Université libre de Bruxelles

ELEC-H304

Physique des Télécommunications

Rapport de projet : Ray Tracing

Auteurs :
Powis De Tenbossche Sylvain
Quiroga Anthony

Professeur :
De Doncker Philippe

Assistant :
Gontier Quentin

Année académique 2022-2023

Table des matières

1 Introduction

Ce projet a été réalisé par des étudiants de 3^e année de bachelier d'ingénieur civil à l'Ecole Polytechnique de Bruxelles dans le cadre d'un cours de physique des télécommunications. L'objectif de ce projet est d'analyser la couverture réseau d'une usine en réalisant un code permettant de faire du ray tracing. Cette analyse est ensuite utilisée pour optimiser le positionnement de nouvelles antennes afin d'obtenir une couverture réseau satisfaisante.

En plus d'optimiser la position des antennes pour obtenir une couverture réseau optimal, il était essentiel de respecter les normes relatives aux puissances maximales acceptables par la législation bruxelloise.

2 Implémentation du code

2.1 Principe du ray tracing

Le ray tracing est une méthode numérique qui permet de simuler le chemin parcouru par les ondes électromagnétiques émises par une antenne émettrice. Les interactions des rayons avec les différents obstacles qui constituent l'environnement sont caractérisés par un coefficient de réflexion ou de transmission selon le type de cas. Tous les coefficients obtenus selon le chemin parcouru par le rayon sont ensuite multipliés. Ce coefficient global s'utilise par la suite dans la formule de Friis pour le calcul de la puissance au récepteur.

2.2 Approximations/simplifications

Le projet a été fait sur base des hypothèses suivantes : les ondes sont planes, l'usine est en 2D, la diffraction n'est pas prise en compte, les calculs sont faits dans le domaine des champs lointain et la présence d'air n'est pas prise en compte. La polarisation a été considérée comme étant perpendiculaire.

De manière générale, l'expression de la puissance reçue au récepteur se calcule de la manière suivante : $P_{RX} = \frac{1}{8R_a} \| \sum h_e(\theta_n, \phi_n) \underline{\overrightarrow{E_n}}(\overrightarrow{r}) \|^2$
Cependant, le projet impose l'utilisation de 3 types d'antennes : TX1, TX2 et TX3. Les spécificités des différentes antennes sont reprises dans le tableau ci-dessous :

TX1 : Antenne $\frac{\lambda}{2}$ verticale, sans pertes, émettant à 20 dBm

TX2 : Idem que TX1 mais émet à 35 dBm

TX3 : Réseaux d'antennes avec réflecteur avec une puissance de 35 dBm ayant pour gain : $G(\theta)[dB] = G_{max}[dB] - 12(\frac{\phi - \delta}{\phi_{3dB}})^2$

Avec les indications reprises ci-dessus, il est possible de simplifier l'équation générale pour la puissance. La hauteur caractéristique d'une antenne $\frac{\lambda}{2}$ s'écrit : $\overrightarrow{h_e} = -\frac{\lambda}{\pi} \frac{\cos(\frac{\pi}{2}\cos\theta)}{\sin^2\theta} \overrightarrow{1z}$.
TX1 et TX2 étant des antennes verticales, $\theta = \frac{\pi}{2} \rightarrow \cos\theta = 0\,;\, \sin\theta = 1 \rightarrow h_e = -\frac{\lambda}{\pi}$.
La hauteur équivalente devient alors une simple constante dans la formule de la puissance. Dès lors, il vient : $P_{RX} = \frac{\lambda^2}{8R_a\pi^2} \| \sum \underline{\overrightarrow{E_n}}(\overrightarrow{r}) \|^2$

L'autre indication mentionne que les antennes sont sans pertes. Cette indication renseigne sur la valeur à donner à R_a car celle-ci est la somme des résistances de rayonnement(R_{ar}) et ohmiques(R_{al}). L'hypothèse d'antenne sans perte permet alors de s'affranchir du terme $R_{al}(R_a = R_{ar})$. La résistance de rayonnement est donnée par : $R_{ar} = \frac{720\pi}{32}$.

Au final, la puissance totale obtenue au récepteur pour une antenne $\frac{\lambda}{2}$ sans pertes s'ob-

tient par la formule : $P_{RX} = \frac{\lambda^2}{8*\frac{720\pi}{32}*\pi^2} \| \sum \underline{\overrightarrow{E_n}}(\overrightarrow{r}) \|^2$

À noter qu'il est également possible de calculer la puissance moyenne reçue au récepteur. Pour cela, il suffit de sommer le module au carré des champs et non plus de prendre la norme au carré de la somme des champs. Ce qui dans le cas d'une antenne $\frac{\lambda}{2}$ sans pertes donne : $< P_{RX} > = \frac{\lambda^2}{8*\frac{720\pi}{32}*\pi^2} \sum \| \underline{\overrightarrow{E_n}}(\overrightarrow{r}) \|^2$

Il faut également prendre en compte la directivité de 1.76 dBi dans le plan xy pour les antennes dipôles.

2.3 Logiciel utilisé

En ce qui concerne le logiciel, celui-ci n'était pas imposé aux étudiants. Cependant, il était fortement recommandé d'utiliser le C++, car celui-ci est environ 10 fois plus rapide que Matlab et 200 fois plus rapide que Python. De plus, étant donné que le ray-tracing est très demandant en calcul, chaque optimisation est bonne à prendre.

Une des recommandations du Professeur De Doncker, si le choix du langage se portait sur C++, était d'utiliser Qt Creator. C'est donc ce qui a été fait. Voici une illustration de l'interface graphique pour le schéma du projet :

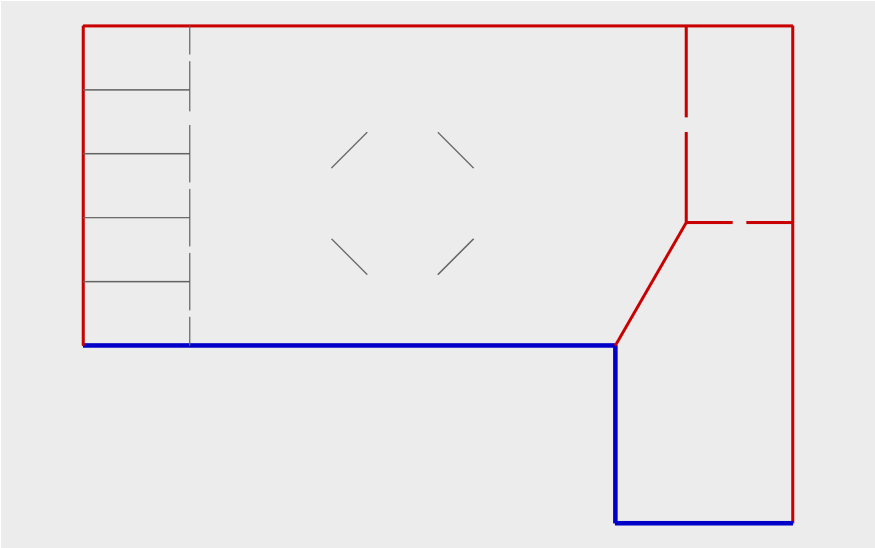


FIGURE 1 – Interface graphique de l'usine du projet

2.4 Trigonométrie

Les différentes fonctions trigonométriques (sinus, cosinus, ...) sont, autant que possible calculées à l'aide de produits scalaires et vectoriels pour éviter l'utilisation des fonctions trigonométriques intégrées qui sont très coûteuses en temps de calcul. Le seul appel à une

fonction trigonométrique est pour calculer l'angle nécessaire au calcul de la directivité de l'antenne TX3.

2.5 Pré-analyse du positionnement des murs

Le code effectue au démarrage une analyse du positionnement relatif des murs les uns par rapport aux autres. Un mur complètement à gauche d'un autre est stocké comme un "-1", un mur complètement à droite comme un "1" et un mur qui serait à la fois à gauche et à droite est stocké comme un 0.

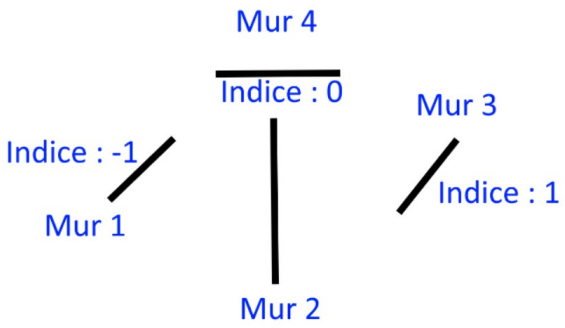


FIGURE 2 – Évaluation des indices par rapport au mur 2

En supposant que lors du calcul des antennes images, un rayon vienne d'un mur 1 et soit réfléchit sur un mur 2. Si ce mur 1 est à gauche du mur 2, le rayon ne peut être réfléchi sur un troisième mur qui serait à droite du mur 2, sinon il passerait au travers du mur 2 tout en changeant d'angle, donnant un rayon sans interprétation physique dans le cadre du projet. Ce cas non physique est illustré ci-dessous.

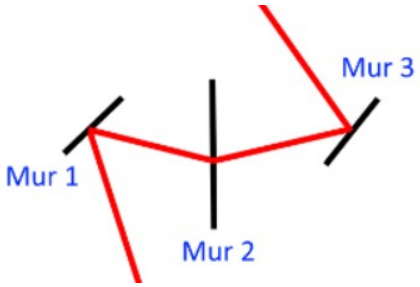


FIGURE 3 – Exemple de rayon non physique

La façon dont les indices introduits précédemment rentrent en compte dans le code est la suivante : le code multiplie les indices (-1,0 ou 1) du mur précédent et suivant [1]. Si cette

1. Dans l'exemple de la Figure 3, les indices du mur 1 et du mur 2 seraient multipliés

multiplication donne -1, c'est que les 2 murs sont des 2 cotés opposés au mur 2, indiquant alors que tout rayon passant par ces 3 murs serait non-physique. Le code abandonne alors directement les calculs et passe à d'autres murs. Le code pour le calcul des antennes images étant récursif, l'optimisation décrite dans cette section permet donc d'éviter de calculer toutes les possibilités et de faire le tri par la suite, ce qui permet de gagner en temps de calcul.

2.6 Multithreading

Afin de tirer pleinement profit d'un processeur, il convient d'utiliser tous ses cœurs. Pour ce faire, il est nécessaire de paralléliser le code et d'utiliser plusieurs "threads", permettant au processeur d'utiliser plusieurs cœurs afin de réaliser plusieurs calculs à la fois.

La parallélisation a été faite au niveau du calcul de la heatmap : plusieurs threads calculent chacun une portion des carrés de 0.5mx0.5m. Une fois l'exécution de tous les threads terminées, tous ces carrés sont rassemblés pour former la heatmap complète.
Le temps de calcul est divisé par le nombre de threads utilisé. Un Ryzen 7 6800H permet par exemple d'utiliser 14 threads et de générer la heatmap pour 2 réflexions en moins de 5 secondes.

L'optimal serait d'utiliser une carte graphique qui est bien plus adaptée pour ce type de calculs, cependant cela se ferait au coût de la lisibilité du code. Au vu du temps d'exécution déjà très rapide, il a été estimé qu'il n'était pas nécessaire de passer les calculs sur la carte graphique.

2.7 Détection de l'intérieur du bâtiment

Une détection automatique des points internes au bâtiment permet de ne calculer la puissance que dans les parties internes au bâtiment. En plus de gagner en temps de calcul, cela permet de n'optimiser que la puissance interne au bâtiment lors du calcul des paramètres optimaux des antennes. Sur la Figure 4, la zone hachurée en blanc, correspond à une zone

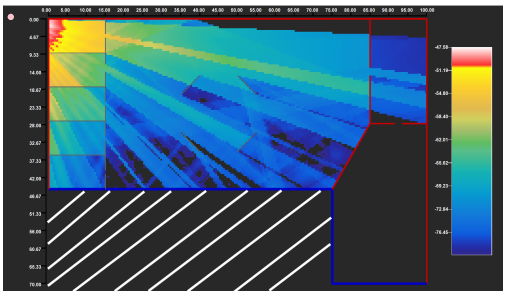


FIGURE 4 – Puissance non calculée dans la zone hachurée

se trouvant en dehors de l'usine et par conséquent, comme expliqué précédemment, il n'est pas utile d'y connaître les valeurs de puissance. Le code ne les calculera donc pas.

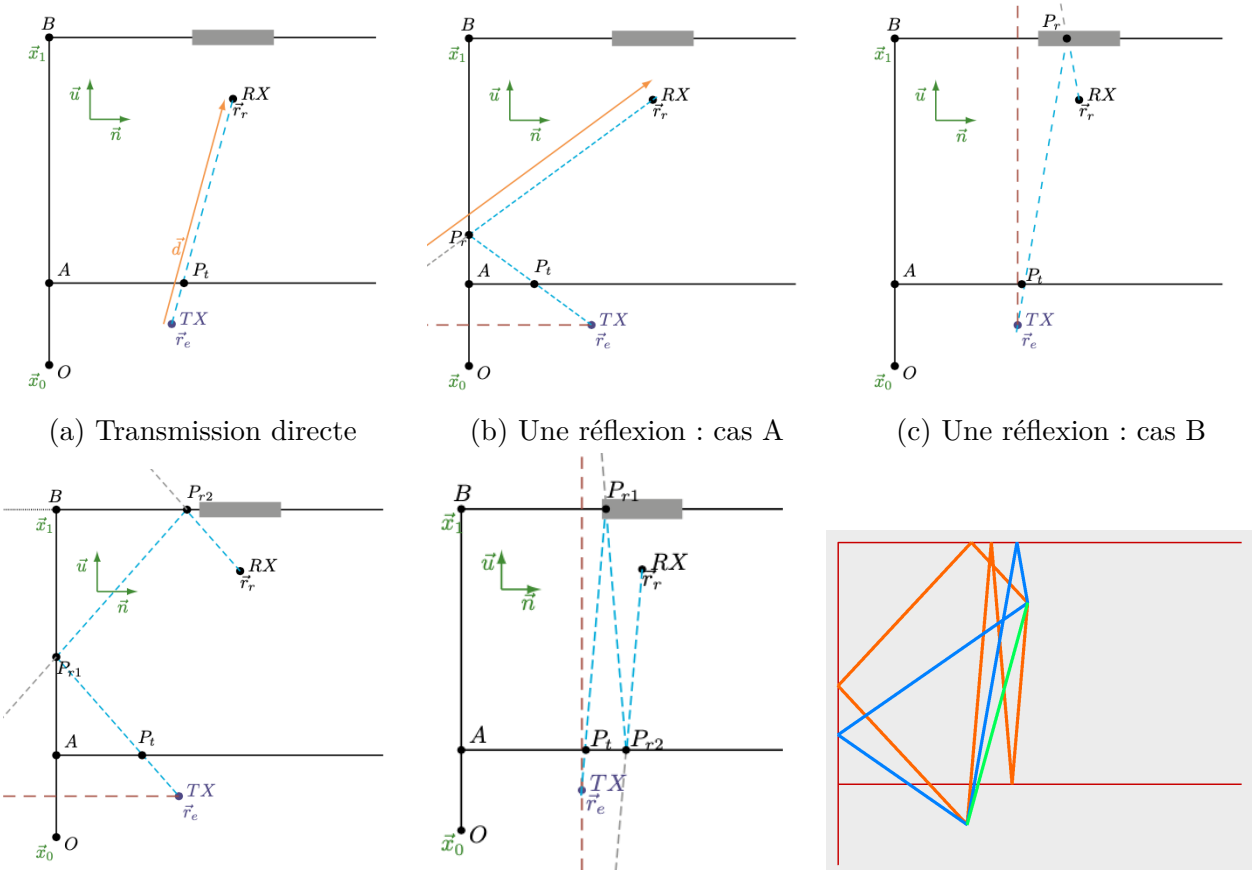
3 Validation du code à l'aide de l'exercice 8.1

Afin de tester le code de ray tracing, une résolution manuelle de l'exercice 8.1 pour une certaine composante multi-trajet sera effectuée. Celle-ci sera ensuite comparée aux valeurs données par le code afin de s'assurer du bon fonctionnement de celui-ci.

3.1 Comparaison avec le correctif

3.1.1 Tracé des rayons

Dans un premier temps, il est primordial de s'assurer que le tracé des rayons (pour un cas relativement simple) calculé par le code correspond bien au tracé fourni dans le correctif. L'illustration ci-dessous permet aisément de voir qu'en effet le tracé des rayons que retourne le code est bien celui donné dans le corrigé.



(a) Transmission directe (b) Une réflexion : cas A (c) Une réflexion : cas B

(d) Deux réflexions : cas A (e) Deux réflexions : cas B (f) Tracé des rayons : code

FIGURE 5 – Tracés des rayons donnés par le correctif et par le code de ray tracing

3.1.2 Résultats pour la transmission directe et le cas à une réflexion :

Le tracé des rayons étant correct, il faut maintenant vérifier si les valeurs numériques que donne le code pour la composante des champs et pour la puissance est similaire aux valeurs fournies par le correctif [2]. Le cadre ci-dessous reprend, sans le détail des calculs, les valeurs du correctif :

Chemin direct :
$T_m = 0.69 + 0.22i$
$\underline{E_1} = (0.0037 - 0.0016i)\text{V/m}$
$P_{RX} = 3,33.10^{-10}$ W $\sim -64,77$ dBm

Une réflexion, cas A :
$\Gamma_m = -0.334 + 0.225i$
$T_m = 0.539 + 0.023i$
$\underline{E_2} = (-5,41.10^{-4} - 4,57.10^{-4}i)\text{V/m}$
$P_{RX} = 2,91.10^{-10}$ W $\sim -65,37$ dBm

Une réflexion, cas B :
$\Gamma_m = -0.2044 + 0.1511i$
$T_m = 0.6910 + 0.2480i$
$\underline{E_3} = (-3,5143.10^{-4} + 5,8005.10^{-4}i)\text{V/m}$
$P_{RX} = 2,0843.10^{-10}$ W $\sim -66,8104$ dBm

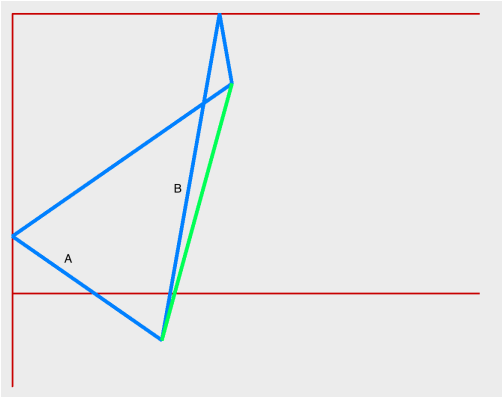


FIGURE 6 – Cas considéré : Transmission directe et une réflection

Les images ci-dessous donnent les valeurs obtenues sur la console de sortie du programme [3].

```
Done reading walls from files.

Coefficient de transmission total du rayon  1 : 0.690657 + 0.233935 j
Found 1 rays.

Power :  3.33544e-10
Time taken : 0.108025 seconds
```

(a) Transmission directe

```
Done reading walls from files.

Coefficient de transmission total du rayon  1 : 0.690657 + 0.233935 j
Coefficient de réflection total du rayon 1 : 1 + 0 j
Coefficient de transmission total du rayon  2 : 0.538698 + 0.022825 j
Coefficient de réflection total du rayon 2 : -0.333844 + 0.225205 j
Coefficient de transmission total du rayon  3 : 0.69137 + 0.24838 j
Coefficient de réflection total du rayon 3 : -0.204106 + 0.151346 j
Found 3 rays.

Power :  2.68657e-10
Time taken : 0.105798 seconds
```

(b) Prise en compte des composantes à une réflexion

FIGURE 7 – Résultat en sortie de console pour le calcul à transmission directe et à une réflection

2. Le détail des calculs effectués pour le cas B à une réflexion se trouvent en Annexe
3. À noter que sur la figure 7(b), le coefficient de réflexion pour le rayon 1 est de 1, car dans le code, la valeur du coefficient est initialisé à 1. Elle est ensuite modifiée lors des différentes réflexions. Puisque dans le cas de la transmission directe, aucune réflexion n'est réalisée, le coefficient reste à 1.

Les résultats obtenus en sortie de console sont bien conformes à ceux attendus par le correctif. La prochaine étape consiste à tester le code pour des cas à deux réflexions. Cela fait l'objet de la sous-section suivante .

