TP: Géométrie 3D

R5.A.06: Sensibilisation à la programmation multimédia

Abdelkader Gouaïch

2023/2024

Introduction

Ce chapitre explore les concepts fondamentaux de la géométrie dans l'espace, passant de la représentation bidimensionnelle (2D) à la tridimensionnelle (3D). Il est organisé avec les sections suivantes :

- Du plan à l'espace, de la 2D à la 3D
  - Transition de la géométrie du plan, vue dans le chapitre précédent, vers la géométrie spatiale, et introduction des coordonnées 3D.
  - Exploration des différences et des similitudes entre les représentations 2D et 3D.
- Géométrie d'un objet 3D
  - Étude des primitives géométriques en 3D (points, lignes, plans, polygones).
  - Construction et manipulation d'objets 3D complexes à partir de primitives.
- Optimisation avec l'orientation des faces
  - Compréhension de l'importance de l'orientation des faces dans le rendu 3D.
  - Techniques d'optimisation pour déterminer la visibilité des faces (culling).
- Optimisation avec le test de profondeur
  - Présentation du test de profondeur (Z-buffering) pour le rendu correct des objets 3D.
  - Optimisation des performances en utilisant des techniques de test de profondeur.
- La projection orthographique
  - Explication de la projection orthographique et de ses applications.

  - Implémentation de la projection orthographique dans des scènes simples.

- La projection perspective simple
  - Introduction à la projection perspective et à son importance dans la perception de la profondeur.
  - Techniques pour construire une matrice de projection perspective.
  - Application de la projection perspective dans des scénarios de rendu réalistes.

- La projection perspective avec field of view
  - Introduction du concept de champ de vision (field of view - FOV) et de son impact sur la projection perspective.
  - Mise en œuvre pratique du FOV dans des projets de rendu 3D.

- Identification de l'objet caméra
  - Compréhension du rôle de la caméra virtuelle dans la visualisation 3D.
  - Techniques pour définir la position, l'orientation et les paramètres de la caméra.
  - Simulation de mouvements de caméra pour naviguer dans des scènes 3D.

Du plan à l'espace, de la 2D à la 3D

Dans le chapitre précédent, nous avons présenté la géométrie du plan en 2D. Nous avons aussi établi un lien avec les espaces vectoriels de dimension 2 et les matrices carrées 3×3.

Dans ce chapitre, nous allons nous intéresser à l'espace euclidien qui sera de dimension 3. Naturellement, nous allons lui associer un espace vectoriel de dimension 3 et nous allons étudier les transformations avec des matrices 4×4.

```
// Notion de Point dans le Plan 2D
class Point3D extends AbstractPoint {
  constructor(public x: number, public y: number, public z: number) {
    super();
  }
  getDimensions(): number {
    return 3;
  }
  getCoordinates(): number[] {
    return [this.x, this.y, this.z];
  }
  getVector(): Euclidean3DVector {
    return new Euclidean3DVector(this);
```

```
  }
}
```

Nous implémentons un point dans l'espace comme une structure qui comporte trois coordonnées. Nous associons également à chaque point un vecteur euclidien de dimension 3.

```
// Vecteur sur le Plan Euclidien
class Euclidean3DVector extends Vector {
  constructor(point: Point3D) {
    super(point.getCoordinates());
  }
  // Calculer la norme du vecteur
  norm(): number {
    return Math.sqrt(this.x * this.x + this.y * this.y + this.z * this.z);
  }

  // Calculer le produit scalaire
  dotProduct(v: Euclidean2DVector): number {
    return this.x * v.x + this.y * v.y + this.z * v.z;
  }
}
```

Un vecteur euclidien de dimension 3 est semblable à son homologue de dimension 2. La différence réside dans la prise en compte d'une nouvelle coordonnée sur l'axe $\vec{z}$. Un vecteur euclidien 3D reste caractérisé par la notion de distance (ou norme) et le produit cartésien qui mesure le degré d'alignement de deux vecteurs.

