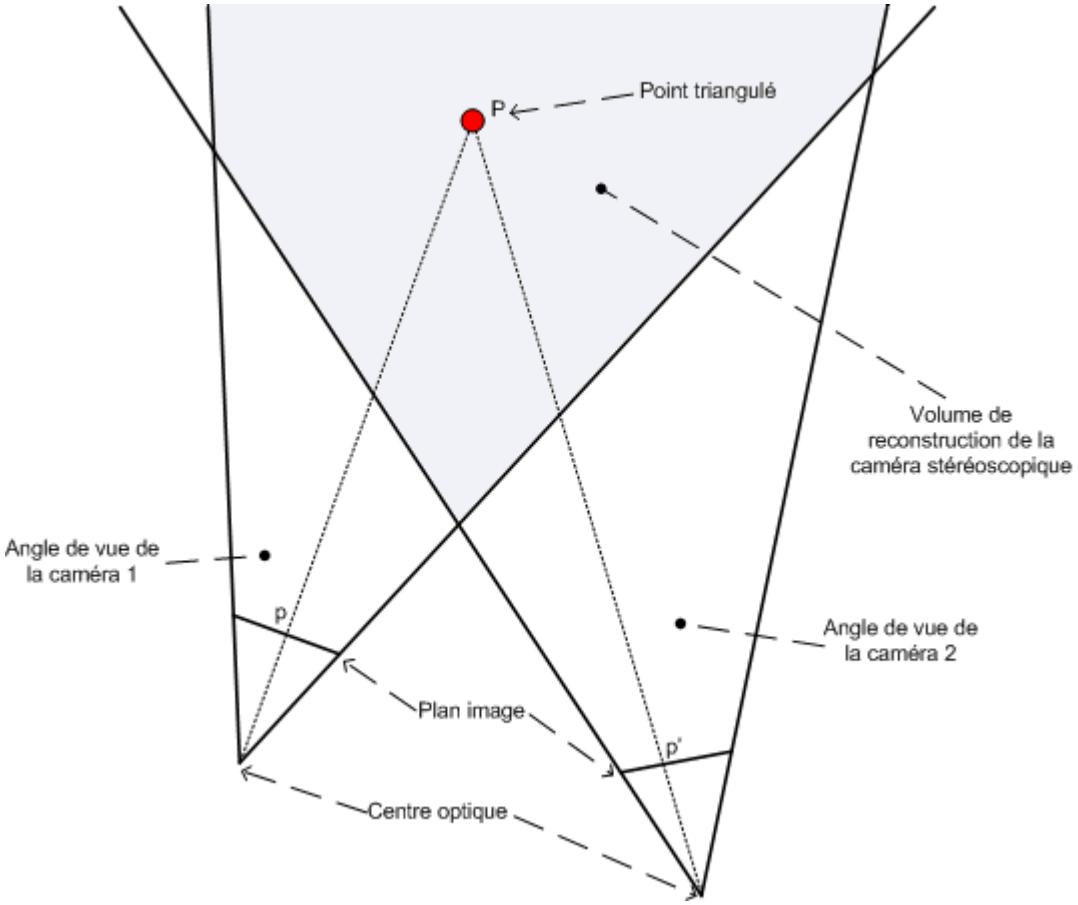

Figure 3.33 – Triangulation d'un point par une caméra stéréoscopique.

Le volume de reconstruction d'un système stéréoscopique est l'union des volumes observés par au moins deux des caméras qui composent ce système (cf. figure 3.33). L'ajout de caméras permet d'agrandir le volume de reconstruction. Il peut également servir à améliorer la précision de la triangulation, car notre système est soumis à plusieurs erreurs de mesure. La raison principale est que nous utilisons des caméras numériques, la position du point observée dans l'image est donc discrétisée selon la précision (*i.e.,* la matrice) du capteur de la caméra. D'autres erreurs viennent également fausser la triangulation. Par exemple lors du calcul des paramètres nous faisons une approximation de la matrice M, ce qui induit inévitablement une erreur lors de la triangulation.

Les paramètres d'une caméra se séparent en deux groupes : les paramètres internes et les paramètres externes. Les paramètres internes ne changent pas, ils définissent le modèle mathématique de l'objectif de la caméra. Les paramètres externes varient en fonction de la position et de l'orientation de la caméra. Pour déterminer les paramètres d'une caméra, il faut la calibrer.

Nous calibrons nos caméras avec une méthode qui utilise l'image des points d'un plan observés depuis plusieurs positions. Le plan est l'écran de l'ordinateur. Pour chaque position d'observation, nous comparons la position d'au moins quatre points du plan, avec leur position dans l'image de la caméra. Plus le nombre de points est grand, plus il permet de réduire l'erreur de calibration (*i.e.,* de mesure). La première fois qu'une caméra est calibrée, nous devons déterminer tous ses paramètres. Pour cela, nous observons le plan depuis trois positions différentes. Plus le nombre de positions d'observation est grand, plus il permet de réduire l'erreur de calibration. Cette première calibration permet de déterminer les paramètres internes de la caméra et une série de paramètres externes correspondants aux positions depuis lesquelles le plan a été observé. Tant que la caméra reste dans l'une de ces positions, nous disposons de tous ses paramètres. Si la caméra est déplacée, nous devons recalibrer ses paramètres externes. Pour cela, nous devons observer le plan depuis seulement une position, car les paramètres internes sont déjà définis.

