

**Université de Strasbourg
CSMI**



Rapport de stage

Keyplan3D

David ALGIS

*Entreprise : QUASARTS
Encadré par : Romain Boisseron*

Table des matières

Introduction	1
1 Formalisation de la hauteur des murs	4
1.1 Rapide présentation de la structure de Keyplan	4
1.2 Appréhension de la difficulté des problèmes	4
1.3 Formalisation mathématiques	5
1.3.1 La fonction de génération naïve : genframe	7
1.3.2 Classification des différentes intersections entre les murs	8
1.3.3 Les fonctions n-maçon	9
1.4 2-maçon	9
1.4.1 Deux murs de même hauteur	10
1.4.2 Deux murs de hauteurs différentes	10
1.5 n-maçon	16
1.5.1 Partitionnement des murs	16
1.5.2 Connexion des murs principaux	19
1.5.2.1 Deux murs principaux	19
1.5.2.2 Plus de deux murs principaux	19
1.5.3 Connexion des murs accessoires	19
2 Implémentation du code	27
2.1 Prototypage	27
2.2 En pratique	30
2.3 Booléens	32
2.4 Débogage	33
3 Esquisse sur l'ajout des étages	37
3.1 Règles pour la création et la modification d'étages	37
3.1.1 Création d'étage	37

TABLE DES MATIÈRES

3.1.2	Idée d'interface	37
3.1.3	Modification des étages	39
3.2	« Rattacher » des étages	39
3.2.1	Génération d'escalier	40

Introduction

Quasarts est une entreprise qui développe une application iOS d'architecture grand public Keyplan3D. Celle-ci permet de créer rapidement et simplement des pièces, d'y rajouter de l'immobilier, des objets de la vie quotidienne, mais aussi de changer le parquet ou encore de personnaliser la tapisserie. Tout ceci, est fait dans un souci de cohérence architecturale, de plus chaque objet de Keyplan3D possèdent des mesures précises (voir [2](#) et [1](#)).

C'est pourquoi, les utilisateurs de Keyplan3D sont en majorité des particuliers, mais aussi on trouve, notamment des experts en assurance, des élèves de primaires en Angleterre, des décoratrices d'intérieur, des agents immobiliers, des architectes d'intérieur. Ils ont tous en commun un besoin de modéliser rapidement et simplement un environnement cohérent et avec des mesures fiables.

L'application est disponible sur l'Appstore sous deux versions :

- Une première version gratuite mais limitée permettant de faire une seule pièce par exemple.
- Une seconde version payante avec toutes les fonctionnalités de base.

De plus, l'application possède aussi un modèle de packs thématiques [1](#) contenant des objets supplémentaires à rajouter dans ses scènes. Ces packs sortent régulièrement et sont payants dans l'application.

Quasarts est une entreprise de deux employés à plein temps : M.Boisseron, le patron de l'entreprise, a développé Keyplan3D et a encadré le stage, ainsi qu'un infographiste qui crée les packs (cf. paragraphe ci-dessus). L'entreprise est située à Montpellier dans un incubateur de startup : le Montpellier International Business Incubator.

Le but du stage était de formaliser puis d'ajouter une fonctionnalité dans Keyplan3D ; Il s'agissait de la hauteur des murs. En effet, la hauteur des murs était une base pour l'ajout de nombreuses autres fonctionnalité comme les étages, mais aussi la toiture. Il est important de préciser que cette fonctionnalité doit s'intégrer dans Keyplan3D, comme nous l'avons dit plus haut, dans un souci de cohérence architecturale. C'est pourquoi, le stage s'est découpé en deux grandes parties :

1. Formalisation du problème.
2. Implémentation dans Keyplan3D et maintenance du code.

De plus, nous avons mené les réflexions plus loin, en réfléchissant sur la problématique d'ajout d'étages auquel nous avons consacré un dernier petit chapitre.

1. Par exemple un pack salon, un pack mobilier japonais, etc...



FIGURE 1 – Impression d'écran de Keyplan3D dans son mode vue du dessus pour tracer les murs et de rajouter des objets.



FIGURE 2 – Impression d'écran de Keyplan3D à un mode 3D pour naviguer dans sa scène

Chapitre 1

Formalisation de la hauteur des murs

Dans ce chapitre, nous allons définir des outils mathématiques afin de faciliter la construction d'un algorithme adéquate par rapport aux exigences de Keyplan3D.

Ainsi le chapitre se divisera en plusieurs parties :

- Nous présenterons rapidement la structure de l'application, et notamment les classes sur lesquelles nous allons travailler.
- Nous appréhenderons la difficulté du problème.
- Nous formaliserons mathématiquement le problème.
- Nous présenterons un algorithme pour connecter deux murs.
- Nous présenterons l'algorithme pour connecter les murs que nous avons utilisé (celui-ci est résumé sous forme d'un graphe dans la dernière page de ce chapitre).

1.1 Rapide présentation de la structure de Keyplan

Avant de commencer, il est nécessaire de faire une rapide présentation de la structure de Keyplan3D, pour plus de détails le lecteur est invité à lire la section [2.2](#) concernant l'implémentation de cette fonctionnalité.

Tout d'abord le projet Keyplan3D est un projet divisé en 3 parties :

- Relight un moteur graphique 3D fait maison avec l'API OpenGL. (C++)
- Vault gère la partie architecture de l'application. (C++)
- Keyplan gère la partie interface et utilisateur. (Objective-C, Swift, et C++)

Dans ce chapitre, nous nous intéressons en grande partie à Vault et en particulier à plusieurs classes définissant les fondations d'un plan :

- KWall les murs.
- KPoint les extrémités des murs représentant les points de connection.

1.2 Appréhension de la difficulté des problèmes

La hauteur est une fonctionnalité qui à première vue semble triviale à implémenter, aussi, nous verrons dans cette section qu'une implémentation de cette fonctionnalité nécessite beaucoup de réflexion.

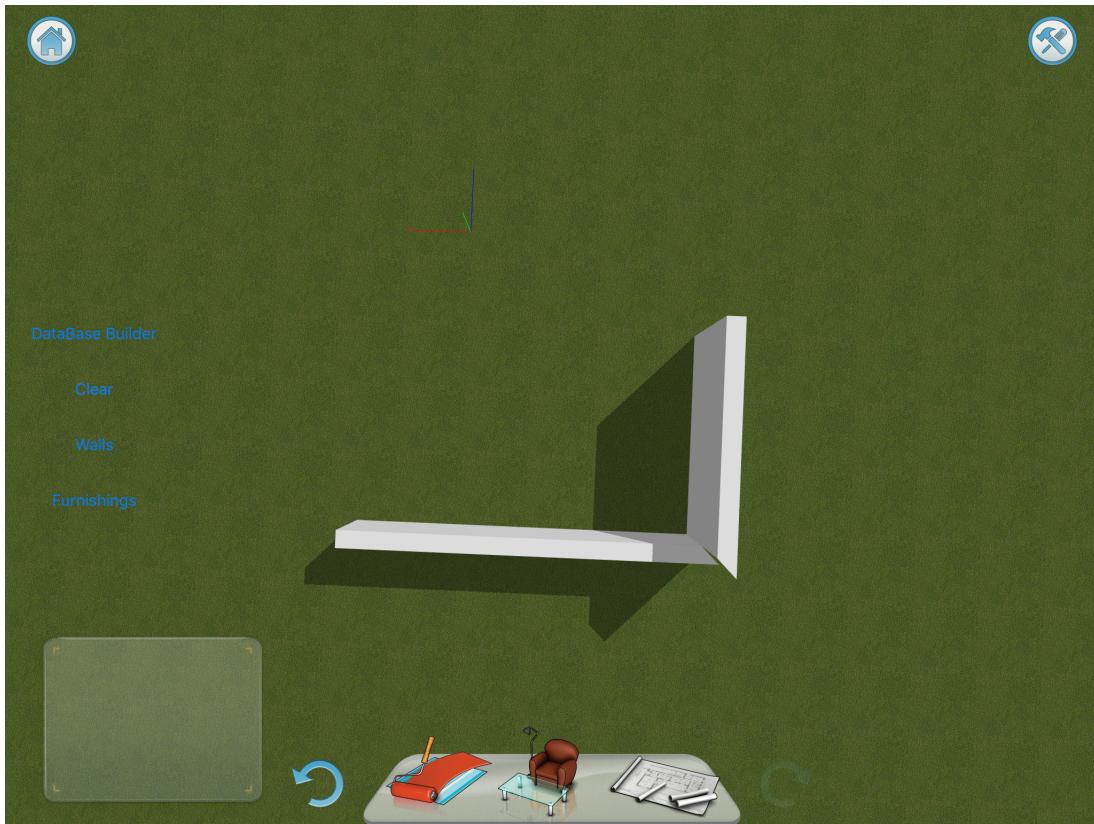


FIGURE 1.1 – Cette figure représente une implémentation naïve de la hauteur des murs. La jonction entre les deux murs pose problème.

En effet, commençons par faire une implémentation naïve, c'est à dire on ajoute uniquement un attribut `height` à la classe des murs `KWall`, cet attribut définit uniquement la hauteur de l'extrusion. On obtient la figure 1.1, on remarque deux problèmes, d'abord il manque certaines faces, cela peut être facilement corrigé, ensuite la jonction entre les faces est une diagonale.

Ce deuxième problème est beaucoup plus complexe à corriger. En effet, il s'agit d'une conséquence de l'ancien algorithme qui supposait que les murs étaient de la même hauteur. Et c'est ici que se pose le problème majeur : réussir à implémenter des hauteurs de murs différentes nécessite une complète révision du côté des jonctions entre les murs.

Reprenons l'exemple de la jonction de la figure 1.1, au lieu d'avoir une jonction diagonale qui n'a que très peu de sens en « matière architecturale », on souhaite que le mur le plus haut soutienne le mur le plus bas comme sur la figure 3.8. En bref, il faut donner une cohérence au sens « architectural » du terme, aux jonctions entre les différents murs.

1.3 Formalisation mathématiques

Tout d'abord, définissons notre problématique de façon informelle :

En partant d'informations données par l'utilisateur sur les murs, comment peut-on obtenir un agencement de ces murs afin qu'ils aient une cohérence architecturale ?

On peut la reformuler de la façon suivante : *Comment peut-on obtenir un agencement des murs qui vérifie des propriétés en fonction des données entrées par l'utilisateur ?* C'est pourquoi on cherche

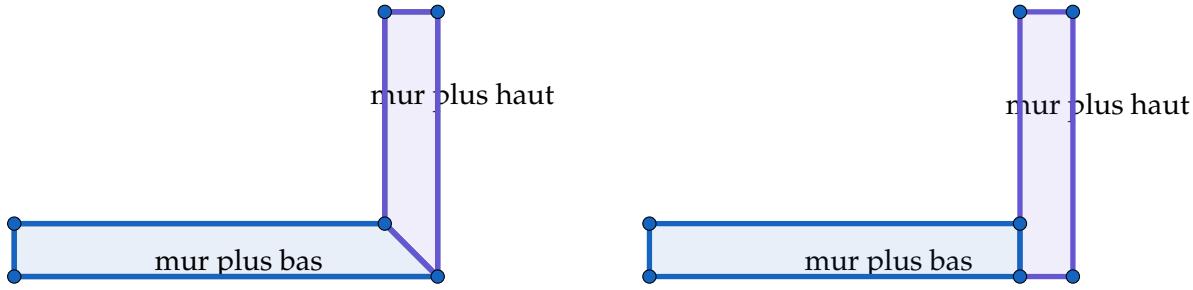


FIGURE 1.2 – A gauche l'implémentation naïve de la hauteur des murs en utilisant l'ancien algorithme, à droite le résultat attendu.

une fonction qui vérifie des propriétés allant de « *l'ensemble des données entrées par l'utilisateur* » vers « *l'ensemble des murs* ». Dans un premier temps, efforçons nous de mieux définir ces ensembles, dans un second temps concentrions nous sur la fonction elle-même.

Afin de définir « *l'ensemble des données entrées par l'utilisateur* », rappelons les actions effectuées par l'utilisateur pour définir un mur.

- L'utilisateur pose son doigt sur ce qui deviendra la première extrémité du mur (le premier KPoint).
 - Il glisse son doigt jusqu'à le relâcher, là où sera l'autre extrémité du mur (le second KPoint).
 - L'utilisateur peut aussi modifier la hauteur et la largeur du mur.

Ainsi, on en déduit que « l'ensemble des données entrées par l'utilisateur » correspond à :

$\mathcal{M} = \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R} \times \mathbb{R}$

Largeur

Hauteur

Points d'extrémités

Maintenant, définissons plus rigoureusement « *l'ensemble des murs* ». Remarquons que nous pouvons étudier notre mur dans le plan, sachant uniquement sa hauteur. De fait, sa hauteur ne varie pas dans l'espace, en pratique il s'agit uniquement de l'extrusion d'un polygone. Ainsi on définit « *l'ensemble des murs* » comme les polygones à 6¹ sommets avec une hauteur, c'est à dire :

$$\mathcal{P} = \mathbb{R}^{12} \times \mathbb{R}$$

Cependant, on souhaite étudier une fonction qui, à plusieurs entrées d'utilisateurs donne l'agencement entre plusieurs murs. Ainsi on s'intéresse aux produits de ces ensembles, de plus

¹. Nous reviendrons sur le choix de 6 dans la section 1.5.3.



FIGURE 1.3 – Partant du vecteur \overrightarrow{OM} dans le plan, on utilise l’algorithme d’orthonormalisation de Gram-Schmidt on définit un repère orthonormée $(O, \frac{\overrightarrow{OM}}{\|\overrightarrow{OM}\|}, \overrightarrow{OX})$, où \overrightarrow{OX} est un vecteur orthonormé orthogonal à \overrightarrow{OM} allant de l’origine O à un point X à une unité sur cet axe.

pour simplifier les notations on note pour $n \in \mathbb{N}$:

$$\mathcal{M}^n = \prod_{i=1}^n \mathcal{M} \quad \text{et} \quad \mathcal{P}^n = \prod_{i=1}^n \mathcal{P}$$

De ce fait, on recherche des fonctions allant de \mathcal{M}^n vers \mathcal{P}^n .