Du côté des shaders, nous devons prendre en compte une matrice de transformation et projection de dimension 4×4. Nous allons utiliser la même astuce que précédemment, en augmentant nos vecteurs 3D avec une coordonnée supplémentaire, notée w, et qui vaudra 1. Cette astuce va nous permettre de traiter toutes les transformations, translation, rotation et dilatation de façon uniforme. De plus, nous pouvons séquencer les transformations sous la forme d'un produit matriciel.

Notre vertex shader devient alors :

```
// la position devient un vecteur avec 4 coordonnées
attribute vec4 a_position;
// la matrice de transformation
uniform mat4 u_matrix;
void main() {
  // opération de transformation.
  // attention, nous récupérons que x et y
  gl_Position = u_matrix * a_position;
}
```

☞ **Note importante :** Le répertoire de travail pour cette section est `step1`.

Pour notre lettre T, nous allons ajouter une composante z pour les coordonnées :

```
// Dessiner la lettre T dans un espace 3D
function fillGeometryCoordinates(gl) {
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
      0,
      0,
      0,
      0,
      10,
      0,
      50,
      0,
      0, //T1
      0,
      10,
      0,
      50,
      0,
      0,
      50,
      10,
      0,
      20,
      10,
      0,
      30,
      10,
      0,
      20,
      10 + 40,
      0,
      20 + 10,
      10,
      0,
      20,
      10 + 40,
      0,
      20 + 10,
      10 + 40,
      0,
    ]),
```

```
    gl.STATIC_DRAW
  );
}
```

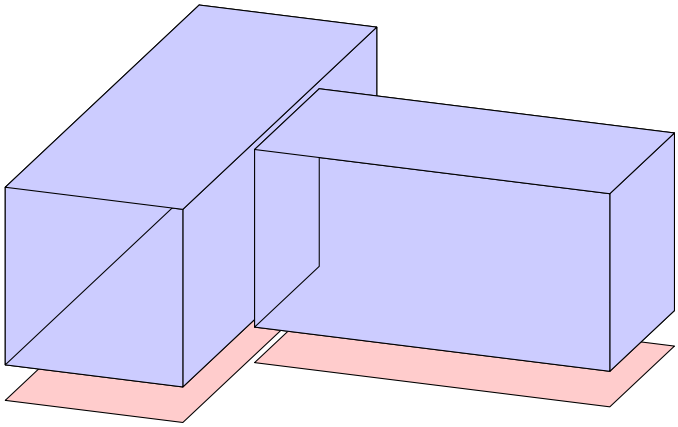
Nous avons maintenant 3 coordonnées par point ; trois points par triangle et un total de 4 triangles. Nous vérifions bien que la taille du tableau est de $36 = 3 \times 3 \times 4$.

✎ **A faire :**

- Visualisez la page `index.html` et manipulez les différents curseurs pour voir l'effet provoqué sur le dessin.
- Vous remarquerez que nous avons 3 angles de rotations qui correspondent aux 3 axes de rotation en 3D. Manipulez les trois curseurs et remarquez que notre lettre est dessinée dans un environnement 3D mais reste 'plate'.

Géométrie d'un objet 3D

☞ **Note importante :** Le répertoire de travail pour cette section est `step2`.



Un objet 3D est composé d'un ensemble de surfaces 3D appelées ses *faces*. Pour la lettre 'T', qui est pour l'instant plate, nous devons définir plusieurs faces qui vont lui donner un volume dans l'espace, comme illustré dans la figure ci-dessus.

Cette figure géométrique est compos

ée :

- De 2×2 rectangles pour la lettre 'T' de base et celle de la profondeur.
- De 4×2 rectangles supplémentaires pour faire le lien entre la base et la profondeur.

Nous avons donc 12 rectangles ; chaque rectangle nécessite 2 triangles et chaque triangle 3 points.

En tout, nous avons besoin de $12 \times 2 \times 3 = 72$ points pour cette figure.

Nous avons également décidé de colorier chaque face avec une couleur différente pour améliorer la visibilité de notre figure.