Cette méthode, appelée « calibration planaire », est avantageuse, car elle permet de calibrer nos caméras dans le système de coordonnées de l'écran. Ce système de coordonnées est un système main droite [64], qui a sont origine dans le coin supérieur gauche de l'écran et dont l'axe des abscisses est dirigé vers la droite parallèlement la largeur de l'écran. Cet avantage est de taille, car nous reconstruisons la vue de l'utilisateur à partir de la position de sa tête par rapport à l'écran. Donc, si les positions renvoyées par notre caméra stéréoscopique n'étaient

pas dans le système de coordonnées de l'écran, il faudrait le localiser pour effectuer un changement de repère.

Mise en œuvre

Les caméras IR que nous utilisons sont des manettes de la console *Nintendo Wii*. Ces manettes sont équipées d'une caméra IR de 1024 par 768 pixels, qui capte 100 images par seconde. Elles utilisent une connexion sans fil, ce qui facilite la mise en place de notre système. Enfin, elles ont l'avantage de renvoyer la position et les dimensions des points observés, plutôt que leur image. Ainsi, nous n'avons pas besoin de chercher les points dans l'image.

Lors de nos tests, nous avons utilisé deux caméras, placées en arrière de l'utilisateur et pointées vers l'écran. Néanmoins, notre système permet en théorie d'inclure un nombre illimité de caméras. En pratique, le programme que nous utilisons pour connecter les manettes de *Wii* à l'ordinateur ne permet pas pour l'instant d'utiliser plus de huit manettes simultanément.

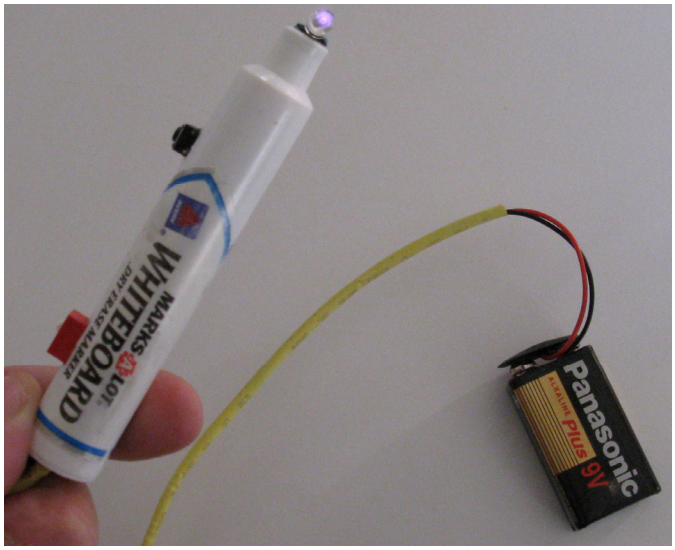

Figure 3.34 – Point IR utilisé pour la calibration.

Figure 3.35 – Point IR utilisé pour la localisation de la tête de l'utilisateur.

Nous utilisons deux points IR différents. Le premier est monté (cf. figure 3.34) sur un feutre et comporte une seule diode IR. Nous l'utilisons pour calibrer les caméras, car il est facile à manipuler et son point IR est petit, ce qui réduit les erreurs de mesure. Le deuxième point IR (cf. figure 3.35) est celui que nous plaçons sur la tête de l'utilisateur. Il est monté sur un serre-tête et comporte deux couronnes de diodes IR qui peuvent être allumées indépendamment. Les diodes sont disposées en quinconce de façon à élargir l'angle d'émission de notre point IR, c'est-à-dire, l'angle sur lequel il est perceptible par les caméras. La première couronne comporte huit diodes, la deuxième en comporte dix. La deuxième couronne a été prévue dans le cas où les caméras seraient très écartées. Cependant, dans la pratique nous n'en avons jamais eu besoin.

Avantage de l'approche stéréoscopique

Chung Lee localise la tête de l'utilisateur à l'aide de deux points IR placés dessus et d'une caméra IR placée sous l'écran [62]. L'idée est de comparer la distance réelle des deux points IR avec leur distance et leurs positions dans l'image de la caméra. Cette approche pose un problème majeur, car elle suppose que l'utilisateur regarde toujours dans la direction du centre de l'écran. En effet, si l'utilisateur tourne la tête par rapport à l'écran, les points IR vont se rapprocher dans l'image de la caméra et le système pensera qu'il a reculé. L'approche stéréoscopique permet de résoudre ce problème, car quelle que soit l'orientation de la tête de l'utilisateur, sa position sera toujours localisée de la même façon.

CHAPITRE 4.