1.3.1 La fonction de génération naïve : genframe

Donnons un premier exemple, si on souhaite construire une représentation naïve du mur en 3D :

$$\text{genframe} : \mathcal{M}^1 \rightarrow \mathcal{P}^1 \tag{1.1}$$

$$(O, M, w, h) \mapsto \text{genframe}(O, M, w, h) = (P, h) \tag{1.2}$$

genframe est définie par la construction suivante :

- Partant du vecteur \overrightarrow{OM} dans le plan engendré par deux vecteurs de la base canonique, en utilisant le produit vectoriel entre le vecteur \overrightarrow{OM} et le dernier vecteur de la base canonique, on définit un nouveau vecteur \overrightarrow{OX} , orthogonal à \overrightarrow{OM} dans le plan. (cf. figure 1.3)
- On définit le point W comme w fois le vecteur \overrightarrow{OX} sur cet axe. De plus on définit aussi W' son symétrique par rapport à l’axe \overrightarrow{OM} . (cf. figure 1.4)
- Enfin définissons le point Z (resp. Z') comme la projection orthogonale du point M sur la droite parallèle à \overrightarrow{OM} passant par le point W (resp. W'). (cf. figure 1.5)

On définit P comme le polygone (W, W', Z, Z') .

On peut généraliser pour $n \in \mathbb{N}$, en définissant :

$$\text{genframe}^n : \mathcal{M}^n \rightarrow \mathcal{P}^n \tag{1.3}$$

$$(O_i, M_i, w_i, h_i)_{1 \leq i \leq n} \mapsto \text{gen frame}(O_i, M_i, w_i, h_i)_{1 \leq i \leq n} \tag{1.4}$$



FIGURE 1.4 – On définit le point W comme w (ici $w = 0.5$) fois le vecteur \overrightarrow{OX} sur cet axe. De plus on définit aussi W' son symétrique par rapport à l'axe \overrightarrow{OM} .

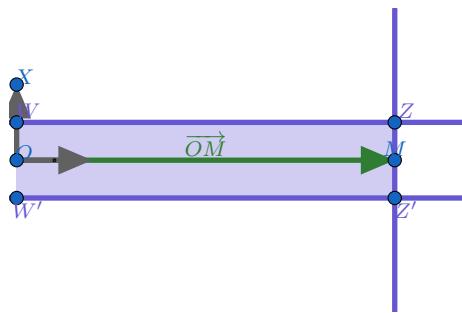


FIGURE 1.5 – Enfin définissons le point Z (resp. Z') comme la projection orthogonale du point M sur la droite parallèle à \overrightarrow{OM} passant par le point W (resp. W').

Remarque 1.3.1. Cette fonction est utilisée en pratique dans Keyplan 3D. Elle génère ce qu'on appelle la « frame » du mur. Celle-ci donne une géométrie primaire sur laquelle se base le reste du processus.

Avant de continuer notre raisonnement, faisons une digression sur les différentes intersections possibles entre deux murs, pour réduire l'ensemble de fonctions à étudier.

1.3.2 Classification des différentes intersections entre les murs

Il existe 6 cas d'intersections entre deux murs. Détailons ces cas représentés sur la figure 1.6 :

1. Les deux murs n'ont aucun points de contact.
2. Les murs ont un point de contact qui n'est pas aux extrémités pour l'un d'eux. Dans ce cas le mur vert est divisé au point de contact, pour créer un 3ème mur.
3. Le mur bleu est complètement compris dans le mur vert, celui-ci est donc ignoré.
4. Le mur bleu et le mur vert se superposent en partie. Dans ce cas, la zone de superposition devient un 3ème mur, et les deux autres sont réduits à leur point de contact avec celui-ci.
5. Les deux murs ont un point de contact aux extrémités.
6. Les deux murs s'intersectent en un point d'intersection. Keyplan 3D n'autorise pas ce cas.

Remarquons tout d'abord, que le fait que le cas (6) soit interdit en pratique, nous permet de déduire que l'ajout d'un 3ème mur ne change rien dans cette classification, il suffira de traiter les cas deux à deux. De plus, si on traite les cas (2),(3) et (4) en amont du programme toutes les intersections reviendront au cas (1) ou au cas (5).

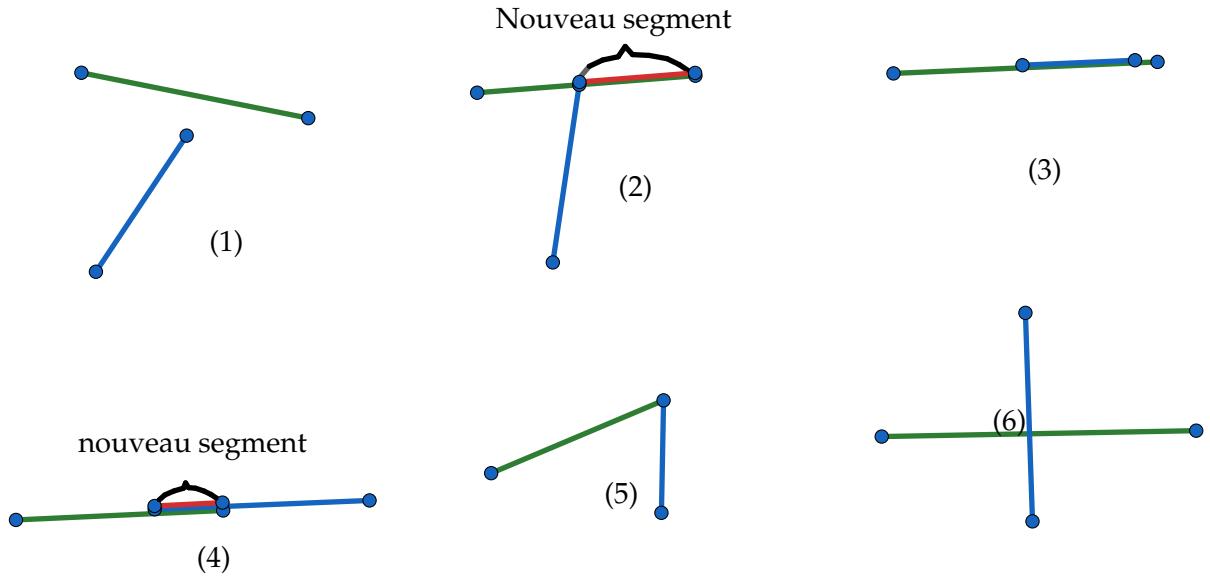


FIGURE 1.6 – Classification des différentes intersections entre deux murs.

1.3.3 Les fonctions n-maçon

A l'aide des deux sections précédentes, nous réduisons le nombres de fonctions à étudier. En effet, nous avons vu dans la section précédente qu'on peut réduire les intersections entre murs à deux cas :

- Les murs ne sont pas en contact.
- Les murs sont en contact par leurs extrémités.

Commençons par étudier le premier cas des murs sans contact. Dans ce cas, il suffit de générer la frame du mur en utilisant la fonction genframe^n définie dans la section 1.3.1. Ce cas est donc trivial et l'ajout de la hauteur ne rajoute aucune difficulté à priori. Ainsi, la difficulté provient des jonctions entre murs.

De ce fait, on se concentre sur l'étude des murs ayant en contact leurs extrémités. Considérons le sous ensemble de \mathcal{M}^n suivant :

$$\mathcal{M}_2^n = \{(M_i)_{1 \leq i \leq n} \in \mathcal{M}^n \mid \forall 1 \leq i, j \leq n \text{ distinct } M_i \text{ et } M_j \text{ possèdent une unique extrémité en commun}\} \quad (1.5)$$

Ainsi, on peut se restreindre à l'ensemble des fonctions suivantes :

Définition 1.3.1. Soit $n \in \mathbb{N}^*$, on appelle **n-maçon** une fonction allant de \mathcal{M}_2^n vers \mathcal{P} .

Dans la suite, on cherche à définir des propriétés satisfaisantes pour les **n-maçon**.

1.4 2-maçon

Dans cette section, nous définissons les règles, que le cas particulier des **2-maçon** doit vérifier et qui sera implémenter en pratique. Ces règles ont été réfléchies puis définies de manière

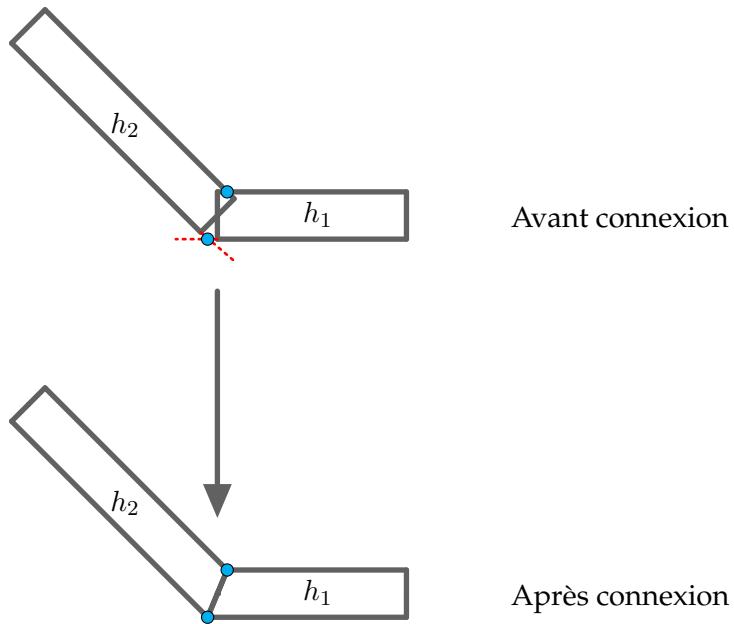


FIGURE 1.7 – Connexion entre deux murs de même taille. ($h_1 = h_2$)

arbitraire par M.Boisseron et moi-même, par ce qui nous semblait le plus à même de représenter la réalité architecturale ainsi que les désirs des utilisateurs.

Plus précisément, nous faisons une classification des différents cas possibles. Un résumé de cette classification peut être vu dans la dernière page de cette section.

1.4.1 Deux murs de même hauteur

Tout d'abord, étudions le cas des murs de même hauteur. Une première règle assez évidente vient du souhait qu'aucune construction fait précédemment ne doit changer. C'est à dire que le nouvel algorithme ne doit pas agir de manière rétroactive. Ainsi on conserve l'ancien algorithme, pour lequel on distingue deux cas :

- D'une part, le cas illustré par la figure 1.7, correspond à deux murs non colinéaires. Ici, les murs vont se lier par les intersections entre les murs consécutifs.
- D'autre part, supposons qu'on souhaite connecter deux murs colinéaires. Dans ce cas, le mur le plus fin se repose sur le mur le plus épais. En terme algorithmique aucun calcul n'est nécessaire comme le montre la figure 1.8.

1.4.2 Deux murs de hauteurs différentes

Pour des murs de différentes hauteurs, nous sommes partis d'une règle simple que nous avons adaptée au fur et à mesure : *pour deux murs de hauteurs différentes, on prolonge le mur le plus haut, afin que le mur le plus bas puisse se reposer sur celui-ci.*

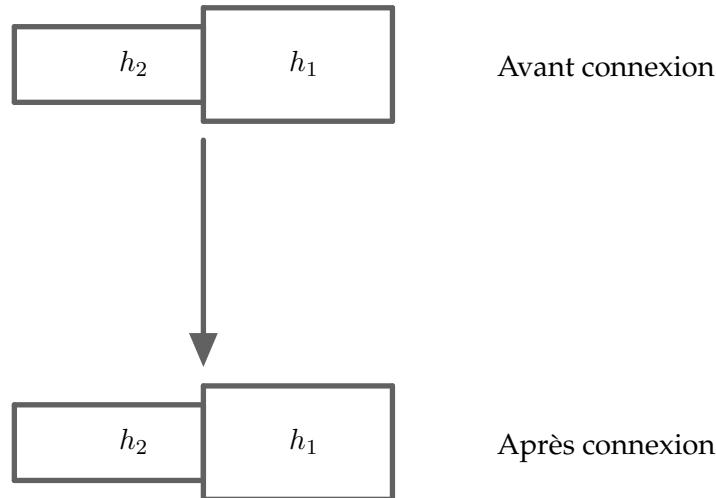


FIGURE 1.8 – Connexion entre deux murs colinéaires de même taille.

Remarque 1.4.1. Le terme *prolonger* peut sembler un peu flou, mais en pratique c'est assez simple, comme l'illustre la figure 1.12.

Nous appliquons cette règle dans la majorité des cas. Cependant on peut distinguer deux autres cas où cette règle n'est pas rigoureusement appliquée :

- Tout d'abord, si deux murs sont colinéaires, on le traite comme dans le cas de deux murs de même hauteur. (cf. figure 1.8).
- Ensuite, après avoir implémenté la règle ci-dessus, nous avons constaté que lorsque les murs étaient pratiquement colinéaires, le prolongement devenait aberrant visuellement (cf. figure 1.9). Ainsi nous avons défini une nouvelle règle : *si le mur le plus haut doit être prolongé plus de deux fois sa largeur, on ne le prolonge pas et on dit que le mur le plus bas se repose sur son côté*. Alors, au lieu d'obtenir la figure 1.9, on obtient la figure 1.10.
- Enfin dans tous les autres cas, on utilise la règle du prolongement décrite précédemment, comme l'illustre la figure 1.12.



