1 Objectifs

Les deux principaux objectifs de ce projet sont de vous faire découvrir comment un environnement en 3D est transformé pour être affiché sur votre écran en 2D ainsi que de vous faire implémenter quelques techniques utilisées pour optimiser la vitesse d'affichage de cet environnement. La transformation se fera de manière logicielle et non matérielle, OpenGL ne sera donc pas utilisé ici. Le projet prendra ainsi la forme d'un challenge au sein du groupe où chaque binôme essaiera d'obtenir le plus grand nombre d'images par secondes pour différents environnements.

Le langage utilisé sera le C couplé à l'utilisation de la bibliothèque logicielle SDL pour l'affichage 2D ainsi que la gestion des événements clavier. Une description succincte de la mise en œuvre de SDL sera présenté un peu plus loin dans ce document.

Qui dit affichage d'environnements dit création d'environnements. Pour simplifier ce processus, nous utiliserons le principe des *heightmaps* pour générer des terrains vallonnés. La création des heightmaps se fera au travers du logiciel de dessin vectoriel Inkscape permettant ainsi de faire varier la résolution de l'image finale librement. Nous utiliserons aussi Inkscape pour définir le trajet que devra suivre la caméra ainsi que son orientation lors des phases d'évaluation des performances. Une description plus complète est présente un peu plus loin dans ce document.

2 Description

2.1 Transformation 3D vers 2D

La première étape dans l'affichage d'un environnement 3D est de savoir comment le projeter sur un écran 2D. Le premier concept impliqué ici est celui de la caméra. Elle représente le point de vue au sein de l'environnement 3D en terme de positionnement, d'orientation mais aussi du volume affiché à l'écran. En effet, tout comme l'œil humain à un champ visuel d'environ 180° le champ de vision de la caméra est limité par une pyramide tronquée comme décrite par la figure 1 en annexe. C'est ici une vue de profil de la pyramide tronquée formée par la zone grise. Ainsi tout point se trouvant à l'extérieure de cette zone grise ne sera pas projeté sur l'écran. Dans le cas présent, il faut déterminer la coordonnée de x_{2D} qui est le point affiché sur l'écran en fonction du point x_{3D} et du point d'origine de la caméra x_{camera}.

Sur la figure 2 en annexe, les deux vues de profil de la pyramide tronquée sont représentées en haut et à droite. Les points noirs en haut et à droite sont les mêmes mais vus de deux points de vue différents. Les points ont les même coordonnées en 3D sur les axes y et z mais voient leur coordonnée x diminuer. Le carré représente l'écran et les points noirs à l'intérieur de celui-ci sont la projection des points 3D sur l'écran 2D. Ainsi on peut observer que les points noirs se déplacent bien vers nous sur l'écran.

Pour effectuer la projection d'un point 3D (x_{3D}, y_{3D}, z_{3D}) de l'environnement en un point 2D sur l'écran, il y a deux étapes. La première est nommée *Camera Transform*. Elle consiste à faire un changement de repère pour passer le point 3D du repère de l'environnement à celui défini par la caméra. Pour ce faire, nous utiliserons les équations suivantes utilisant la position (x_c, y_c, z_c) et l'orientation $(\theta_x, \theta_y, \theta_z)$ de la caméra au sein de l'environnement :

$$\begin{aligned}
x'_{3D} &= \cos\theta_y(\sin\theta_z(y_{3D}-y_c)+\cos\theta_z(x_{3D}-x_c))-\sin\theta_y(z_{3D}-z_c)\\
y'_{3D} &= \sin\theta_x(\cos\theta_y(z_{3D}-z_c)+\sin\theta_y(\sin\theta_z(y_{3D}-y_c)+\cos\theta_z(x_{3D}-x_c)))\\
&\quad +\cos\theta_x(\cos\theta_z(y_{3D}-y_c)-\sin\theta_z(x_{3D}-x_c))\\
z'_{3D} &= \cos\theta_x(\cos\theta_y(z_{3D}-z_c)+\sin\theta_y(\sin\theta_z(y_{3D}-y_c)+\cos\theta_z(x_{3D}-x_c)))\\
&\quad -\sin\theta_x(\cos\theta_z(y_{3D}-y_c)-\sin\theta_z(x_{3D}-x_c))
\end{aligned}$$

Si la caméra ne peut pas tourner, le changement de repère se résume alors une simple translation :

$$\begin{aligned}
x'_{3D} &= x_{3D}-x_c\\
y'_{3D} &= y_{3D}-y_c\\
z'_{3D} &= z_{3D}-z_c
\end{aligned}$$

Maintenant que nous avons le point 3D transformé $(x'_{3D}, y'_{3D}, z'_{3D})$ et appartenant au repère défini par la caméra, la deuxième étape est de projeter le point 3D pour obtenir un point 2D. Cette étape est nommée *Camera Projection*. Cette opération est assez simple et repose principalement sur le théorème de Thalès (voir figure 1.) Les équation utilisées pour obtenir le point 2D (x_{2D}, y_{2D}) sur l'écran sont :

$$\begin{aligned}
x_{2D} &= \lceil \text{width}((y'_{3D}f_n/x'_{3D})-f_l)/(f_r-f_l)\rceil\\
y_{2D} &= \lceil \text{height}(1-(((z'_{3D}f_n/x'_{3D})-f_d)/(f_t-f_d)))\rceil
\end{aligned}$$

avec la définition de la pyramide tronqué donnée par $(f_l, f_r, f_t, f_d, f_n, f_f)$ décrit par la figure 3 en annexe et width et height respectivement la résolution en largeur et en hauteur en pixels de l'écran. Il est à noter que le point $(0,0)$ sur l'écran se situe en haut à gauche, l'axe y_{2D} étant donc inversé. La pyramide tronquée que nous allons utiliser dans ce

projet sera définie par $(-1, 1, 1, -1, 1.5, 4096)$. La distance maximal d'affichage est donc de 4096 unités mais devra être paramétrable. On fera attention aux divisions par zéro.