VALIDATION

Nous avons réalisé une étude exploratoire pour évaluer EOD. Cette étude a consisté à présenter à nos sujets l'évolution d'un logiciel dans EOD. Puis nous leur avons demandé de réaliser cinq tâches chronométrées de lecture des valeurs des métriques à différents moments de l'évolution du code source. Enfin, ils devaient remplir un questionnaire d'évaluation sur les avantages et les inconvénients d'EOD.

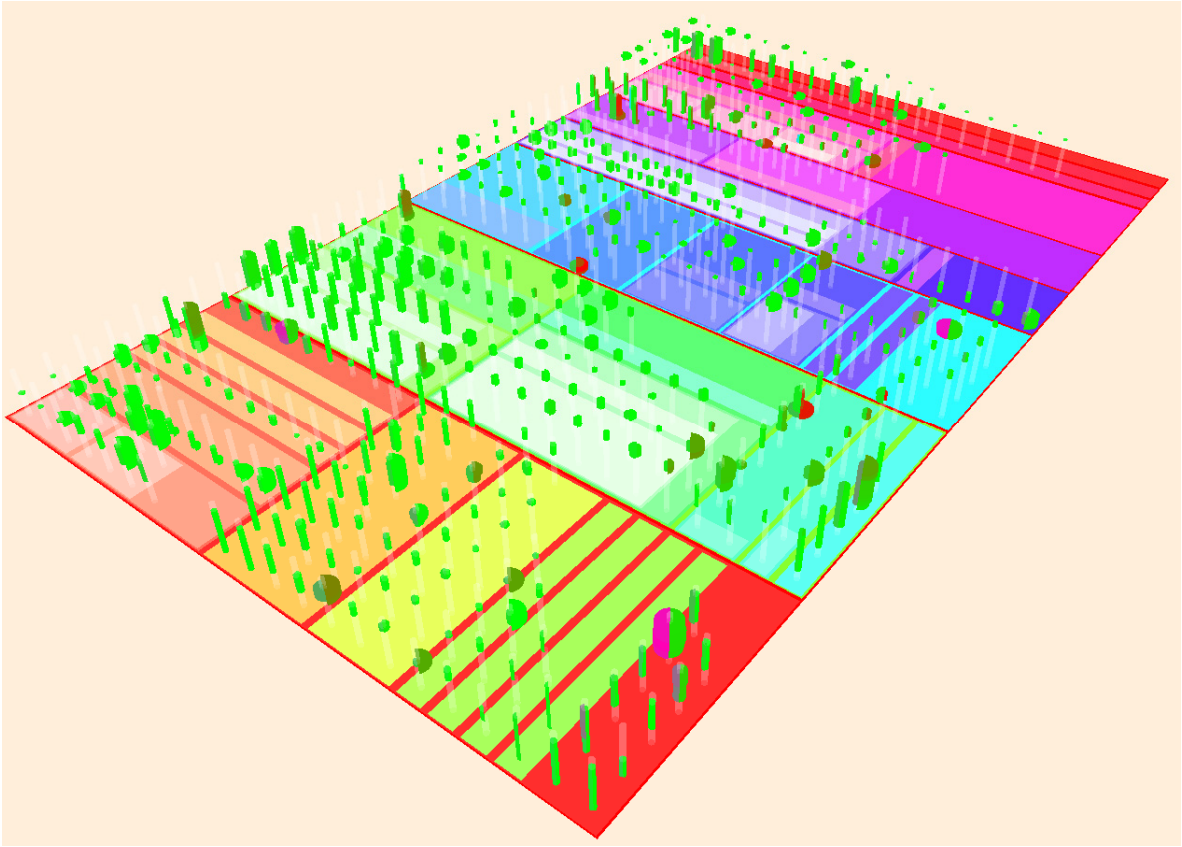

Figure 4.1 – Scène présentée à nos sujets pour réaliser les tâches.

4.1. Scène visualisée

La scène visualisée (cf. figure 4.1) présente l'évolution du module d'EOD qui permet d'extraire et d'analyser le code contenu dans un SGV. Nous avons choisi ce logiciel, car nous connaissons bien ses points forts et ses points faibles. La scène

visualisée présente 462 classes et interfaces sur 33 clichés et affiche 19 835 valeurs de métriques.

Les métriques visualisées étaient associées aux VV de la façon suivante :

- la couleur de la moitié gauche des bâtiments était associée à deux métriques : le nombre de méthodes déclarées (NMD) et le nombre d'attributs déclarés (NAD);
- la couleur de la moitié droite était associée à une métrique : le couplage entre les objets (CBO);
- le rayon de la moitié gauche était associé au poids des méthodes par classe (WMC);
- le rayon de la moitié droite était associé à la métrique de cohésion (LCOM5);

4.2. Les sujets

L'étude a été réalisée de façon anonyme avec 14 sujets, dont 6 étudiants en doctorat, 6 étudiants en maîtrise et 2 étudiants en baccalauréat. Nous avons demandé aux sujets quelle est leur aisance à manipuler des scènes en 3D : 6 s'estimaient débutant, 6 moyen et 2 confirmé. Seulement 5 sujets avaient déjà manipulé un SV en 3D et 4 avaient déjà utilisé des métriques pour évaluer la qualité.