FIGURE 1.9 – Connexion entre deux murs de tailles différentes pratiquement colinéaires. ($h_1 > h_2$)

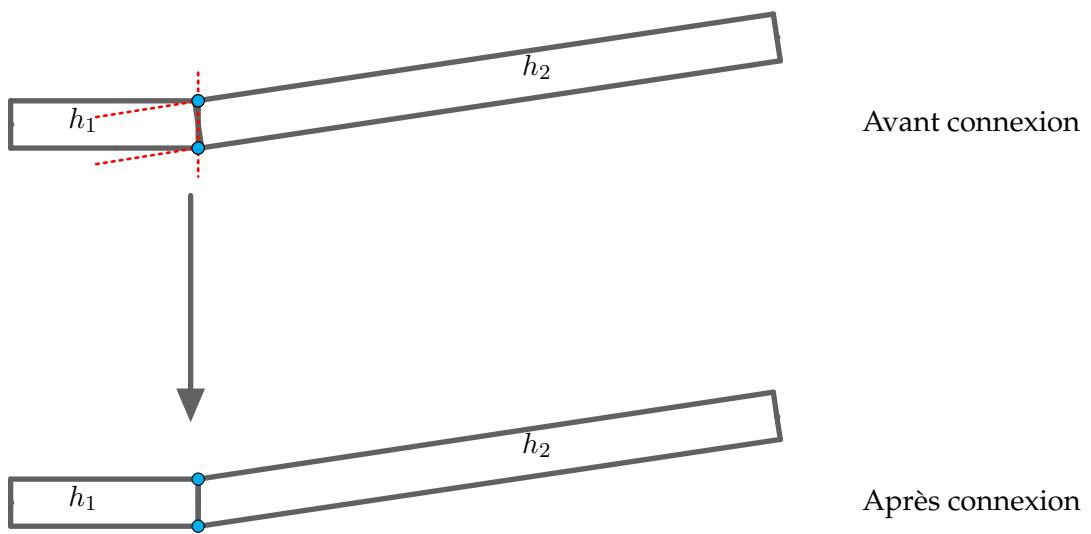


FIGURE 1.10 – Connexion entre deux murs de tailles différentes pratiquement colinéaires. ($h_1 > h_2$)

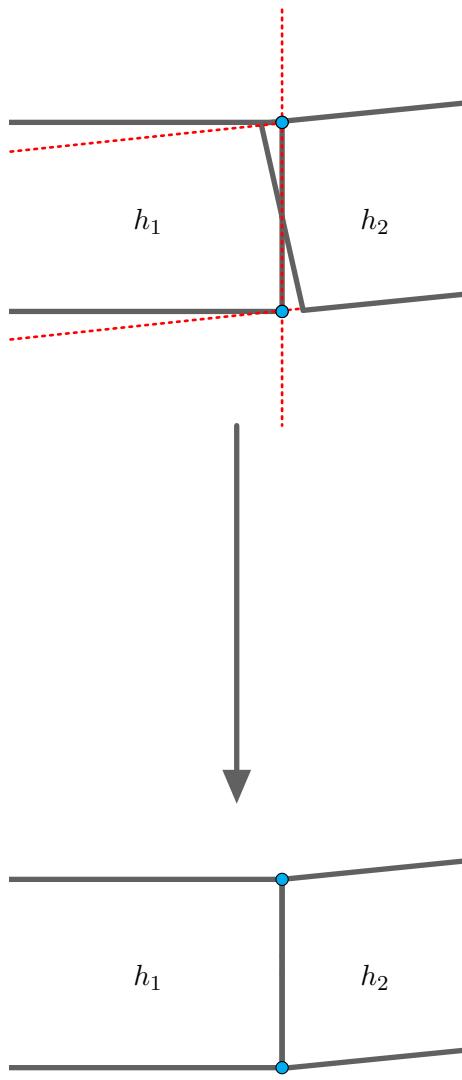


FIGURE 1.11 – Zoom - Connexions entre deux murs de tailles différentes, si le mur est trop proche d'être colinéaire. ($h_1 > h_2$)

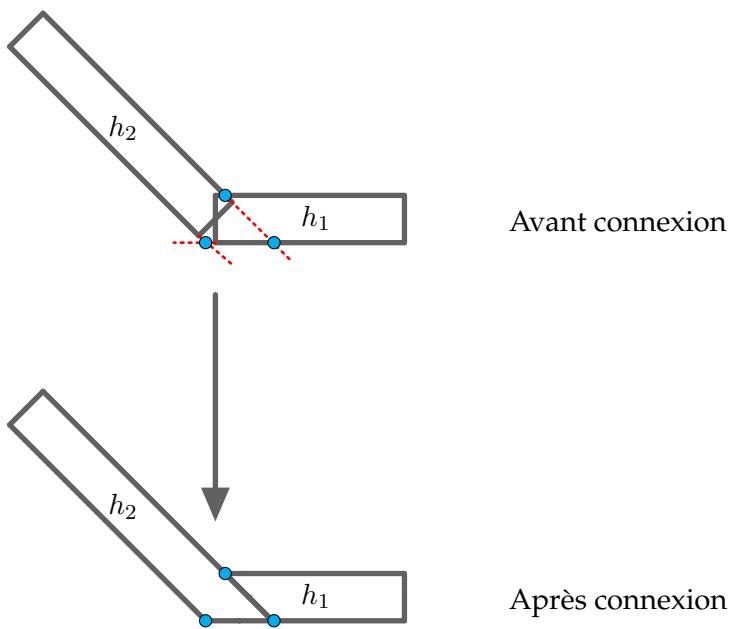


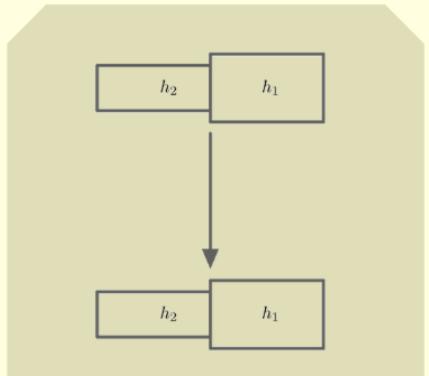
FIGURE 1.12 – Connexions entre deux murs de tailles différentes dans le cas d'un prolongement.
($h_1 > h_2$)

Classification des connexions entre deux murs

Soit deux murs de hauteurs h_1 et h_2 , on distingue tout d'abord deux cas :

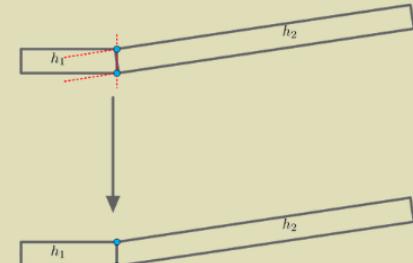
$$h_1 = h_2$$

$$h_1 > h_2$$

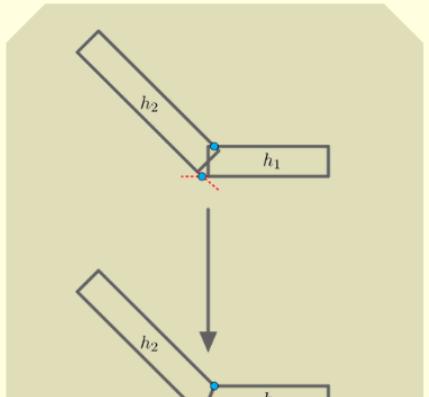


Colinéaires

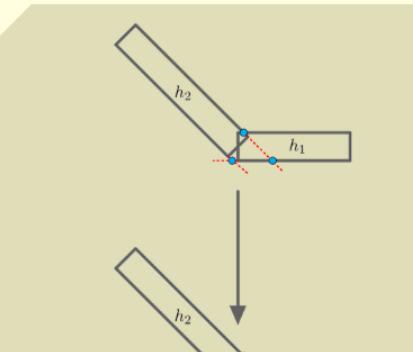
Colinéaires



Pratiquement colinéaires



Non colinéaires



Sinon prolongement

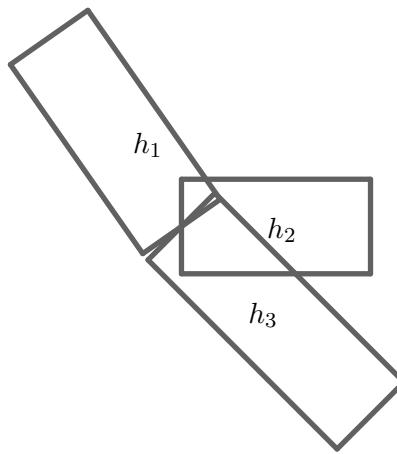


FIGURE 1.13 – Connexion entre 3 murs de hauteurs différentes $h_1 > h_2 = h_3$.

1.5 n-maçon

Dans cette section, nous introduisons les règles que le n-maçon doivent vérifier. Pour rappelle, le n-maçon représente une fonction qui connecte n murs étant en contact pour l'une de leurs extrémités, de même que pour le 2-maçon il s'agit de règles arbitraires.

1.5.1 Partitionnement des murs

Une première problématique est de partitionner les murs afin d'appliquer à chacun d'eux les bonnes règles. Pour répondre à cette problématique, nous devons trouver une généralisation du principe que nous avons évoqué dans la section précédente : *ayant deux murs d'hauteurs différentes, le mur le plus bas se repose sur le mur le plus haut.*

Ce principe fonctionne parfaitement pour deux murs connectés, cependant il pose problème lorsqu'il y a plus de deux murs, on peut illustrer ces difficultés par la figure 1.13.

Commençons par traiter le cas de la figure 1.13, ce qui paraît assez naturel serait que le mur de hauteur h_2 , se repose sur les murs h_1 et h_3 , bien que l'un d'eux soit de même hauteur. Ainsi, on traiterait d'abord le mur d'hauteur h_1 , le mur d'hauteur h_3 , et enfin le mur d'hauteur h_2 . De ce fait, l'ordonnancement des murs ne semblerait pas être uniquement une affaire d'hauteur mais aussi d'**angle**.

On peut avoir l'impression que le mur de hauteur h_2 se repose sur le couple de mur de hauteur (h_1, h_3) ; cela vient du fait que le mur de hauteur h_3 est proche du prolongement du mur h_1 (cf. figure 1.14).

Définition 1.5.1. *On définit la fonction qui donne l'angle entre deux murs :*

$$\begin{aligned} \theta : \mathcal{M}_2^2 &\rightarrow [0, 2\pi] \\ ((O, M_1, w_1, h_1), (O, M_2, w_2, h_2)) &\mapsto \min(\widehat{M_1OM_2}, \widehat{M_2OM_1}) \end{aligned}$$

Ainsi, on donne la définition du terme « proche du prolongement », comme un problème

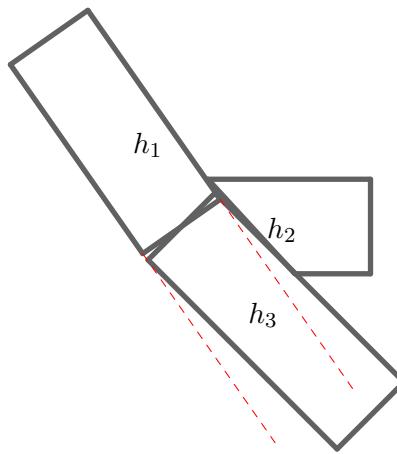


FIGURE 1.14 – Le mur de hauteur h_3 est quasiment dans le prolongement (en rouge ici) du mur de hauteur h_1 . $h_1 > h_2 > h_3$

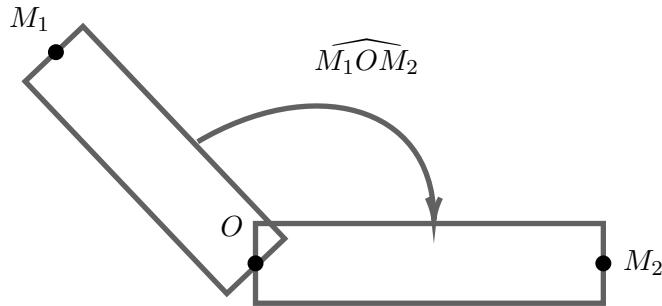


FIGURE 1.15 – Exemple de calcul d’angle

d’optimisation d’angle :

Définition 1.5.2. Soit $(m_1, m_2, \dots, m_n) \in \mathcal{M}_2^n$ n murs, on dit que m_i est proche du prolongement de m_1 , s’il résout le problème suivant :

$$m_i = \min_{m_j, 2 \leq j \leq n} |180 - \theta(m_1, m_j)| \quad (1.6)$$

Donnons un second exemple un peu plus sophistiqué avec la figure 1.16. Dans celle-ci, nous avons 4 murs, dans l’ordre décroissant un mur de hauteur h_1 , deux murs de hauteur h_2 et h_3 , et un mur plus bas de hauteur h_4 . Bien que le mur de hauteur h_4 vérifie l’équation 1.6, il est plus petit que les murs de hauteur h_2 et h_3 , ainsi cela contredit la règle d’un mur plus bas se repose sur un mur plus haut. C’est pourquoi, lorsque l’on recherche les murs les plus proches du prolongement du mur de hauteur h_1 , on considère uniquement les murs les plus proches en taille de celui-ci. Dans notre cas on considère uniquement les murs de même hauteur h_2 et h_3 et ce sera le mur de hauteur h_2 qui vérifierait l’équation 1.6.

De ce fait, on en déduit la règle suivante : Soit $(m_1, m_2, \dots, m_n) \in \mathcal{M}_2^n$ n murs avec $n > 2$. On suppose que m_1 est strictement le mur le plus haut, de plus on considère m_i le mur le plus proche du prolongement de m_1 parmi les murs de la deuxième hauteur la plus grande. Alors quel que soit j

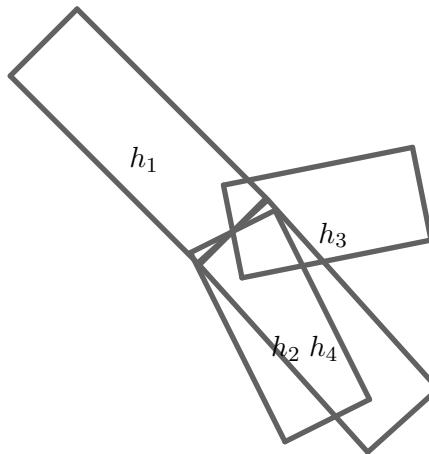


FIGURE 1.16 – Connexion entre 4 murs de hauteurs différentes $h_1 > h_2 = h_3 > h_4$.

différent de 1 et i , le mur m_j se repose sur le couple (m_1, m_i) .

Ce qui nous permet de donner les définitions suivantes :

Définition 1.5.3. En reprenant les notations et les hypothèses de la règle ci-dessus, on appelle :

- **Murs principaux** les murs m_1 et m_i , car ils sont la base sur laquelle va se reposer tout les autres murs.
- **Murs accessoires** les murs m_j avec j différent de 1 et i , car ils se reposent sur les murs principaux.