✎ **A faire :**

- Consultez le code source du vertex shader et du fragment shader. Pouvez-vous identifier les changements afin de prendre en compte la coloration des faces ?
- Consultez le code de la fonction `setupGL()` dans `main.js`. Pouvez-vous identifier les changements pour dessiner la nouvelle forme 3D ?
- Consultez le code de la fonction `setupGL()` dans `main.js`. Pouvez-vous identifier les changements pour intégrer la coloration des faces ?
- Manipulez les curseurs pour voir le résultat. Que remarquez-vous concernant les faces colorées ?

Optimisation avec l'orientation des faces

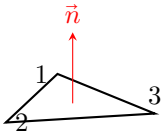
☞ **Note importante :** Le répertoire de travail pour cette section est `step3`.

Vous avez remarqué que les couleurs des faces de notre forme géométrique ne semblaient pas cohérentes. Certaines faces sont dessinées par-dessus d'autres ; de plus, les faces en arrière-plan peuvent être dessinées au-dessus des faces en avant-plan.

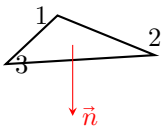
Ce phénomène a une explication simple : l'ordre de coloration est celui du dessin. Si une face est déjà dessinée sur la scène, alors ses couleurs seront écrasées par la coloration des faces dessinées après.

Nous allons voir dans cette section une technique qui va nous permettre d'indiquer si une surface se trouve face à l'observateur et doit être dessinée, ou si elle est orientée à l'opposé de l'observateur et peut dans ce cas être ignorée. Cela se justifie : nous ne souhaitons dessiner que les faces qui sont orientées vers l'observateur et pouvons ignorer celles qui sont opposées.

Pour cela, nous allons donner une orientation aux faces qui va dépendre de l'ordre dans lequel nous dessinons les points.



Prenons l'exemple du triangle de la figure ci-dessus. Si l'ordre de dessin des points est contraire au mouvement de l'aiguille d'une montre, alors la direction de cette surface est vers l'observateur. Cette surface fait donc face à l'observateur.



Si, par contre, l'ordre de dessin des points est celui du mouvement de l'aiguille d'une montre, alors la direction de cette surface est opposée à l'observateur. Cette surface est opposée à l'observateur.

Nous pouvons indiquer à OpenGL d'ignorer toutes les faces opposées avec l'option :

```
gl.enable(gl.CULL_FACE);
```

Avec l'option du culling activée, nous allons dessiner uniquement les faces qui ont une orientation vers l'observateur.

L'avantage de cette technique est qu'avec des rotations, le vecteur normal qui indique l'orientation d'une surface est aussi modifié. Ainsi, une surface qui était cachée, avec une rotation, peut devenir visible et inversement une surface visible peut de

venir cachée.

✎ **A faire :**

- Activez l'option de culling dans la fonction `updateScene`.
- Définissez des fonctions d'aide pour remplir les points des différentes faces correctement.

Après le culling, nous allons obtenir le résultat suivant :

Optimisation avec le test de profondeur

Avec le culling, nous donnons une indication à WebGL quant aux faces qu'il faut dessiner, en utilisant le vecteur normal à la surface qui indique sa direction. Le résultat n'est pas complètement satisfaisant car, comme nous pouvons le voir dans la figure 1, il y a toujours une superposition de surfaces qui ont la même orientation.

Nous allons introduire un nouvel outil pour améliorer l'affichage : le test de profondeur.

Z-buffer

Le Z-buffer est une technique de gestion de la profondeur utilisée pour déterminer quelle surface d'un objet doit être affichée lorsqu'il y a des objets qui se chevauchent dans une scène. Le Z-buffer est essentiel pour résoudre le problème de visibilité et d'occultation des surfaces dans les graphiques 3D.