3.2 Calcul détaillé pour une des composantes à deux réflections

Cette section présentera la résolution détaillée pour une des composantes multi-trajet à deux réflexions. Le trajet choisi est présenté sur l'illustraion suivante :

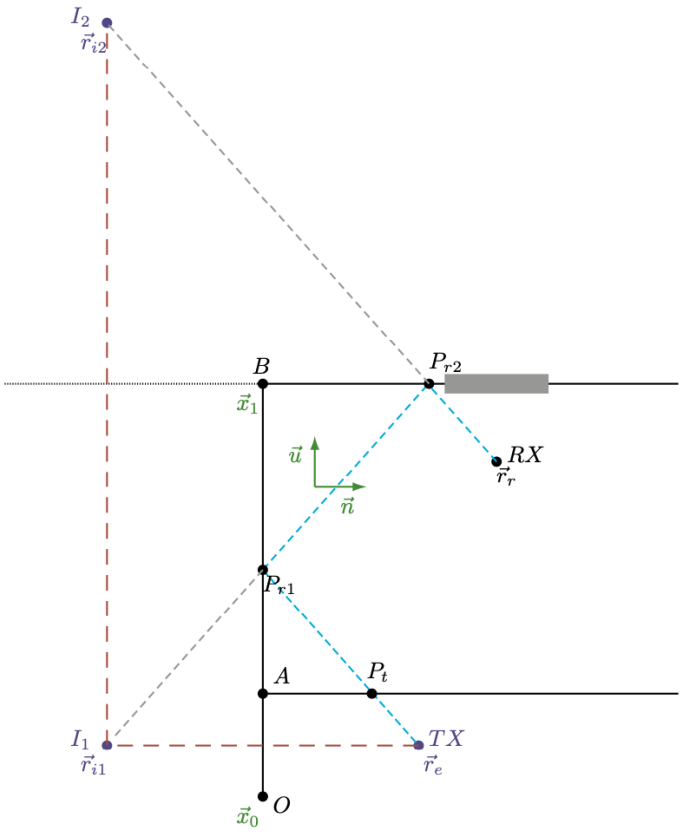


FIGURE 8 – Chemin du rayon pour le cas B à deux réflexions

La première étape du calcul consistera à déterminer la position des antennes images pour ensuite pouvoir trouver les points sur les murs par lesquels passe le rayon.

Les coordonnées des antennes images est assez aisée à obtenir puisqu'une simple symétrie orthogonale a été effectuée pour les obtenir. Pour la coordonnée I_1 : $(-32; 10)$. Pour la coordonnée I_2 : $(-32; 150)$. Il est désormais possible calculer les coordonnées de P_{r2}, P_{r1},P_t

3.2.1 Calcul de $P_{r2}/P_{r1}/P_t$:

Le calcul des coordonnées des points P_n se fait en commençant par la fin. P_{r2} est un point intermédiaire sur le chemin $\overrightarrow{I_2RX}$. Pour ce faire, il est nécessaire de déterminer le vecteur reliant ces deux points et de le normer.

$\vec{v_1} = (47-(-32); 65-150) = (79; -85) \rightarrow \|\vec{v}\| = \sqrt{79^2+(-85)^2} = 116.0431$
Vecteur v_1 normé : $\frac{\vec{v_1}}{\|\vec{v_1}\|} = (0.6808; -0.7325)$
$P_{r2} = I_2 + \lambda * \frac{\vec{v_1}}{\|\vec{v_1}\|} \rightarrow (x; 80) = (-32; 150) + \lambda * \frac{\vec{v_1}}{\|\vec{v_1}\|}$
$\lambda = \frac{80-150}{-0.7325} = 95.5631 \rightarrow x = -32 + 95.5631 * 0.6808 = 33.0593 \rightarrow P_{r2} = (33.0593; 80)$

Le point P_{r2} connu, il est possible de calculer le point P_{r1} de la même façon en reliant cette fois-ci I_1 et P_{r2}

$\vec{v_2} = (33.0593-(-32); 80-10) = (65.0593; 70) \rightarrow \|\vec{v_2}\| = \sqrt{(65.0593)^2+(70)^2} = 95.5652$

Vecteur v_2 normé : $\frac{\vec{v_2}}{\|\vec{v_2}\|} = (0.6808; 0.7325)$
$P_{r1} = I_1 + \lambda * \frac{\vec{v_2}}{\|\vec{v_2}\|} \rightarrow (0; y) = (-32; 10) + \lambda * \frac{\vec{v_2}}{\|\vec{v_2}\|}$
$\lambda = \frac{0-(-32)}{0.6808} = 47.0035 \rightarrow y = 10 + 47.0035 * 0.7325 = 44.4301 \rightarrow P_{r1} = (0; 44.4301)$

Le dernier point à calculer est le point P_t. La même méthode de résolution est à nouveau appliquée.

$\vec{v_3} = (0-32; 44.4301-10) = (-32; 34.4301) \rightarrow \|\vec{v_3}\| = \sqrt{(-32)^2+(34.4301)^2} = 47.0046$

Vecteur v_3 normé : $\frac{\vec{v_3}}{\|\vec{v_3}\|} = (-0.6808; 0.7325)$
$P_t = TX + \lambda * \frac{\vec{v_3}}{\|\vec{v_3}\|} \rightarrow (x; 20) = (32; 10) + \lambda * \frac{\vec{v_3}}{\|\vec{v_3}\|}$
$\lambda = \frac{20-10}{0.7325} = 13.6519 \rightarrow x = 32 + 13.6519 * (-0.6808) = 22.7058 \rightarrow P_t = (22.7058; 20)$

3.2.2 Coefficients de réflexion/transmission

Le rayon subit une transmission et deux réflexions au cours de son trajet. Il y a donc 2 coefficients de réflexion à calculer ainsi qu'un coefficient de transmission.

Calcul du coefficient Γ_2 : La composante normale du mur au point P_{r2} est $(0; -1)$. La projection de $\vec{v_1}$ sur le vecteur normal donne : $\cos\theta_i = < \frac{\vec{v_1}}{\|\vec{v_1}\|}; (0; -1) > = 0.7325$. De là, il est

aisé d'obtenir $\sin\theta_i$: $\sqrt{1-0.7325^2} = 0.6808$.

$Sin\theta_t$ s'obtient en prenant en compte la permittivité du mur et en appliquant la loi de Snell$\rightarrow \sin\theta_t = \sqrt{\frac{1}{\epsilon_r}}\sin\theta_i = \sqrt{\frac{1}{4.8}} * 0.6808 = 0.3107 \rightarrow \cos\theta_t = \sqrt{1-0.3107^2} = 0.9505$
s $= \frac{l}{\cos\theta_t} = \frac{0.15}{0.9505} = 0.1578$
Coefficient de réflexion : $\Gamma_b = \frac{Z_m\cos\theta_i - Z_0\cos\theta_t}{Z_m\cos\theta_i + Z_0\cos\theta_t} = \frac{(171.57+j6.65)*0.7325-377*0.9505}{(171.57+j6.65)*0.7325+377*0.9505} = -0.4805 + j0.0149$

Calcul de Γ_2 :

$\Gamma_2 = \Gamma_b - (1-\Gamma_b^2) * \frac{\Gamma_b e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}}{1-\Gamma_b^2 e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}}$ = -0.4188 + 0.2462j

Pour le deuxième calcul du coefficient Γ_2, la même démarche est appliquée.

Calcul du coefficient Γ_2 : Le cosinus incident est obtenu en projetant le vecteur $\vec{v_2}$ normé sur la normale du mur correspondant $(1;0) \rightarrow \cos\theta_i = <(0.6808;0.7325);(1;0)> = 0.6808$.
Les valeurs suivantes en découlent [4] : $\sin\theta_i = 0.7325$; $\sin\theta_t = 0.3343$; $\cos\theta_t = 0.9425$; s = 0.1591
Coefficient de réflexion : $\Gamma_a = \frac{Z_m\cos\theta_i - Z_0\cos\theta_t}{Z_m\cos\theta_i + Z_0\cos\theta_t} = \frac{(171.57+j6.65)*0.6808-377*0.9425}{(171.57+j6.65)*0.6808+377*0.9425} = -0.5050 + j0.0144$

Calcul de Γ_1 :

$\Gamma_1 = \Gamma_a - (1-\Gamma_a^2) * \frac{\Gamma_a e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}}{1-\Gamma_a^2 e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}}$ = -0.4710 + 0.2518j

Le deuxième coefficient de réflexion ayant été calculé, il ne reste plus qu'à calculer le coefficient de transmission pour pouvoir passer au calcul de la puissance.

Calcul du coefficient T_1 :
$\cos\theta_i = <(-0.6808;0.7325);(0;1)> = 0.7325$. A nouveau, les autres valeurs découlent de $\cos\theta_i \rightarrow \sin\theta_i = 0.6808$; $\sin\theta_t = 0.3107$; $\cos\theta_t = 0.9505$; s = 0.1578.
Coefficient de réflexion : $\Gamma_c = \frac{Z_m\cos\theta_i - Z_0\cos\theta_t}{Z_m\cos\theta_i + Z_0\cos\theta_t} = \frac{(171.57+j6.65)*0.7325-377*0.1578}{(171.57+j6.65)*0.7325+377*0.1578} = -0.5082 + j0.0144$

Calcul de T_1 :

Transmission $= \frac{(1-\Gamma^2)e^{-\gamma_m * s}}{1-\Gamma^2 e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}}$ = 0.6295 + 0.0890j

4. De manière analogue à ce qui a été fait au point précédent

Avant de passer au calcul de la puissance, il est nécessaire de connaître la distance totale qu'aura parcouru le rayon. Pour cela, il suffit de faire le trajet de TX à RX en prennant le module du vecteur.

Trajet TX/P_t $\sqrt{(22.7058-32)^2+(20-10)^2} = 13.6522$
Trajet P_t/P_{r1} $\sqrt{(0-22.7058)^2+(44.4301-20)^2} = 33.3524$
Trajet P_{r2}/P_{r1} $\sqrt{(33.0593-0)^2+(80-44.4301)^2} = 48.5606$
Trajet P_{RX}/P_{r2} $\sqrt{(47-33.0593)^2+(65-80)^2} = 20.4779$

Distance totale = 116.0431

Une autre façon de connaître la distance totale (bien plus efficace) est tout simplement de calculer la distance qui sépare la dernière antenne image du récepteur. Dans le cas présent, il s'agit de I_2 de coordonnées (-32 ;150). La distance qui sépare RX de I_2 est : $\sqrt{(47-(-32))^2+(65-150)^2} = 116.0431$ qui est bien identique à la valeur trouvée précédemment.

3.2.3 Calcul du champ et de la puissance reçue au récepteur

La valeur du champ se calcule grâce à la formule suivante :

$\underline{E_n} = \Gamma_1\Gamma_2T_1\sqrt{60G_{TX}P_{TX}}\frac{e^{-j\beta d_n}}{d_n}$

$\Gamma_1\Gamma_2T_1$ = (-0.4710 + 0.2518j)*(-0.4188 + 0.2462j)*(0.6295 + 0.0890j) = 0.1048 - 0.1273j

$\rightarrow \underline{E_n} = (0.0159 - 0.0496j)\sqrt{60*1.64*10^{-3}}\ \frac{e^{-j\frac{2\pi*868.3*10^6}{3*10^8}*116.0431}}{116.0431}$ = 4.4519*10^{-4} - 2.1816*10^{-5}j

Avec toutes les valeurs trouvés, il est désormais possible de calculer la puissance totale reçue au récepteur, elle se calcule comme suit : $\frac{\lambda^2}{8\pi^2R_a}\|E_n\|^2$

$P_{RX} = \frac{(\frac{3*10^8}{868.3*10^6})^2}{8\pi^2*73} * \|4.4519*10^{-4} - 2.1816*10^{-5}j\|^2 = 4.1145*10^{-12}$ W

P_{RX} = -83.8568 dBm

3.3 Comparaison entre la résolution manuelle et le programme

Grâce aux différents coefficients calculés lors du point précédent, il est maintenant possible de calculer la puissance reçue au récepteur pour la composante considérée. Voici les résultats

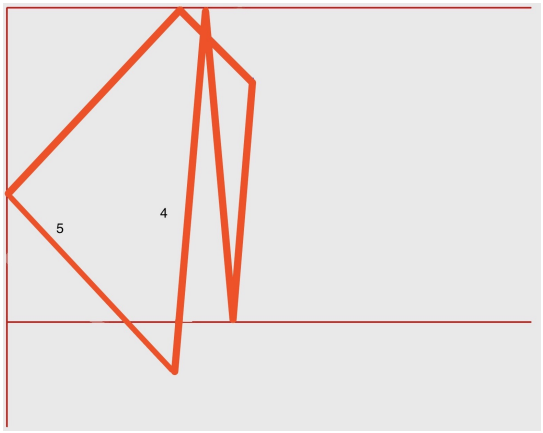
affichés par le programme lors de la prise en compte de 2 réflexions :

```
Done reading walls from files.

Coefficient de transmission total du rayon  1 : 0.690657 + 0.233935 j
Coefficient de réflection total du rayon 1 : 1 + 0 j
Coefficient de transmission total du rayon  2 : 0.538698 + 0.022825 j
Coefficient de réflection total du rayon 2 : -0.333844 + 0.225205 j
Coefficient de transmission total du rayon  3 : 0.69137 + 0.24838 j
Coefficient de réflection total du rayon 3 : -0.204106 + 0.151346 j
Coefficient de transmission total du rayon  4 : 0.69144 + 0.256821 j
Coefficient de réflection total du rayon 4 : 0.0177809 + -0.0570236 j
Coefficient de transmission total du rayon  5 : 0.630094 + 0.0891045 j
Coefficient de réflection total du rayon 5 : 0.134864 + -0.221442 j
Found 5 rays.

Power :  3.26161e-10
Time taken : 0.100998 seconds
```

(a) Console de sortie du code



(b) Trajet du rayon 4 et 5

FIGURE 9 – Résultat en sortie de console pour le calcul à transmission directe et à une réflection

Attention, le rayon pour lequel les calculs de la sous-section précédente ont été effectués est le rayon 5. Ce rayon est indiqué sur l'illustration 7(b). En comparant les coefficients de transmission et de réflexion obtenus à ceux de la sous-section précédente, on peut noter leur similitude au centième près. Pour ce qui est de la puissance totale, il sera difficile de la prendre en compte puisque le calcul pour le rayon 4 n'as pas été effectué manuellement. Il a donc été décidé de calculer la puissance de chaque rayon indépendamment des autres. En modifiant légèrement le code, la sortie suivante est obtenue.

```
Champ E du rayon 1 : 0.00381879 + 0.00123364 j
Power :  3.33544e-10
Champ E du rayon 2 : 0.000234736 + -0.000667634 j
Power :  1.03726e-11
Champ E du rayon 3 : -0.000452365 + -0.000505728 j
Power :  9.53501e-12
Champ E du rayon 4 : -7.85299e-05 + 5.15574e-06 j
Power :  1.28271e-13
Champ E du rayon 5 : 0.000445572 + -2.15585e-05 j
Power :  4.12138e-12
Found 5 rays.
```

FIGURE 10 – Chemin du rayon pour le cas B à deux réflexions

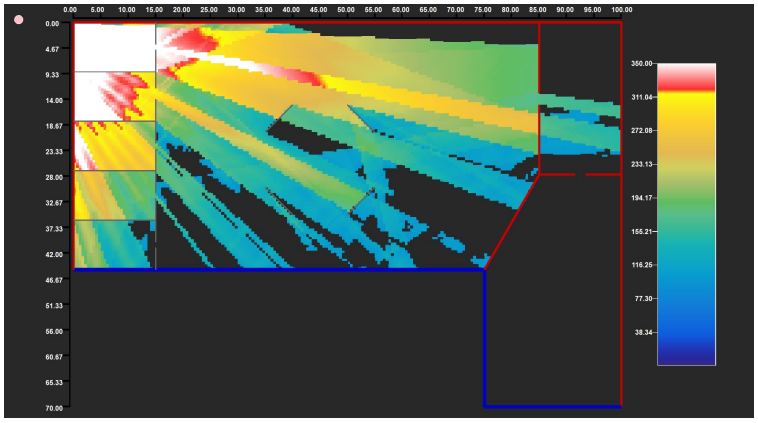
En observant les valeurs relatives au rayon 5, il est aisé de constater la similitude avec les valeurs numériques obtenues par résolution manuelle. Sur base des différents test et comparaison qui ont été effectués dans cette section, il semble que le code fonctionne correctement.

4 Application du code à la problématique du projet

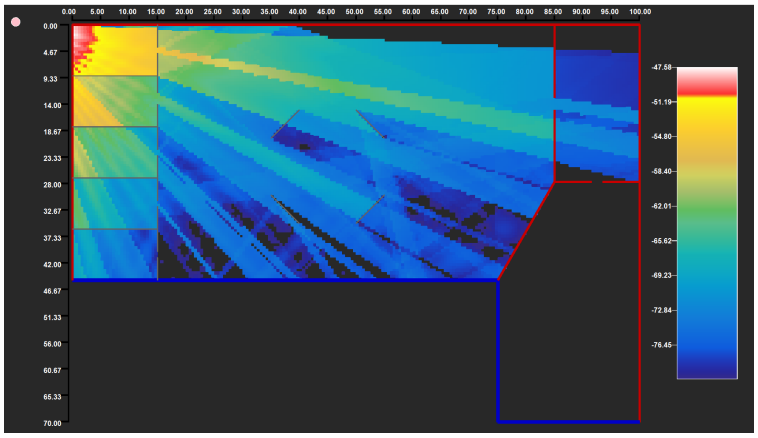
Les précédents tests effectués sur l'exercice 8.1 s'étant révélés corrects. Il est maintenant possible de s'attaquer à la résolution de la problématique du projet. L'objectif étant de respecter le seuil légal d'exposition aux ondes électromagnétiques tout en assurant un débit suffisant à l'intérieur de l'usine [5].

4.1 Résultats obtenus pour une antenne émetrice TX2

En utilisant comme antenne émettrice l'antenne TX2, les résultats suivants sont obtenus [6]



(a) Débit binaire



(b) Puissance des champs (dBm)

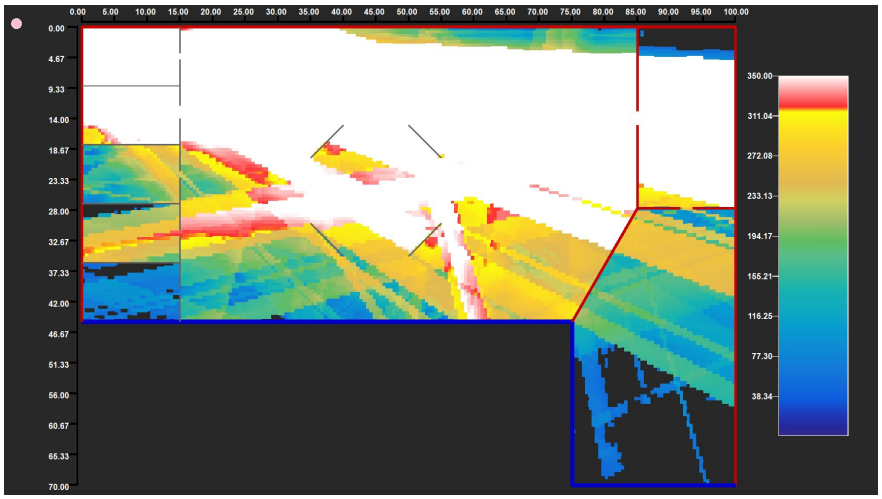
En se référant au code couleur affiché sur les résultats, il est possible de voir qu'un débit suffisant (100 Mb/s) n'est pas atteint dans une grande partie de l'usine. Cependant, en ce qui concerne les seuils légaux d'expositions aux champs électromagnétiques, il est possible

5. Certaines illustrations mises dans cette section se retrouverons en annexe avec un affichage plus grand pour une meilleure lisibilité. Lorsque ce sera le cas, une note de bas de page le mentionnera.
6. Illustrations mises en annexes (Figure 20 et 21)

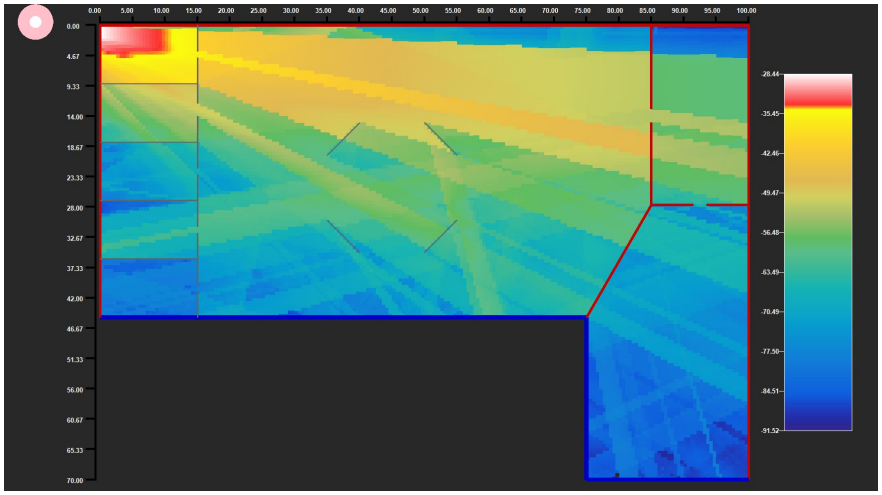
de voir que toutes les zones de l'usine sont conformes à la législation. La section suivante introduira l'antenne TX3 qui est l'antenne qui sera réellement placée.