4.3. Le protocole

Lors de cette étude, nous n'avons pas activé le système de réduction de l'occlusion. Avant de commencer les tâches, nous avons lu aux sujets un texte expliquant le fonctionnement d'EOD et l'association des métriques avec les VV. À la suite de cette lecture, les sujets étaient libres de poser des questions pour clarifier

les explications. Puis, nous avons demandé aux sujets de réaliser les tâches, selon le protocole suivant :

1. La tâche à réaliser est lue au sujet.
2. Le sujet peut poser toutes les questions qu'il estime nécessaires pour clarifier la tâche à réaliser.
3. La scène est affichée et le sujet peut commencer à réaliser la tâche.
4. Pour répondre, le sujet clique simplement sur l'entité qui est selon lui la réponse de la tâche à réaliser.
5. Nous notons manuellement la réponse qui s'affiche à l'écran.
6. La scène est cachée pour énoncer la tâche suivante (retour en 1).

Après avoir réalisé toutes les tâches, le sujet pouvait prendre le temps qu'il désirait pour remplir le questionnaire sur les avantages et inconvénients d'EOD, ce qui terminait l'expérience.

4.4. Les tâches

Voici les tâches demandées à nos sujets :

1. Trouver une classe ou une interface dont le couplage augmente au fil du temps.
2. Trouver une classe ou une interface ayant le plus fort couplage au début de son l'évolution visualisée ou à sa création.
3. Trouver une classe ou une interface ayant le plus grand nombre d'attributs à la fin de l'évolution visualisée ou à sa destruction.
4. Trouver une classe ou une interface qui a le plus grand nombre de méthodes à la fin de l'évolution visualisée ou à sa destruction.

5. Trouver une classe ou une interface qui a le plus grand nombre de méthodes et d'attributs à la fin de l'évolution visualisée ou à sa destruction.

4.5. Le questionnaire

Voici le questionnaire que les sujets devaient remplir après avoir effectué les tâches :

1. Selon vous, quelle est la difficulté de se positionner dans la scène?
2. Lorsque vous observez une tour, quelle est la difficulté de différencier les variations de couleurs?
3. Lorsque vous observez une tour, quelle est la difficulté de différencier les variations de sa largeur?
4. Quelle est la difficulté d'interprétation de l'association d'une métrique avec une couleur?
5. Quelle est la difficulté d'interprétation de l'association de deux métriques avec une couleur?
6. Quelle est la difficulté de comparer plusieurs tours.
7. Quelle est la difficulté de distinguer la base des tours.
8. Pensez-vous qu'un tel principe de visualisation peut aider à la compréhension de l'évolution du logiciel?
9. Maintenant que vous savez qu'EOD existe, seriez-vous prêt à l'utiliser lors de vos développements?

Pour les questions 1 à 7, les sujets devaient noter la difficulté entre 1 et 5 ; 1 signifiant « aucune difficulté », 3 « difficulté moyenne » et 5 « impossible ». Pour la question 8, les sujets devaient noter l'utilité d'EOD entre 1 et 5 ; 1 signifiant « inutile » et 5 « très utilise ». Enfin, pour la question 9, les sujets devaient noter

s'ils étaient prêts à utiliser EOD entre 1 et 5 ; 1 signifiant « pas du tout » et 5 « certainement ».

4.6. Les résultats

4.6.1. Résultats des tâches

Les résultats de cette étude sont présentés dans les tableaux 4.1 et 4.2.

De cette étude exploratoire, nous déduisons qu'EOD permet de remplir aisément les 4 premières tâches, car elles ont toutes un taux de réussite supérieur à 85%. Parmi ces quatre tâches, la première est la seule qui demandait aux sujets d'explorer à la fois les entités et leur évolution. Le nombre de valeurs à comparer était donc plus important que pour les autres tâches, ce qui la rendait plus délicate à réaliser. Nous pensons que c'est la raison pour laquelle elle affiche un taux de réussite (85,7%) légèrement inférieur aux autres. La deuxième tâche est la seule qui demandait aux sujets d'observer la base des bâtiments. Or, comme notre système de réduction de l'occlusion n'était pas activé, les sujets devaient se déplacer dans la scène pour résoudre l'occlusion. Nous pensons que c'est la raison pour laquelle la deuxième tâche a le plus long temps de réponse moyen. Cependant, nous constatons que les déplacements ne semblent pas avoir faussés les résultats du processus analytique, puisque cette tâche à un taux de réussite de 92,9%.