Dans ce projet, on se limitera à l'affichage en fil de fer de triangle. Nous venons de voir rapidement comment un point 3D est transformé en un point 2D sur l'écran, il faut maintenant les relier entre eux. Pour ce faire, vous utiliserez l'algorithme de tracé de segment de Bresenham.

2.2 Utilisation de SDL

Pour effectuer l'affichage 2D ainsi que la gestion des évènements clavier, nous utiliserons la bibliothèque logicielle SDL dans sa version 1.2. Nous allons maintenant présenter succinctement les fonctions de base permettant d'initialiser une fenêtre d'affichage de la dimension voulue et éventuellement en plein écran. Il vous est fortement recommandé de faire un tour dans la documentation en ligne de SDL. *

2.2.1 Initialisation de SDL et de l'affichage

La première étape dans l'utilisation de la bibliothèque logicielle SDL est son initialisation. SDL est décomposée en sous-système que l'on peut démarrer ou non. Le premier qui nous intéresse et qui est le plus important est le sous-système vidéo. La fonction utilisée pour démarrer un ou plusieurs sous-systèmes est la suivante :

```
int SDL_Init(Uint32 flags);
```

Le paramètre `flags` permet de définir quels sous-systèmes nous souhaitons initialiser. Dans notre cas, le paramètre serait `SDL_INIT_VIDEO`. On veillera à traiter la valeur de retour et en cas d'erreur d'afficher sa nature par un appel à la fonction :

```
char* SDL_GetError(void);
```

Une fois le sous-système initialisé, il faut le paramétrer par un appel à la fonction :

```
SDL_Surface* SDL_SetVideoMode(int width, int height, int bitsperpixel, Uint32 flags);
```

`width` et `height` étant respectivement le nombre de pixel en largeur et en hauteur de la fenêtre, `bitsperpixel` étant le nombre de bits représentant un pixel (vous mettrez `0` pour simplifier) et `flags` étant les options passées au système vidéo. Puisque nous allons travailler essentiellement au niveau des pixels affichés à l'écran et non avec des textures chargées en mémoire et comme recommandé par la documentation de SDL, nous passerons l'option `SDL_SWSURFACE` pour travailler en mémoire système plutôt qu'en mémoire vidéo. Pour les détails sur ce choix, je vous recommande de vous référer à la documentation de SDL. Une autre option pouvant être passé est `SDL_FULLSCREEN` permettant de demander un affichage en pleine écran. Pour passer plusieurs options, on utilisera le ou bit à bit du langage C `|` tel que :

```
SDL_SWSURFACE | SDL_FULLSCREEN
```

Cette fonction retourne un pointeur vers le type `SDL_Surface` qui représente la surface de l'écran. Nous nommerons cette surface `screen` dans le reste du document. Le type `SDL_Surface` vous est détaillé dans la documentation SDL.

2.2.2 Manipulation des pixels

Dans un premier temps et pour vous simplifier la vie, vous utiliserez la fonction `SDL_FillRect` pour manipuler les pixels de la surface `screen`. La prototype de cette fonction est le suivant :

```
int SDL_FillRect(SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color);
```

Le premiere paramètre sera la surface `screen`, le deuxième paramètre `dstrect` définit le rectangle dans lequel la couleur représentée par le troisième paramètre `color` sera appliquée. Voici un exemple de code permettant de dessiner un pixel noir à la position `x,y` :

```
SDL_Rect pixel = {0, 0, 1, 1};

pixel.x = x;
pixel.y = y;

if(SDL_FillRect(screen, &pixel, SDL_MapRGBA(screen->format, 0, 0, 0, 255)) == -1) {
    fprintf(stderr, "Writing a pixel failed: %s\n", SDL_GetError());
    SDL_Quit();
    exit(EXIT_FAILURE);
}
```

Une fois les manipulations effectuer, il faut mettre à jour l'affichage en faisant un appel à la fonction :

```
void SDL_UpdateRect(SDL_Surface *screen, Sint32 x, Sint32 y, Sint32 w, Sint32 h);
```

*. http ://www.libsdl.org/cgi/docwiki.cgi

avec comme paramètre `screen,0,0,0,0` pour mettre à jour l'ensemble de l'écran.

Il est a noter que `screen` n'est pas réinitialisé automatiquement dans le sens où après un appel à `SDL_UpdateRect`, `screen` n'a pas été modifié. Il convient donc avant toute chose de remplir `screen` d'une couleur pas défaut, par exemple blanc :

```
SDL_FillRect(screen, NULL, SDL_MapRGBA(screen->format, 255,255,255,255));
```

2.2.3 Évènements clavier

Maintenant que nous savons comment afficher et modifier des pixels à l'écran, nous allons voir comment gérer les évènements clavier. Ceux-ci sont gérer par SDL qui les stockent dans une queue d'évènements n'attendant que d'être traités par nos soins. La fonction permettant d'extraire une évènement de la queue est la suivante :

```
int SDL_PollEvent(SDL_Event *event);
```

Cette fonction prend en paramètre un pointeur vers le type `SDL_Event` et permettra d'obtenir toutes les informations relatives à l'évènement. Elle retournera `1` si évènement a été extrait, `0` sinon.