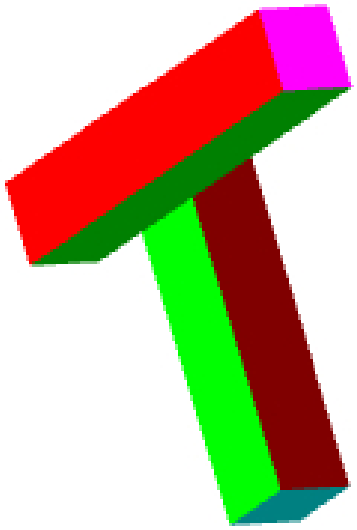


Figure 1: T en 3D avec culling.

Le fonctionnement du Z-buffer peut être résumé en suivant ces étapes :

- **Initialisation** : Lorsque le rendu d'une scène commence, le Z-buffer est initialisé avec une valeur de profondeur maximale. Cette valeur représente la distance la plus éloignée possible dans la scène, souvent définie comme la valeur de clipping loin (far clipping plane) de la caméra.
- **Stockage des valeurs Z** : Chaque fois qu'un pixel est dessiné, le Z-buffer stocke la valeur de profondeur (coordonnée Z) de ce pixel. La valeur Z représente alors la distance entre le point sur la surface de l'objet et la caméra le long de l'axe Z de la vue.
- **Comparaison des profondeurs** : Lorsqu'un nouveau pixel doit être dessiné au même emplacement (x, y) sur l'écran, le système compare sa valeur Z avec la valeur Z déjà stockée dans le Z-buffer pour cet emplacement :
  - Si la nouvelle valeur Z est inférieure (ce qui signifie que l'objet est plus proche de la caméra), le pixel est dessiné et le Z-buffer est mis à jour avec la nouvelle valeur Z.
  - Si la nouvelle valeur Z est supérieure (l'objet est plus éloigné), le pixel n'est pas dessiné car il est occulté par un objet plus proche déjà dessiné.

Ce processus est répété pour chaque pixel de chaque objet de la scène. À la fin, le Z-buffer assure que seules les surfaces visibles les plus proches de l'observateur sont rendues, et les surfaces cachées sont occultées.

✎ **A faire :**

- Activez le test de profondeur avec l'instruction `gl.enable(gl.DEPTH_TEST);` dans la fonction `updateScene()`.

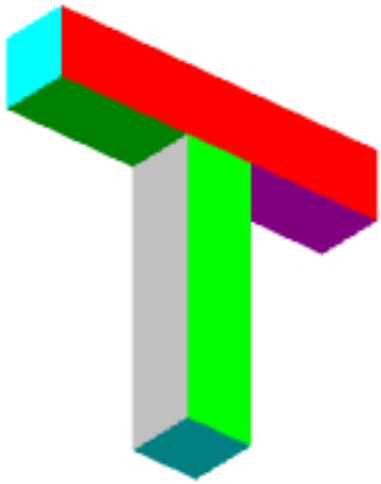


Figure 2: Modèle 3D avec culling et test de profondeur.

Après l'activation du test de profondeur, voici un exemple d'affichage du modèle 3D. Nous sommes maintenant capables de dessiner correctement des objets 3D complexes !

La projection orthographique

Revenons maintenant sur la phase de projection. Il s'agit d'une transformation des points de notre géométrie, dans notre cas un espace euclidien de dimension 3, vers un plan de dessin en 2D.

Dans l'exemple précédent, nous avons utilisé une projection orthographique.

Pour comprendre cette projection, il faut imaginer que notre cadre de dessin est parallèle au plan formé par les vecteurs $(\vec{x}, \vec{y})$. Nous allons projeter les points sur notre cadre de dessin simplement en conservant leurs coordonnées x et y et en mettant la coordonnée z à la valeur de notre plan de dessin.