Ainsi lorsque l'on distingue les murs principaux et accessoires, on partitionne l'ensemble des murs $(m_1, m_2, \dots, m_n) \in \mathcal{M}_2^n$. Nous avons supposé que le mur m_1 est **strictement** plus haut que les autres murs, néanmoins ce n'est pas forcément vrai. Il est nécessaire de considérer le cas où il existe k murs de même taille plus grand que les autres. Dans ce cas, on définit les murs principaux comme ces k murs, et les autres murs comme les murs accessoires.

Définition 1.5.4 (Partitionnement général des murs). Soit $(m_1, m_2, \dots, m_n) \in \mathcal{M}_2^n$, on distingue deux cas :

1. Soit, il existe un unique mur plus haut m_1 (sans perte de généralité à une permutation près), dans ce cas les murs principaux sont définis comme le couple (m_1, m_i) où m_i vérifie le problème d'optimisation 1.6, et les autres murs sont les murs accessoires.
2. Soit, il existe k murs de même taille plus haut, dans ce cas les murs principaux sont les k murs de même taille, et les autres murs sont les murs accessoires.

Finalement avec ce partitionnement, on a découpé le problème de la construction d'un bon n-maçon en deux plus petits problèmes :

- Connecter les murs principaux entre eux.
- Connecter les murs accessoires entre eux sur les murs principaux.

Nous les traiterons dans la suite.

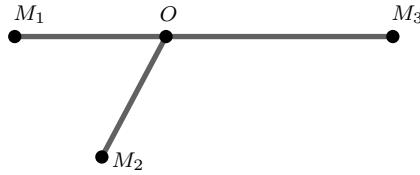


FIGURE 1.17 – On part de trois murs $m_i = (O, M_i, w_i, h)$ pour $1 \leq i \leq 3$

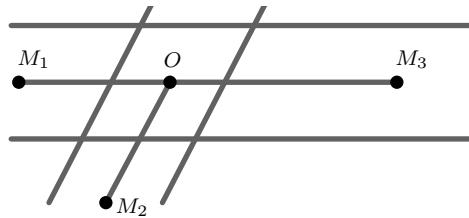


FIGURE 1.18 – Comme pour la construction de la fonction genframe (ie. section 1.3.1) on trace les parallèles pour définir l'épaisseur.

1.5.2 Connexion des murs principaux

Comme nous l'avons expliqué dans la définition 1.5.4 il existe deux types de partitionnement pour les murs principaux, nous étudions ces deux cas dans cette section :

1.5.2.1 Deux murs principaux

Pour deux murs principaux, nous allons pouvoir utiliser la fonction 2-maçon que nous avons construit dans la section 1.4.

1.5.2.2 Plus de deux murs principaux

Comme nous l'avons expliqué dans la définition 1.5.4, s'il y a plus de deux murs principaux, cela veut dire que les murs principaux ont tous la même hauteur. Dans ce cas on utilise l'ancien algorithme de connexion des murs. Un exemple pour 3 murs est donné par les figure 1.17, 1.18, et 1.19.

1.5.3 Connexion des murs accessoires

Les murs accessoires se reposent par définition sur les murs principaux, par exemple dans la figure 1.14. Nous avons constaté que le mur de hauteur h_3 se repose sur les deux murs principaux. Néanmoins, il convient de distinguer certains cas, où le mur accessoire se repose uniquement sur un seul des deux murs principaux (cf. figure 1.20 (1)) On peut s'interroger sur le fait que le mur m_3 sur la figure 1.20 se repose dans le cas (1) uniquement sur m_2 , et dans le cas (2) sur les murs m_1 et m_2 ?

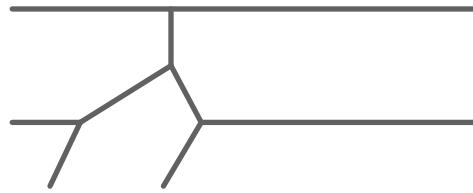


FIGURE 1.19 – Enfin on prend la première intersection de chaque faces consécutives.

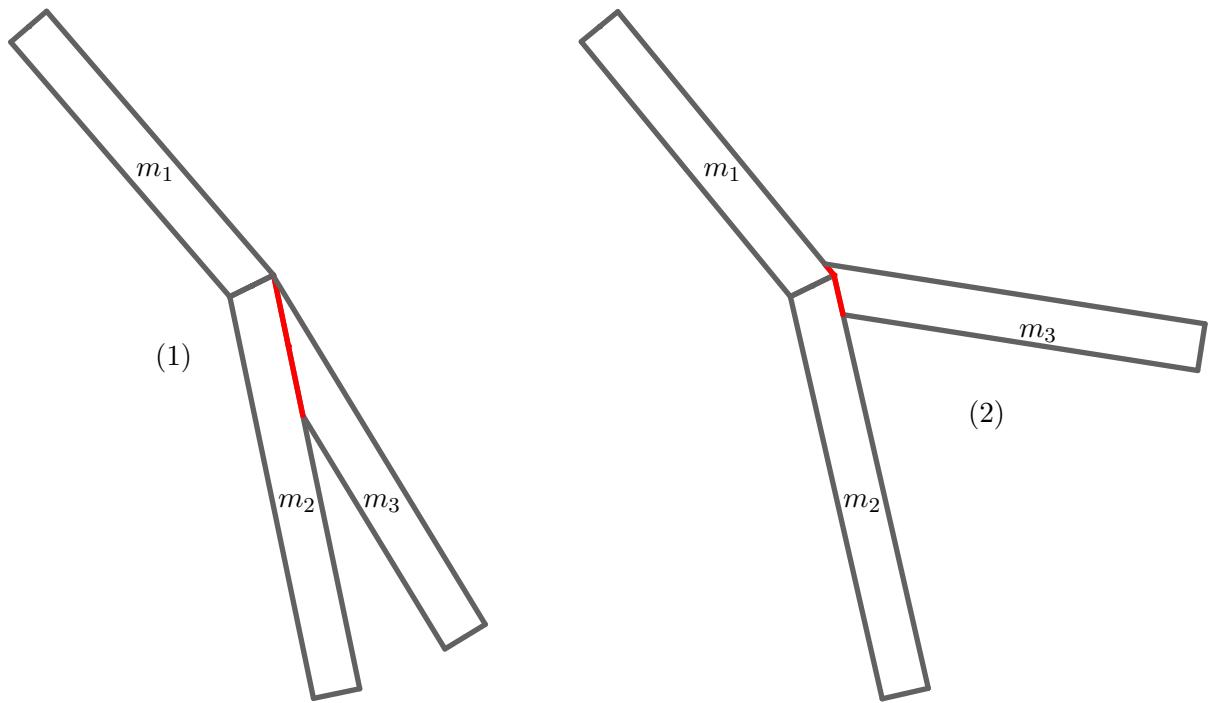


FIGURE 1.20 – A gauche (1) le mur m_3 se repose **uniquement** sur m_2 . A droite (2) le mur m_3 se repose sur m_1 et m_2 .

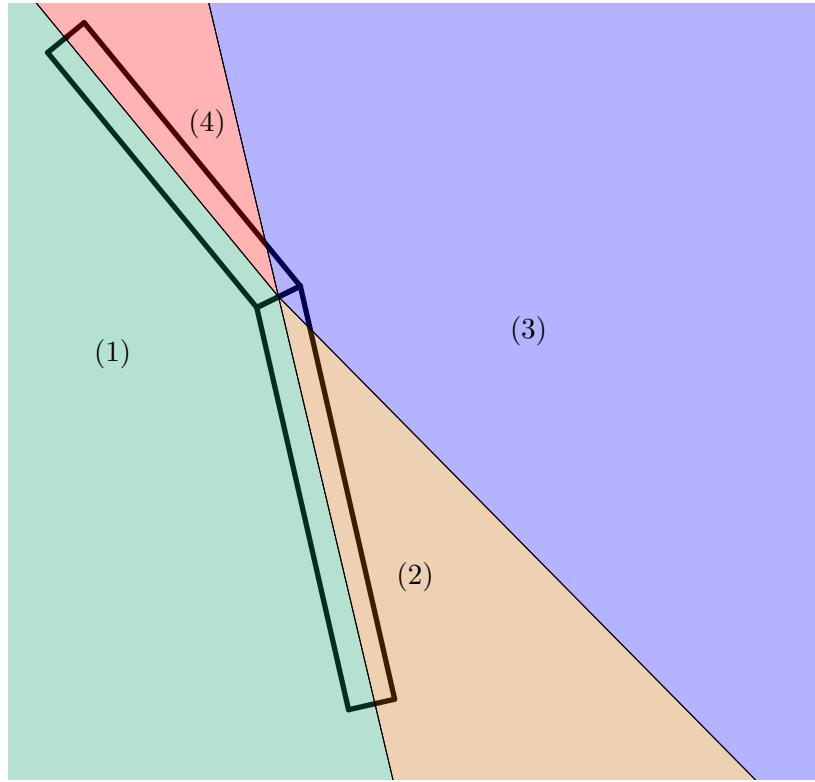


FIGURE 1.21 – L’intervalle $[0, \theta(m_1, m_2)]$, dans ce cas le mur accessoire m_3 se repose sur m_1 et m_2 (1). L’intervalle $[\theta(m_1, m_2), 180]$, dans ce cas le mur accessoire m_3 se repose uniquement sur m_2 (2). L’intervalle $[180, \theta(m_1, m_2) + 180]$, dans ce cas le mur accessoire m_3 se repose sur m_1 et m_2 (3). L’intervalle $[\theta(m_1, m_2) + 180, 360]$, dans ce cas le mur accessoire m_3 se repose uniquement sur m_1 (4).

En testant avec plusieurs murs accessoires ayant différents angles (par rapport au mur m_1), on remarque qu’entre certains angles, le mur m_3 se repose sur un seul des deux murs principaux. Plus précisément, on a la proposition suivante :

Propriété 1.5.1. Soit deux murs principaux $(m_1, m_2) = ((O, M_1, w_1, h_1), (O, M_2, w_2, h_2)) \in \mathcal{M}_2^2$. Alors quelque soit $m_3 \in \mathcal{M}_2$ un mur accessoire que l’on rajoute, il existe un unique partitionnement de l’ensemble des angles avec le mur m_1 (cf. définition 1.5.1) permettant de connaître les murs sur lesquels m_3 se repose :

1. L’intervalle $[0, \theta(m_1, m_2)]$, dans ce cas le mur accessoire m_3 se repose sur m_1 et m_2 .
2. L’intervalle $[\theta(m_1, m_2), 180]$, dans ce cas le mur accessoire m_3 se repose uniquement sur m_2 . On appelle cette intervalle **la première zone limite**.
3. L’intervalle $[180, \theta(m_1, m_2) + 180]$, dans ce cas le mur accessoire m_3 se repose sur m_1 et m_2 .
4. L’intervalle $[\theta(m_1, m_2) + 180, 360]$, dans ce cas le mur accessoire m_3 se repose uniquement sur m_1 . On appelle cette intervalle **la seconde zone limite**.

On peut voir un exemple de partitionnement dans la 1.21.

Remarquons que la première et la seconde zone limites peuvent être réduites à l’ensemble vide, dans le cas $\theta(m_1, m_2) = 180$. De plus, on a considéré le cas de deux murs principaux, mais on peut généraliser le propos pour $(m_1, m_2, \dots, m_n) \in \mathcal{M}_2^n$ murs, en étudiant uniquement le couple de murs (m_1, m_k) où k vérifie le problème suivant :

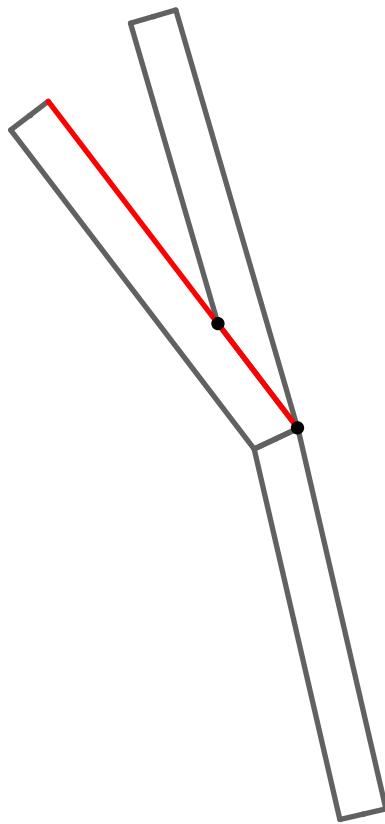


FIGURE 1.22 – Le mur accessoire est dans la première zone limite. De ce fait, on calcule les intersections avec la face consécutive en rouge.

$$\max_i(\theta(m_1, m_i)) \quad (1.7)$$

Ainsi, une fois que l'on a connecté un mur accessoire, on le rajoute à la liste des murs principaux.

On en déduit une façon de connecter les murs accessoires par disjonction des cas énoncés ci-dessus, que l'on résume dans les figures 1.22 à 1.25. On remarque qu'un troisième point d'intersection est nécessaire pour définir le mur dans les figures 1.23 et 1.25. Nous avons nommé ce point le **middle point**, dans les autres cas, il est uniquement défini comme le milieu des deux autres points d'intersection.

En conclusion de ce chapitre, nous avons donné une façon de construire un n-maçon, en partitionnant n murs en murs principaux/accessoires et en traitant ces murs différemment.

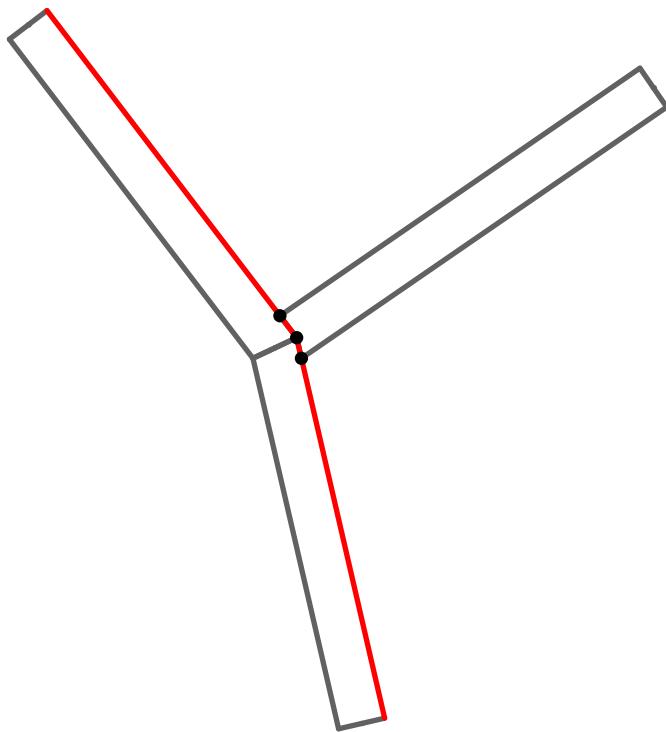


FIGURE 1.23 – Le mur accessoire se repose sur les deux murs principaux, sur les faces consécutives en rouges.