Le type `SDL_Event` est une union dont l'utilisation est un peu différente d'une structure. Dans le cadre de SDL, le champs `type` permet de déterminer le type d'évènement que l'on vient d'extraire. Une fois le type déterminé, on accédera au champs correspondant. Dans notre cas, l'utilisation la plus courante sera quand la touche *Échap* sera pressée ce qui correspond à avoir la valeur `SDL_KEYDOWN` dans le champs `type`. Le champs a utiliser alors sera `key` de type `SDL_KeyboardEvent`. Le code suivant permet de quitter le programme lorsque la touche Échap est pressée :

```
SDL_Event event;
while(SDL_PollEvent(&event) != 0)
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_ESCAPE)) {
        SDL_Quit();
        exit(EXIT_SUCCESS);
    }
```

2.3 Format du fichier d'environnement

Pour définir l'environnement à afficher ainsi que le parcours de la caméra à l'intérieur de celui-ci, on utilisera le format de fichier SVG permettant de décrire une image vectorielle. L'intérêt d'utiliser un format vectoriel est qu'on facilement changer la résolution de l'image exportée sans devoir retoucher à la définition de l'environnement.

Comme présenté au début de ce document, l'environnement se basera sur le principe des *heightmaps*. C'est à dire que chaque pixel de l'image correspondra à un point dans l'environnement 3D. Ainsi la position du pixel au sein de l'image définira les coordonnées x et x du point 3D tandis que la couleur du pixel définira la coordonnée z de ce même point 3D. On peut ainsi voir une heightmap comme une vue du dessus d'une carte en 3D où les points les plus élevés sont en noir (`0xFFFFFFFF` en RGBA) et les points les plus bas sont en blanc (`0x00000000` en RGBA). On peut ainsi deviner qu'une heightmap sera traditionnellement en noir et blanc et qu'une seule des composantes RGB nous intéressera.

Tout comme un fichier vectoriel peut être exporté à une résolution plus élevée, il peut être aussi exporté à une résolution plus basse tout aussi facilement. L'intérêt d'avoir un environnement avec une plus faible résolution et donc avec moins de points 3D est l'utilisation du principe du niveau de détail (*level of detail*(LOD) en anglais). Ce principe est de réduire le nombre de polygones au plus la distance augmente dans un environnement 3D. Ainsi les objets les plus éloignés seront moins détaillés que ceux juste à côté de nous. Il peut ainsi être intéressant dans notre recherche du plus haut nombre d'images par seconde d'implémenter un tel principe.

Le polygone de base que vous utiliserez pour afficher l'environnement est le triangle et chaque triangle utilisera donc 3 points issus du heightmap pour être afficher. Il peut être intéressant d'optimiser leur affichage en utilisant le principe des *triangle strip* présent dans OpenGL. Un exemple de modélisation de l'environnement par des triangles vous est montré par les figures 4 et 5 en annexe.

On utilisera donc Inkscape qui est un logiciel d'imagerie vectorielle pour créer les fichiers d'environnement. En plus de contenir la heightmap, un fichier d'environnement contiendra le chemin ainsi que des indications d'orientation pour la caméra. Le chemin de la caméra sera défini par un chemin formé de courbe de Bézier cubique tracé sur un calque nommé "camera" dans Inkscape. L'idée est d'extraire les points de contrôle du chemin ainsi défini et de les utiliser pour déplacer la camera le long de ce chemin. En effet, à partir des points de contrôle et en utilisant la formule associée aux courbes de Bézier cubique, vous pouvez obtenir des coordonnées 2D en x et y suivant la courbe. L'altitude de la caméra doit varier pour garder une distance fixe entre elle et l'environnement. Cette distance sera de 2 unités par défaut mais devra être paramétrable.

En plus du chemin, il faut définir l'orientation de la caméra le long de celui-ci. Pour simplifier les choses, on définit un ensemble de point le long de la trajectoire. À tout instant du trajet de la caméra, celle-ci devra s'orienter vers le point le plus proche et s'orientera suivant une vitesse de rotation définie vers le nouveau pour le plus proche. Un exemple est illustré par la figure 6 en annexe.

Pour simplier les choses, on supposera que les heightmaps ont une résolution verticale et horizontale qui soit une puissance de 2. Cela permet de diviser facilement la résolution par deux dans le cadre de la mise en place du principe de niveau de détail.

Enfin pour étendre l'environnement en dehors des limites définies par la heightmap, on accolera à chaque bord de l'environnement un miroir de lui même. Pour bien comprendre la chose, référez vous à la figure 7 en annexe.

Cette définition de fichier d'environnement vous laisse libre du ratio entre une unité de distance et un pixel dans la heightmap. Par exemple, si vous avancez de une unité dans votre environnement, de combien de pixel vous serez vous déplacé dans la heightmap. Par défaut, on supposera que le ratio est de 1.

Il est fortement encouragé de calculer hors-ligne ce qui peut l'être. N'hésitez donc pas à définir votre propre format de fichier contenant par exemple les points de passage de la caméra, la heightmap déjà simplifiée, etc...

2.4 Code fourni et outils logiciels conseillés

Pour vous mettre en route ainsi que pour vous aidez dans la compréhension du projet, nous vous fournissons quelques bout de code C ainsi qu'un fichier d'environnement au format SVG. Il vous est fortement recommandé d'y jeter un coup d'œil. Les codes fournis sont :

- Export du fichier SVG en heightmap au format BMP et des données de parcours et d'orientation de la caméra ;
- Lecture d'un fichier BMP (partiellement) ;
- Initialisation de SDL, affichage d'un pixel et gestion de la touche Échap ;
- Calcul du nombre d'images par seconde.

Un makefile pour compiler ces trois fichiers vous est fourni aussi.