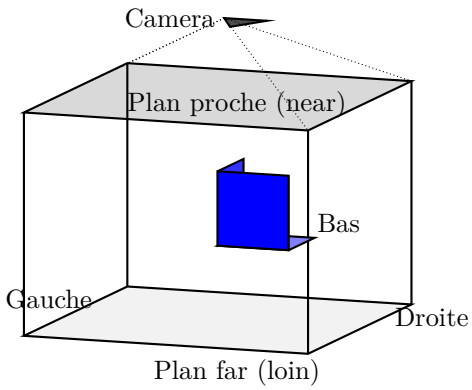
Pour projeter un point de l'espace vers le cadre de dessin, nous allons dessiner une ligne passant par ce point et parallèle à l'axe par $\vec{z}$. Le point d'intersection entre cette ligne et le plan de dessin sera le résultat de la projection. Ensuite, nous pouvons réaliser une dilatation pour les points projetés afin d'agrandir ou de diminuer l'image des objets projetés. Nous ne devons pas oublier qu'il nous

faut toujours transformer les coordonnées pour les faire correspondre au système de coordonnées du cadre de dessin qui va de -1 à 1.

Dans la programmation graphique, il est d'usage de paramétrer la projection orthographique avec les paramètres suivants :

- point gauche
- point droit
- point bas
- point haut
- distance de près
- distance de loin

Ces paramètres vont permettre de définir un volume d'observation, qui est un cube, comme l'illustre la figure **??**. Tous les points des objets se trouvant dans ce volume vont être projetés avec des lignes de projection perpendiculaires à la surface de projection. Les objets, ou parties d'objets, qui sont en dehors du cube seront ignorés par la projection orthographique.



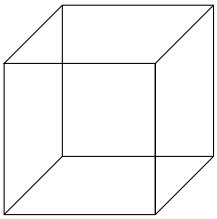
Tous les objets se trouvant dans ce volume vont être dessinés dans notre cadre de dessin avec les propriétés suivantes :

- les distances relatives seront conservées
- les angles entre les lignes seront conservés

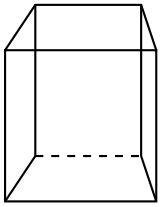
Cette projection a la propriété de conserver les angles invariants ainsi que les proportions entre distances. Cela signifie que deux lignes parallèles dans un plan parallèle à celui de la projection seront projetées comme deux lignes parallèles et que deux objets auront proportionnellement la même taille dans le cadre de rendu.

La projection perspective simple

☞ **Note importante :** Le répertoire de travail pour cette section est `etape4`.



Projection orthographique d'un cube



Projection d'un cube en perspective

Figure 3:

La projection orthographique permet de représenter des objets 3D sur une scène 2D. Elle conserve les angles et les proportions entre distances. Cependant, elle souffre d'un manque de réalisme. En effet, elle ne nous permet pas d'apprécier la profondeur et les distances qui nous séparent des objets dans l'espace.

Notre vision est basée sur une projection en perspective qui rend compte de la profondeur des scènes 3D en modifiant les distances en proportion de leur distance à notre œil. Ainsi, un cube aura ses côtés les plus éloignés de nous plus petits que les côtés les plus proches. Cette information est utilisée par notre cerveau pour reconstituer l'information sur la profondeur 3D qui est perdue au moment de projeter sur un plan 2D.

Nous pouvons illustrer la différence entre les deux projections dans la Figure 3. Nous avons une scène avec un cube en 3D. A gauche nous avons réalisé la projection orthographique de ce cube. Nous remarquons que les cotés du cube sont tous égaux dans la project. Les angles droits des plans parallèle à celui de la projection sont aussi consérvés comme droit et lignes parallèles sont aussi parallèles dans notre projection.

A droite nous avons une projection en perspective. Nous notons que les distances ne sont pas les mêmes entre les cotés. Les cotés les plus éloignés du plan de projection sont plus cours que ceux qui sont proches. Les lignes parallèles ne sont plus parallèles, elles semblent se rejoindre si nous continuons à les dessiner.