Les trois dernières tâches traitaient de l'association de deux métriques avec une couleur. La troisième et la quatrième tâche demandaient aux sujets d'observer une seule des deux métriques. La quatrième tâche demandait aux sujets d'observer les deux métriques à la fois. La quatrième tâche a un taux de réussite inférieur aux

Sujet	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Exactitude de la réponse														
tâche 1	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	×	✓	✓
tâche 2	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
tâche 3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tâche 4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓
tâche 5	✓	×	×	✓	✓	✓	×	×	✓	✓	✓	×	✓	×
Temps de réponse (en seconde)														
tâche 1	18	20	45	35	41	26	50	115	20	26	57	50	20	16
tâche 2	15	40	90	45	20	49	10	120	15	46	60	25	30	18
tâche 3	13	10	75	10	25	20	40	25	25	25	15	28	35	17
tâche 4	11	10	35	5	20	15	15	25	15	11	57	15	16	15
tâche 5	35	30	37	20	45	25	45	20	25	113	10	56	28	36
Temps moyen de réponse	18,4	22	56,4	23	30,2	27	32	61	20	44,2	39,8	34,8	25,8	20,4
Temps moyen pour une réponse exacte	18,4	20	61,25	23	30,2	27	28,75	25	20	44,2	35,5	22,67	25,8	16,5
Temps moyen pour une réponse inexacte	0	30	37	0	0	0	45	85	0	0	57	53	0	36

Tableau 4.1 – Résultats des tâches classés par sujet.

	Pourcentage de bonnes réponses	Temps de réponse moyen (secondes)	Temps moyen d'une réponse exacte (secondes)	Temps moyen d'une réponse inexacte (secondes)
tâche 1	85,71%	38,50	31,17	82,50
tâche 2	92,86%	41,64	35,62	120,00
tâche 3	100,00%	25,93	25,93	0,00
tâche 4	92,86%	18,93	16,00	57,00
tâche 5	57,14%	37,50	37,63	37,33

Tableau 4.2 – Moyenne des résultats classés par tâche.

autres (57%). Pourtant, ce n'est pas le cas de la troisième et de la quatrième tâche qui ont des taux de réussite de 92,9% et 100%. Nous pensons que la raison de ces différences est qu'il n'est pas possible d'associer une VV à elle-même. En effet, pour la troisième et la quatrième tâche, nous demandions aux sujets de diriger leur attention seulement sur une échelle de teintes d'une couleur, alors que pour la quatrième tâche, nous demandions aux sujets de suivre deux échelles de teintes sur une seule couleur.

4.6.2. Résultats du questionnaire

Les résultats du questionnaire sont présentés dans le tableau 4.3.

Sujet	0	1	2	3	4	5	6	7	8	9	10	11	12	13	moyenne
Question 1	1	1	3	3	2	1	2	1	2	3	1	3	2	1	1,86
Question 2	3	2	3	2	3	1	1	1	2	4	3	1	2	3	2,21
Question 3	1	3	1	2	3	2	1	1	4	1	1	2	1	2	1,79
Question 4	1	1	1	1	4	1	1	1	3	3	2	2	1	1	1,64
Question 5	4	3	2	1	4	4	3	2	2	1	1	3	1	3	2,43
Question 6	3	2	2	1	2	3	2	3	3	1	1	1	3	2	2,07
Question 7	1	4	4	4	2	2	1	2	4	3	4	3	3	1	2,71
Question 8	5	4	3	5	2	5	2	5	5	4	5	4	4	4	4,07
Question 9	5	5	2	4	3	5	2	5	4	4	4	4	3	5	3,93

Tableau 4.3 – Résultats du questionnaire classés par sujets.

Les 7 premières questions portent sur la difficulté d'utilisation d'EOD. Pour toutes ces questions la difficulté moyenne notée par les sujets n'excède pas 3. Nous en déduisons que les utilisateurs trouvent globalement que EOD est facile à utiliser.

Les difficultés notées par les sujets pour les questions 1, 3 et 4 sont les plus faibles, elles n'excèdent pas 2. Nous en déduisons que les sujets se déplacent facilement dans la scène (question 1) et que le concept de la pyramide des âges est bien compris (question 3 et 4).

Les difficultés notées par les sujets pour les questions 5 et 7 sont le plus importantes, ce qui s'explique facilement. En effet, la question 5 demandait aux sujets quelle était la difficulté d'interpréter l'association de deux métriques avec une couleur et la question 7 demandait quelle était la difficulté d'observer la base des bâtiments. Cependant, nous espérons que la difficulté de distinguer la base des bâtiments perçue par les sujets baissera une fois que notre système de réduction de l'occlusion sera activé.

Pour finir, les questions 8 et 9 portaient sur l'utilité d'EOD et son attrait pour les sujets. Nous déduisons des résultats de ces questions que les sujets trouvent EOD utile et qu'ils sont prêts à l'utiliser pour analyser l'évolution de la qualité d'un logiciel.

CHAPITRE 5.

CONCLUSION

Dans ce mémoire, nous avons présenté un système de visualisation de l'évolution du logiciel sur une seule vue. Ce système, *Eyes Of Darwin*, utilise une métaphore avec des bâtiments et des quartiers d'une ville. Chaque bâtiment est associé à une entité du code et permet de suivre l'évolution de 2 à 6 données qui la caractérise.