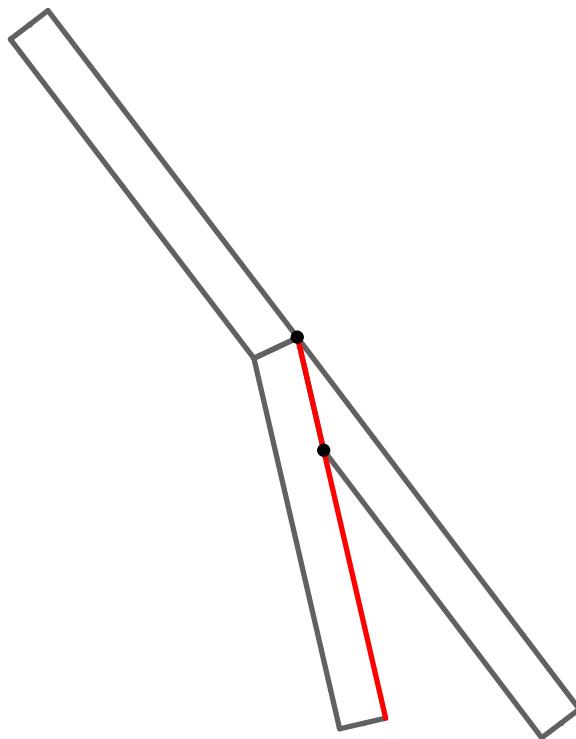


FIGURE 1.24 – Le mur accessoire est dans la seconde zone limite. De ce fait, on calcule les intersections avec la face consécutive en rouge.

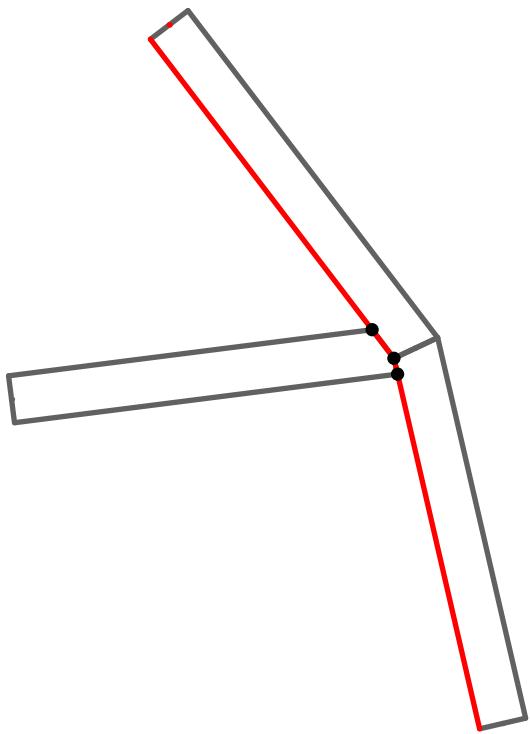
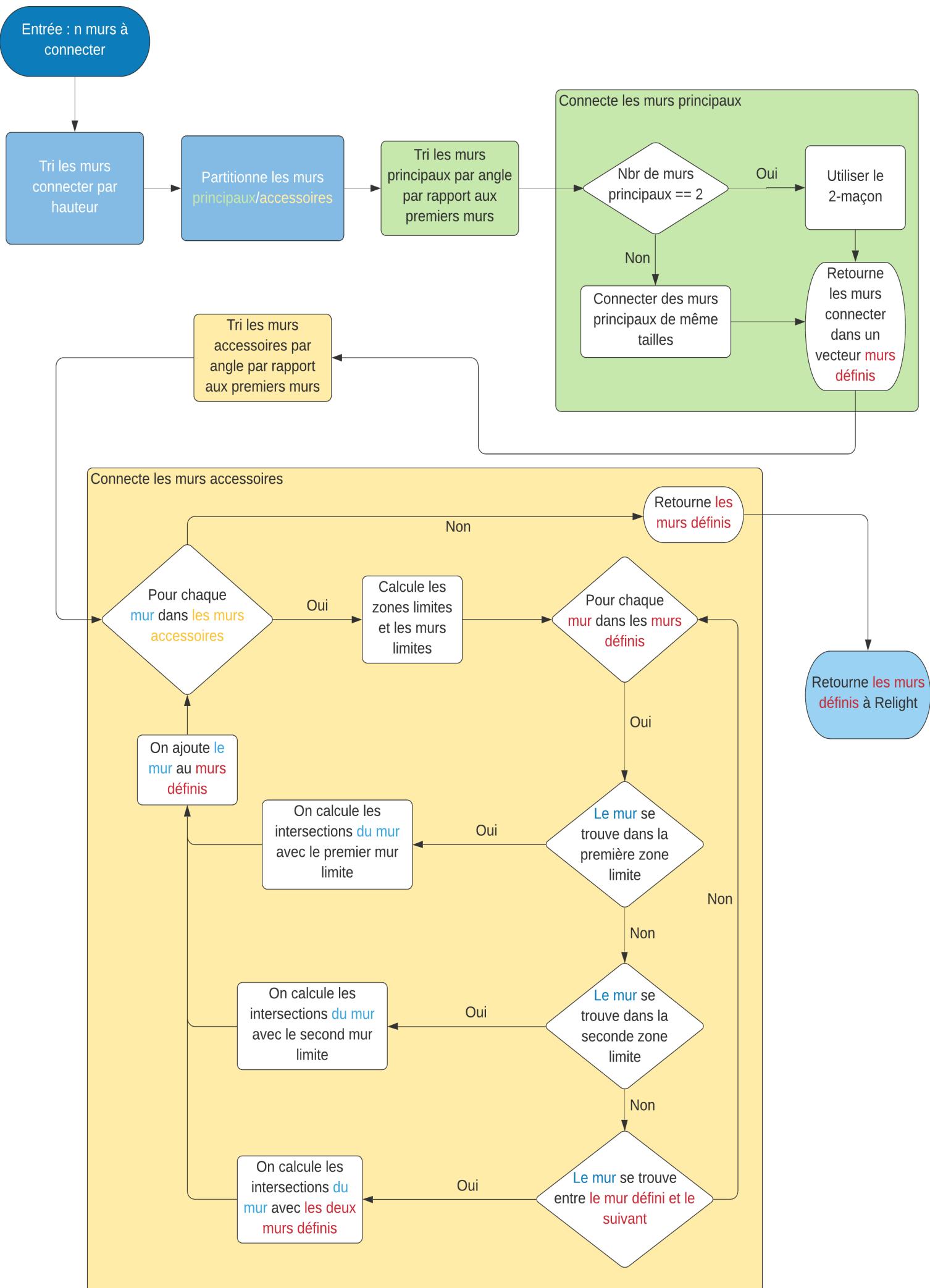


FIGURE 1.25 – Le mur accessoire se repose sur les deux murs principaux, sur les faces consécutives en rouge.



Chapitre 2

Implémentation du code

Dans ce chapitre, nous allons décrire l'implémentation de l'algorithme décrite dans le chapitre précédent.

2.1 Prototypage

Avant de rentrer « dans le dur » du code, il était nécessaire de prototyper l'algorithme, cela dans deux buts :

- Tester les algorithmes que l'on souhaite implémenter pour avoir un aperçu de leurs limites.
- Appréhender les difficultés de l'implémentation finale.

Ce prototypage a été fait en python sur un Notebook Colaboratory dans la grande tradition des TP du master CSMI. Nous avons utilisé les modules numpy, shapely pour la partie mathématiques, et matplotlib pour la partie graphique.

Le prototype a été développé pendant plusieurs semaines, jusqu'à ce que ses résultats soient concluant (comme l'illustre les figures 2.1-2.1), mais encore loin d'être parfait. Néanmoins, il est important de préciser que l'algorithme que nous avons prototypé ne correspondait pas complètement au n-maçon décrit dans le chapitre précédent. En effet, il s'agissait d'une première version dont des détails se trouvent ci-dessous. Ainsi, le prototypage a permis de voir des faiblesses qui ont été confirmées dans l'implémentation de cet algorithme. Celui-ci a été abandonné au bout de deux semaines au profit du n-maçon que nous avons traité dans le chapitre précédent.

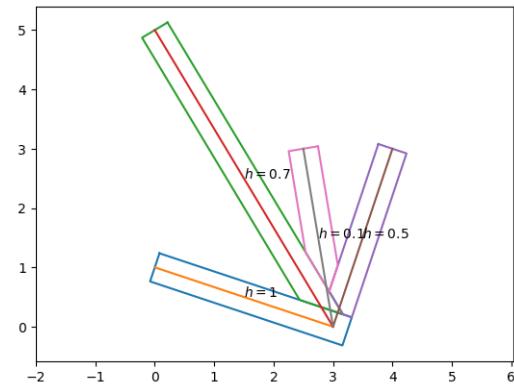
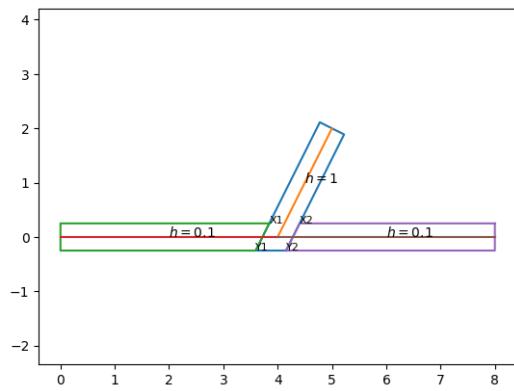
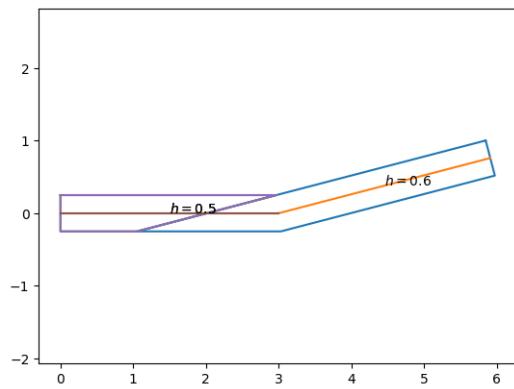
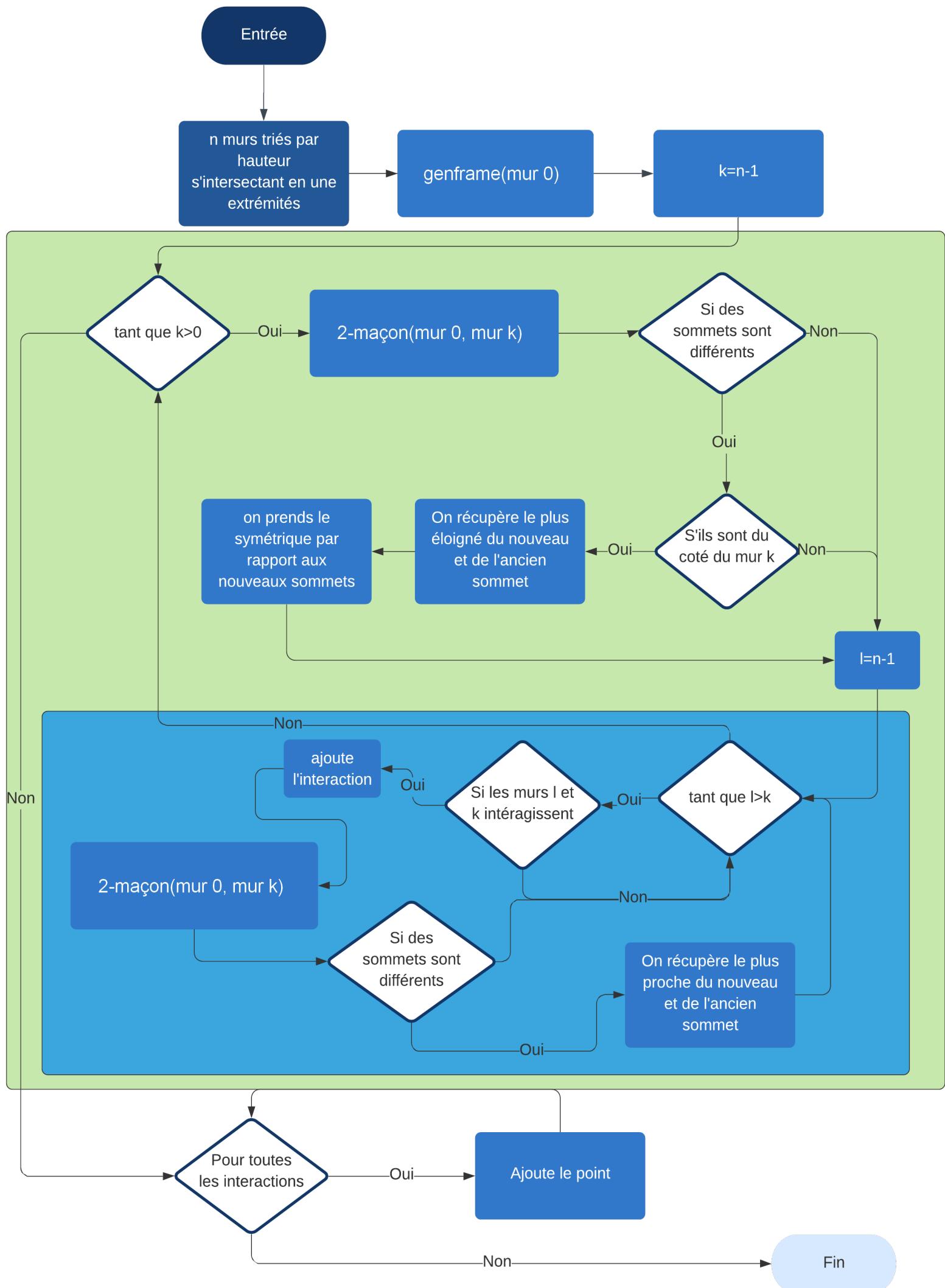


FIGURE 2.1 – Exemple après le prototypage



2.2 En pratique

Comme nous l'avons évoqué dans la section 1.1, Keyplan3D est divisé en trois parties Relight, Keyplan, Vault. En général, le travail s'effectue sur les classes KWall et KPoint dans Vault. Cependant, il a fallu ajouter de l'interface pour pouvoir paramétriser la hauteur et donc de développer en Objective-C sur Keyplan. Mais aussi, il a été nécessaire de travailler sur la partie moteur 3D dans le cadre d'ajout de booléen.