Il vous est fortement recommandé d'utiliser GDB et Valgrind pour déboguer votre code et trouver les fuites mémoires. De plus pour évaluer quelle portion du code mérite d'être optimisé, il vous est conseillé d'utiliser Gprof. La documentation de ces 3 programmes est accessible en ligne, il vous est donc aussi conseillé de la lire.

3 Résumé et objectifs à atteindre

Vous aurez bien compris qu'il n'est fait qu'y qu'une brève introduction à tous les concepts nécessitant d'être mise en œuvre pour atteindre les différents objectifs. Le but premier est de développer votre autonomie surtout au niveau de la recherche de documentation ainsi que du partage du travail au sein du binôme.

- Affichage d'une heightmap en fil de fer et en utilisant des triangles ;
- Parcours et orientation de la caméra ;
- Navigation au clavier et à la souris ;
- Extension de l'environnement par effet miroir ;
- Aucun warning à la compilation ;
- Aucune fuite mémoire ;
- Optimisation du code ;
- Utilisation de technique d'optimisation comme le *back-face culling*, les *bounding volumes*, les *quad-tree* ou le *level of detail* ;
- Évolution chiffrée des performances.

N'hésitez pas à créer d'une fichier d'environnement en utilisant Inkscape et de les partager avec les autres binômes. Un classement des meilleurs nombre d'images par seconde pour chaque fichier d'environnement sera publié sur Moodle.