Les bâtiments d'*Eye Of Darwin* réutilisent le concept de la pyramide des âges. Chaque tranche de la pyramide représente une période de l'évolution de l'entité du code à laquelle le bâtiment est associé et l'ensemble de la pyramide représente l'évolution complète de cette entité. Par ailleurs, les quartiers permettent de visualiser l'organisation géographique des entités visualisées. Cependant, cette approche impose d'afficher de l'information sur le côté des bâtiments, ce qui génère des problèmes d'occlusion. C'est pourquoi nous avons doté *Eyes Of Darwin* d'un module qui transforme l'écran en une fenêtre ouverte sur la scène qu'il affiche. Ainsi, pour régler les problèmes d'occlusion, l'utilisateur peut simplement déplacer sa tête par rapport à l'écran, ce qui est un réflexe et donc qui ne perturbe pas le processus analytique. Enfin, dans ce mémoire nous avons présenté une étude exploratoire qui montre qu'EOD permet de remplir les tâches pour lesquelles il a était conçu et qu'il est facile à utiliser.

Dans le futur, nous souhaitons réaliser une expérience plus poussée pour compléter la validation d'EOD. Plus précisément, nous souhaitons valider (1) la facilité d'interprétation de notre *treemap*, (2) les filtres, (3) l'association des métriques avec les VV et (4) le système de réduction de l'occlusion.

De plus, nous souhaitons modifier l'association de deux métriques avec une couleur. En effet, l'étude exploratoire que nous avons menée tend à montrer que le concept actuellement utilisé est difficile à comprendre. C'est pourquoi, nous proposons d'associer la première métrique à une échelle de couleurs variant entre deux teintes et la deuxième métrique à la saturation (*i.e.,* l'intensité) de cette couleur. Comme la teinte et l'intensité d'une couleur sont deux VV différentes, nous pensons que cela pourrait permettre à l'utilisateur de lire les deux métriques à la fois.

Nous souhaitons également qu'EOD offre plus d'interactivité à ses utilisateurs, pour leur permettre de consulter les valeurs des métriques qui ne sont pas visualisées sur les bâtiments et pour leur permettre d'annoter la scène avec les résultats de leurs analyses.

Pour finir, nous souhaitons faciliter les déplacements au sein de la scène à l'aide d'une troisième manette de *Wii*. En effet, en plus de contenir une caméra IR, ces manettes contiennent 3 accéléromètres qui permettent de connaître leur orientation par rapport au plan horizontal (*i.e.,* leur assiette). En utilisant à la fois les accéléromètres et la caméra IR, nous pouvons associer la position et l'orientation de la manette à celles du plan d'affichage de la scène. Ainsi, l'utilisateur pourrait se déplacer dans la scène en déplaçant simplement la manette qu'il tient dans ses mains.

BIBLIOGRAPHIE

[1]. R. Pressman, *Software Engineering: A Practitioner's Approach*. 6th ed. 2005: McGraw-Hill. pp. 880.

[2]. S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, *Does code decay? Assessing the evidence from change management data.* Transactions on Software Engineering, 2001. **27**(1): IEEE Computer Society Press. p. 1–12.

[3]. M.M. Lehman. *Laws of Software Evolution Revisited*. in *Proceedings of the 5th Workshop on Software Process Technology*. 1996: Springer-Verlag. p. 108–124.

[4]. S.R. Chidamber and C.F. Kemerer. *Towards a metrics suite for object oriented design*. in *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*. 1991: ACM Press. p. 197–211.

[5]. S.R. Chidamber and C.F. Kemerer, *A Metrics Suite for Object Oriented Design.* Transactions on Software Engineering, 1994. **20**(6): IEEE Computer Society Press. p. 476–493.

[6]. T.J. McCabe. *A complexity measure*. in *Proceedings of the 2nd Conference on Software Engineering*. 1976: IEEE Computer Society Press. p. 407.

[7]. E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides, *Design patterns : elements of reusable object-oriented software*. Professional computing series. 1995: Addison-Wesley. pp. 395.

[8]. M. Fowler and K. Beck, *Refactoring : improving the design of existing code*. Object technology series. 1999: Addison-Wesley. pp. 431.

[9]. W.J. Brown, R.C. Malveau, W.S.M. Hays, and T.J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. 1998: Wiley. pp. 336.

[10]. B. Berliner. *CVS II: Parallelizing Software Development*. in *Proceedings of the Winter 1990 USENIX Conference*. 1990. p. 341–352.

[11]. B. Collins-Sussman, B.W. Fitzpatrick, and C.M. Pilato, *Version control with Subversion*. 2004: O'Reilly Media.

[12]. D. Grune, *Concurrent Versions System, A Method for Independent Cooperation*. 1986, Vrije Universiteit Brussels.

[13]. IBM. *IBM Rational ClearCase*. 2009 [last visisted on 25/09/2009]; Available from: http://www-01.ibm.com/software/awdtools/clearcase/.