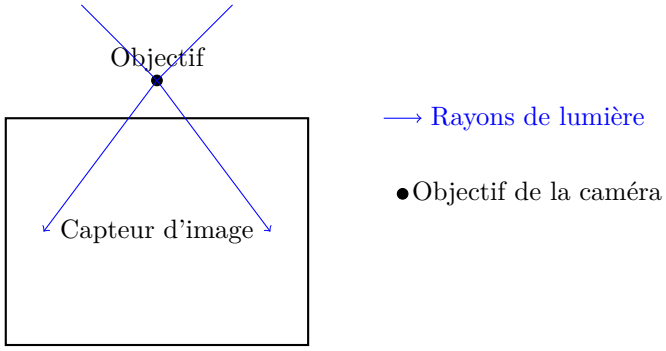
La scène de droite nous semble plus familière car elle correspond à notre expérience de perception qui est en perspective.

Dans cette section nous allons comprendre comment réaliser les projections perspectives.

Comprendre le phénomène de projection perspective

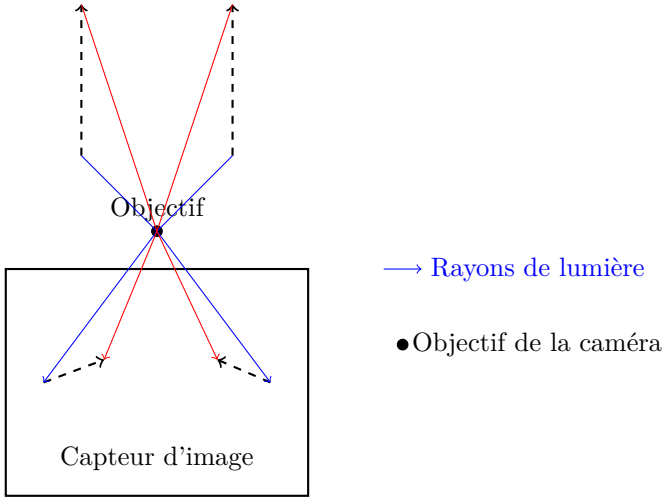
Pour bien comprendre les principes de la projection perspective et s'initier à la géométrie projective qui est un cadre plus général, nous proposons de réaliser l'expérimentation suivante: Prenons un appareil photo, dans le domaine du

graphisme nous parlons d'une caméra; plaçons cette caméra au point p_c et de sorte que son axe de visée soit aligné avec l'axe $\vec{z}$.



Schématiquement, une caméra est un objet composé par :

- un cadre de capture de l'image qui construit la photo suite à la réception des rayons lumineux provenant des différents objets de la scène.
- un objectif qui permet aux rayons lumineux de traverser depuis l'extérieur et rejoindre le cadre de capture.



Comme illustré dans la figure ci-dessus, prenons maintenant l'initiative de déplacer nos deux points pour nous éloigner de l'objectif de la caméra. Il s'agit donc d'un déplacement le long de l'axe $\vec{z}$. Observons ce qui se passe au niveau du cadre de capture d'image.

Nous allons nous rendre compte que les nouvelles coordonnées sont modifiées de sorte que :

- Les nouveaux points sont aussi déplacés vers le haut mais avec une amplitude moindre.

- Les nouveaux points se rapprochent alors que les points dans la scène conservent la même distance entre eux.

Si nous continuons ce processus, les différents points sur le cadre d'image vont dessiner deux lignes qui se croisent à un point sur la ligne de l'horizon.

Nous pouvons à présent énoncer une des propriétés remarquables de la projection perspective : les lignes qui sont parallèles dans la scène vont correspondre à des lignes qui se croisent, à l'infini, dans le cadre de projection.

Modéliser le phénomène de projection perspective

Une première approche

☞ **Note importante :** Le répertoire de travail pour cette section est `step4`.

Passons maintenant à la phase de modélisation pour reproduire le phénomène observé.

Notre première approche de modélisation est en deux étapes:

- premièrement nous allons calculer les coordonnées d'une projection orthographique classique
- dans un deuxième temps nous allons alterer les coordonnées de la projection orthgraphique pour prendre en compte la distance de l'objet par rappoart à la caméra selon l'axe de vision $\vec{z}$.

La projection orthographique, issue de la multiplication matricielle, donne un vecteur $p(x, y, z, w)$ avec les valeurs de x, y, z comprises entre -1 et 1.