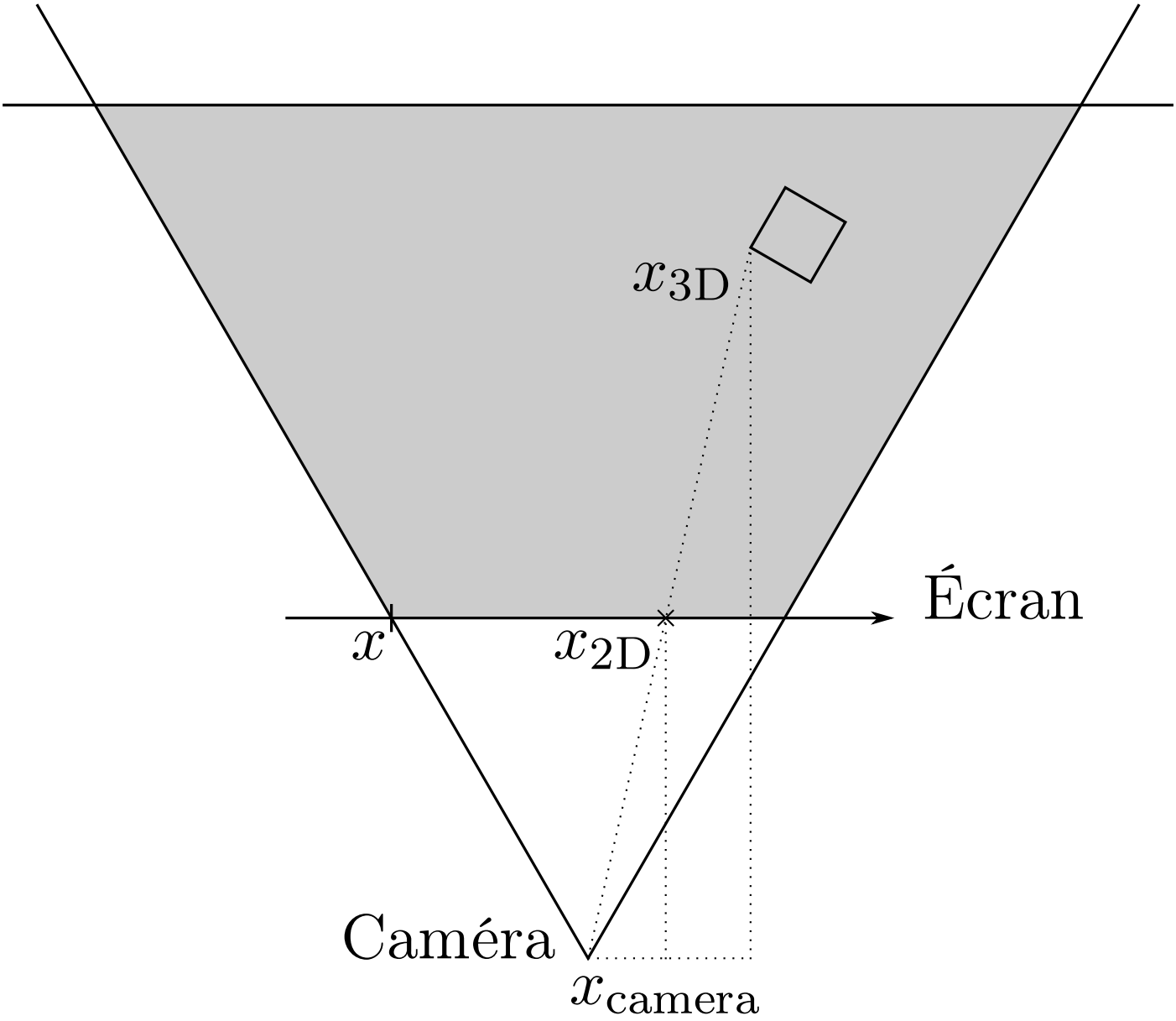

FIGURE 1 – Représentation de la caméra par une pyramide tronquée

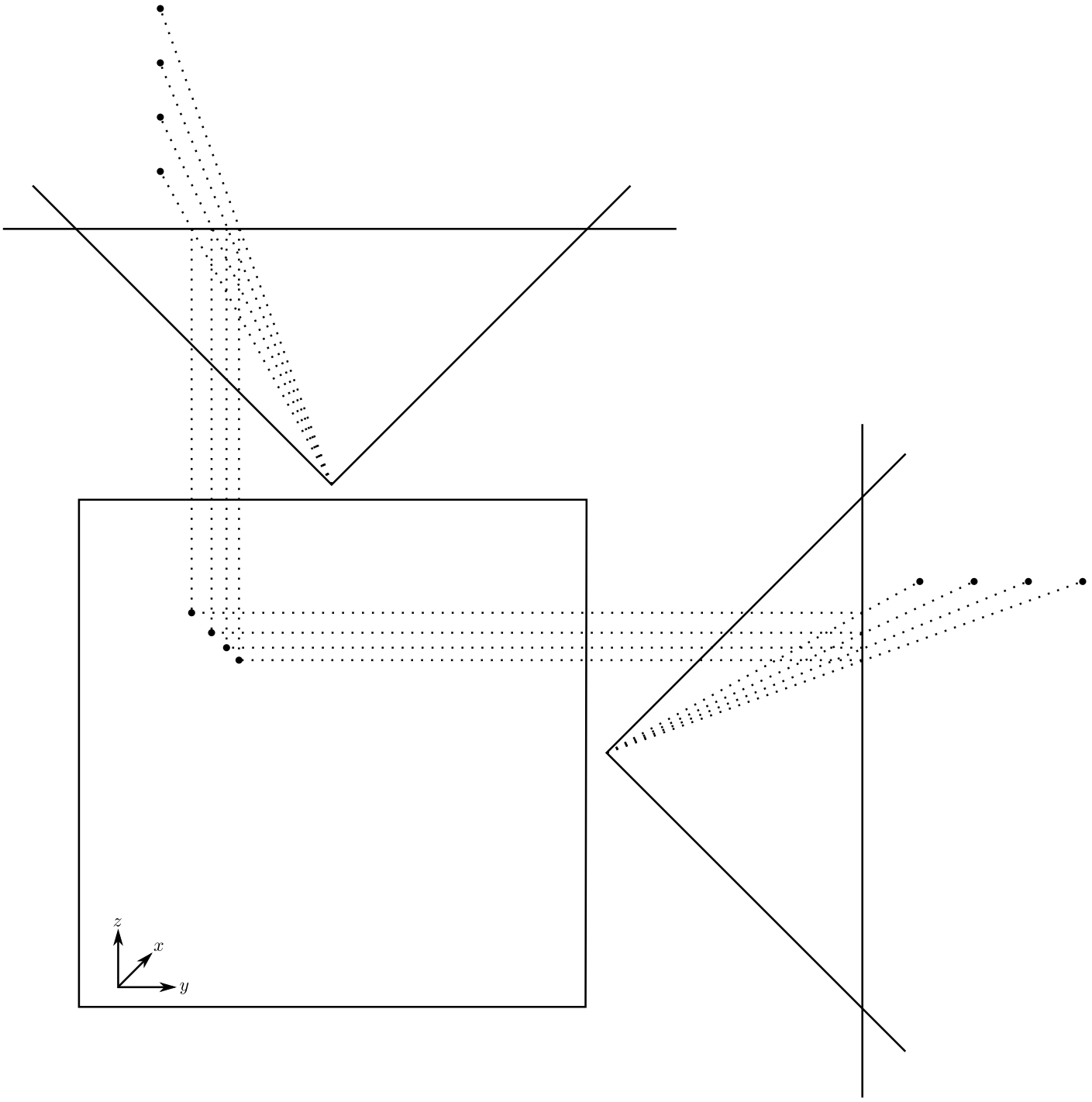

FIGURE 2 – Projection d'un point 3D sur un écran 2D