[14]. J. Loeliger, *Version Control with Git: Powerful tools and techniques for collaborative software development*. 2009: O'Reilly Media. pp. 328.

[15]. Microsoft. *Introduction à Visual SourceSafe*. 2005 [last visisted on 25/09/2009]; Available from: http://msdn.microsoft.com/fr-fr/library/3h0544kx%28VS.80%29.aspx.

[16]. Perforce. *Perforce Server*. 2009 [last visisted on 25/09/2009]; Available from: www.perforce.com/perforce/products/p4d.html.

[17]. M.J. Rochkind, *The Source Code Control System.* Transactions on Software Engineering, 1975. **1**(4): IEEE Computer Society Press. p. 364–370.

[18]. W.F. Tichy, *RCS–a system for version control.* Software–Practice & Experience, 1985. **15**(7): Wiley. p. 637–654.

[19]. S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc. *Prevention and Cure of Software Defects: Lessons from the Study of God Classes*. in *Proceedings of the 16th Working Conference of Reverse Engineering*. 2009.

[20]. D.E. Knuth, *Structured Programming with go to Statements.* Computer Survey, 1974. **6**(4): ACM Press. p. 261–301.

[21]. E.W. Dijkstra, *Letters to the editor: go to statement considered harmful.* Communications, 1968. **11**(3): ACM Press. p. 147–148.

[22]. E. Arisholm. *Dynamic Coupling Measures for Object-Oriented Software*. in *Proceedings of the 8th Symposium on Software Metrics*. 2002: IEEE Computer Society Press. p. 33–43.

[23]. V.R. Basili, L.C. Briand, and W.L. Melo, *A Validation of Object-Oriented Design Metrics as Quality Indicators.* Transactions on Software Engineering, 1996. **22**(10): IEEE Computer Society Press. p. 751–761.

[24]. D. Glasberg, K. El-Emam, W. Memo, and N. Madhavji, *Validating Object-Oriented Design Metrics on a Commercial Java Application*. 2000, NRC-CNRC. pp. 54.

[25]. T. Gyimothy, R. Ferenc, and I. Siket, *Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction.* Transactions on Software Engineering, 2005. **31**(10): IEEE Computer Society Press. p. 897–910.

[26]. N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. Le Meur, *DECOR: A Method for the Specification and Detection of Code and Design Smells.* Transactions on Software Engineering, 2009: IEEE Computer Society Press. p. 1–17.

[27]. Y.-G. Guéhéneuc and G. Antoniol, *DeMIMA: A Multilayered Approach for Design Pattern Identification.* Transactions on Software Engineering, 2008. **34**(5): IEEE Computer Society Press. p. 667–684.

[28]. N. Moha, *DECOR : Détection et correction des défauts dans les systèmes orientés objet*, in *DIRO*. 2008, Université de Montréal - Université des Sciences et Technologies de Lille: Montréal. pp. 155.

[29]. C. Knight and M. Munro. *Comprehension with[in] Virtual Environment Visualisations*. in *Proceedings of the 7th Workshop on Program Comprehension*. 1999: IEEE Computer Society Press. p. 4–11.

[30]. A.D. Baddeley, N. Thomson, and M. Buchanan, *Word length and the structure of short-term memory.* Journal of Verbal Learning and Verbal Behavior, 1975. **14**: American Psychological Association. p. 575–589.

[31]. G.A. Miller, *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.* The Psychological Review, 1956. **63**: American Psychological Association. p. 81–97.

[32]. J. Bertin, *Sémiologie graphique*. 1967, Paris: Mouton/Gauthier-Villars.

[33]. K. Mullet and D. Sano, *Designing visual interfaces: communication oriented techniques*. 1995: Prentice-Hall. pp. 273.

[34]. J. Nielsen, *Usability Engineering*. 1994: Morgan Kaufmann. pp. 340.

[35]. V.L. Averbukh, *Visualization Metaphors.* Programming and Computing Software, 2001. **27**(5): Plenum Press. pp. 227-237.

[36]. E.R. Tufte, *The visual display of quantitative information*. 1986: Graphics Press. pp. 200.

[37]. Wikipedia. *Psychologie de la forme.* 2009 [last visisted on 07/10/2009]; Available from: http://fr.wikipedia.org/wiki/Psychologie_de_la_forme.

[38]. M. Handford, *Où est charlie?* 1998: Gründ.

[39]. E.G. Louie, D.W. Bressler, and D. Whitney, *Holistic crowding: Selective interference between configural representations of faces in crowded scenes.* Journal of Vision, 2007. **7**(2). p. 1–11.

[40]. M. Martelli, N.J. Majaj, and D.G. Pelli, *Are faces processed like words? A diagnostic test for recognition by parts.* Journal of Vision, 2005. **5**(1). p. 58–70.

[41]. S.C. Eick, J.L. Steffen, and E.E. Sumner, Jr., *Seesoft-a tool for visualizing line oriented software statistics.* Transactions on Software Engineering, 1992. **18**(11): IEEE Computer Society Press. p. 957–968.