4.2 Résultats avec une antenne TX3

Une antenne TX3 avec ses paramètres originaux donne le résultat suivant [7] :



(a) Débit binaire



(b) Puissance des champs (dBm)

La norme sur la puissance [8] est alors dépassée dans le coin supérieur gauche du bâtiment. Cependant, en utilisant les paramètres optimisés pour mieux répartir la puissance dans le bâtiment et maximiser le débit binaire (voir suite), la norme est respectée.

7. Illustrations en annexes, Figure 22 et 23

8. A noter qu'ici, la puissance calculé est en réalité la puissance moyenne. La puissance "classique" a uniquement été utilisée pour la section concernant l'exercice 8.1 car c'est ce qui a été considéré dans le correctif. Dans toute cette section, lorsque l'on parlera de puissance, il s'agira de la puissance moyenne.

4.3 Optimisation

L'amélioration du débit se fait grâce à l'introduction d'antennes TX1 à l'intérieur de l'usine tout en faisant attention à ne pas dépasser le seuil légal imposé. Il y a également une optimisation qui doit se faire sur l'angle à donner à l'antenne TX3.

Pour ce faire, le choix des différents paramètres (positions des antennes TX1 et angle de l'antenne TX3) a été effectué en testant toutes les possibilités et en leur attribuant un score. Le paramètre donnant le meilleur score est ensuite conservé.

Le score a été défini avec comme objectif de minimiser le nombre d'emplacements en dessous du seuil minimum de connexion. Les scores sont donc $-1 * N$, N étant le nombre de cases en dessous de ce seuil. Si aucune case n'est déconnectée, l'objectif est alors de maximiser le débit. Le score vaut alors la somme du débit de chacune des cases.

L'angle de TX3 est le premier paramètre qui a été optimisé. En essayant tous les angles avec un pas de 1 degré, l'angle trouvé est -0.541052 radiant, donnant la heatmat suivante :

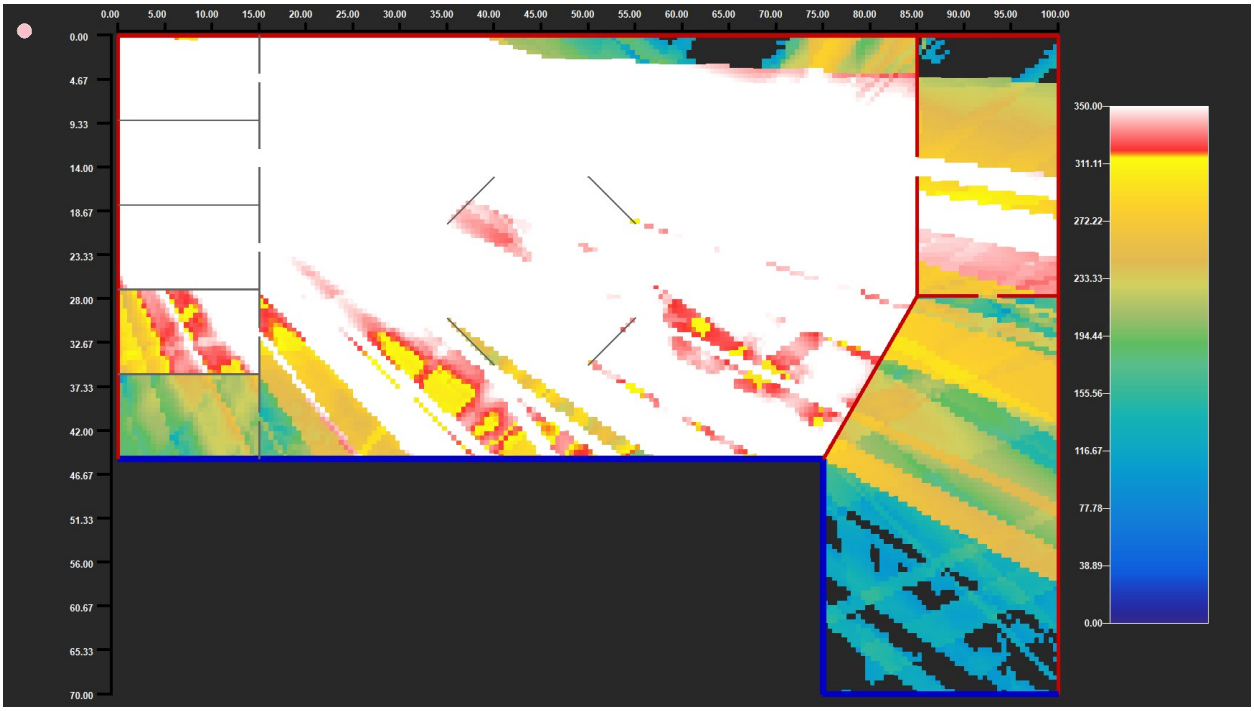


FIGURE 13 – Première optimisation

Une première antenne TX1 a ensuite été posée en essayant toutes les cases comme position. La position donnant le score le plus élevé est (91,-41), donnant le résultat illustré à la Figure 14. Toutes les cases sont alors au-dessus du seuil de connexion, ce qui change le gradient de couleur et peut donner l'impression que certaines zones ont un débit plus faible.

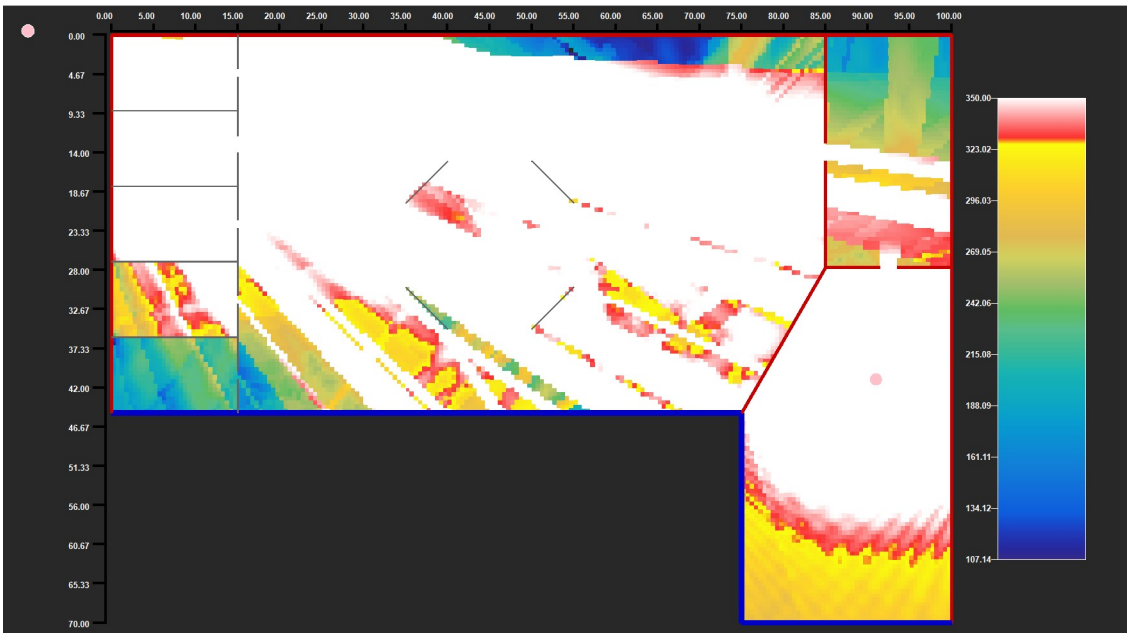


FIGURE 14 – Pose d'une première antenne TX1

L'angle de TX3 a ensuite été optimisé une seconde fois en appliquant la même méthode qu'auparavant, donnant un angle de -0.62858 radiant [9].

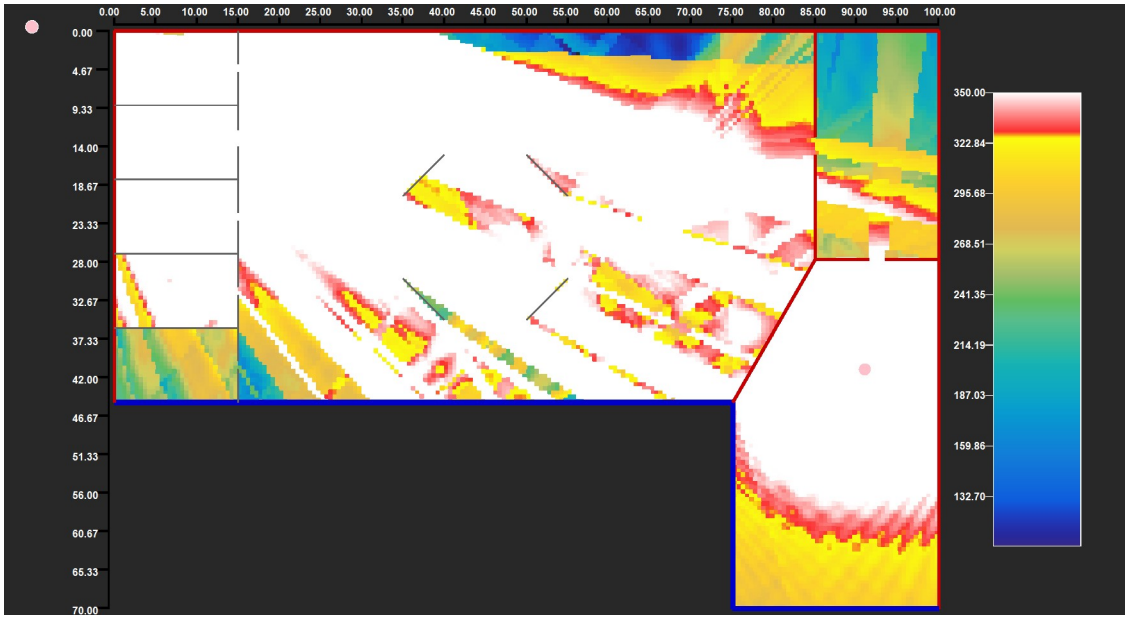


FIGURE 15 – Antenne TX1 avec optimisation de l'angle de TX3

9. Illustration en annexes, figure 24 et 25

Cette configuration permet déjà de couvrir toute l'usine avec un débit suffisant (supérieur ou égal à 100 Mb/s).

Ensuite, pour le cas où un débit supérieur serait nécessaire, la meilleure position pour une 2e antenne TX1 a été déterminée, toujours avec la même méthode, donnant une position de (88, -7), donnant le résultat suivant [10] :

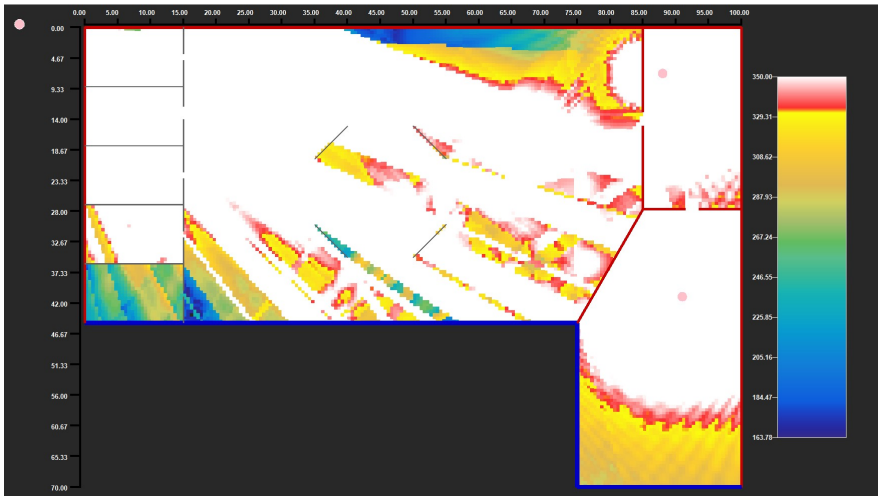


FIGURE 16 – Pose d'une deuxième antenne TX1

Ce qui donne en dBm [11] :

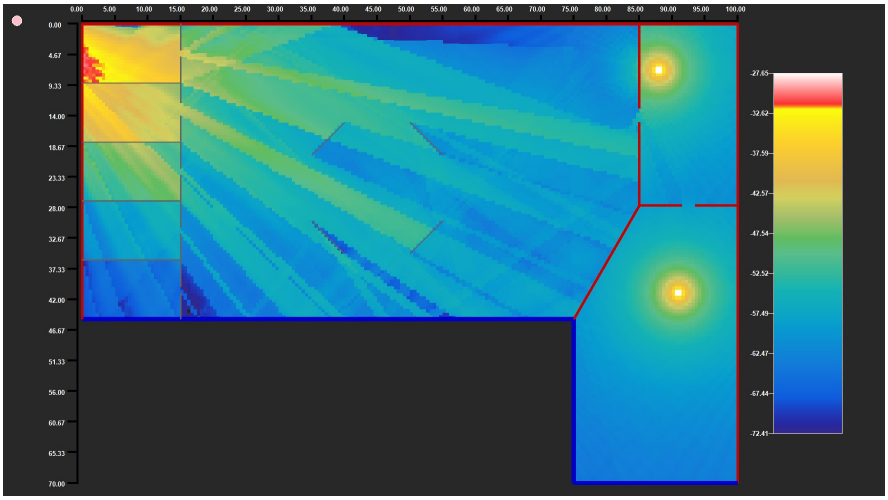


FIGURE 17 – Pose d'une deuxième antenne TX1, résultat en dBm

10. Mise en annexe, figure 26
11. Mise en annexe, figure 27

Il peut être observé qu'à part aux emplacements des antennes TX1 (c'est-à-dire dans la zone d'interdiction), le seuil légal d'exposition n'a pas été dépassé.

Une deuxième antenne n'est pas nécessaire pour couvrir toute l'usine et répondre au cahier des charges, mais permet néanmoins d'augmenter significativement la vitesse de la connexion dans le coin supérieur droit.

5 Nouvelle situation

Cette section traite d'une nouvelle situation : quel serait l'impact de l'ajout d'une seconde usine en dessous de la première, si seule l'antenne TX3 avec ses paramètres initiaux était présente ?

L'ajout de la seconde usine mène à la heatmap suivante :

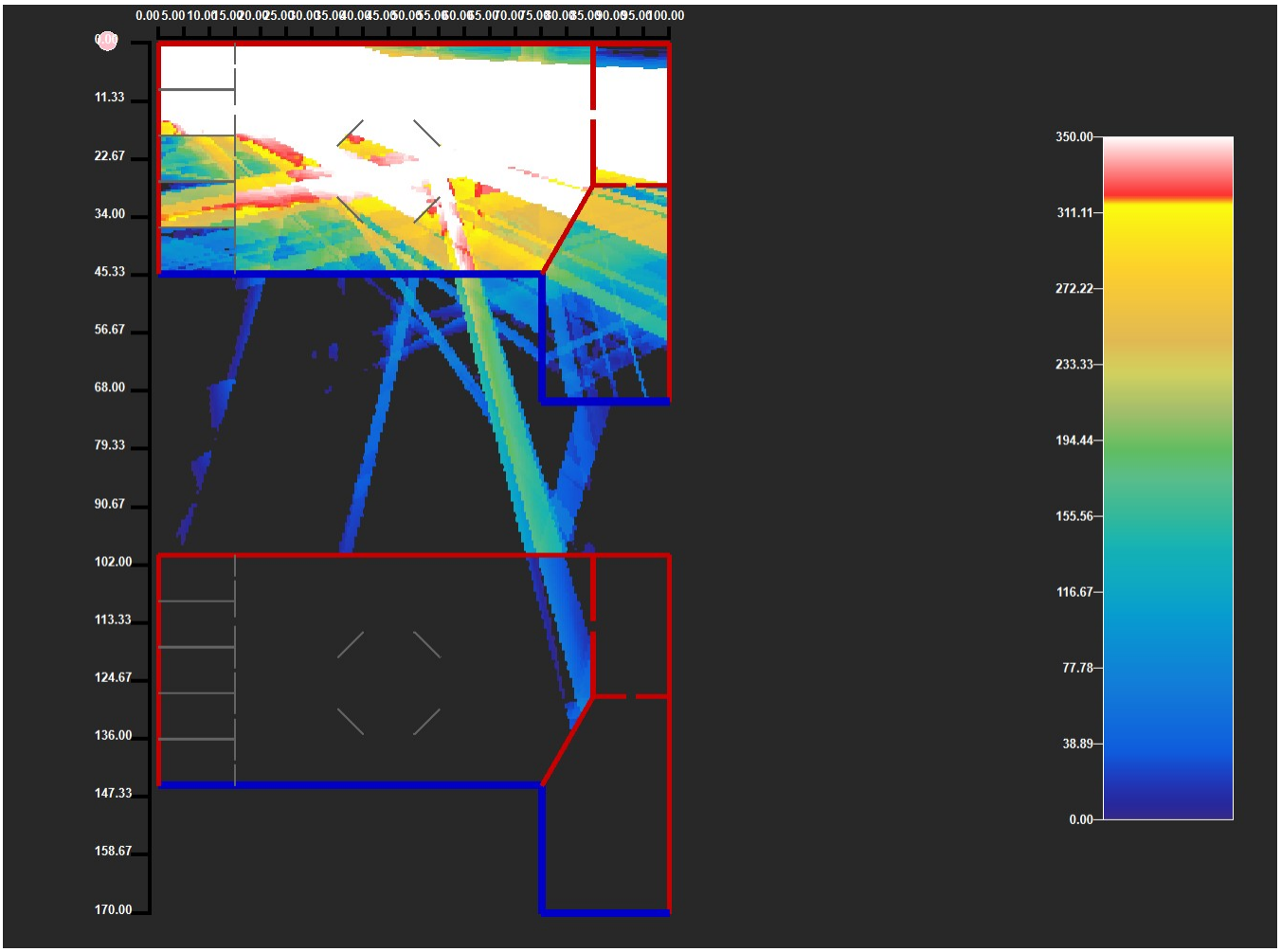


FIGURE 18 – Cas avec une deuxième usine à coté,identique à la première

Aucune différence notable avec la situation sans la seconde usine n'apparait. La 2e usine étant trop éloignée, les rayons revenant à la première usine après un rebond sur la seconde sont tellement affaiblis que leur impact est négligeable.

6 Conclusion

Pour terminer, les objectifs ont été atteints, autant au niveau de la norme de la puissance des champs électromagnétiques qui respectent les seuils légaux imposés que pour le débit binaire minimal (100 Mb/s).

Les points fort du code sont le temps de calcul, l'interface graphique et l'adaptabilité à de nouvelles situations (il suffit de changer un fichier .txt contenant les murs et les types de murs). Cependant, le point faible du code est que sa structure a dû s'adapter à la syntaxe de QtCreator, nuisant à la lisibilité du code.

Deuxième partie
Annexes

A Résolution du cas B à une réflexion

A.0.1 Cas à une réflexion : Trajet B

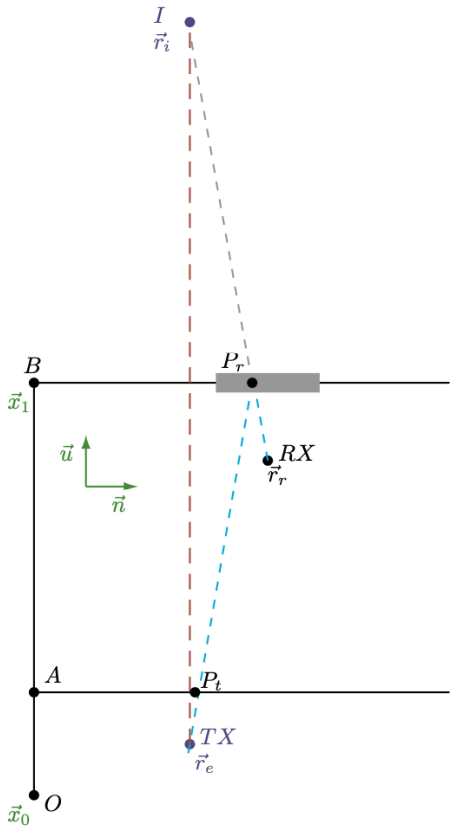


FIGURE 19 – Chemin du rayon dans le cas B pour une réflexion

Dans un premier temps, il convient de calculer la position de l'antenne image. Celle-ci est symétrique à Tx par rapport au mur horizontal de coin B. La coordonnée de I est donc (32 ;150). Il y a deux parties à traiter, une avec transmission et l'autre avec réception. Pour la réflexion :

Vecteur reliant I et RX : $\vec{v} = (47 - 32; 65 - 150) = (15; -85)$

Norme de $\vec{v}$: $\sqrt{15^2 + (-85)^2} = 86.3134 \rightarrow (\frac{15}{86.3134}; \frac{-85}{86.3134}) = (0.1738; -0.9848)$

Coordonnée de l'intersection avec le mur : $P_r = I + \lambda * \frac{\vec{v}}{\|\vec{v}\|} \rightarrow (x; 80) = (32; 150) + \lambda * \frac{\vec{v}}{\|\vec{v}\|}$

$\lambda = \frac{80-150}{-0.9848} = 71.0804 \rightarrow x = 32 + 71.0804 * 0.1738 = 44.3538 \rightarrow P_r = (44.3538; 80)$

Distance entre P_r et RX : $\sqrt{(47 - 44.14)^2 + (65 - 80)^2} = 15.2316$

Projection du vecteur normé sur la normale au mur : $\cos\theta_i = < \frac{\vec{v}}{\|\vec{v}\|}; (0; -1) >= 0.9848$

$\sin\theta_i = \sqrt{1 - (\cos\theta_i)^2} = 0.1737 \rightarrow \sin\theta_t = \sqrt{\frac{1}{\epsilon_r}^a} \sin\theta_i = 0.0793 \rightarrow \cos\theta_t = 0.9968$

Distance du trajet effectué dans le mur de 15 cm d'épaisseur : s = $\frac{0.15}{\cos\theta_t} = 0.1505$ m

Coefficient de réflexion : $\Gamma = \frac{Z_m{}^b \cos\theta_i - Z_0 \cos\theta_t}{Z_m \cos\theta_i + Z_0 \cos\theta_t} = -0.3795 + 0.0166i$

Réflexion : $\Gamma - (1 - \Gamma^2) * \frac{\Gamma e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}}{1 - \Gamma^2 e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}} = -0.2044 + 0.1511i$

a. La permittivité relative est donnée dans l'énoncé de l'exercice et vaut 4.8
b. Valeur numérique donnée dans le corrigé de l'exercice : (171.57 + j6.65)Ω

La réflexion connue, il est maintenant temps de passer à la partie transmission. Pour cela, les deux points à considérer sont l'émetteur TX ainsi que le point P_r dont les coordonnées ont été calculés.