Dans cette section nous allons nous concentrer sur Vault qui, on le rappelle, définit l'architecture de Keyplan3D. Dans les grandes lignes Vault est constitué de :

- Des fondations qui définissent un plan, des murs, des pièces, etc...
 - Des opérateurs qui permettent de bouger les murs, les pièces, etc...
 - Des architectes qui orchestrent tout ces choses.

Vault est exclusivement développé en C++, en utilisant la librairie standard et la librairie mathématiques d'OpenGL : GLM¹.

De plus, un grand nombre de fonctions étaient déjà implémentées avec l'ancien programme, notamment des fonctions de géométrie pour calculer des intersections, mais aussi des fonctions pour ordonner des faces par rapport aux KPoint.

C'est pourquoi, il a fallu dans un premier temps appréhender le code qui régit les KPoint et les KWall. Ainsi nous avons développé un graphe des dépendances fonctionnelles, comme l'illustre la figure 2.2.

Les connexions se faisaient principalement dans la fonction `update_connected_Kwall_contour()`. De ce fait la plupart des ajouts et changements ont été opérés dans la suite de cette fonction.

Voici un exemple d'implémentation de la fonction qui partitionne (cf section 1.5.1).

```

1 void
2 KPoint::get_partition_mains_accessories_walls( const std :: vector<KWall*>&
n_walls_sorted , std :: vector<KWall*>& mains_walls , std :: vector<KWall*>&
accessories_walls ) const {
3
4     const auto nbr_of_walls = n_walls_sorted . size ();
5
6     if ( nbr_of_walls < 3) {
7         mains_walls = n_walls_sorted ;
8         accessories_walls . clear ();
9     }
10    else
11    {
12        const float main_height = n_walls_sorted[0] -> get_height ();
13
14        if ( main_height != n_walls_sorted[1] -> get_height () ) {
15            mains_walls = std :: vector<KWall*>(n_walls_sorted . begin () , n_walls_sorted
.begin () + 2 );
16            accessories_walls = std :: vector<KWall*>(n_walls_sorted . begin () + 2 ,
n_walls_sorted . end () );
17        }
18        else
19        {
20            std :: vector<KWall*>:: size_type k;
21            //case with main wall of the same height
22            for ( k = 2; k < nbr_of_walls ; k++ ) {
23                if ( n_walls_sorted [k] -> get_height () != main_height )

```

1. <https://glm.g-truc.net/0.9.9/index.html>

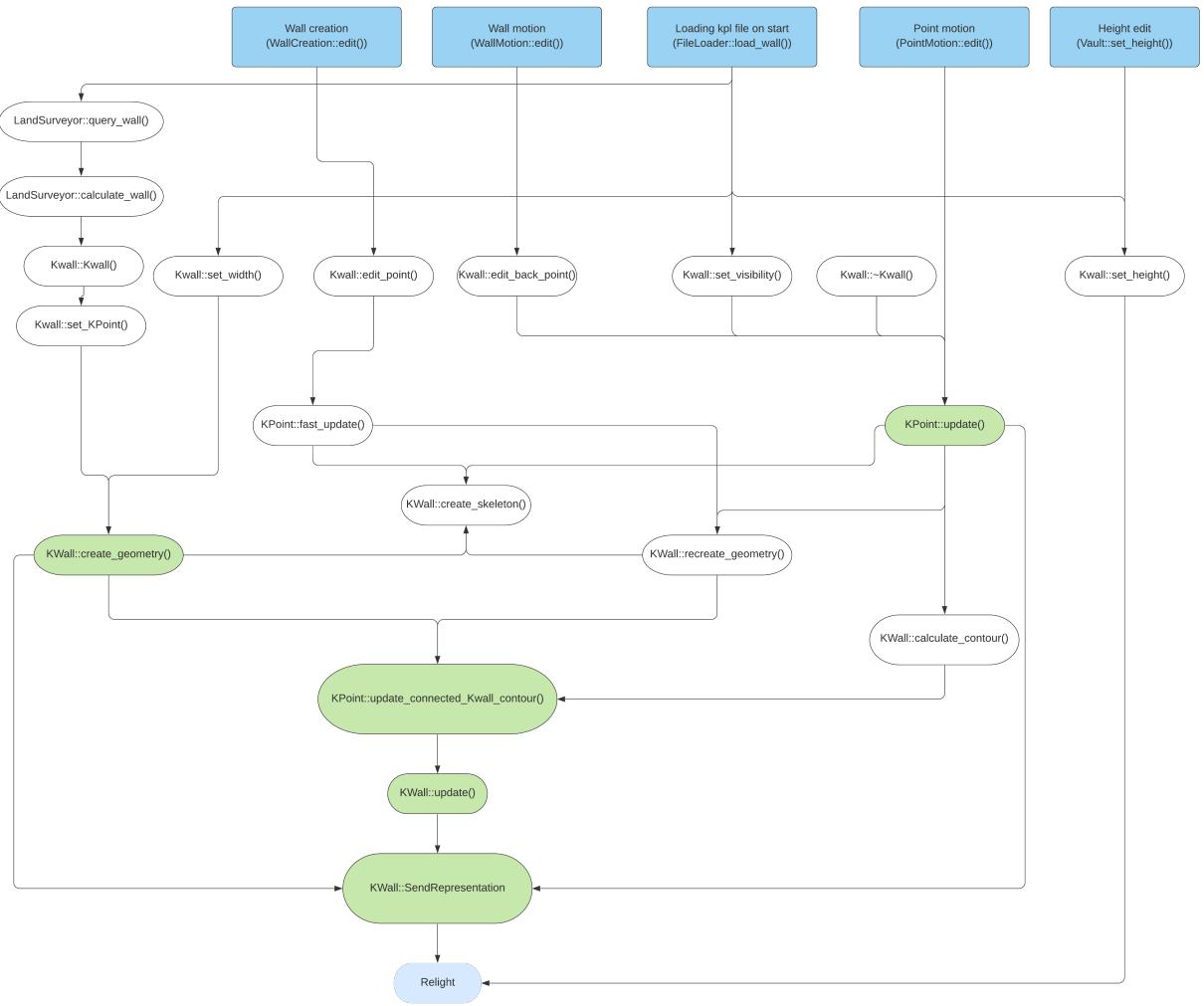


FIGURE 2.2 – Graphe des dépendances fonctionnelles de l'ancien programme pour gérer les connections entre murs

S



FIGURE 2.3 – La face (en bleu ici) du mur le plus bas qui intersecte le mur le plus haut n'a pas de sens physique.

```

24                     break;
25                 }
26
27             mains_walls = std::vector<KWall*>(n_walls_sorted.begin(), n_walls_sorted
28 .begin() + k);
29             accessories_walls = std::vector<KWall*>(n_walls_sorted.begin() + k,
30 n_walls_sorted.end());
31         }
31 }
```

De plus, il est important de préciser que le développement a été accompagné par les outils suivant :

- Un environnement de développement Xcode.
- Deux outils de gestion de version avec Gitlab et Fork.

2.3 Booléens

Au milieu du stage, nous avons fait face à un problème, celui-ci émane d'un soucis de cohérence de mesure. En effet, on obtient une face qui n'a pas de sens physique, dans le cas où d'un mur le plus haut qui se prolonge afin que le mur le plus bas se repose celui-ci (cf. section 1.4) comme l'illustre la figure 2.3.

De ce fait, il faut trouver un moyen de supprimer cette face. Plusieurs moyens sont imaginés, notamment une refonte complète de la notion de la géométrie du mur dans Keyplan3D. Néanmoins, la solution la plus simple fut d'utiliser des booléens. Nous rappelons qu'il s'agit de modèles 3D que nous allons « soustraire »² à un autre modèle 3D. Dans notre cas le modèle 3D en question ne sera qu'une surface rectangulaire.

Un système de booléens avait déjà été implémenté dans Keyplan3D comme le montre la figure 2.4. Il avait été développé par un ancien développeur sans aucune documentation. De plus, les informations des booléens étaient hardcodées dans les modèles booléens. De ce fait, il

2. au sens de supprimer l'intersection des deux modèles 3D.

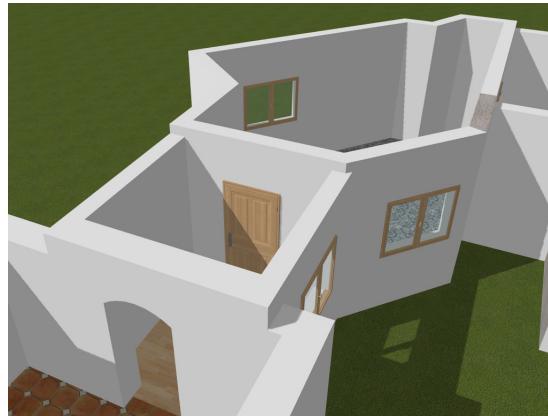


FIGURE 2.4 – Exemples de booléens : des portes, des fenêtres, etc...

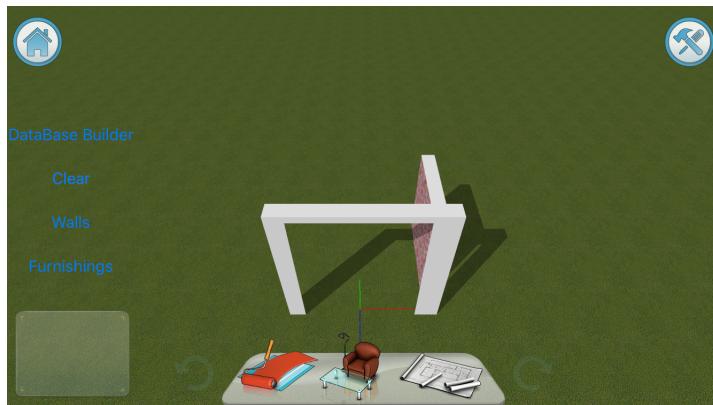


FIGURE 2.5 – Booléens sur les faces des murs après l’implémentation

a fallu apprendre comment ce système fonctionnait et finalement rajouter des fonctions spécifiques à l’ajout de ces murs.

2.4 Débogage

Le débogage a été une tâche récurrente durant le stage.

Pour s’habituer au moteur, notre première tâche était de corriger un bug découvert par un utilisateur, un exemple de ce bug est représenté sur la figure 2.6. Après des recherches, nous avons découvert et résolu ce bug qui provenait d’une boucle qui essayait d’accéder à des valeurs en dehors d’un vecteur de booléens.

Au démarrage de l’implémentation de l’algorithme décrit dans le chapitre précédent, nous avons constaté que certains cas n’étaient pas correctement traités. De plus, de nombreux bugs sont apparus.

Au départ nous avons utilisé les outils de debug standard de Xcode, et récupéré les valeurs des murs que nous avons dessinés sur papier. Toutefois, celui-ci n’était pas très convenable pour avoir un retour visuel et géométrique de l’évolution des murs. Aussi pour faciliter le débogage, nous avons mis en place une petite interface, mise à jour tout au long du stage pour afficher différentes valeurs de débogage.

Cette interface a été développée en python avec la classe `tkinter`, les graphiques sont



FIGURE 2.6 – Exemple du bug sur l’extrusion des trous d’un mur.