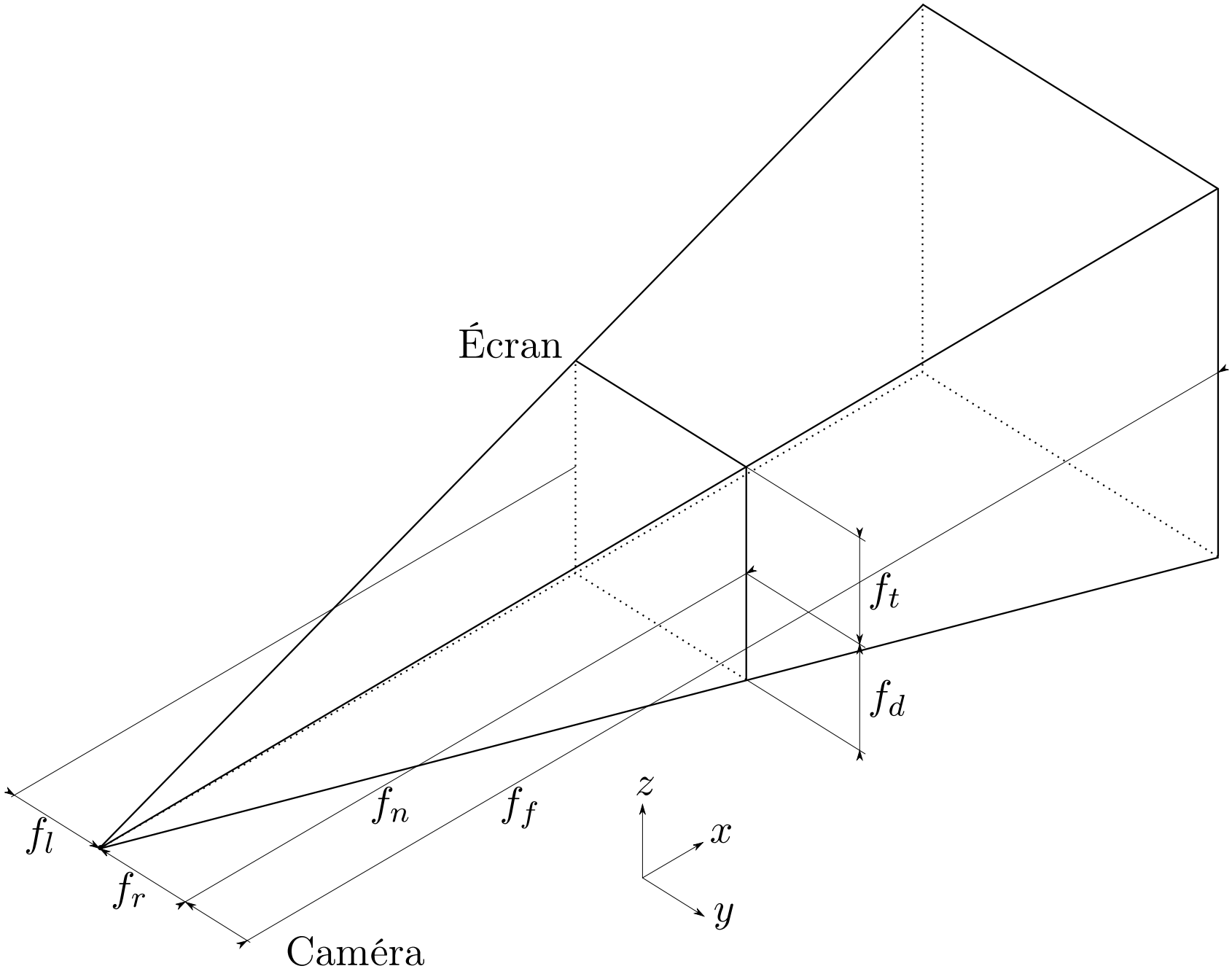

FIGURE 3 – Dimension de la pyramide tronquée

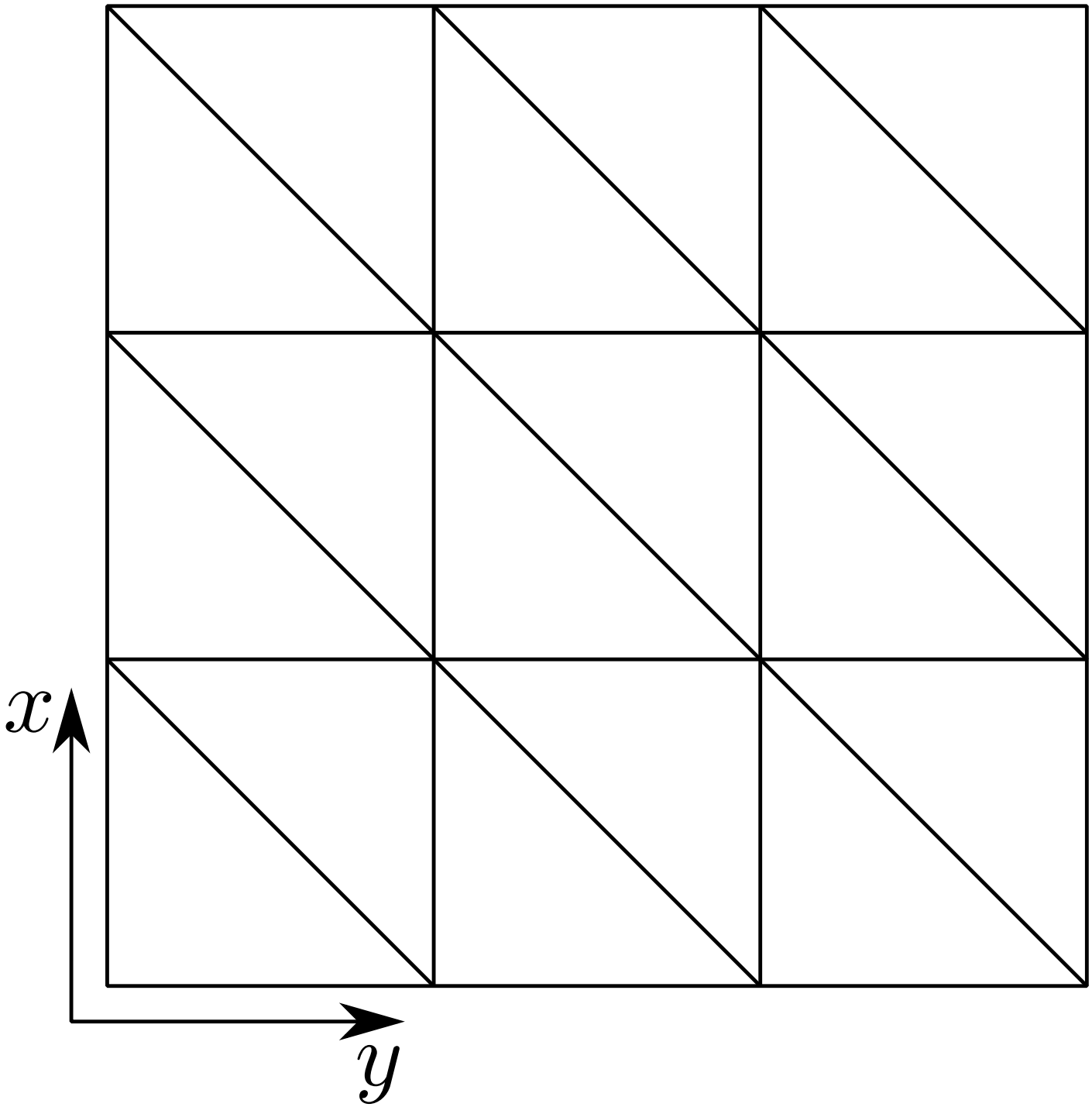

FIGURE 4 – Vu du dessus des triangles pour un environnement issus d'une heigthmap de 4x4 pixels