Vecteur reliant Tx et P_r : $\vec{v} = (44.3538 - 32; 80 - 10) = (12.3538; 70)$

Norme de $\vec{v} = \sqrt{12.14^2 + (70)^2} = 71.0817 \rightarrow (\frac{12.3538}{71.0817}; \frac{70}{71.0817}) = (0.1738; 0.9848)$

Projection du vecteur normé sur la normale au mur : $\cos\theta_i = < \frac{\vec{v}}{\|\vec{v}\|}; (0; 1) >= 0.9848$

$\sin\theta_i = \sqrt{1 - (\cos\theta_i)^2} = 0.1737 \rightarrow \sin\theta_t = \sqrt{\frac{1}{\epsilon_r}^a} \sin\theta_i = 0.0793 \rightarrow \cos\theta_t = 0.9968$

Distance du trajet effectué dans le mur de 15 cm d'épaisseur : s = $\frac{0.15}{\cos\theta_t} = 0.1505$ m

Coefficient de réflexion : $\Gamma = \frac{Z_m{}^b \cos\theta_i - Z_0 \cos\theta_t}{Z_m \cos\theta_i + Z_0 \cos\theta_t} = -0.3795 + 0.0166i$

Transmission = $\frac{(1-\Gamma^2)e^{-\gamma_m * s}}{1-\Gamma^2 e^{-2\gamma_m * s} e^{j\beta 2s * \sin\theta_i * \sin\theta_t}} = 0.6910 + 0.2480i$

Distance totale parcourue par le rayon : Distance entre TX et P_r + distance entre P_r et RX $\rightarrow 15.2316 + 71.0817 = 86.3133$

a. La permittivité relative est donnée dans l'énoncé de l'exercice et vaut 4.8
b. Valeur numérique donnée dans le corrigé de l'exercice : (171.57 + j6.65)Ω

Les coefficients de réflexion et de transmission étant connus, il est désormais possible de calculer le champ électrique de la composante multi-trajet. Pour ce faire, la loi de Friis est utilisée : $\underline{E_n} = \Gamma_1\Gamma_2\Gamma_3...T_1T_2T_3\sqrt{60 G_{TX} P_{TX}} \frac{e^{-j\beta d_n}}{d_n}$ [12]

$$\underline{E_3} = (0.6910 + 0.2480i) * (-0.2044 + 0.1511i)\sqrt{60 * 1,64.10^{-3}} \frac{e^{-j\frac{2\pi * 868.3 * 10^6}{3 * 10^8} 86.3133}}{86.3133}$$