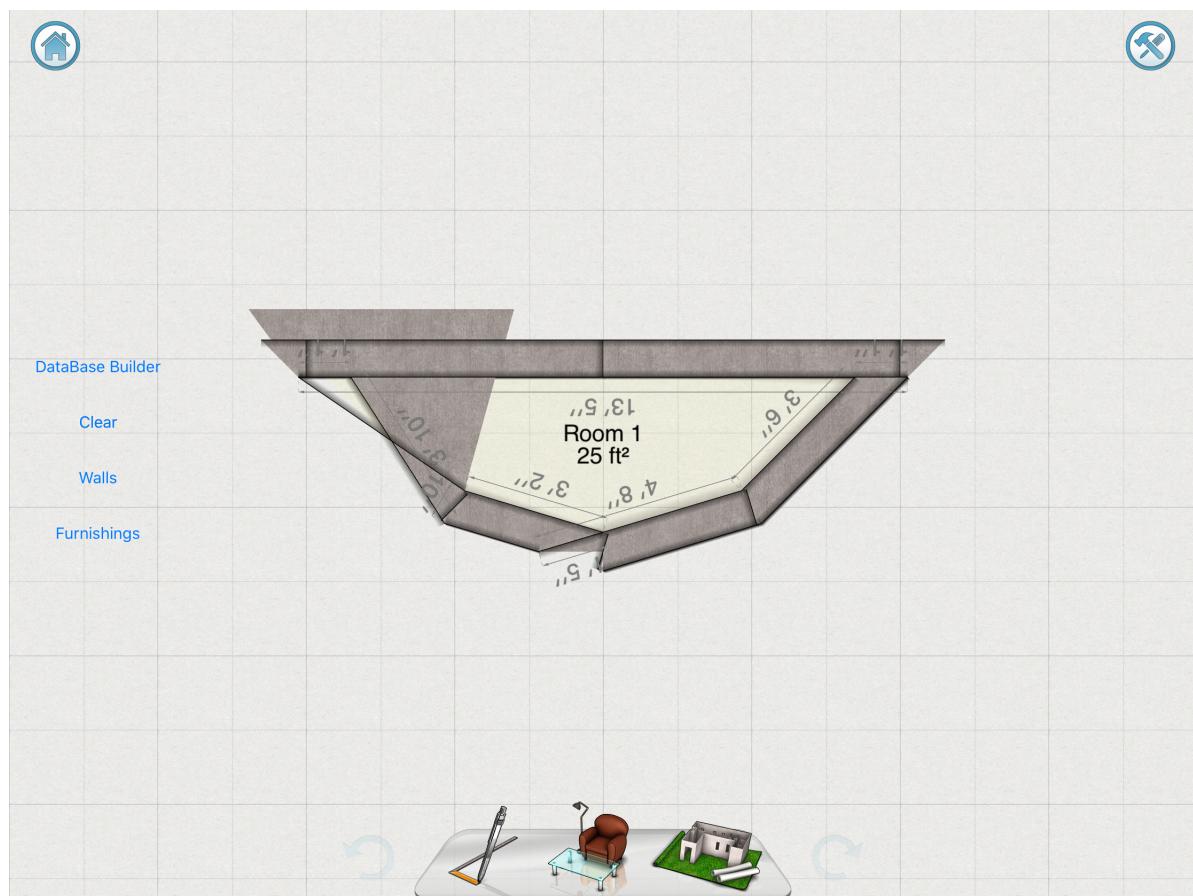


FIGURE 2.7 – Exemple de bug

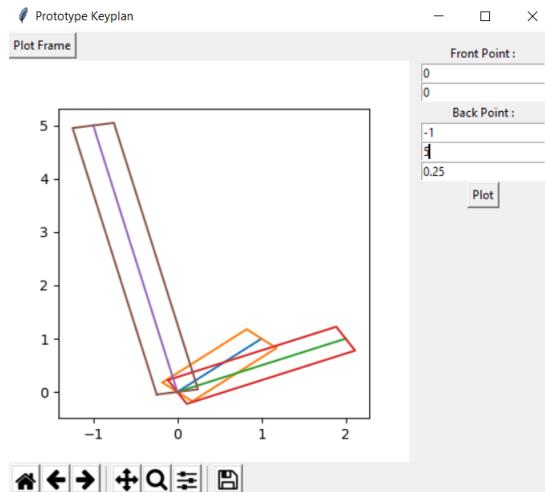


FIGURE 2.8 – v0.1

issus de la classe `matplotlib`. Comme il l'a été notifié ci-dessus, l'interface a eu plusieurs mises à jour :

- v0.1 une version qui permet d'afficher les frames des murs (cf. figure 2.8).
- v0.2 une version qui permet d'afficher les zones limites (cf. définition 1.5.1) en fonction des points des murs (cf. figure 2.9).
- v0.3 une version qui permet d'afficher des murs en fonction des points des faces (cf. figure 2.10).
- v0.4 une version qui simplifie l'entrée des points des murs qui ne se font plus un à un mais en rentrant un tableau `numpy` avec les coordonnées. Pour simplifier, plusieurs fonctions ont été définies dans `Keyplan` pour exporter directement un murs en tableau `numpy`.
- v0.5 permet d'afficher des tableaux 3D. Afin de visualiser les booléens appliqués aux murs.
- v0.6 permet d'exporter l'affichage dans le package `latex` : `tikz`. Cette version a permis de créer les graphiques de ce rapport.

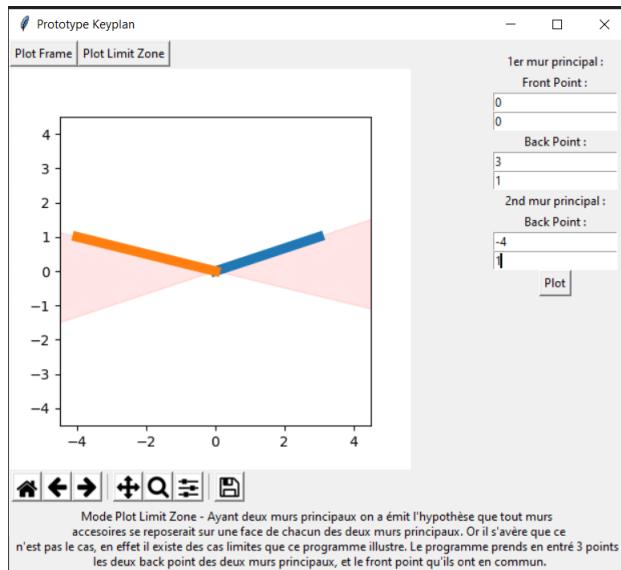


FIGURE 2.9 – v0.2

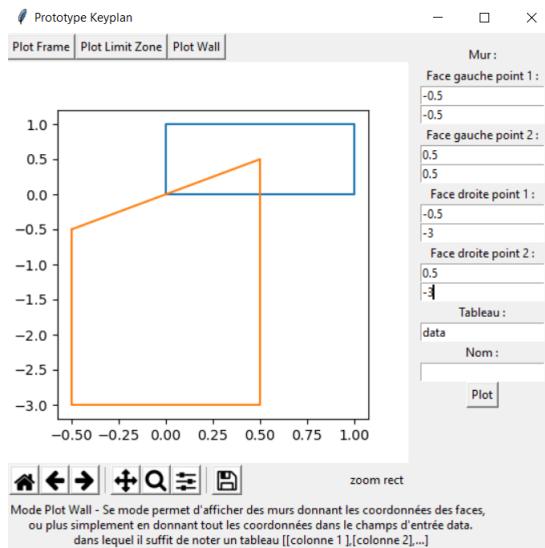


FIGURE 2.10 – v0.3

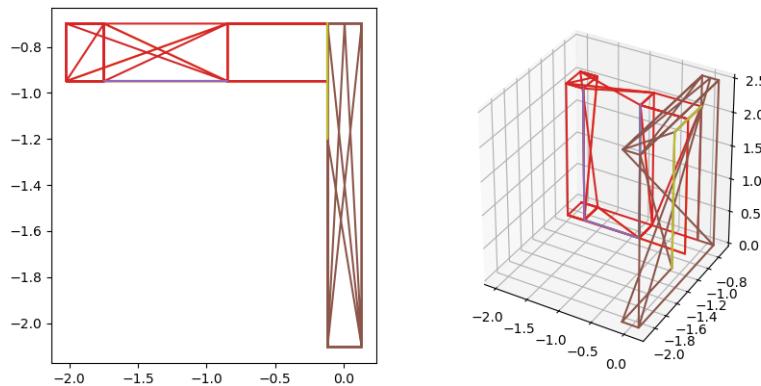


FIGURE 2.11 – v0.5

Chapitre 3

Esquisse sur l'ajout des étages

On souhaite implémenter une fonctionnalité dans Keyplan3D qui permet de créer des étages. Tout ceci doit être fait dans un souci de cohérence architecturale. Par manque de temps dans le stage, nous présentons au lecteur des réflexions sur ce sujet.

3.1 Règles pour la création et la modification d'étages

Dans cette section, nous décrirons des règles afin de cadrer un minimum la création et la modification des étages.

3.1.1 Création d'étage

Tout d'abord, il est nécessaire de définir comment un étage peut être créé. A priori, pouvoir créer un étage « généralisé »¹ pose trop de problème. En effet, les pièces peuvent être de hauteur différentes (voire posséder des murs de hauteurs différentes en leur sein); dans ces deux cas, la création d'un étage ne semble pas raisonnable. De plus, pour un ensemble de pièces non connexe, un utilisateur pourrait vouloir des étages uniquement sur certaines pièces.

De ce fait, il semble plus raisonnable de définir les étages à partir **d'un ensemble connexe de pièces possédant des murs de même hauteur**.

3.1.2 Idée d'interface

En pratique, on crée un étage avec un bouton sur les pièces (cf. figure 3.3), ensuite une pièce identique est générée dans un nouveau plan. Si cette pièce devait être connectée à un ensemble de pièces possédant des murs de même hauteur, alors un po-pup proposerait de créer l'étage pour tout l'ensemble.

Une fois l'étage créé, l'utilisateur possède un bouton pour passer d'un plan à l'autre contenant chaque étage, comme l'illustre les figures 3.1 et 3.2. Nous précisons qu'un même plan peut contenir plusieurs pièces non connexes tant que leur base est à la même hauteur.

1. C'est à dire on associe un étage à l'ensemble des pièces

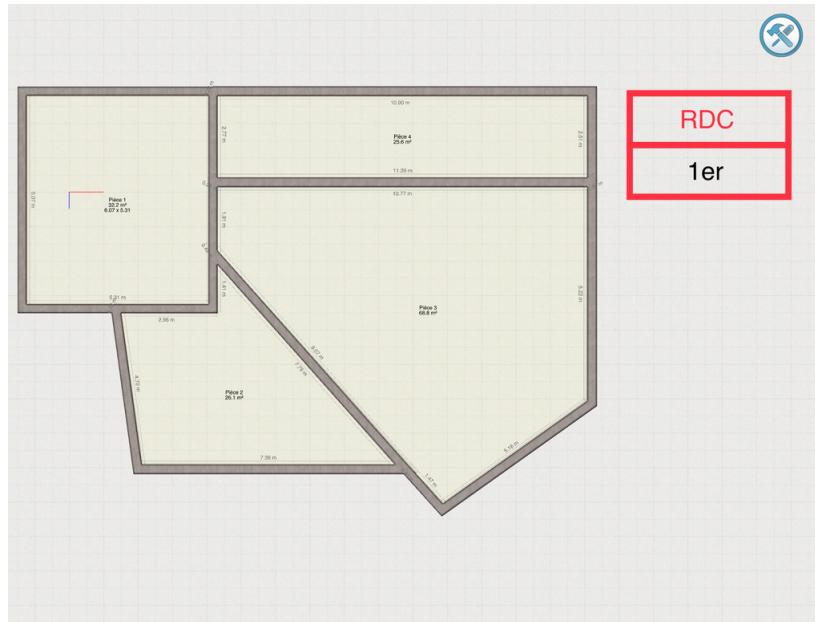


FIGURE 3.1 – Exemple de plan comportant deux étages avec un bouton pour passer de l'un à l'autre.

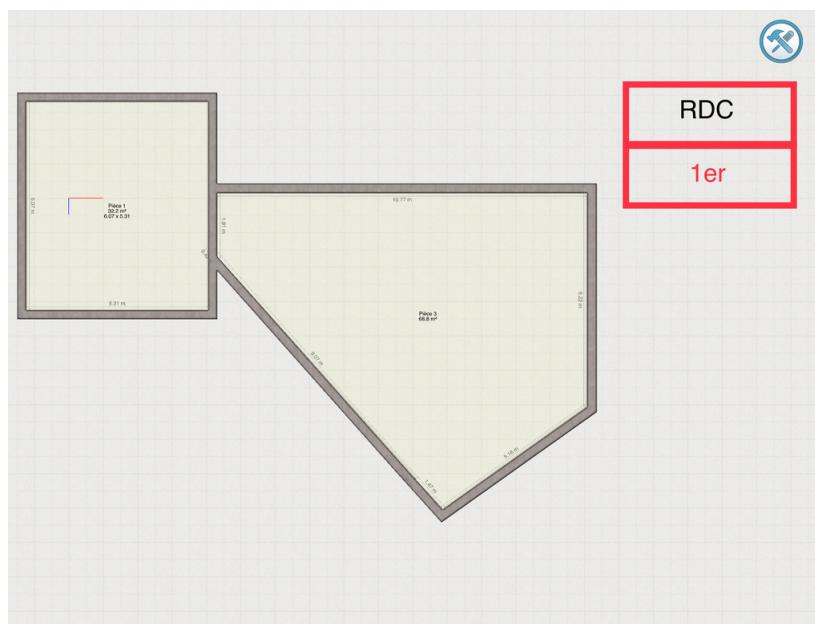


FIGURE 3.2 – Exemple de plan comportant deux étages avec un bouton pour passer de l'un à l'autre.

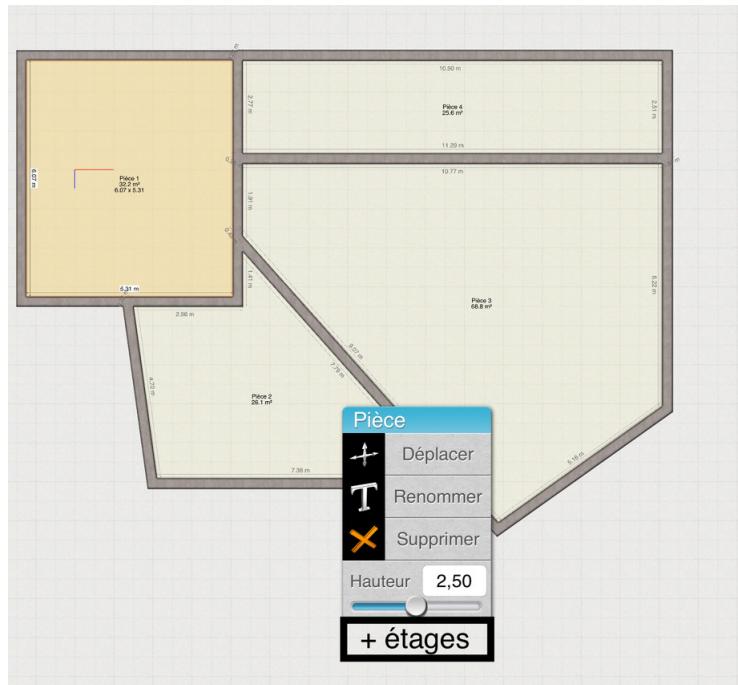


FIGURE 3.3 – Exemple de bouton pour ajouter un étage sur une pièce.



FIGURE 3.4 – Exemple de maison avec un étage plus large que le l'étage inférieur.

3.1.3 Modification des étages

Laisser libre la modification des pièces au niveau des étages pose des problèmes. Par exemple imaginons la situation suivante : l'utilisateur souhaite agrandir la pièce et il bouge l'un des murs de sorte que l'étage soit dix fois plus grand que le rez-de-chaussée (cf. figure 3.5). Il s'agit bien évidemment d'une situation complètement aberrante. Toutefois, la possibilité d'augmenter dans une moindre mesure la taille de l'étage par rapport à un étage inférieur doit être possible, car elle est fréquente en architecture (cf. figure 3.4).