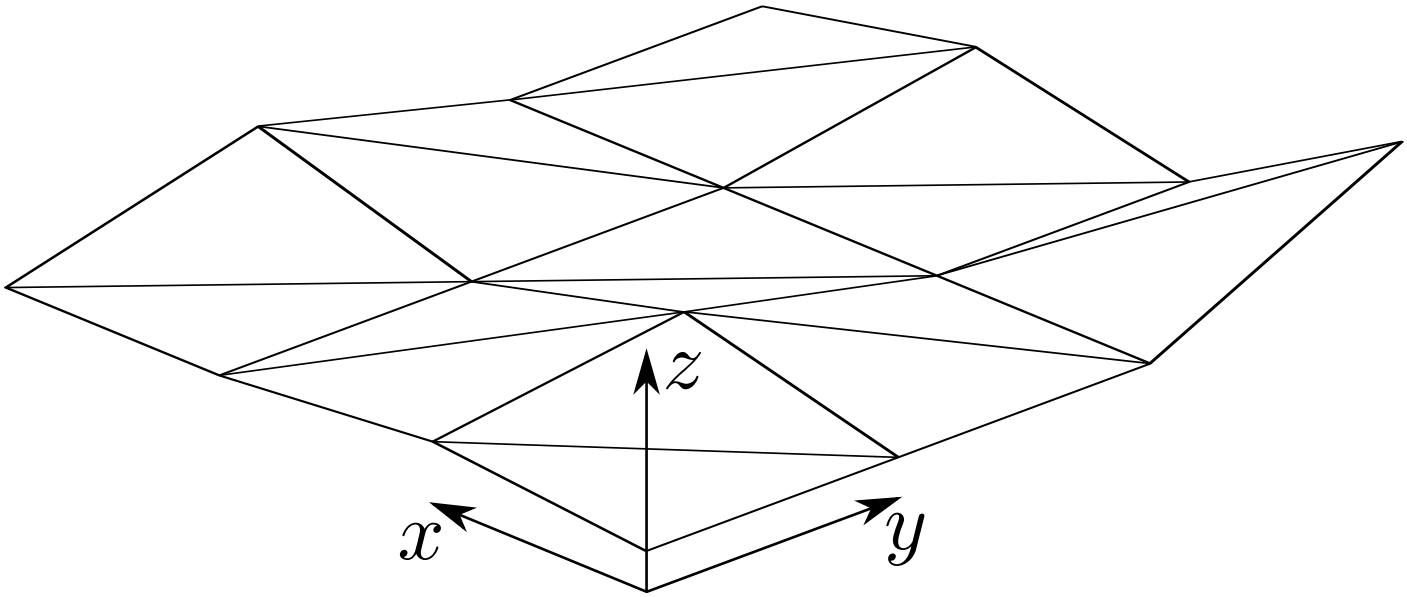

FIGURE 5 – Vu en perspective d'un environnement issus d'une heightmap de 4x4 pixels