$$\underline{E_3} = -3.5143 * 10^{-4} + 5.8005 * 10^{-4} i$$

12. Dans cet exercice, le produit $G_{TX}P_{TX}$ vaut 2.15 dBm soit environ 1.64 mW

B Illustrations

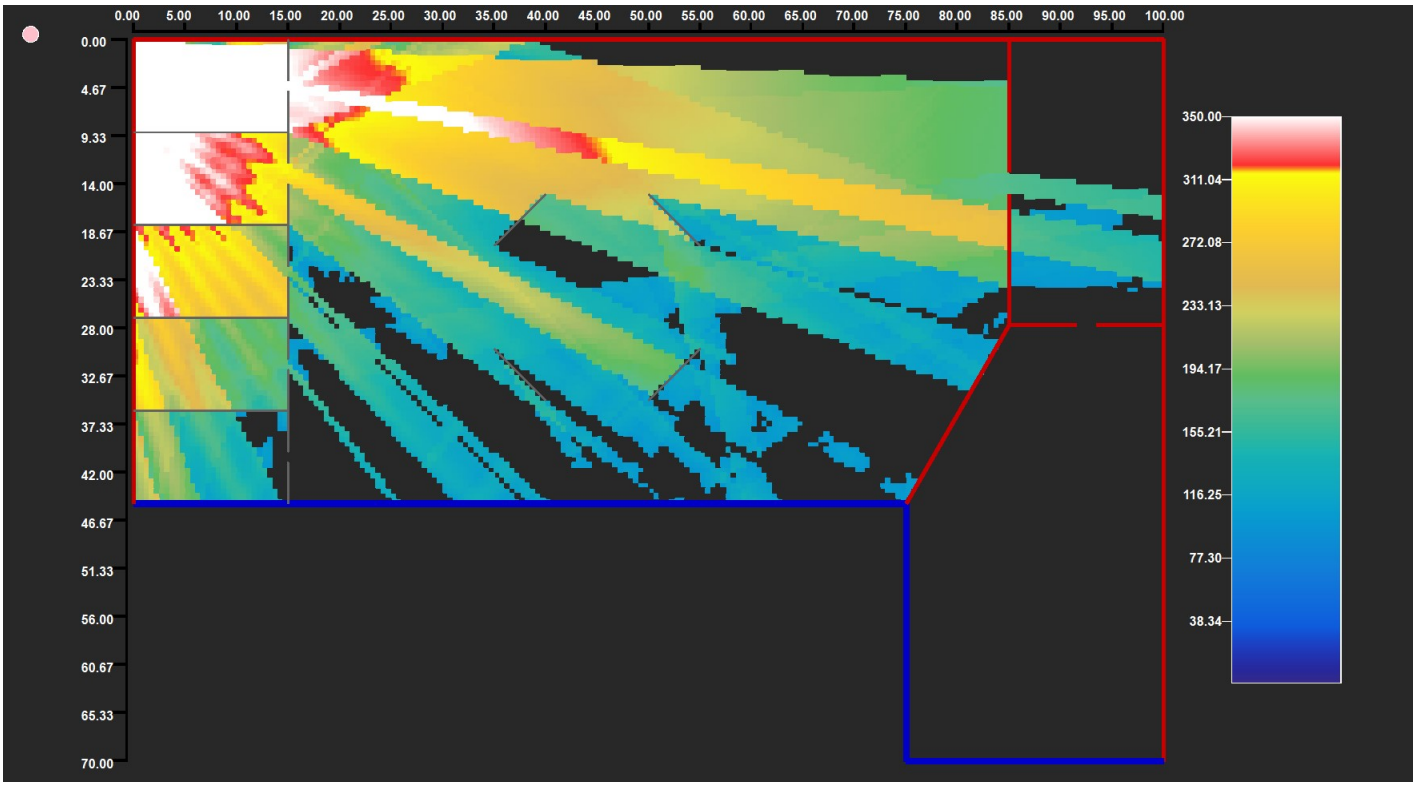


FIGURE 20 – Débit binaire pour antenne TX2 seule

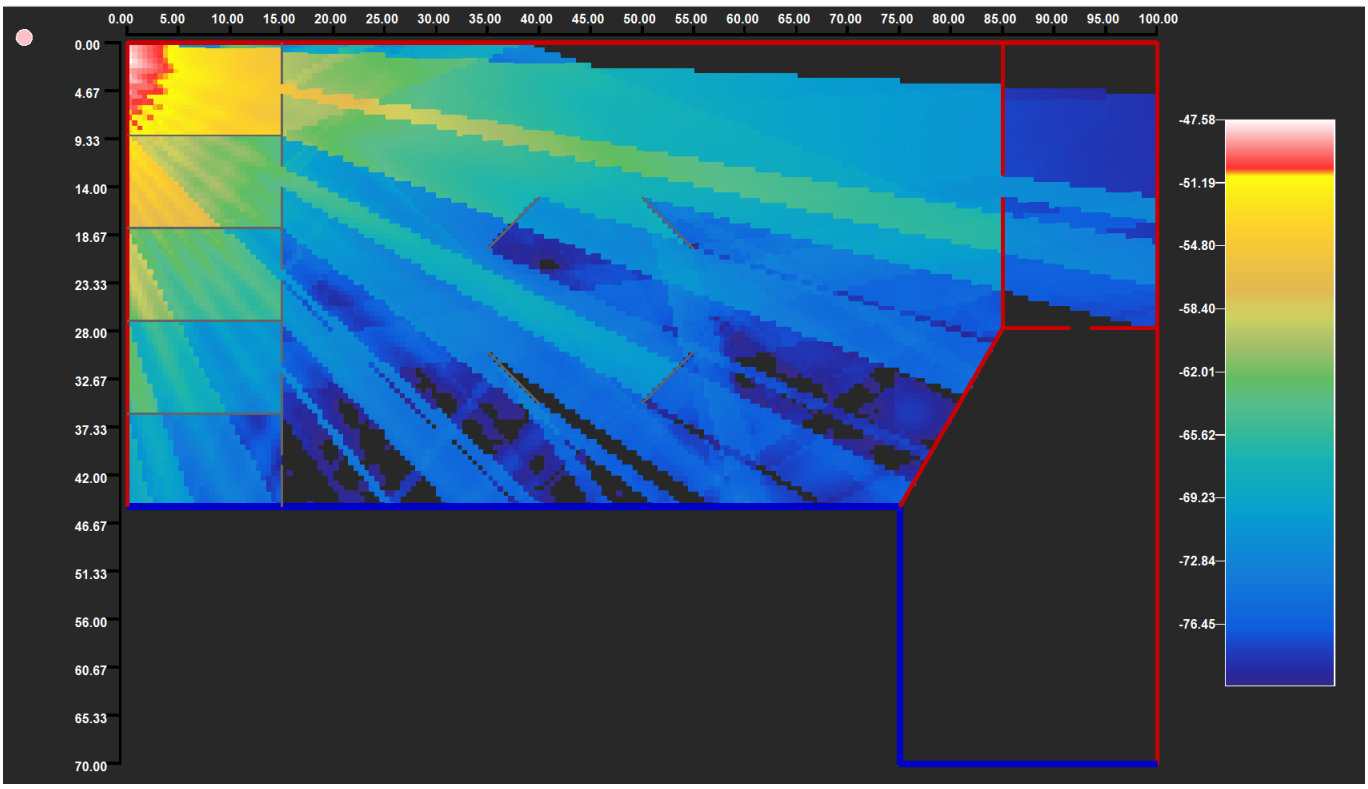


FIGURE 21 – Puissance pour antenne TX2 seule

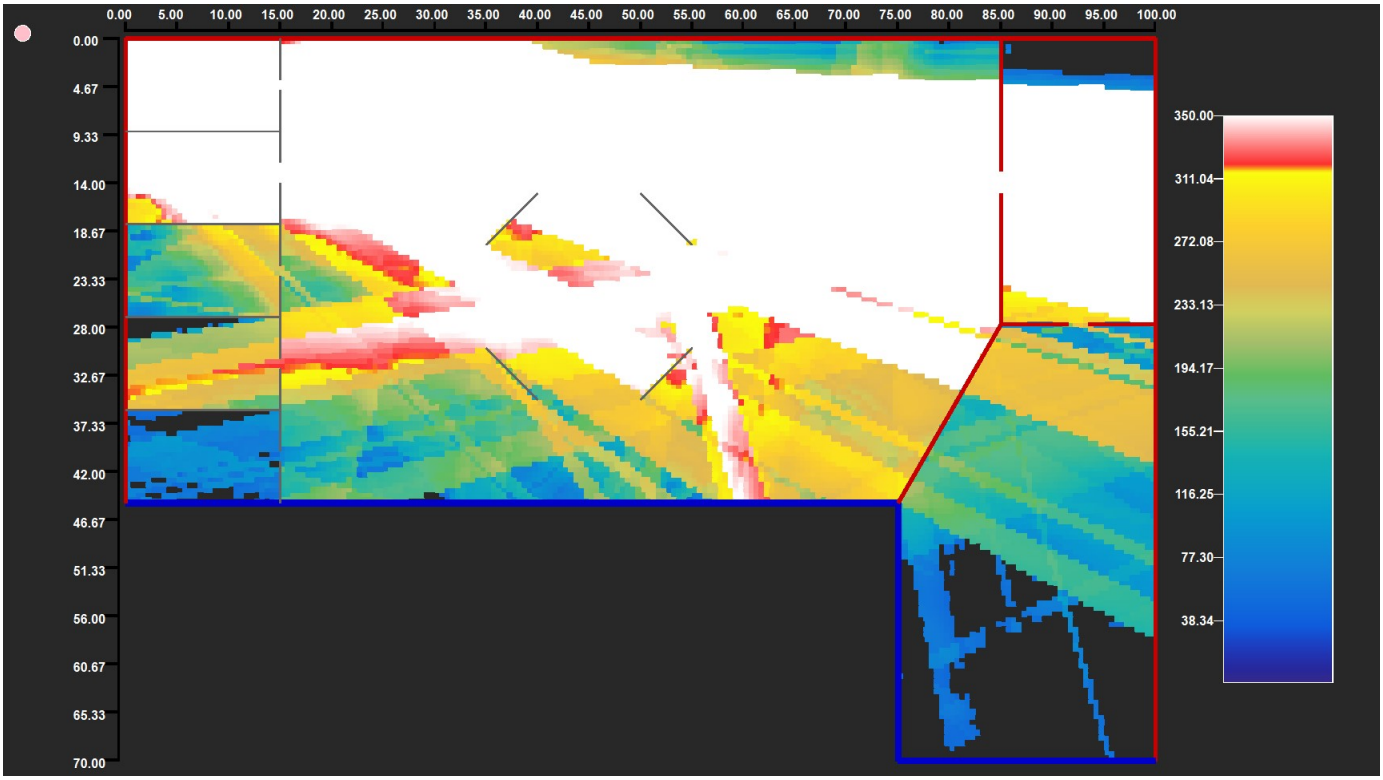


FIGURE 22 – Débit binaire antenne TX3 avec paramètres de base

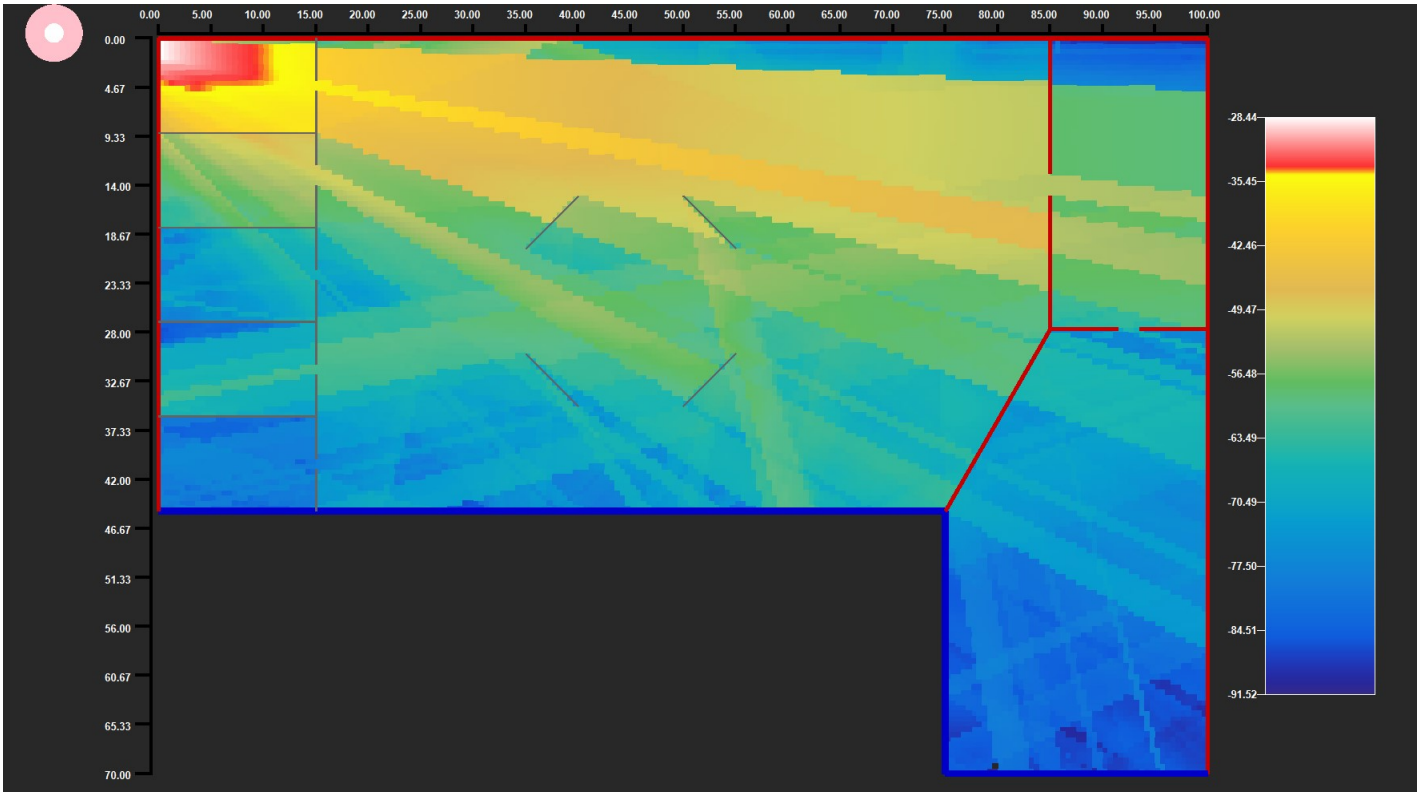


FIGURE 23 – Puissance pour antenne TX3 avec paramètres de base

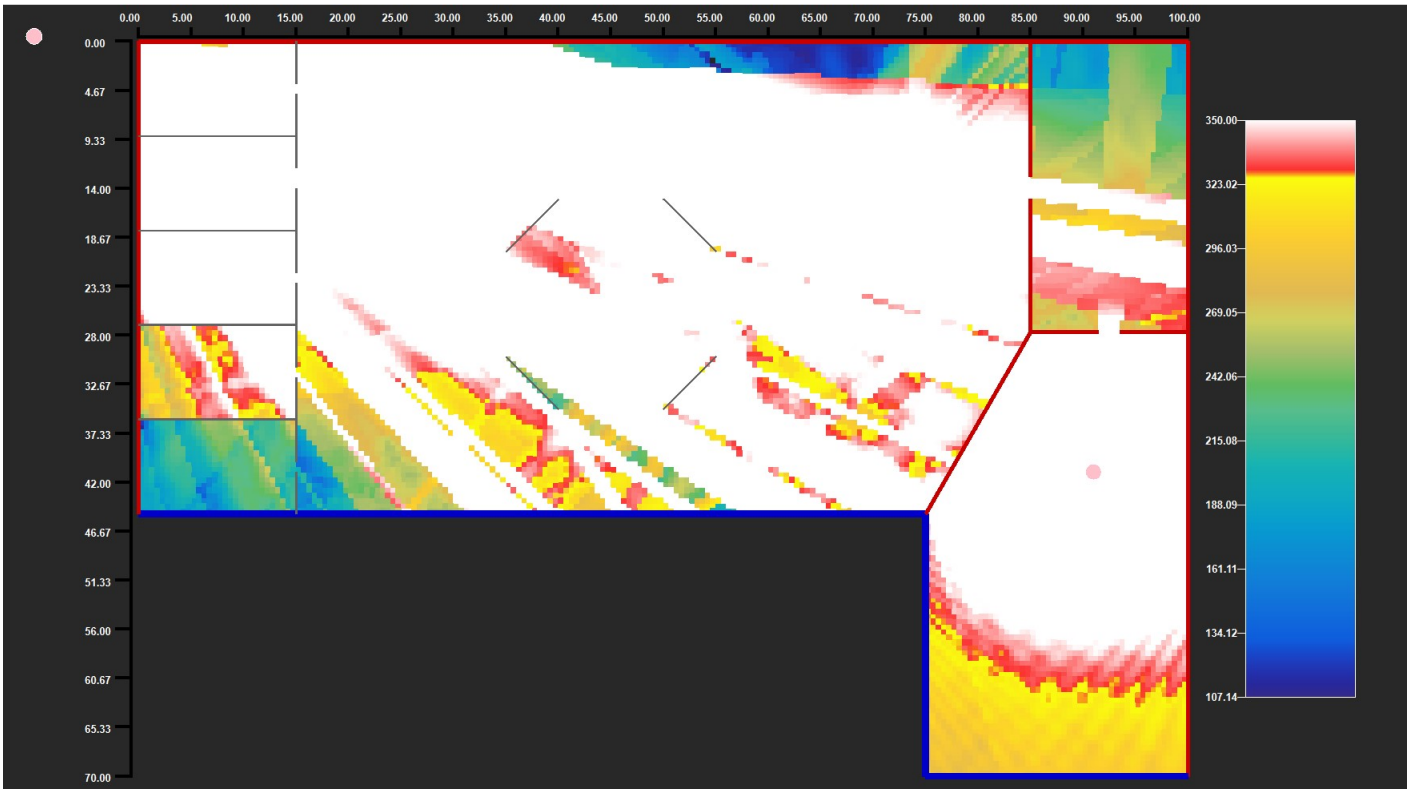


FIGURE 24 – Pose d'une première antenne TX1

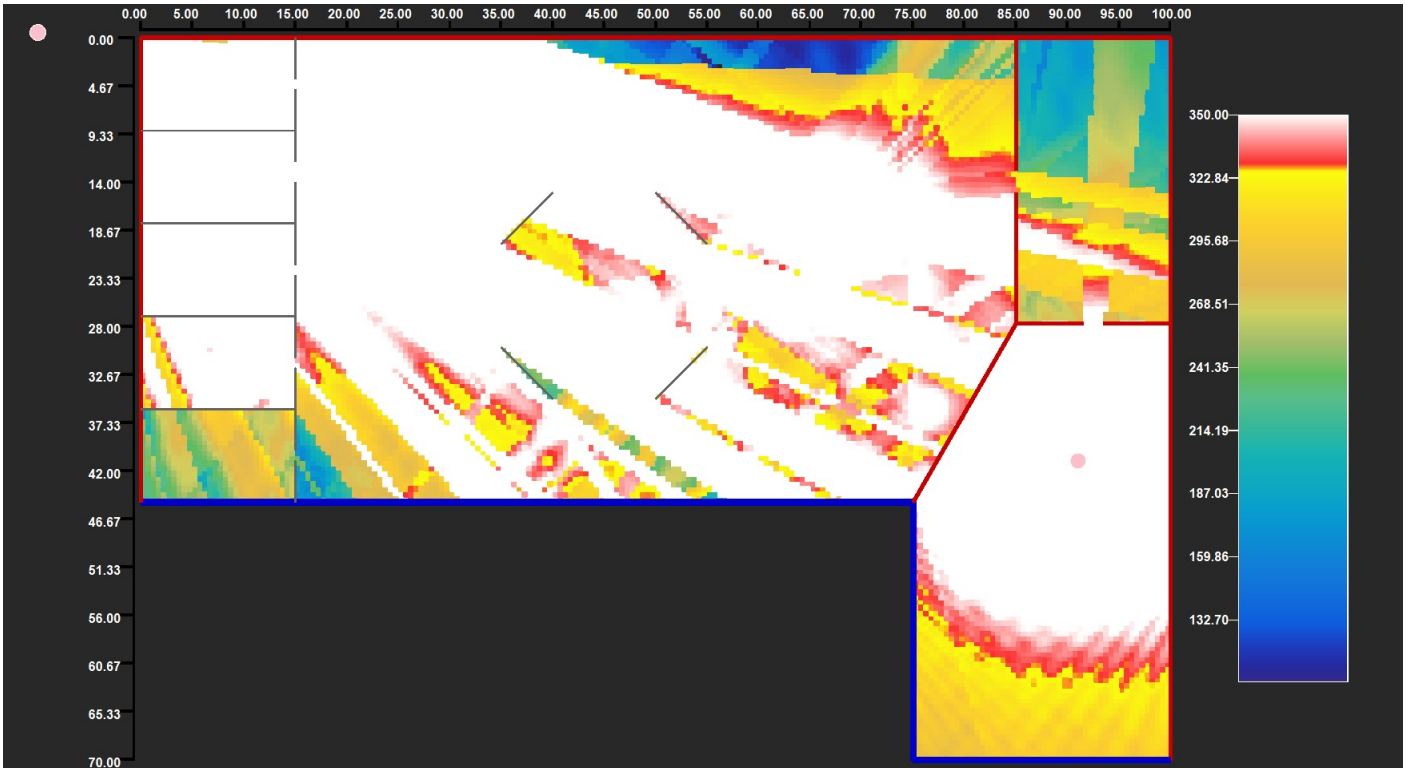


FIGURE 25 – Antenne TX1 avec optimisation de l'angle de TX3

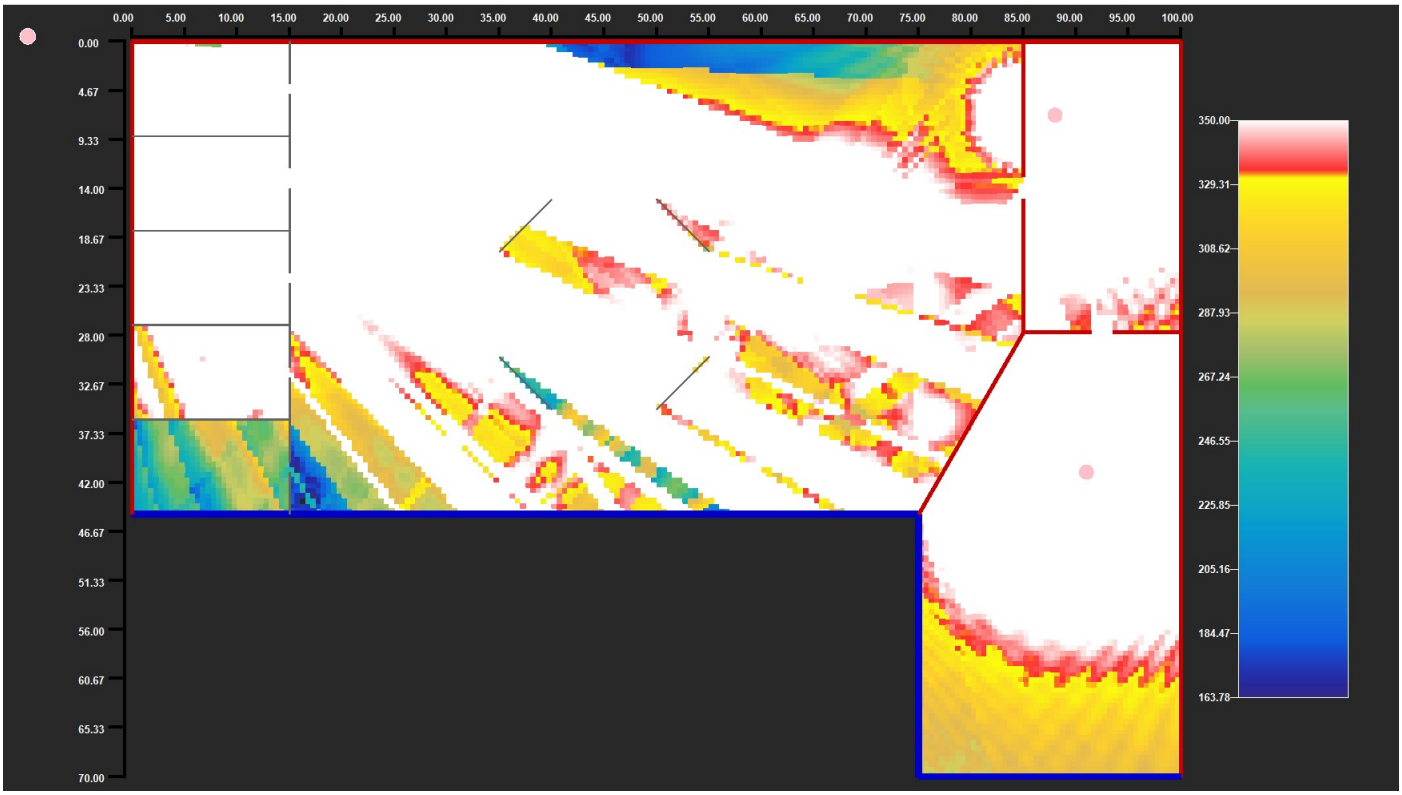


FIGURE 26 – Débit binaire lors de l'ajout de 2 antennes TX1 (optimisé)

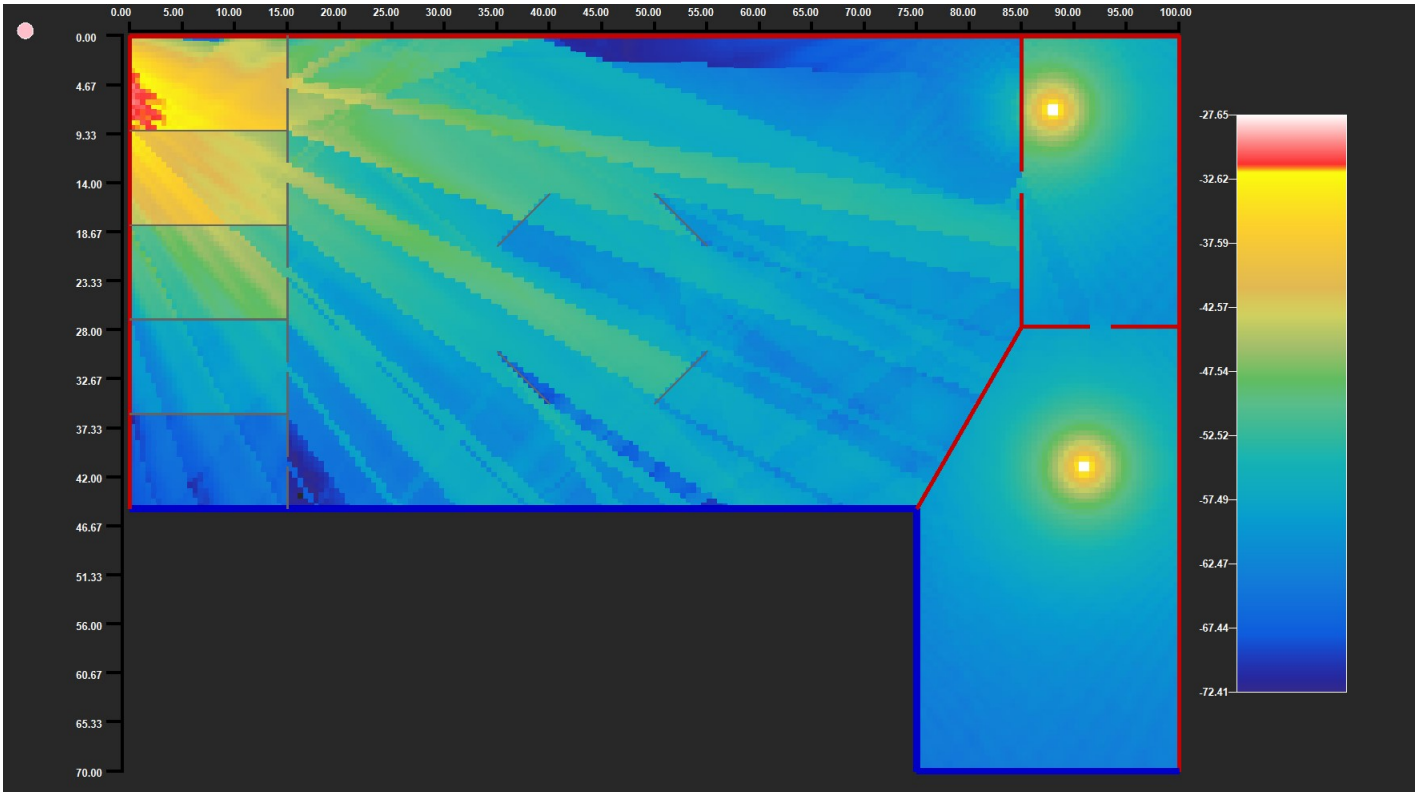


FIGURE 27 – Puissance pour antenne TX3 avec 2 antennes TX1 (optimisé)

C Code

antenna.h :

```
#ifndef ANTENNA_H
#define ANTENNA_H

#include "qpoint.h"
#include <complex>

 class Antenna
{
public:
    Antenna(double f, double l, double prx, QPointF pos, double grxy = 1);
    double ra, prx, grxy;
    QPointF pos;
    virtual std::complex<double> PtxGtx(float delta);
};

class TX3:public Antenna
{
public:
    TX3(double f, double l, double prx, double delta, double phi3db, double GMax, QPointF pos);
    std::complex<double> PtxGtx(float delta) override;
    double delta, phi3db, GMaxDB;
};

#endif // ANTENNA_H
```

antennamanager.h :

```
#ifndef ANTENNAMANAGER_H
#define ANTENNAMANAGER_H

#include "Antenna.h"
#include <QList>
QList<Antenna*> GetAntennas();
void DeleteAntennas(QList<Antenna*> antennas);
Antenna* GenerateTx3();
double GetLambda();
#endif // ANTENNAMANAGER_H
```

boardwidget.h :

```
#ifndef BOARDWIDGET_H
#define BOARDWIDGET_H

#include <QWidget>
#include <QPainter>
#include <QBrush>
#include <QPolygon>
#include <QColor>
#include <iostream>
#include "colorcase.h"
#include "ray.h"
#include "wall.h"

class BoardWidget : public QWidget
{
public:
    BoardWidget(Wall **_walls, int _wallCount, QList<Ray> _rays, QList<ColorCase> _cases);
private:
    QList<Ray> rays;
    QList<ColorCase> cases;
    Wall **walls;
    int numWalls;
protected:
    void paintEvent(QPaintEvent *event) override;
};



#endif // BOARDWIDGET_H
```

colorcase.h :

```
#ifndef COLORCASE_H
#define COLORCASE_H


class ColorCase
{
public:
    ColorCase(float _x, float _y, float _value, float _width);
```

```
    float x, y, value, width;
};

#endif // COLORCASE_H
```

filemanager.h :

```
#ifndef FILEMANAGER_H
#define FILEMANAGER_H

#include "wall.h"
#include "walltype.h"
#include <string>

void ReadWallsFromFile(const std::string fileName, Wall**& walls, int& numWalls,std::unordered_map<std:
std::unordered_map<std::string, WallType*> ReadWallTypesFromFile(const std::string fileName);

#endif // FILEMANAGER_H
```

graphicsmanager.h :

```
#ifndef GRAPHICSMANAGER_H
#define GRAPHICSMANAGER_H

#include "colorcase.h"
#include "qpainter.h"
#include "ray.h"
#include "wall.h"

void Display(QPainter *painter, Wall **walls, int numWalls, QList<Ray> rays, QList<ColorCase> cases, fl
void CreateBoard();
#endif // GRAPHICSMANAGER_H
```

heatmapmanager.h :

```
#ifndef DISPLAYMANAGER_H
#define DISPLAYMANAGER_H
#include "antenna.h"
#include "colorcase.h"
#include "ray.h"
#include "wall.h"
```

```
7
8  void GenerateRaysAndPower(Wall **walls, int wallCount, QList<Antenna*> *antennas, QList<Ray> *rays, std
9  void GetHeatmap(Wall **walls, int wallCount, QList<Antenna*> *antennas, QList<ColorCase> *cases);
10 float DBmToSpeed(float dbm);
11 bool isPointInsideBuilding(QPointF point, Wall** walls, int wallCount);
12 #endif // DISPLAYMANAGER_H
```

mainwindow.h :

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5
6  QT_BEGIN_NAMESPACE
7  namespace Ui { class MainWindow; }
8  QT_END_NAMESPACE
9
10 class MainWindow : public QMainWindow
11 {
12     Q_OBJECT
13 public:
14     MainWindow(QWidget *parent = nullptr);
15     ~MainWindow();
16 private:
17     Ui::MainWindow *ui;
18 };
19 #endif // MAINWINDOW_H
```

optimiser.h :

```
1  #ifndef OPTIMISER_H
2  #define OPTIMISER_H
3
4  void Optimise();
5
6  #endif // OPTIMISER_H
```

params.h :

```
1  #ifndef PARAMS_H
2  #define PARAMS_H
```

```
3
4  #define MAX_RECU 2
5
6  #define TARGET proj //8 pour ex 8, proj pour la situation du projet
7
8
9
10 #if TARGET != 8 //Factory
11
12 #define F (26 * pow(10,9))
13
14 #define WallFilePath  "../Test3/WallsV1.txt"
15 #define WallTypeFilePath "../Test3/WallTypes.txt"
16
17 #define TX_RX std::make_pair(QPointF(-10.0, 0.5),QPointF(0.25, -0.25))
18
19 #define ANTENNA 3 //3 for tx3
20 #define PrxDbm 35.0
21 #define GMAXDB 21.5836
22 #define PHI3DB 30.0/360.0*2.0*PI
23
24 #else //EX 8
25
26 #define F 868.3 * pow(10,6)
27
28 #define TX_RX std::make_pair(QPointF(32, 10),QPointF(47, 65))
29
30 #define WallFilePath  "../Test3/Ex8.1.txt"
31 #define WallTypeFilePath "../Test3/WallTypesExo8.txt"
32
33 #define PrxDbm 2.15
34 #define RA 73.0
35 #define GMAXDB 0
36 #define PHI3DB 0
37
38 #endif
39
40 #define Prx (pow(10, PrxDbm / 10) / 1000)
41
42
43 #define SQUARE_SIZE 0.5 //0 for ray only
44 #define NUM_THREADS 15
45
```

```
#define PI 3.14159

#define POWER_MODE 1 // 0 : sum, 1 : take largest
#define DISPLAY_THREAD true
#endif // PARAMS_H
```

powercalculator.h :

```
#ifndef POWERCALCULATOR_H
#define POWERCALCULATOR_H

#include "ray.h"
#include "wall.h"
#include "antenna.h"
QList<Wall> wallsBetweenTwoPoints(QPointF pointA, QPointF pointB, QList<Wall> AllWalls, QList<Wall> wal

std::complex<double> Reflection(QPointF first_point, QPointF second_point, QPointF wall_normal, float e
std::complex<double> GetPower(QList<Ray> rays, Wall **allWalls, int wallCount, double f, Antenna *an);

#endif // POWERCALCULATOR_H
```

ray.h :

```
#ifndef RAY_H
#define RAY_H

#include <QList>
#include <QPointF>
#include <complex>
#include "wall.h"
class Ray
{
public:
    Ray();
    QList<QPointF> points;
    QList<Wall> walls;
};

#endif // RAY_H
```

raytracingmanager.h :

```
#ifndef RAYTRACINGMANAGER_H
#define RAYTRACINGMANAGER_H

#include "qpainter.h"
#include "qpoint.h"
#include "ray.h"
#include "wall.h"

QPolygonF RayToPolygon(const Ray& ray, float centerX, float centerY, float scale, float screenHeight, fl
QList<Ray> GenerateRays(Wall **walls, int numWalls, QPointF startPos, QPointF endPos, int maxRecu);

QPointF FindRayWallIntersection(QPointF TX, QPointF mPoint, Wall wall, bool* valid);
#endif // RAYTRACINGMANAGER_H
```

wall.h :

```
#ifndef WALL_H
#define WALL_H

#include "walltype.h"
#include <QPointF>

class Wall
{
public:
    Wall(float x0, float y0, float x1, float y1,  WallType type);
    WallType wallType;
    float x0, y0, x1, y1, length;
    bool outer;
    QPointF normal;
    bool operator==(const Wall& other) const {return (this->x0 == other.x0 && this->y0 == other.y0 && th
};

#endif
```

walltype.h :

```
#ifndef WALLTYPE_H
#define WALLTYPE_H
#include <QColor>
#include <complex>
```

```
5
6  class WallType
7  {
8  public:
9      WallType(std::string name,float permitivity, float width, float conductivity, QColor color);
10     QColor color;
11     std::string name;
12     float permitivityRela, width, conductivity;
13     std::complex<double> Z;
14     std::complex<double> Z0;
15     std::complex<double> gamma;
16     std::complex<double> beta0;
17     std::complex<double> Z1;
18 };
19
20 #endif
```

antenna.cpp :