Imaginons une autre situation où l'utilisateur souhaite bouger une des pièces de l'étage. L'utilisateur pourrait la changer complètement d'emplacement et d'obtenir une pièce qui « flotte » dans les airs. Cette situation est bien plus préoccupante, et nécessite une limitation. C'est pourquoi, on pourrait laisser libre la modification des pièces, dans la mesure où une partie de la pièce après modification est toujours au dessus de l'étage inférieur.

3.2 « Rattacher » des étages

Dans cette section, nous définirons une façon de « rattacher » des étages entre eux.

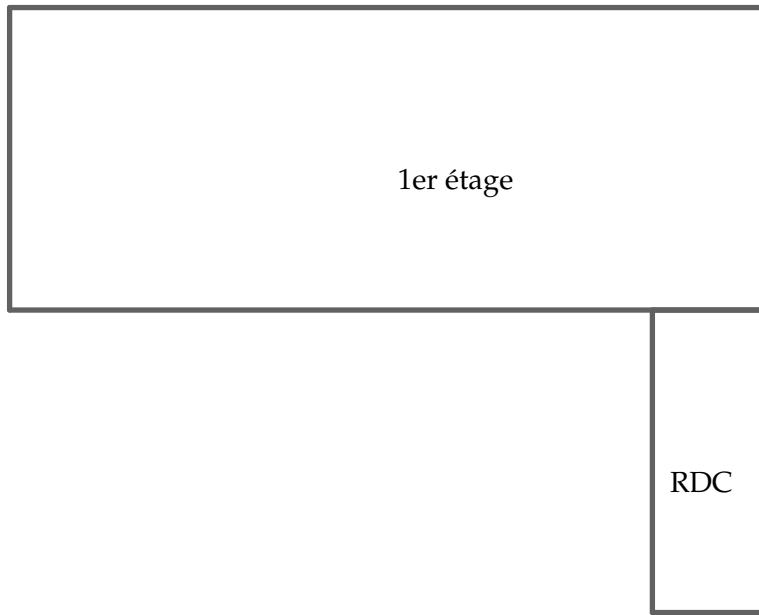


FIGURE 3.5 – Vue latérale d'une maison, aberration architecturale.

L'idée serait de poser un escalier dans l'étage le plus bas. Selon l'endroit où l'escalier estposé, le sol de l'étage sera automatiquement Troué. Cette idée qui paraît simple à première vue, pose un problème. Supposons que l'escalier que l'on pose soit d'une taille x et que la hauteur de l'étage soit de $10x$, dans ce cas, on pourrait juste augmenter manuellement la taille du modèle de l'escalier. Cependant, on aurait des marches de tailles disproportionnées et un modèle pas très esthétique. De ce fait, la solution idéale serait de générer un escalier selon un certain nombre de paramètres.

3.2.1 Génération d'escalier

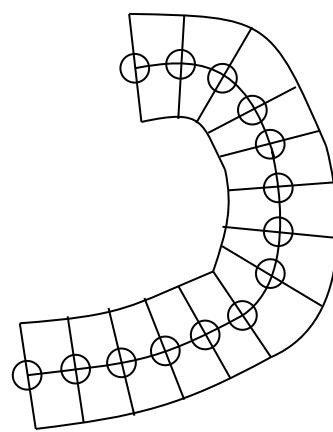
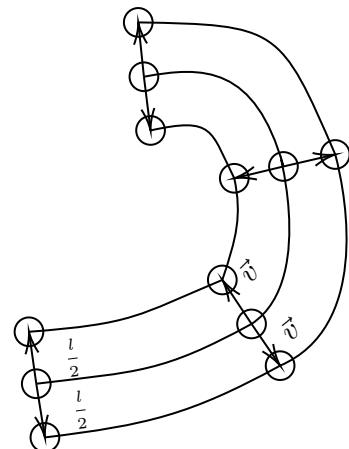
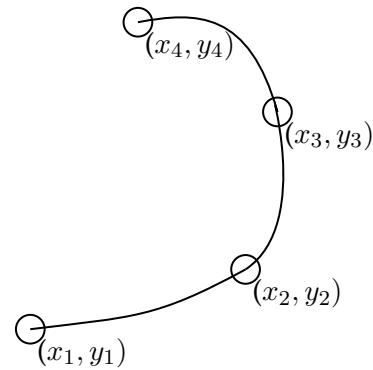
Afin de résoudre ce problème, nous donnons une piste d'algorithme ci-dessous :

Soit $(x_i, y_i) \in \mathbb{R}^2$ un ensemble de point définissant la courbe que va suivre l'escalier à l'aide d'une *splines*. Soit $l \in \mathbb{R}$ la largeur de l'escalier. Et soit $n \in \mathbb{N}$ le nombre de marches.

Dans un premier temps on va définir les bords de l'escalier, par deux ensembles de points $(x'_i, y'_i) \in \mathbb{R}^2$ et $(x''_i, y''_i) \in \mathbb{R}^2$. Les points (x'_i, y'_i) sont définis comme la translation des points (x_i, y_i) , par l'application T_v où v est définie comme le vecteur orthogonale à la courbe de longueur $l/2$. On procède de même pour les points (x''_i, y''_i) mais dans la direction opposée.

Enfin on discrétise la courbe en n points successifs, ceux-ci forment deux à deux les marches de l'escalier. La i -ième marche sera finalement extrudée d'une hauteur $i\frac{n}{h}$ pour h la hauteur d'un étage.

En conclusion, dans ce chapitre, nous avons traiter trois concepts qui permettront de mieux appréhender la hauteur des murs pour une futur implémentation.



Conclusion

L'objectif de ce stage était d'implémenter la fonctionnalité de la hauteur des murs dans l'application Keyplan3D.

C'est pourquoi, dans un premier temps nous avons réfléchi et débattu sur ce que devait être cet algorithme. En effet, un algorithme de hauteur des murs possède forcément un grand nombre de choix arbitraires. Ainsi, les débats ont mené à un algorithme qui a une cohérence architecturale et est transparent pour l'utilisateur.

Afin de mieux formaliser l'algorithme nous avons défini un cadre mathématiques sur lequel s'appuyer (cf. chapitre 1). Celui-ci a permis de définir l'algorithme le plus précisément possible.

Enfin, l'implémentation dans le moteur a nécessité plusieurs mois. Dans un premier temps, quelques semaines ont été nécessaires pour s'habituer aux codes. Dans un second temps, nous avons fait face à de nombreux imprévus comme par exemple les booléens décrits dans le chapitre 2. Finalement, les derniers mois on été consacrés à la correction des nombreux bugs et faiblesses de l'algorithme.

De plus, nous avons eu le temps de réfléchir à l'implémentation d'une autre fonctionnalité : « l'ajout d'étages ». (cf. chapitre 3).

En conclusion, ce stage a été très enrichissant sur de nombreux points de vue :

- La mise en pratique du C++ appris tout au long du master.
- Le travail en équipe avec M. Boisseron, et les nombreux débats que nous avons eus.
- La formalisation d'un problème original assez éloigné des concepts que nous manipulons à la faculté.

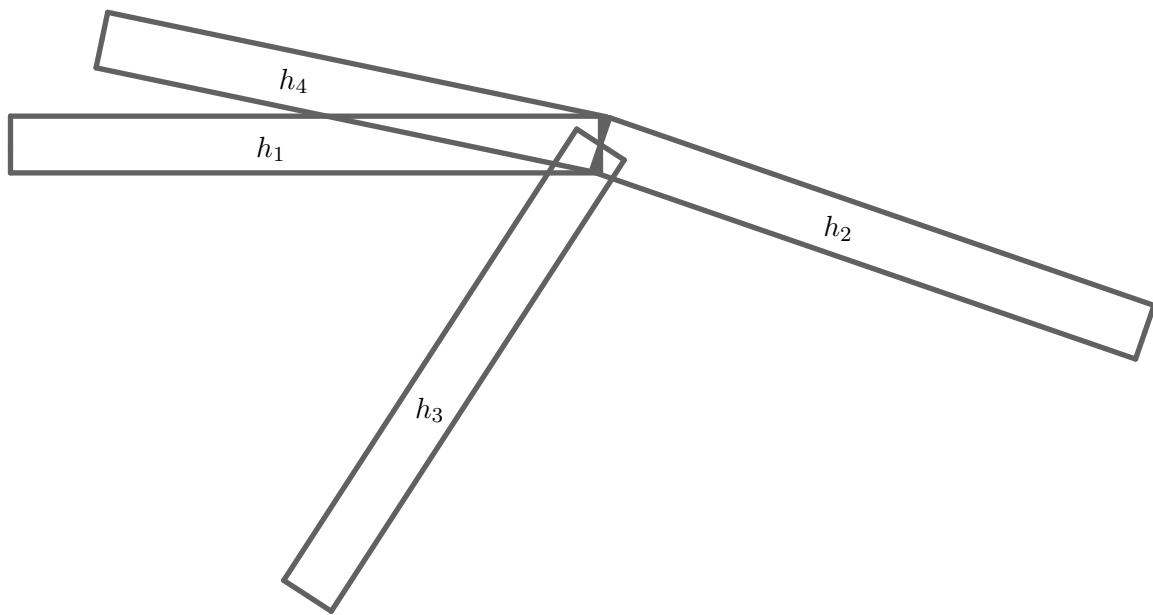


FIGURE 3.6 – A gauche l’implémentation naïve de la hauteur des murs, à droite le résultat attendu.

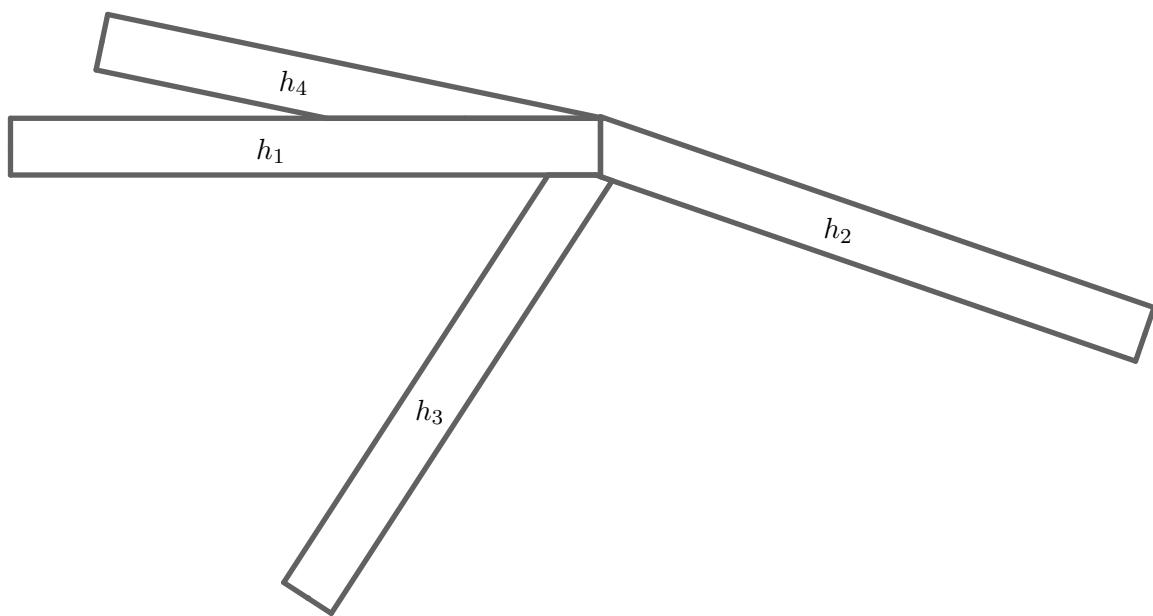


FIGURE 3.7 – A gauche l’implémentation naïve de la hauteur des murs, à droite le résultat attendu.

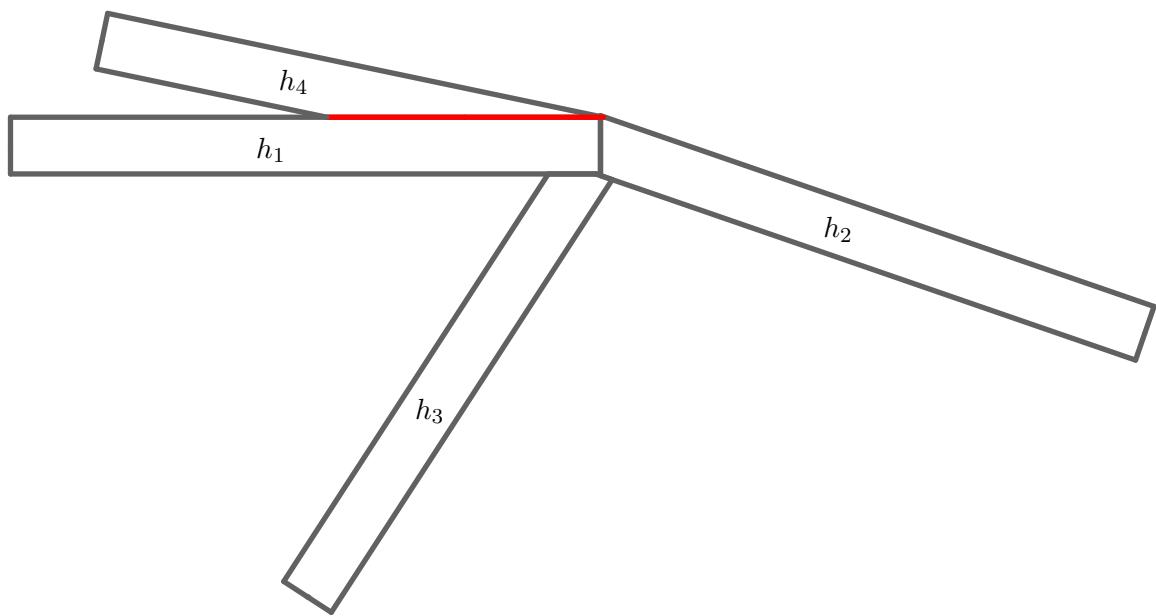


FIGURE 3.8 – A gauche l’implémentation naïve de la hauteur des murs, à droite le résultat attendu.