[42]. C. Mesnage and M. Lanza. *White Coats: Web-Visualization of Evolving Software in 3D.* in *Proceedings of the 3rd Workshop on Visualizing Software for Understanding and Analysis.* 2005: IEEE Computer Society Press. p. 1–6.

[43]. D.M. Germán and A. Hindle, *Visualizing the Evolution of Software Using Softchange.* International Journal of Software Engineering and Knowledge Engineering, 2006. **16**(1): World Scientific Publishing Co. p. 5–22.

[44]. A. Marcus, L. Feng, and J.I. Maletic. *3D representations for software visualization.* in *Proceedings of the Symposium on Software visualization.* 2003: ACM Press. p. 27–36.

[45]. S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, *Visualizing software changes.* Transactions on Software Engineering, 2002. **28**(4): IEEE Computer Society Press. p. 396–412.

[46]. J. Froehlich and P. Dourish. *Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams.* in *Proceedings of the 26th Conference on Software Engineering.* 2004: IEEE Computer Society Press. p. 387–396.

[47]. T. Panas, R. Berrigan, and J. Grundy. *A 3D Metaphor for Software Production Visualization.* in *Proceedings of the 7th Conference on Information Visualization.* 2003: IEEE Computer Society Press. p. 314–319.

[48]. B. Shneiderman, *Tree visualization with tree-maps: 2-d space-filling approach.* Transactions on Graphics, 1992. **11**(1): ACM Press. p. 92–99.

[49]. M. Balzer, O. Deussen, and C. Lewerentz. *Voronoi treemaps for the visualization of software metrics.* in *Proceedings of the Symposium on Software Visualization.* 2005: ACM Press. p. 165–172.

[50]. H. Byelas and A. Telea. *Visualizing metrics on areas of interest in software architecture diagrams.* in *Proceedings of the Symposium on Software Visualization* 2009: IEEE Computer Society Press. p. 33–40.

[51]. M. Lanza and S. Ducasse. *A categorization of classes based on the visualization of their internal structure: the class blueprint.* in *Proceedings of the 16th Conference on Special Interest Group on Programming Languages.* 2001: ACM Press. p. 300–311.

[52]. A. McNair, D.M. German, and J. Weber-Jahnke. *Visualizing Software Architecture Evolution Using Change-Sets.* in *Proceedings of the 14th Working Conference on Reverse Engineering.* 2007: IEEE Computer Society Press. p. 130–139.

[53]. M. Termeer, C.F.J. Lange, A. Telea, and M.R.V. Chaudron. *Visual Exploration of Combined Architectural and Metric Information*. in *Proceedings of the 3rd Workshop on Visualizing Software for Understanding and Analysis*. 2005: IEEE Computer Society Press. p. 1–6.

[54]. S. Alam and P. Dugerdil. *EvoSpaces: 3D Visualization of Software Architecture*. in *Proceedings of the 19th Conference on Software Engineering and Knowledge Engineering*. 2007: IEEE Computer Society Press. p. 500–505.

[55]. G. Langelier, *Visualisation de la qualité des logiciels de grandes tailles*, in *DIRO*. 2006, Université de Montréal: Montréal. pp. 117.

[56]. R. Wettel and M. Lanza. *Visually localizing design problems with disharmony maps*. in *Proceedings of the 4th Symposium on Software Visualization*. 2008: ACM Press. p. 155–164.

[57]. M. Bruls, K. Huizing, and J.v. Wijk. *Squarified Treemap*. in *Proceedings of the Joint Eurographics and TCVG Symposium on Visualization*. 1999: IEEE Computer Society Press. p. 33–42.

[58]. M. Lanza. *The evolution matrix: recovering software evolution using software visualization techniques*. in *Proceedings of the 4th Workshop on Principles of Software Evolution*. 2001: ACM Press. p. 37–42.

[59]. M. Pinzger, H. Gall, M. Fischer, and M. Lanza. *Visualizing multiple evolution metrics*. in *Proceedings of the Symposium on Software Visualization*. 2005: ACM Press. p. 67–75.

[60]. R. Wettel and M. Lanza. *Visual Exploration of Large-Scale System Evolution*. in *Proceedings of the 15th Working Conference on Reverse Engineering*. 2008: IEEE Computer Society Press. p. 219–228.

[61]. R. McGill, J.W. Tukey, and W.A. Larsen, *Variations of Box Plots*. The American Statistician, 1978. **32**(1): American Statistical Association. p. 12–16.

[62]. J. Chung Lee. *Head Tracking for Desktop VR Displays using the Wii Remote*. 2008 [last visisted on 2/12/2009]; Available from: http://johnnylee.net/projects/wii/.

[63]. D.A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*. 1st ed. 2002: Prentice Hall. pp. 693.

[64]. Wikipedia. *Règle de la main droite*. 2009 [last visisted on 2/12/2009]; Available from: http://fr.wikipedia.org/wiki/R%C3%A8gle_de_la_main_droite.