```
1  #include "antenna.h"
2  #include "params.h"
3  #include <cmath>
4  #include "params.h"
5  #include <iostream>
6
7  Antenna::Antenna(double f, double l, double _prx, QPointF _pos, double _grxy)
8  {
9      ra = 720 * PI / 32; //Pour une antenne dipôle
10     //ra = 80 *pow(PI * l / lambda,2); //Ra est changé après l'initialisation dans les autres cas
11     grxy = _grxy;
12     prx = _prx;
13     pos = _pos;
14 }
15
16 TX3::TX3(double f, double l, double _prx, double _delta, double _phi3db, double _GMaxDB, QPointF _pos):
17     Antenna(f, l, _prx, _pos), delta(_delta), phi3db(_phi3db), GMaxDB(_GMaxDB)
18 {}
19
20 std::complex<double> TX3::PtxGtx(const float phi)  //Implémente la directivité de l'antenne tel que don
21 {
22     double Gdb = (GMaxDB - 12.0 * pow(((phi - delta) / phi3db),2));
23     double G = pow(10.0, Gdb / 10.0);
```

```
24      return prx * G;
25  }
26
27  std::complex<double> Antenna::PtxGtx(const float delta)
28  {
29      return prx*grxy;
30  }
31
```

---

antennamanager.cpp :

```
1   #include "antennamanager.h"
2   #include "params.h"
3   #include <QList>
4
5   Antenna* GenerateTx1(QPointF pos)
6   {
7       double lambda = 3.0 * pow(10,8) / F;
8       float pDbm = 20;
9       float p = (pow(10, pDbm / 10) / 1000);
10      return new Antenna(F, lambda/2, p, pos, 1.5);
11  }
12  double GetLambda()
13  {
14      return 3.0 * pow(10,8) / F;
15  }
16  #if TARGET != 8
17  Antenna* GenerateTx3()
18  {
19      //Basic angle  : -0.1 First optimisation : -0.726056 Second : -0.575958
20      TX3 *toReturn = new TX3(F, GetLambda()/2, Prx, -0.1,PHI3DB, GMAXDB, TX_RX.first);
21      toReturn->ra = 73;
22      return toReturn;
23  }
24  #endif
25  Antenna* GenerateTx2()
26  {
27      return new Antenna(F, GetLambda()/2, Prx, TX_RX.first, 1.5);
28  }
29
30  QList<Antenna*> GetAntennas()
31  { //Retourne les bonnes antennes selon les paramètres donnés dans params.h
32      QList<Antenna*> antennas;
```

```
33
34  #if TARGET != 8
35  #if ANTENNA == 3
36      antennas.append(GenerateTx3());
37  #else
38      antennas.append(GenerateTx2());
39  #endif
40
41      QPointF posTx1(90,-36);
42      antennas.append(GenerateTx1(posTx1)); //First antenna
43       QPointF posTx1V2(88, -3);
44      antennas.append(GenerateTx1(posTx1V2));
45
46
47  #else
48
49      QPointF txPos = TX_RX.first;
50      double lambda = 3.0 * pow(10,8) / F;
51      antennas.append(new Antenna(F, lambda/2, Prx, txPos));
52      antennas[antennas.length()-1]->ra = RA;
53  #endif
54      return antennas;
55  }
56  void DeleteAntennas(QList<Antenna*> antennas)
57  {//Supprime les antennes pour éviter des memory leaks
58      for (int i = 0; i < antennas.length()-1; i++)
59      {
60          delete antennas[i];
61      }
62  }
```

boardwidget.cpp :

```
1   #include "boardwidget.h"
2   #include "wallmanager.h"
3   #include "graphicsmanager.h"
4   #include "params.h"
5   #include <QPointF>
6
7
8   BoardWidget::BoardWidget(Wall **_walls, int _wallCount, QList<Ray> _rays, QList<ColorCase> _cases) :
9       QWidget{nullptr}, walls(_walls), numWalls(_wallCount), rays(_rays), cases(_cases) {}
10
```

```
void BoardWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    Display(&painter, walls, numWalls, rays, cases, width(), height());
}
```

colorcase.cpp :

```
#include "colorcase.h"

ColorCase::ColorCase(float x, float y, float value, float width):x(x), y(y), value(value), width(width)
{}
//Utilisé pour stoquer la puissance ou le débit et l'afficher dans une heatmap
```

filemanager.cpp :

```
#include "filemanager.h"
#include "qdebug.h"
#include "qline.h"
#include "wall.h"
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>

WallType* FindWallType(const std::string& name, std::unordered_map<std::string, WallType*> wallTypeMap)
{  //Trouve le type de mur correspondant au nom donné dans le fichier
    auto iter = wallTypeMap.find(name);
    if (iter != wallTypeMap.end())
    {
        return iter->second;
    }
    else
    {
        return nullptr;
    }
}

bool pointIsSurrounded(QPointF point, Wall** walls, int numWalls) //Détermine si un point est entourré
//Utilisé pour déterminer quel mur est un mur extérieur de l'usine et lequel n'en est pas un
{
    bool hasTop = false, hasBottom = false, hasLeft = false, hasRight = false;
```

```
27      for (int i = 0; i < numWalls; i++) {
28          Wall* wall = walls[i];
29          bool inBetweenX = (wall->x0 >= point.x() != wall->x1 >= point.x());
30          bool inBetweenY = (wall->y0 >= point.y() != wall->y1 >= point.y());
31          if (inBetweenX)
32          {
33              if (wall->y0 > point.y() || wall->y1 > point.y())
34              {
35                  hasTop = true;
36              }
37              if (wall->y0 < point.y() || wall->y1 < point.y())
38              {
39                  hasBottom = true;
40              }
41          }
42          if (inBetweenY)
43          {
44              if (wall->x0 > point.x() || wall->x1 > point.x())
45              {
46                  hasRight = true;
47              }
48              if (wall->x0 < point.x() || wall->x1 < point.x())
49              {
50                  hasLeft = true;
51              }
52          }
53      }
54      return hasTop && hasBottom && hasLeft && hasRight;
55  }
56
57
58
59
60  void setOuterWalls(Wall**& walls, int& numWalls) {
61      // Détermine quel mur est un mur extérieur et lequel n'en est pas un
62      for (int i = 0; i < numWalls; i++)
63      {
64          if (pointIsSurrounded(QPointF(walls[i]->x0,walls[i]->y0), walls, numWalls)
65                  || pointIsSurrounded(QPointF(walls[i]->x1,walls[i]->y1), walls, numWalls))
66          {
67              walls[i]->outer = false;
68          }
69          else
```

```
70          {
71              walls[i]->outer = true;
72
73          }
74      }
75  }
76
77
78
79  void ReadWallsFromFile(const std::string filename, Wall**& walls, int& numWalls,std::unordered_map<std:
80  {
81      //Lis les murs à partir d'un fichier .txt
82      std::ifstream file(filename);
83
84      if (file.is_open())
85      {
86          std::string line;
87          int numLines = std::count(std::istreambuf_iterator<char>(file), std::istreambuf_iterator<char>(
88          file.seekg(0); //To go back at the begining of the file
89
90          walls = new Wall*[numLines];
91
92          while (std::getline(file, line))
93          {
94              std::stringstream sSt(line);
95              float x0, y0, x1, y1;
96              std::string typeName;
97
98              if (sSt >> x0 >> y0 >> x1 >> y1 >> typeName)
99              {
100                 WallType wallType = *FindWallType(typeName, types);
101                 Wall* wall = new Wall(x0, y0, x1, y1, wallType);
102                 walls[numWalls] = wall;
103                 numWalls++;
104             }
105         }
106         setOuterWalls(walls, numWalls);
107         file.close();
108     }
109     else
110     {
111         qDebug() << "Unable to open file!\n";
112     }
```

```
}

std::unordered_map<std::string, WallType*> ReadWallTypesFromFile(const std::string filename)
{
    //Lis les "types" de murs (béton, cloison, ...) à partir d'un fichier txt
    std::unordered_map<std::string, WallType*> wallTypeMap;
    std::ifstream file(filename);
    if (!file)
    {
        return wallTypeMap;
    }

    std::string line;
    while (std::getline(file, line))
    {
        std::stringstream sSt(line);
        std::string name;
        float permitivity, conductivity, width;
        int r, g, b;

        if (sSt >> name >> permitivity >> conductivity >> width >> r >> g >> b)
        {
            QColor color(r, g, b);
            WallType* wallType = new WallType(name, permitivity, width, conductivity, color);
            wallTypeMap[name] = wallType;
        }
    }
    return wallTypeMap;
}
```

graphicsmanager.cpp :

```
#include "graphicsmanager.h"
#include "antennamanager.h"
#include "boardwidget.h"
#include "colorcase.h"
#include "filemanager.h"
#include "heatmapmanager.h"
#include "params.h"
#define SCALE_DRECREASER 1.1 //Pour que les mus ne soient pas collés exactement contre les bords de l'é


```

```
12  QColor ColorGradient(float value, float minValue, float maxValue)
13  {
14      //Donne la couleur à employer à partir de la valeur en un point ansi que la valeur la plus basse et
15      //Inspiré d'un gradients de couleur matlab
16      if (value == minValue)
17      {
18          return QColor(40,40,40);
19      }
20      QColor colors[11] = {
21          QColor(53, 42, 135), QColor(15, 92, 221), QColor(18, 125, 216),
22          QColor(7, 156, 207), QColor(21, 177, 180), QColor(89, 189, 140),
23          QColor(165, 190, 107), QColor(225, 185, 82), QColor(252, 206, 46),
24          QColor(249, 251, 14), QColor(255, 255, 255)
25      };
26
27      int index = static_cast<int>((value - minValue) / (maxValue - minValue) * 10.0f);
28
29      index = std::max(0, std::min(index, 10));
30
31      float fraction = (value - minValue) / (maxValue - minValue) * 10.0f - index;
32      QColor color = colors[index].toHsv();
33      QColor nextColor = colors[index + 1].toHsv();
34      color.setHsvF(qBound(0.0, color.hueF() + fraction * (nextColor.hueF() - color.hueF()), 1.0),
35                    qBound(0.0, color.saturationF() + fraction * (nextColor.saturationF() - color.saturati
36                    qBound(0.0, color.valueF() + fraction * (nextColor.valueF() - color.valueF()), 1.0));
37      return color;
38  }
39
40  void CalculateCorners(Wall** walls, int numWalls, float* minX, float* minY, float* maxX, float* maxY)
41  {
42      //Trouve les 4 coins les plus éloignés, utile pour l'affichage des murs
43      *minX = walls[0]->x0;
44      *minY = walls[0]->y0;
45      *maxX = walls[0]->x1;
46      *maxY = walls[0]->y1;
47
48      for (int i = 1; i < numWalls; i++)
49      {
50          if (walls[i]->x0 < *minX) {
51              *minX = walls[i]->x0;
52          }
53          if (walls[i]->y0 < *minY) {
54              *minY = walls[i]->y0;
```

```
        }
        if (walls[i]->x1 > *maxX) {
            *maxX = walls[i]->x1;
        }
        if (walls[i]->y1 > *maxY) {
            *maxY = walls[i]->y1;
        }
    }
}

void CalculateDimensions(Wall** walls, int numWalls, float* width, float* height, float* centerX, float*
{
    //Trouve le centre géométrique du bâtiment ainsi que ses coordonnées les plus éloignées
    float minX, minY, maxX, maxY;
    CalculateCorners(walls, numWalls, &minX, &minY, &maxX, &maxY);

    *width = maxX - minX;
    *height = maxY - minY;

    *centerX = minX + (*width / 2);
    *centerY = minY + (*height / 2);
}

QPointF Normalize(const QPointF& point)
{
    qreal length = qSqrt(point.x() * point.x() + point.y() * point.y());
    if (length == 0) {
        return QPointF(0, 0);
    } else {
        return QPointF(point.x() / length, point.y() / length);
    }
}

QPointF WallToScreenCoordinate(float x, float y, float centerX, float centerY, float scale, float screen
{
    //Convertion des coordonnées de type "mur" (comme dans le plan donné dans un fichier .txt)
    // vers les coordonnées utilisées par l'afficheur graphique
    float XToReturn =(x - centerX) * scale + screenWidth / 2;
    float YToReturn = (centerY-y) * scale + screenHeight / 2;
    QPointF ToReturn = QPointF(XToReturn, YToReturn);
    return ToReturn;
}

```

```
98  QPolygonF CaseToPolygon(ColorCase toDraw,  float centerX, float centerY, float scale, float screenHeigh
99  {
100     //Convertit une case vers un polygon affichable par le moteur graphique
101     QPointF center = WallToScreenCoordinate(toDraw.x, toDraw.y, centerX, centerY, scale, screenHeight, 
102     QPointF halfHeight(0, (float)SQUARE_SIZE / 2 * scale);
103     QPointF halfWidth((float)SQUARE_SIZE / 2 * scale, 0);
104     QPolygonF polygon;
105     polygon << center + halfHeight + halfWidth <<  center - halfHeight + halfWidth <<  center - halfHei
106     return polygon;
107 }
108 
109 QPolygonF WallToPolygon(const Wall& wall, float centerX, float centerY, float scale, float screenHeight
110 {
111     //Convertit un mur vers un polygon affichable par le moteur graphique
112     QPointF p1((wall.x0 - centerX) * scale + screenWidth / 2, (centerY-wall.y0) * scale + screenHeight 
113     QPointF p2((wall.x1 - centerX) * scale + screenWidth / 2, (centerY-wall.y1) * scale + screenHeight 
114 
115     QPointF v = (p2 - p1) / QLineF(p1, p2).length();
116     QPointF n(-v.y(), v.x());
117 
118     float calculatedWidth = wall.wallType.width;
119 
120     if (screenHeight < screenWidth)
121     {
122         calculatedWidth *= screenHeight / 70;
123     }
124     else
125     {
126         calculatedWidth *= screenWidth / 70;
127     }
128 
129     QPointF c1 = p1 + calculatedWidth / 2 * n;
130     QPointF c2 = p1 - calculatedWidth / 2 * n;
131     QPointF c3 = p2 - calculatedWidth / 2 * n;
132     QPointF c4 = p2 + calculatedWidth / 2 * n;
133 
134     QPolygonF polygon;
135     polygon << c1 << c2 << c3 << c4;
136     return polygon;
137 }
138 
139 QPolygonF RayToPolygon(const Ray& ray, float centerX, float centerY, float scale, float screenHeight, fl
140 {
```

```
141     //Convertit un rayon vers un polygon affichable par le moteur graphique
142     QPolygonF poly;
143     float calculatedWidth = 1;
144 
145     if (screenHeight < screenWidth)
146     {
147         calculatedWidth *= screenHeight / 290;
148     }
149     else
150     {
151         calculatedWidth *= screenWidth / 290;
152     }
153 
154     QList<QPointF> pts;
155     QPointF pt1(((ray.points[num]).x() - centerX) * scale + screenWidth / 2, (centerY - (ray.points[num]
156     QPointF pt2(((ray.points[num+1]).x() - centerX) * scale + screenWidth / 2, (centerY - (ray.points[nu
157 
158     QPointF delta = pt2 - pt1;
159     QPointF normal = Normalize(QPointF(-delta.y(), delta.x())) * calculatedWidth;
160     QPointF perp1 = pt1 + normal;
161     QPointF perp2 = pt2 + normal;
162     pts.append(perp1);
163     pts.append(perp2);
164     perp1 = pt2 - normal;
165     perp2 = pt1 - normal;
166     pts.append(perp1);
167     pts.append(perp2);
168     return QPolygonF(pts);
169 }
170 
171 
172 
173 void DrawRay(Ray toDraw, QPainter* painter, float centerX, float centerY, float scale, float screenHeigh
174 {
175     QColor color = QColor::fromHsv((toDraw.points.length() * 70) % 256, 255, 255);
176     QBrush brush(color);
177     painter->setBrush(brush);
178     painter->setPen(color);
179     for (int i = 0; i < toDraw.points.length() -1; i++)
180     {
181         QPolygonF poly = RayToPolygon(toDraw, centerX, centerY, scale, screenHeight, screenWidth, i);
182         painter->drawPolygon(poly);
183     }
```

```
184 }
185
186 void FindMinMaxValues(const QList<ColorCase>& cases, float& minValue, float& maxValue)
187 {
188     if (cases.isEmpty())
189     {
190             minValue = 0;
191             maxValue = 0;
192             return;
193     }
194
195     minValue = cases.first().value;
196     maxValue = cases.first().value;
197
198     for (int i = 1; i < cases.size(); i++) {
199         const ColorCase& item = cases.at(i);
200         if (item.value < minValue) {
201             minValue = item.value;
202         }
203         if (item.value > maxValue) {
204             maxValue = item.value;
205         }
206     }
207 }
208
209 void DrawVerticalGradientRectangle(float minValue, float maxValue, int screenHeight, int screenWidth, Q
210 {
211     //Affiche le gradient de couleur à droite de l'écran
212
213     // Define the rectangle's dimensions
214     int rectHeight = screenHeight / 1.4;
215     int rectWidth = screenWidth / 14; // 1/4th of the screen width
216     int rectX = screenWidth - rectWidth - screenWidth / 20;
217     int rectY = (screenHeight - rectHeight)/2;
218
219     // Draw the rectangle
220     QRectF rectangle(rectX, rectY, rectWidth, rectHeight);
221     QLinearGradient gradient(rectangle.topLeft(), rectangle.bottomLeft());
222
223     QFont font("Helvetica", 10);
224     font.setBold(true);
225     painter->setFont(font);
226     float laxtTxtPrinted = -19;
```

```
227     for (int i = 0; i <= rectHeight; i++) {
228             float value = maxValue - (i * (maxValue - minValue)) / rectHeight;
229             QColor color = ColorGradient(value, minValue, maxValue);
230             gradient.setColorAt((float)i / rectHeight, color);
231 
232             // Draw the value next to the color
233             QString valueString = QString::number(value, 'f', 2);
234 
235 
236             // Set the text color to white (or any other color that contrasts with the gradient)
237 
238             float prct = (float)i / rectHeight * 100;
239             if (prct - 100/9.1 >= laxtTxtPrinted)
240             {
241                 laxtTxtPrinted = prct;
242                 painter->setPen(Qt::white);
243                 QRectF textRect(rectX - 60, i - 10 + rectY, 50, 20);
244                 painter->drawText(textRect, Qt::AlignRight | Qt::AlignVCenter, valueString);
245                 painter->drawLine(rectX + 10, i + rectY, rectX - 10, i + rectY);
246             }
247         }
248          painter->setBrush(QBrush(gradient));
249          painter->drawRect(rectangle);
250 }
251 
252 
253 void DrawCase(ColorCase toDraw, float maxVal, float minVal, QPainter *painter,  float centerX, float ce
254 {
255     QColor col = ColorGradient(toDraw.value, minVal, maxVal);
256     QBrush brush(col);
257     painter->setBrush(brush);
258     painter->setPen(col);
259     QPolygonF poly = CaseToPolygon(toDraw, centerX, centerY, scale, screenHeight, screenWidth);
260     painter->drawPolygon(poly);
261 }
262 
263 void DrawWall(Wall toDraw, QPainter* painter, float centerX, float centerY, float scale, float screenHe
264 {
265     QBrush brush(toDraw.wallType.color);
266     painter->setBrush(brush);
267     painter->setPen(toDraw.wallType.color);
268     QPolygonF poly = WallToPolygon(toDraw, centerX, centerY, scale, screenHeight, screenWidth);
269     painter->drawPolygon(poly);
```

```
}
void BlackBackround(QPainter *painter, float width, float height)
{    QBrush brush(QColor(40,40,40));
     painter->setBrush(brush);
     painter->setPen(QColor(40,40,40));
    QPointF topRCorner(width,0),topLCorner(0, 0),botRCorner(width, height),botLCorner(0, height);
    QPolygonF polygon;
    polygon << topRCorner << topLCorner << botLCorner << botRCorner;
    painter->drawPolygon(polygon);
}

void DrawGraduatedLines(QPainter* painter, float centerX, float centerY, float scale, int screenWidth, i
{
    //Affiche les coordonnées sur l'écran sous forme d'une ligne graduée verticale et horizontales.
    painter->setPen(QPen(Qt::black, screenWidth/400));

    QPointF wallTopLeft = WallToScreenCoordinate(0, 0, centerX, centerY, scale, screenHeight, screenWid
    QPointF wallBotRight = WallToScreenCoordinate(widthWall, -heightWall, centerX, centerY, scale, scre
    painter->drawLine(wallTopLeft.x(), screenHeight/100*4, wallBotRight.x(), screenHeight/100*4);

    for (float i = 0; i <= widthWall; i += widthWall/20.0) {
        painter->setPen(QPen(Qt::black, screenWidth/400));
        QPointF start = WallToScreenCoordinate(i, 10, centerX, centerY, scale, screenHeight, screenWidt
        QPointF end = WallToScreenCoordinate(i, 0, centerX, centerY, scale, screenHeight, screenWidth);
        painter->drawLine(start.x(), screenHeight/100*4, end.x(), screenHeight/100*3);
        QRectF textRect(start.x()-screenWidth/80, screenHeight/150, start.x(), screenHeight/100*2);
        painter->setPen(Qt::white);
        painter->drawText(textRect, Qt::AlignLeft | Qt::AlignVCenter,  QString::number(i, 'f', 2));
    }


    painter->setPen(QPen(Qt::black, screenWidth/400));

    painter->drawLine(wallTopLeft.x() -  screenWidth/200, wallTopLeft.y(), wallTopLeft.x() -  screenWid

    for (float i = 0; i <= heightWall; i += heightWall/15.0) {
        painter->setPen(QPen(Qt::black, screenWidth/400));
        QPointF start = WallToScreenCoordinate(0, -i, centerX, centerY, scale, screenHeight, screenWidt
        QPointF end = WallToScreenCoordinate(0, -i, centerX, centerY, scale, screenHeight, screenWidth)
        painter->drawLine(start.x()-screenWidth/200, end.y(),  start.x() -  screenWidth/200*3, end.y())
        QRectF textRect(start.x()-screenWidth/200*7.5, end.y()-heightWall/15.0,  start.x() -  screenWid
        painter->setPen(Qt::white);
        painter->drawText(textRect, Qt::AlignLeft | Qt::AlignTop,  QString::number(i, 'f', 2));
```