Dans la projection orthgraphique nous avons utilisé directement les valeurs de (x, y).

Pour avoir un effet de perspective, nous proposons la transformation suivante: $(x' = \frac{x}{1+z}, y' = \frac{y}{1+z})$.

Ainsi les valeurs de x' et y' vont être inversement proportionnelles aux valeurs de z: plus z sera grand en se rapprocherant de 1, plus x et y seront petits et se rapprocheront du point $(\frac{x}{2}, \frac{y}{2})$

✎ **A faire :**

- Modifiez le vertex shader pour réaliser la projection en perspective avec comme formule $(\frac{x}{1+z}, \frac{y}{1+z})$
- Vérifiez que vous avez un résultat comparable à la figure 5

Une généralisation

☞ **Note importante :** Le répertoire de travail pour cette section est `step5`.

L'effet de perspective que nous obtenons est intéressant mais nous souhaitons l'accentuer.

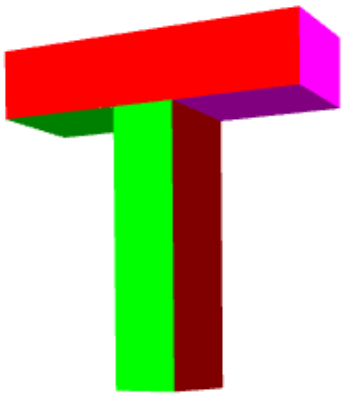


Figure 4: T en 3D avec projection en perspective $(\frac{x}{1+z}, \frac{y}{1+z})$.

Pour cela nous proposons d'introduire un paramètre multiplicatif γ pour réaliser la projection suivante: $(x' = \frac{x}{1+(\gamma \cdot z)}, y' = \frac{y}{1+(\gamma \cdot z)})$.

✎ **A faire :**

- Nous avons défini une variable globale `gamma` ainsi qu'un ascenseur qui permet de la modifier; la scène sera redessiner à chaque modification.
- Modifier le vertex shader pour réaliser une projection avec la formule $(x' = \frac{x}{1+(\gamma \cdot z)}, y' = \frac{y}{1+(\gamma \cdot z)})$
  - Introduisez une variable unifome, `u_gamma`, de type float dans le vertex shader
  - Déclarez une variable `gammaLocation` pour avoir la localisation de l'uniform `u_gamma`
  - Transférez la valeur de l'uniform `gamma` avec la fonction `gl.uniform1f(gammaLocation, gamma);`
- Observez les effets de `gamma` sur la projection perspetive

Le système de coordonnées homogènes

Revenons maintenant sur notre sytème de coordonnées. Nous remarquons qu'avec la projection perspective nous allons transformer un point de l'espace 3D vers un point du plan de dessin en 2D.

Dans sa version simple, cette transformation prend la forme :

$$(x, y, z) \mapsto (x/z, y/z, 1)$$

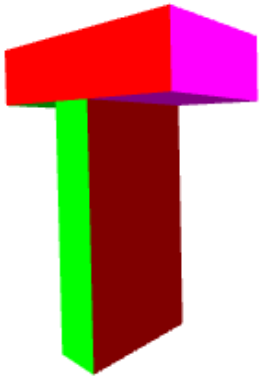


Figure 5: T en 3D avec projection en perspective $(\frac{x}{1+\gamma z}, \frac{y}{1+\gamma z})$ avec $\gamma = 3$.

D'un autre coté, nous avons utilisé des vecteurs 4D pour manipuler les transformations géométrique dans espace 3D. Nos points ont donc été augmenté d'une dimension et sont écrits sous la forme (x, y, z, w).

Pour la projection perspective nous pourrions généraliser ce que nous avons présenté précédemment et la réaliser comme:

$$(x, y, z, w) \mapsto (x/w, y/w, z/w, 1)$$

Il s'agit bien d'une généralisation, car nous retrouvons exactement le même résultat que précédemment pour $w = z$.