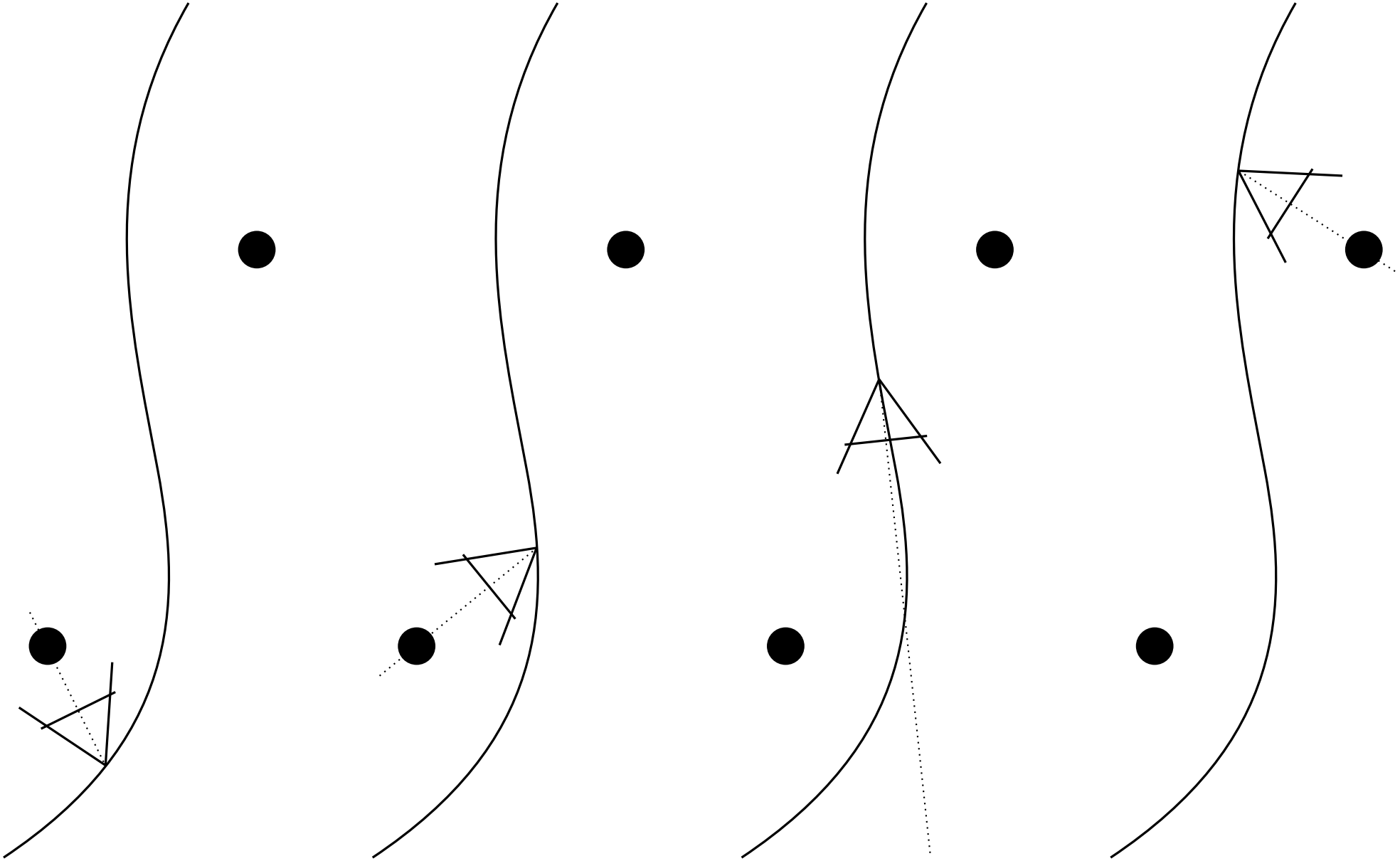

FIGURE 6 – Trajet de la caméra le long d'un chemin avec changement d'orientation

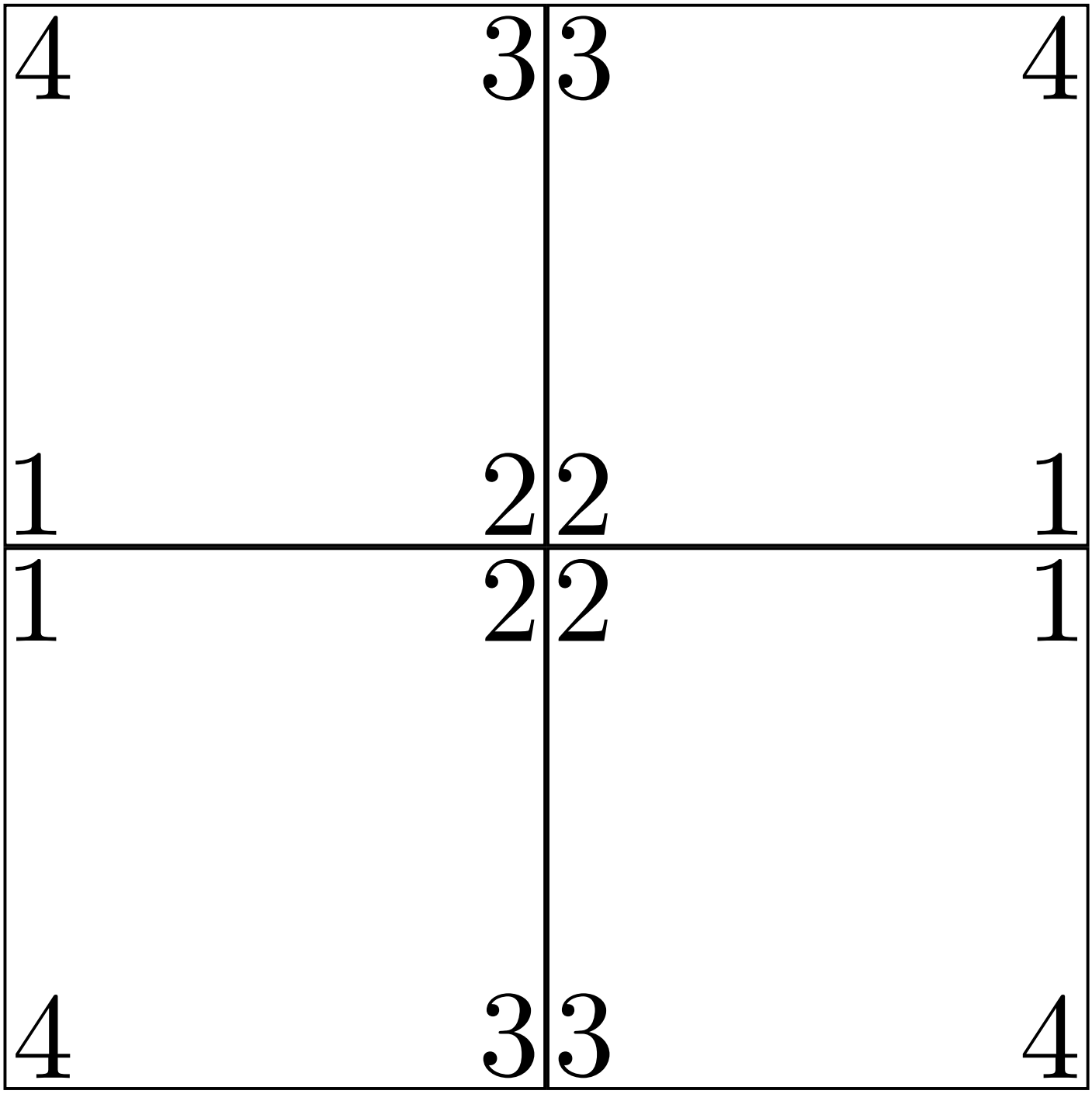

FIGURE 7 – Extension de l'environnement par effet miroir