```
313     }
314
315 }
316 void DisplayTx(QPointF pos, QPainter* painter, float centerX, float centerY, float scale, int screenWid
317 {
318     //Afffiche une antenne
319     painter->setBrush(QBrush(QColorConstants::Svg::pink));
320     QPointF screenPos = WallToScreenCoordinate(pos.x(), pos.y(), centerX, centerY, scale, screenHeight,
321     painter->drawEllipse(screenPos,  screenWidth/180,  screenWidth/180);
322 }
323
324 void Display(QPainter *painter, Wall **walls, int numWalls, QList<Ray> rays, QList<ColorCase> cases, fl
325 {
326     painter->setRenderHint(QPainter::Antialiasing, false);
327     float widthWall, heightWall, centerX, centerY;
328     CalculateDimensions(walls, numWalls, &widthWall, &heightWall, &centerX, &centerY);
329
330     float scale = width / widthWall;
331
332     if (height / heightWall < scale)
333     {
334         scale = height / heightWall;
335     }
336
337     scale /= SCALE_DRECREASER; //We don't want to sitck to the edges of the screen
338
339     BlackBackround(painter, width, height);
340     float minVal, maxVal;
341     FindMinMaxValues(cases, minVal, maxVal);
342
343     for (int i = 0; i < cases.length(); i++)
344     {
345         DrawCase(cases[i], maxVal, minVal, painter, centerX, centerY, scale, height, width);
346     }
347
348     for (int i = 0; i < numWalls; i++)
349     {
350         DrawWall(*walls[i], painter, centerX, centerY, scale, height, width);
351     }
352
353     for (int i = rays.length() - 1; i >= 0; i--)
354     {
355         DrawRay(rays[i], painter, centerX, centerY, scale, height, width);
```

```
356     }
357     if (cases.length() > 0)
358     {
359         DrawVerticalGradientRectangle(minVal, maxVal, height, width, painter);
360     }
361     DisplayTx(TX_RX.first,painter, centerX, centerY, scale, width, height, widthWall, heightWall);
362     DrawGraduatedLines(painter, centerX, centerY, scale, width, height, widthWall, heightWall);
363    /*  QPointF posTx1(90,-36);
364     DisplayTx(posTx1,painter, centerX, centerY, scale, width, height, widthWall, heightWall);
365    QPointF posTx1V2(88, -3);
366     DisplayTx(posTx1V2,painter, centerX, centerY, scale, width, height, widthWall, heightWall);*/
367
368 }
369
370 float DoubleToDbmT(double toConvert)
371 {
372     return 10 * log10(toConvert * 1000); //*1000 as / 1 mW
373 }
374 /*
375 void DuplicateWallsLower(Wall **&walls, int &numWalls)
376 {
377     //Duplication des murs plus bas pour créer une seconde usine
378     Wall** newWalls = new Wall*[numWalls * 2];
379
380     for (int i = 0; i < numWalls; i++)
381     {
382         newWalls[i] = walls[i];
383     }
384
385     for (int i = 0; i < numWalls; i++)
386     {
387         Wall* wall = walls[i];
388         Wall* newWall = new Wall(wall->x0, wall->y0 - 100, wall->x1, wall->y1 - 100, wall->wallType);
389         newWalls[numWalls + i] = newWall;
390     }
391
392     numWalls *= 2;
393
394     delete[] walls;
395     walls = newWalls;
396 }*/
397
398 void CreateBoard()
```

```
{
    std::unordered_map<std::string, WallType*> wallTypeMap = ReadWallTypesFromFile(WallTypeFilePath);
    Wall **walls;
    int wallCount = 0;
    ReadWallsFromFile(WallFilePath, walls, wallCount, wallTypeMap);

    qDebug() << "Done reading walls from files.\n";
    QList<Antenna*> antennas = GetAntennas();
    QList<Ray> rays;
    QList<ColorCase> cases;

    if (SQUARE_SIZE != 0)
    {
        GetHeatmap(walls, wallCount, &antennas, &cases);
    }
    else
    {
        std::complex<double> power;
        GenerateRaysAndPower(walls, wallCount, &antennas, &rays, power);
        qDebug("Found %lli rays.\n", rays.length());
        qDebug() << "Power : " << power.real();
        qDebug() << "Power dbm : " <<  DoubleToDbmT(power.real());

    }

    BoardWidget* board = new BoardWidget(walls, wallCount, rays, cases);
    board->show();
}
```

heatmapmanager.cpp :

```
#include "colorcase.h"
#include "wall.h"
#include <QObject>
#include <functional>

#include "qdebug.h"
#include "qpoint.h"
#include "ray.h"
#include "wall.h"
#include <thread>
#include "raytracingmanager.h"
#include "powercalculator.h"
```

```
13  #include <cmath>
14  #include "params.h"
15  #include "wallmanager.h"
16  #include "antenna.h"
17
18  float DoubleToDbm(double toConvert)
19  {
20      return 10 * log10(toConvert * 1000); //*1000 as / 1 mW
21  }
22
23  float DBmToSpeed(float dbm)
24  {
25      if (dbm < -80)
26      {
27          return 0;
28      }
29      float val = dbm * 15 + 1250;
30      if (val < 100)
31      {
32          return 0;
33      }
34      if (val > 350)
35      {
36          return 350;
37      }
38      return val; //TODO : verif
39  }
40
41  void GetLineRowCount(QPointF topRight, QPointF bottomLeft, int& rowCount, int& lineCount, int& total)
42  {
43      //Donne le nombre de ligne et de colonnes dans la heatmap
44      float minX = bottomLeft.x();
45      float maxX = topRight.x();
46      float minY = bottomLeft.y();
47      float maxY = topRight.y();
48
49      rowCount = ceil((maxX-minX) / SQUARE_SIZE);
50      lineCount = ceil((maxY-minY) / SQUARE_SIZE);
51      total = rowCount * lineCount;
52  }
53  void DisplayProgress(const int thread_id, const int currentProgress, int& progressDisplayed)
54  {
55      //Affiche le progrès réalisé par un thread
```

```
56      if (DISPLAY_THREAD && thread_id == 0)
57      {
58          if (currentProgress % 10 == 0 && currentProgress != progressDisplayed)
59          {
60              progressDisplayed = currentProgress;
61              qDebug() << "Thread 0 progress: " << progressDisplayed << "%";
62          }
63      }
64  }
65
66
67
68  bool isPointInsideBuilding(QPointF point, Wall** walls, int wallCount) {
69      //Returne true si le pointn est à l'intérieur du bâtiment, false sinon
70      //Si un point est dans le bâtiment, c'est qu'il y a un nombre impair de mur(s) extérieurs à sa droi
71      int count = 0;
72      for (int i = 0; i < wallCount; i++) {
73          Wall* wall = walls[i];
74          if (wall->outer)
75          {
76              bool inBetweenY = (wall->y0 >= point.y() != wall->y1 >= point.y());
77              if (inBetweenY && wall->x0 >= point.x())
78              {
79                  ++count;
80              }
81          }
82      }
83      return (count % 2) == 1;
84  }
85
86
87
88
89
90  void HeatMapThread(Wall **walls, int wallCount, QList<ColorCase> *cases, int thread_id, QPointF topRigh
91  {
92      //Un thread générant une partie de la heatmap
93      int rowCount, lineCount, total;
94      GetLineRowCount(topRight, bottomLeft, rowCount, lineCount, total);
95      QList<ColorCase> casesToAdd;
96      int progressDisplayed = 0;
97
98      for (int i = thread_id; i <  total; i+=NUM_THREADS)
```

```
99      {
100         float x = i % rowCount * SQUARE_SIZE +  bottomLeft.x() + SQUARE_SIZE / 2;
101         float y = floor(i / rowCount) * SQUARE_SIZE + bottomLeft.y() + SQUARE_SIZE / 2;
102         std::complex<double> PX = 0;
103
104         if (isPointInsideBuilding(QPointF(x, y), walls, wallCount))
105         {
106             for (int j = 0; j < antennas->length(); j++)
107             {
108                 QList<Ray> rays = GenerateRays(walls, wallCount, (antennas->at(j))->pos, QPointF(x, y),
109                 std::complex<double> power = GetPower(rays, walls, wallCount, F, antennas->at(j));
110 #if POWER_MODE == 0  //Si l'on veut sommer la puissance de toute les antennes, pour vérifier que la nor
111                 PX += power;
112 #else //Si l'on veut uniquement la plus grande puissance pour calculer le débit de donnée
113                 if (power.real() > PX.real())
114                 {
115                     PX = power;
116                 }
117 #endif
118             }
119            casesToAdd.append(ColorCase(x, y, DBmToSpeed(DoubleToDbm(PX.real())), SQUARE_SIZE));
120
121             //   casesToAdd.append(ColorCase(x, y, DoubleToDbm(PX.real()), SQUARE_SIZE));
122         }
123
124         DisplayProgress(thread_id, (i * 100) / total, progressDisplayed);
125
126     }
127     cases->append(casesToAdd);
128 }
129
130 void GenerateRays(Wall **walls, int wallCount, QList<Antenna*> *antennas, QList<Ray> *rays)
131 {
132     //Génère les rayons pour chaque antenne
133     for (int i = 0; i < antennas->length(); i++)
134     {
135         QPointF rxPos = TX_RX.second;
136         rays->append(GenerateRays(walls, wallCount, (antennas->at(i))->pos, rxPos, MAX_RECU));
137     }
138 }
139
140
141 void GetHeatmap(Wall **walls, int wallCount, QList<Antenna*> *antennas, QList<ColorCase> *cases)
```

```
{
    float width, height, centerX, centerY;
    CalculateDimensions(walls, wallCount, &width, &height, &centerX, &centerY);
    QPointF topRight = QPointF(centerX + width /2, centerY + height / 2);
    QPointF bottomLeft = QPointF(centerX - width /2, centerY - height / 2);
    std::vector<std::thread> threads;
    QList<QList<ColorCase>*> casesForThreads;

    // Initialise les threads
    for (int i = 0; i < NUM_THREADS; i++)
    {
        casesForThreads.append(new QList<ColorCase>);
        threads.emplace_back(HeatMapThread, walls, wallCount, casesForThreads[i], i, topRight, bottomLe
    }

    // Wait for the threads to finish
    for (auto& thread : threads) {
        thread.join();
    }

    for (int i = 0; i < NUM_THREADS; i++)
    {
        cases->append(*casesForThreads[i]);
        delete casesForThreads[i];
    }

}
```

mainwindow.cpp :

```
#include "mainwindow.h"
#include "./ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
```

```
}

```

optimiser.cpp :

```
#include "qdebug.h"
#include "colorcase.h"
#include "filemanager.h"
#include "heatmapmanager.h"
#include "antennamanager.h"
#include "params.h"
#include "optimiser.h"

int NumCasesBehindTreshold(QList<ColorCase> cases)
{//Nombre de case en desosus du seuil minimal
    int toReturn = 0;
    float treshold = 0;//En Mb/s
    for (int i = 0; i < cases.length(); i++)
    {
        if (cases[i].value <= treshold)
        {
            toReturn++;
        }
    }
    return toReturn;
}

double totalSpeed(QList<ColorCase> cases)
{//Somme du débit de toutes les cases
    double toReturn = 0;
    for (int i = 0; i < cases.length(); i++)
    {
        toReturn += cases[i].value/100; // /100 to avoid reaching number so high the float compression
    }
    return toReturn;
}

double GetScore(QList<ColorCase> cases)
{ //Attribue un score à une heatmap
    int casesUnderTre = NumCasesBehindTreshold(cases);
    if (casesUnderTre > 0)
    {
        return -casesUnderTre;
```

```
39      }
40      return totalSpeed(cases);
41  }
42
43  void OptimiseAngle()
44  { //Optimise l'angle de TX3
45      std::unordered_map<std::string, WallType*> wallTypeMap = ReadWallTypesFromFile(WallTypeFilePath);
46      Wall **walls;
47      int wallCount = 0;
48      ReadWallsFromFile(WallFilePath, walls, wallCount, wallTypeMap);
49      qDebug() << "Done reading walls from files.\n";
50      QList<Antenna*> antennas;
51      TX3 tx3(F, GetLambda()/2, Prx,-0.1,PHI3DB, GMAXDB, TX_RX.first);
52      tx3.ra = 73;
53      antennas.append(&tx3);
54      double lambda = 3.0 * pow(10,8) / F;
55      float pDbm = 20;
56      float p = (pow(10, pDbm / 10) / 1000);
57      Antenna tx1AldreadySetup(F, lambda/2, p, QPointF(90.0,-36.0),1.5);
58      antennas.append(&tx1AldreadySetup);
59      float bestScore = -INFINITY;
60      double bestAngle = -INFINITY;
61      int numSteps = 360;
62
63      for (int i = 0; i < numSteps; i++)
64      {
65          QList<ColorCase> cases;
66          double angle = -PI + 2 * PI * i / numSteps;
67          tx3.delta = angle;
68          qDebug() << "Testing angle " << angle;
69          GetHeatmap(walls, wallCount, &antennas, &cases);
70          float score = GetScore(cases);
71          if (score > bestScore)
72          {
73              bestScore = score;
74              bestAngle = angle;
75          }
76          qDebug() << "Score " << score;
77          int currentProgress = i * 100 / numSteps;
78          qDebug() << "Progress: " << currentProgress << "%";
79
80      }
81      qDebug() << "Best angle " << bestAngle;
```

```
82      qDebug() << "with a score of" << bestScore;
83
84  }
85
86  void OptimisePos()
87  {//Optimise la position des TX1
88      std::unordered_map<std::string, WallType*> wallTypeMap = ReadWallTypesFromFile(WallTypeFilePath);
89      Wall **walls;
90      int wallCount = 0;
91      ReadWallsFromFile(WallFilePath, walls, wallCount, wallTypeMap);
92      qDebug() << "Done reading walls from files.\n";
93      QList<Antenna*> antennas;
94      TX3 tx3(F, GetLambda()/2, Prx,-0.575958,PHI3DB, GMAXDB, TX_RX.first);
95      tx3.ra = 73;
96      antennas.append(&tx3);
97      double lambda = 3.0 * pow(10,8) / F;
98      float pDbm = 20;
99      float p = (pow(10, pDbm / 10) / 1000);
100     Antenna tx1(F, lambda/2, p, QPointF(0.0,0.0),1.5);
101     antennas.append(&tx1);
102     Antenna tx1AldreadySetup(F, lambda/2, p, QPointF(90.0,-36.0),1.5);
103     antennas.append(&tx1AldreadySetup);
104
105     double bestScore = -INFINITY;
106     double bestX = -INFINITY;//X et Y donnant le meilleur score
107     double bestY= -INFINITY;
108     float startX = 0, endX = 100;
109     float startY = -75, endY = 0;
110     float step = 8;
111     int totalSteps = ceil((endX - startX) / step) * ceil((endY - startY) / step);
112     int currentStep = 0;
113
114     for (float x = startX; x <= endX; x += step)
115     {
116         for (float y = startY; y <= endY; y += step)
117         {
118             if (isPointInsideBuilding(QPointF(x,y), walls, wallCount))
119             {
120                 QList<ColorCase> cases;
121                 qDebug() << "Testing coordinates " << x << " ; " << y;
122                 tx1.pos = QPointF(x,y);
123                 GetHeatmap(walls, wallCount, &antennas, &cases);
124                 double score = GetScore(cases);
```

```
125                 if (score > bestScore)
126                 {
127                     bestScore = score;
128                     bestX = x;
129                     bestY = y;
130                 }
131                 // qDebug() << "Score " << score;
132                 int currentProgress = (int)((float)currentStep / (float)totalSteps * 100.0f);
133                 qDebug() << "Progress: " << currentProgress << "%";;
134                 ++currentStep;
135             }
136         }
137     }
138 
139     qDebug() << "Best x " << bestX;
140     qDebug() << "Best y " << bestY;
141     qDebug() << "with a score of" << bestScore;
142 }
143 void Optimise()
144 {
145     OptimisePos();
146 
147 
148 }
```

powercalculator.cpp :

```
1  #include "qdebug.h"
2  #include "powercalculator.h"
3  #include "params.h"
4  #include "wall.h"
5  #include "raytracingmanager.h"
6  #include "antenna.h"
7  
8  using ComplexD = std::complex<double>;
9  
10 QList<Wall> WallsBetweenTwoPoints(QPointF pointA, QPointF pointB, Wall** allWalls, int wallCount, QList
11 {
12     //Retourne tout les murs présents entre 2 points
13     QList<Wall> walls;
14     for (int i = 0; i < wallCount; i++) {
15         Wall wall = *allWalls[i];
16 
```

```
17          if (wallsToAvoid.contains(wall)) {
18              continue;
19          }
20
21          QPointF intersection;
22          bool hits;
23          intersection = FindRayWallIntersection(pointA, pointB, wall, &hits); //TODO : optimise to not c
24          if (hits) {
25              float dist0Squared = pow(intersection.x() - pointA.x(),2) + pow(intersection.y() -pointA.y()
26              float dist1Squared = pow(intersection.x() - pointB.x(),2) + pow(intersection.y() - pointB.y
27              float distPointAPointBSquared = pow(pointA.x() - pointB.x(),2) + pow(pointA.y() - pointB.y()
28              if (dist0Squared <= distPointAPointBSquared && dist1Squared <= distPointAPointBSquared )
29              {
30                  walls.append(wall);
31              }
32          }
33      }
34      return walls;
35  }
36
37  ComplexD Reflection(QPointF first_point, QPointF second_point, QPointF wall_normal, float eps_rel, Compl
38      //Calcule le coefficient de réflexion
39      QPointF vector_d = second_point - first_point;
40      float vector_d_length = sqrt(vector_d.x() * vector_d.x() + vector_d.y() * vector_d.y());
41      ComplexD cosIncidence = abs(QPointF::dotProduct(vector_d / vector_d_length, wall_normal));
42      ComplexD sinIncidence = sqrt(1.0 - pow(cosIncidence, 2));
43      ComplexD sinTransmission = sqrt(1.0/ eps_rel) * sinIncidence;
44      ComplexD cosTransmission = sqrt(1.0 - pow(sinTransmission, 2));
45      ComplexD s = l / cosTransmission;
46      ComplexD reflectionCoeff = (Zm * cosIncidence - Z0 * cosTransmission) / (Zm * cosIncidence + Z0 * cos
47
48      ComplexD term1 = 1.0 - pow(reflectionCoeff, 2);
49      ComplexD term2 = reflectionCoeff * exp(-2.0 * gamma * s);
50      ComplexD term3 = exp(ComplexD(0, 1.0) * beta * 2.0 * s * sinIncidence * sinTransmission);
51      ComplexD term5 = exp(ComplexD(0, 1.0) * beta * 2.0 * s * sinIncidence * sinTransmission);
52      ComplexD term4 = 1.0 - pow(reflectionCoeff, 2) * exp(-2.0 * gamma * s)*term5;
53
54
55      ComplexD numerator = term2 * term3 * term1;
56      ComplexD denominator = term4;
57      ComplexD fraction = numerator / denominator;
58
59      ComplexD totalReflection = reflectionCoeff - fraction;
```

```
60
61      return totalReflection ;
62  }
63
64  double ComplexNormSquared(ComplexD  z)
65  {
66      return pow(z.real(),2)+pow(z.imag(),2);
67  }
68
69  ComplexD Transmission(QPointF firstPoint, QPointF secondPoint, QPointF wallNormal, float epsRela, Compl
70  {
71      QPointF vectorD = secondPoint - firstPoint;
72      float vectorDLength = sqrt(vectorD.x() * vectorD.x() + vectorD.y() * vectorD.y());
73      ComplexD  cosIncidence = abs(QPointF::dotProduct(vectorD / vectorDLength, wallNormal)); // Produit s
74      ComplexD  sinIncidence = sqrt(1.0 - pow(cosIncidence, 2));
75      ComplexD  sinTransmission = sqrt(1.0 / epsRela) * sinIncidence;
76      ComplexD  cosTransmission = sqrt(1.0 - pow(sinTransmission, 2));
77      ComplexD  s = l / cosTransmission; //
78      ComplexD  reflectionCoeff = (Zm * cosIncidence - Z0 * cosTransmission) / (Zm * cosIncidence + Z0 * c
79      ComplexD  totalTransmission =
80              (ComplexD (1.0,0.0) - pow(reflectionCoeff, 2)) * exp(ComplexD(-1.0,0.0) * gamma * s)
81               / (
82                  ComplexD (1.0,0.0) - pow(reflectionCoeff, 2) * exp(-2.0 * gamma * s)
83               * exp(ComplexD (0,1.0) * beta * ComplexD (2.0,0.0) * s * sinIncidence * sinTransmission)
84                  );
85
86      return totalTransmission;
87  }
88
89  double AngleBetweenPoints(QPointF a, QPointF b)//In radiants
90  {
91      //Utilisé pour la directivité de TX3
92      double dx = b.x() - a.x();
93      double dy = b.y() - a.y();
94      double angleRadians = qAtan2(dy, dx);
95      return angleRadians;
96  }
97
98  ComplexD TransmissionOneRay(Ray ray, Wall **allWalls, int wallCount)
99  {
100     //Trouve les coefficients de transmissions pour tout les murs par lesquels passe un rayon
101     ComplexD  globalTransmission = (1.0) ; // Coefficient global de transmission pour un rayon donné
102     for (int j = 0; j < ray.points.length() - 1; j++)
```

```
103     {
104         QPointF point1 = ray.points[j] ; //Pt avant mur
105         QPointF point2 = ray.points[j+1]; //Pt après
106         QList<Wall> wallsToAvoid;
107 
108         if (j-1 >= 0)
109         {
110             Wall wall1 = ray.walls[j-1];
111             wallsToAvoid.append(wall1);
112         }
113         if (j < ray.walls.length())
114         {
115             Wall wall2 = ray.walls[j];
116             wallsToAvoid.append(wall2); //To ensure we don't consider the transmission for the walls we
117         }
118 
119         QList<Wall> transmissionWalls = WallsBetweenTwoPoints(point1, point2, allWalls, wallCount, walls
120 
121         for (int w = 0; w < transmissionWalls.length(); w++)
122         {
123             Wall toUse = transmissionWalls[w];
124             ComplexD  complexTransmission =
125                     Transmission(point1,point2,
126                                  toUse.normal,
127                                  toUse.wallType.permitivityRela,toUse.wallType.width,
128                                  toUse.wallType.Z,toUse.wallType.Z0,toUse.wallType.gamma,toUse.wallType
129             globalTransmission *= complexTransmission  ; // On récupère tout les coefficients pour un r
130         }
131 
132     }
133 
134     return globalTransmission;
135 }
136 
137 ComplexD GetRayLength(Ray ray)
138 {
139     //Distance parcourue par un rayon
140     ComplexD  dist = 0.0;
141     for (int j = 0; j < ray.points.length() - 1; j++)
142     {
143                 QPointF point_1 = ray.points[j] ; //Pt avant mur
144                 QPointF point_2 = ray.points[j+1]; //Pt après
145                 ComplexD  module = sqrt(pow(point_2.x()-point_1.x(),2)+pow(point_2.y()-point_1.y(),2));
```

```
146                 dist = dist + module ;
147     }
148     return dist;
149 }
150 ComplexD ReflexOneRay(Ray ray)
151 {
152     //Tout les coefficients de réflexion pour un rayon
153     ComplexD  globalReflexion = (1.0) ;
154     for (int j = 0; j < ray.points.length() - 2; j++){
155         QPointF point1 = ray.points[j] ;  //Pt avant le mur
156         QPointF point2 = ray.points[j+1]; //Pt sur le mur
157         Wall wall = ray.walls[j];
158         ComplexD  complexReflexion = Reflection(point1, point2,wall.normal,wall.wallType.permitivityRela
159         globalReflexion = globalReflexion * complexReflexion  ; // coeff. reflexion global en multiplia
160     }
161 
162     return globalReflexion;
163 }
164 
165 ComplexD GetPower(QList<Ray> rays, Wall **allWalls, int wallCount, double f, Antenna *an){ // On veut c
166     std::vector<ComplexD > complexListEField ; // Liste contenant les valeurs des différents champs
167     //Calcule la puissance à partir de tout les rayons trouvés précédemment
168     ComplexD  lambda = (3.0*pow(10,8)/f)  ; // longueur d'onde
169     ComplexD  ETotal = 0.0 ; // Somme de tout les champs E
170     ComplexD  PRX; // puissance reçue au récepteur
171     ComplexD beta0 = 2 * PI * f / (3 * pow(10,8));//= 2 * 3.14 * 868.3 * pow(10,6) / (3 * pow(10,8));
172 
173     for (int i = 0; i < rays.length(); i++)
174     {
175         Ray rayI = rays[i];
176 
177         ComplexD  dn = GetRayLength(rays[i]); // trajet total effectué par le rayon
178         ComplexD allEffects = TransmissionOneRay(rays[i], allWalls, wallCount)*ReflexOneRay(rays[i]) ; 
179         double ang = AngleBetweenPoints(rays[i].points[0], rays[i].points[1]);
180 
181         ComplexD E = allEffects*sqrt(60.0*an->PtxGtx(ang))*exp(-ComplexD(0.0,1.0)*dn*beta0)/dn;
182         complexListEField.push_back(E); // Ajoute à la liste std::complexe la valeur du champ E corresp
183 
184         double normSquaredETot = pow(E.real(),2) + pow(E.imag(),2);
185 
186         ComplexD PRXT = (pow(lambda,2)/(8*pow(PI,2)*an->ra))*normSquaredETot;
187 
188     }
```

```
189
190 #if TARGET == 8
191     for (int i = 0; i < complexListEField.size(); i++)   { // Somme de toutes les composantes des diffé
192         ETotal += complexListEField.at(i) ;
193         }
194
195     double normSquaredETot = pow(ETotal.real(),2) + pow(ETotal.imag(),2);
196     PRX = (pow(lambda,2)/(8*pow(PI,2)*an->ra))*normSquaredETot;
197 #else
198     double val = 0;
199     for (int i = 0; i < complexListEField.size(); i++)
200     { // Somme de toutes les composantes des différents champs
201         val += ComplexNormSquared(complexListEField.at(i));
202     }
203     ComplexD R = an->ra;
204     PRX = pow(lambda,2)/(pow(PI,2)*8*an->ra)*val;
205 #endif
206     return PRX;
207 }
208
209
```

ray.cpp :

```
1 #include "ray.h"
2 #include <QPointF>
3 #include <complex>
4 Ray::Ray()
5 {}
```

raytracingmanager.cpp :

```
1 #include "wall.h"
2 #include "ray.h"
3 #include <iostream>
4 #include <vector>
5 #include <QPolygonF>
6 #include <QColor>
7 #include<cmath>
8
9
10 int PointRelativePositionToWall(const Wall& wall, QPointF point) {
```

```
11      QPointF n_normal = wall.normal;
12      QPointF v = point - QPointF(wall.x0, wall.y0);
13      float dotProduct = v.x() * wall.normal.x() + v.y() * wall.normal.y();
14      if (dotProduct > 0) {
15          return 1; // the point is on the right side of the wall
16      } else if (dotProduct < 0) {
17          return -1; // the point is on the left side of the wall
18      } else {
19          return 0; // the point is on the wall's track
20      }
21  }
22
23  int WallPosRelativeToWall(Wall& wall1, Wall& wall2)
24  {
25      //Calcul l'indice d'un mur par rapport à un autre - voir partie "optimisation du code" du rapport
26      int startPos = PointRelativePositionToWall(wall1, QPointF(wall2.x0, wall2.y0));
27      int endPos = PointRelativePositionToWall(wall1, QPointF(wall2.x1, wall2.y1));
28
29      if (startPos == endPos)
30      {
31          return startPos;
32      }
33      else
34      {
35          return 0;
36      }
37  }
38
39  QPointF ReflectPoint(const QPointF& point, Wall mirror)
40  {
41      //Donne l'image mirroir d'un point par un mur
42      QPointF normal = mirror.normal;
43      QPointF vector = point - QPointF(mirror.x0, mirror.y0);
44      qreal projectionLength = QPointF::dotProduct(vector, normal);
45      QPointF projection = projectionLength * normal;
46
47      projection *= 2; //Doubling the projection allows us to go to the mirror point
48
49      QPointF reflectedPoint = point - projection;
50
51      return reflectedPoint;
52  }
53
```

```
54
55  QPointF FindRayWallIntersection(QPointF TX, QPointF mPoint, Wall wall, bool* valid)
56  {
57      *valid = false; //Valid est mit à true si il y a une intersection
58      //Et reste à false si il n'y en a pas
59      QPointF direction = mPoint - TX;
60      QPointF normal = wall.normal;
61      qreal dotProduct = QPointF::dotProduct(direction, normal);
62
63      if (qAbs(dotProduct) < 1e-6) {//Parallel to the wall
64          return QPointF();
65      }
66
67      qreal distance = QPointF::dotProduct(QPointF(wall.x0, wall.y0) - TX, normal) / dotProduct;
68      QPointF intersection = TX + distance * direction;
69
70      QPointF v1 = intersection - QPointF(wall.x0, wall.y0);
71      QPointF v2 = intersection - QPointF(wall.x1, wall.y1);
72
73      qreal dotProduct1 = QPointF::dotProduct(v1, v2);
74
75      if (dotProduct1 <= 0) { //Negative if the two vectors have opposite directions
76          //If the two vectors have opposite directions,
77          //it means that the point is between the start and end of the wall
78          // The intersection point is on the wall
79          *valid = true;
80          return intersection;
81      }
82
83      // The intersection point is outside of the wall
84      return QPointF();
85  }
86
87  qreal CrossProduct(const QPointF& v1, const QPointF& v2) {
88      return v1.x() * v2.y() - v1.y() * v2.x();
89  }
90
91  bool ArePointsOnSameSideOfWall(const QPointF& p1, const QPointF& p2, const Wall& wall)
92  {
93      QPointF wallVector = QPointF(wall.x1 - wall.x0, wall.y1 - wall.y0);
94
95      QPointF p1Vector = QPointF(p1.x() - wall.x0, p1.y() - wall.y0);
96      QPointF p2Vector = QPointF(p2.x() - wall.x0, p2.y() - wall.y0);
```

```
97
98      qreal crossProduct1 = CrossProduct(wallVector, p1Vector);
99      qreal crossProduct2 = CrossProduct(wallVector, p2Vector);
100
101     if (crossProduct1 * crossProduct2 > 0) {
102         return true;
103     }
104     else {
105         return false;
106     }
107 }
108
109 Ray DirectRay(QPointF startPos, QPointF endPos, bool *generatedRay)
110 {//Génère le rayon direct
111     Ray ray;
112     ray.points.append(endPos);
113     ray.points.append(startPos);
114     *generatedRay = true;
115     return ray;
116 }
117 void AddReversedWallsToRay(Ray& ray, Wall **walls, int wallCount)
118 {//Ajoute les murs en ordre inversé au rayon - c'est l'ordre utilisé plus tard dans le code
119     QList<Wall> reversedWalls;
120     for (int i = wallCount - 1; i >= 0; i--) {
121         reversedWalls.append(*walls[i]);
122     }
123     ray.walls = reversedWalls;
124 }
125
126 bool GenerateRayMiddlePoints(Ray& ray, Wall **walls, int wallCount,  QList<QPointF> images)
127 {
128     //Génère les points du rayon à partir des murs par lesquels ce rayon doit passer
129     bool valid = true;
130     int index = 1;
131     for (int i = wallCount-2; i >= 0; i--)
132     {
133         index++;
134         ray.points.append(QPointF(FindRayWallIntersection(ray.points[index-1], images[i], *walls[i], &va
135
136         if (valid == false || !ArePointsOnSameSideOfWall(ray.points[ray.points.length()-3], ray.points.l
137             return false;
138     }
139     return true;
```

```
140 }
141
142 Ray GenerateRay(Wall **walls, int wallCount, QPointF startPos,  QPointF endPos, bool *generatedRayIsVali
143 {
144     *generatedRayIsValid = false; //Mit a true si le rayon est valide
145
146     if (wallCount == 0)
147     {
148         return DirectRay(endPos, startPos, generatedRayIsValid);
149     }
150
151     Ray ray;
152     bool valid = true;
153     ray.points.append(QPointF(startPos));
154     ray.points.append(QPointF(FindRayWallIntersection(startPos,images[images.length()-1],*walls[wallCou
155     if (!valid)
156         return ray;
157
158     if (!GenerateRayMiddlePoints(ray, walls, wallCount, images))
159     {
160         *generatedRayIsValid = false;
161         return ray;
162     }
163
164     ray.points.append(QPointF(endPos));
165     if (!ArePointsOnSameSideOfWall(ray.points[ray.points.length()-3], ray.points.last(), *walls[0]))
166         return ray;
167
168     AddReversedWallsToRay(ray, walls, wallCount);
169     *generatedRayIsValid = true;
170     return ray;
171 }
172
173
174 void TryToAddRay(QList<Ray>& rays, Wall **usedWalls, int numUsedWalls, QPointF startPos, QPointF endPos
175 {
176     //Ajoute le rayon à la liste si il est valide
177     bool validRay = true;
178     Ray genRay = GenerateRay(usedWalls, numUsedWalls, startPos, endPos, &validRay, mirrorImages);
179     if (validRay)
180        rays.append(genRay);
181 }
182
```

```
183 bool WallIsValid(int wallId, int lastWallId, int numUsedWalls, Wall **usedWalls, int** wallRelativePos,
184 {
185     if (wallId == lastWallId)
186     {
187         return false; //On ne peut rebondir 2x sur le même mur
188     }
189 
190     if (numUsedWalls >= 2)
191     {
192         Wall before = *usedWalls[numUsedWalls-2];
193         int indexOfWallBefore = -1;
194         for (int i = 0; i < numWalls; i++) {
195           if (*allWalls[i] == before) {
196             indexOfWallBefore = i;
197             break;
198           }
199         }
200         int relativePosWallBefore = wallRelativePos[lastWallId][indexOfWallBefore];
201         int relativePosWallAfter = wallRelativePos[lastWallId][wallId];
202         if (relativePosWallAfter * relativePosWallBefore == -1)
203         {
204             return false;
205         } //Utilisation des indices précalculés - voir section "Optimisation du code " du rapport
206     }
207     return true;
208 }
209 
210 QList<Ray> RecuRayGenerator(Wall **allWalls, int numWalls, int** wallRelativePos, Wall **usedWalls, int
211 { //Fonction récursive générant les rayons
212     QList<Ray> toReturn;
213     if (recu <= 0)
214     {
215         TryToAddRay(toReturn, usedWalls, numUsedWalls, startPos, endPos, mirrorImages);
216         return toReturn;
217     }
218     for (int i = 0; i < numWalls; i++)
219     {
220         if (WallIsValid(i, lastWallId, numUsedWalls, usedWalls, wallRelativePos, allWalls, numWalls))
221         {
222             QVector<Wall*> n_usedWalls(numUsedWalls + 1);
223             std::copy(usedWalls, usedWalls + numUsedWalls, n_usedWalls.begin());
224             n_usedWalls[numUsedWalls] = allWalls[i];
225 
```

```
226             QList<QPointF> n_mirror = mirrorImages;
227             n_mirror.append(n_mirror.isEmpty() ? ReflectPoint(endPos, *allWalls[i]) : ReflectPoint(n_mi
228 
229             toReturn.append(RecuRayGenerator(allWalls, numWalls, wallRelativePos, n_usedWalls.data(), nu
230         }
231     }
232 
233     return toReturn;
234 }
235 
236 void GenerateRelativePosArray(Wall **walls, int wallCount, int **wallRelativePos)
237 {
238     //Génération des indices des murs les uns par rapport aux autres - voir section "Optimisation du co
239     for (int i = 0; i < wallCount; i++) {
240         wallRelativePos[i] = new int[wallCount];
241     }
242 
243     for (int i = 0; i < wallCount; ++i)
244     {
245         for (int j = 0; j < wallCount; ++j)
246         {
247             wallRelativePos[i][j] =  WallPosRelativeToWall(*walls[i], *walls[j]);
248         }
249     }
250 }
251 
252 
253 QList<Ray> GenerateRays(Wall **walls, int numWalls, QPointF startPos, QPointF endPos, int maxRecu)
254 {
255     //Récupère les paramètres requis puis utilise la fonction récursive - permet de ne pas recalculer t
256     QList<Ray> rays;
257     int** wallRelativePos = new int*[numWalls];
258 
259     GenerateRelativePosArray(walls, numWalls, wallRelativePos);
260 
261     for (int i = 0; i <= maxRecu; i++)
262     {
263         QList<QPointF> mirror;
264         Wall** usedWalls = new Wall*[0];
265         rays.append(RecuRayGenerator(walls, numWalls, wallRelativePos, usedWalls, 0, startPos, endPos, 
266         delete[] usedWalls;
267     }
268 
```

```
269     delete[] wallRelativePos;
270     return rays;
271 }
272
273
274
```

---

wall.cpp :

```
1  #include "wall.h"
2  #include "qmath.h"
3  #include "walltype.h"
4  #include <QPointF>
5
6
7  Wall::Wall(float x0, float y0, float x1, float y1, WallType type)
8      : wallType(type), x0(x0), y0(y0), x1(x1), y1(y1)
9      {
10     QPointF direction(x1 - x0, y1 - y0);
11     QPointF _normal = QPointF(-direction.y(), direction.x());
12     float n_length = qSqrt(_normal.x() * _normal.x() +  _normal.y() *  _normal.y());
13     normal = QPointF(_normal.x() / n_length, _normal.y() / n_length);
14     length = qSqrt(pow(x1 - x0,2) + pow(y1 - y0,2));
15     }
16
17
```

---

walltype.cpp :

```
1  #include "walltype.h"
2  #include <complex>
3  #include "params.h"
4
5
6
7  WallType::WallType(std::string _name, float _permitivityRela, float _width, float _conductivity, QColor
8  : name(_name), permitivityRela(_permitivityRela), width(_width), conductivity(_conductivity), color(_co
9  {
10     std::complex<double> Zt, _Z0, beta0t;
11     std::complex<double> w = (2.0*PI*F);
12     std::complex<double> mu0 = (4.0) * PI * pow(10,-7) ; // permeabilité du vide
13     std::complex<double> epsilon0 = (8.854) * pow(10,-12) ; // permittivité du vide
```

```
    std::complex<double> c = (3.0) * pow(10,8) ; // vitesse de la lumière
    std::complex<double> permi = permitivityRela ;
    std::complex<double> conduct = conductivity ;


    std::complex<double> ewall = std::complex<double>(permitivityRela,0.0) *
                                    epsilon0 - std::complex<double>(0.0, conductivity) / w;


    Zt = sqrt(mu0 / ewall);
    _Z0 = sqrt(mu0 / epsilon0);
    beta0t = (w / c);


    std::complex<double> alpha = (w*sqrt((mu0*epsilon0*permi)/2.0)*sqrt(sqrt(1.0+pow(conduct/(w*epsilon
    std::complex<double> beta = (w*sqrt((mu0*epsilon0*permi)/2.0)*sqrt(sqrt(1.0+pow(conduct/(w*epsilon0


    gamma = std::complex<double>(alpha.real(), beta.real());
    Z = std::complex <double> (Zt);
    Z0 = std::complex <double> (_Z0);

    beta0 = std::complex <double>(w / c);


}
```

Et pour les fichiers .txt utilisés pour stocker les informations des murs :
WallsV1.txt (Murs de l'usine) :

```
0 0 100 0 Brick
100 0 100 -70 Brick
75 -70 100 -70 Concrete
75 -70 75 -45 Concrete
0 -45 75 -45 Concrete
0 -45 0 0 Brick
15 0 15 -4 Partition
0 -9 15 -9 Partition
15 -5 15 -12 Partition
75 -45 85 -27.6795 Brick
0 -18 15 -18 Partition
15 -14 15 -22 Partition
0 -27 15 -27 Partition
```

```
15 -23 15 -31 Partition
0 -36 15 -36 Partition
15 -32 15 -40 Partition
15 -41 15 -45 Partition
35 -20 40 -15 Partition
50 -15 55 -20 Partition
35 -30 40 -35 Partition
50 -35 55 -30 Partition
85 0 85 -12.8398 Brick
85 -15 85 -27.67955 Brick
85 -27.6795 91.5 -27.6795 Brick
93.5 -27.6795 100 -27.6795 Brick
```

WallTypes.txt (Type de murs présent dans l'usine) :

```
Brick 4.6 0.02 0.3 200 0 0
Concrete 5 0.014 0.5 0 0 200
Partition 2.25 0.04 0.1 100 100 100
```

Ex8.1.txt (Murs de l'exercice 8) :

```
0 0 0 80 Brick
0 20 100 20 Brick
0 80 100 80 Brick

```

WallTypesExo8.txt (Type de mur présent dans l'exercice 8) :

```
Brick 4.8 0.018 0.15 200 0 0
```