Observons attentivement cette projection en 4D: si nous considérons un point de départ et multiplions toutes les composantes par un scalaire (kx, ky, kz, kw) alors la projection perspective ce point sera associé à:

$$(kx, ky, kz, kw) \mapsto (x/w, y/w, z/w, 1)$$

Nous remarquons alors que tous les points de la forme $k \cdot (x, y, z, w)$ ($k \neq 0$ et $w \neq 0$) seront projetés vers le même point $(x_p, y_p, z_p, 1)$. Nous pouvons prendre le point $(x_p, y_p, z_w, 1)$ comme représentant de tous les autres points.

Le point à l'infini

Que se passe-t-il pour les points de la forme $(x, y, z, 0)$?

Nous ne pouvons pas les projeter à cause de la division par zéro. Nous allons les utiliser pour modéliser les points à l'infini. Ce sont les points de rencontre,

à l'infini, des droites parallèles dans le plan Euclidien. Le point à l'infini dans la direction (x, y, z) sera représenté comme $(x, y, z, 0)$.

Coordonnées homogènes

Pour résumer, les coordonnées homogènes permettent une représentation unifiée des opérations géométriques standards telles que les translations, les rotations et les projections.

Dans un espace à n dimensions, un point est normalement représenté par n coordonnées. En coordonnées homogènes, ce point est représenté par $n + 1$ coordonnées.

Il découle de cette représentation les propriétés suivantes:

- **Représentation multiple**: Tous les points de la forme (kx, ky, kz, kw) $kwneq0$ sont équivalents; il peuvent être représenté par un point $(x, y, z, 1)$.
- **La projection**: L'opération de projection d'un point en coordonnées homogènes (x, y, z, w) fait correspondre ce point à $(x/w, y/w, z/w, 1)$
- **Points à l'infini**: Nous pouvons introduire de nouveaux points qui modélisent l'intersection de droites parallèles à l'infini. Ils sont de la forme $(x, y, z, 0)$. L'ensemble de ces points forment une droite : la droite à l'infini (ou la ligne de l'horizon)

WebGL et coordoonées homogènes

L'utilisation des coordonnées homogènee est standard dans le monde du graphisme par ordinateur.

WebGL adopte également cette repérentation des points. La conséquence est qu'une opération comme:

```
vec4 p = vec4( x , y , z , w);
gl_Position = p;
```

va automatiquement convertir le point `p` vers son représentant en coordonnées homogènes $(x/w, y/w, z/w, 1)$.

Nous allons mettre à profit cette propriété pour simplifier la phase de projection. En effet, il nous suffit de placer dans w une valeur appropriée qui sera ensuite utilisée comme dénominateur pour diviser les autres coordonnées.

Par exemple si $w = 1 + (\gamma \cdot z)$ alors nous allons réaliser la projection $(\frac{x}{1+(\gamma \cdot z)}, \frac{y}{1+(\gamma \cdot z)}, \frac{z}{1+(\gamma \cdot z)}, 1)$

La bonne nouvelle est que nous pouvons réaliser cette correspondance de valeur entre w et z par un calcul matricielle.

En effet, observons le résultat de la multiplication cette matrice par un vecteur $(x, y, z, 1)$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Le résultat de $P \cdot (x, y, z, 1)$ est:

$$\begin{bmatrix} x \\ y \\ z \\ z+1 \end{bmatrix}$$

Le point $(x, y, z, z+1)$ sera associé automatiquement, par WebGL, au point $(\frac{x}{1+z}, \frac{y}{1+z}, \frac{z}{1+z}, 1)$

Pour ajouter un facteur multiplicateur γ, il suffit d'utiliser cette matrice :

$$P_\gamma = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \gamma & 1 \end{bmatrix}$$

Pour finir, voici le code de notre librairie `m4` qui permet de générer notre matrice de projection perspective :

```
createPerspectiveMatrixFromZ: function (gamma) {
    return [
        1, 0, 0, 0,
        0,1, 0, 0,
        0,0, 1, gamma,
        0,0, 0, 1,
    ];
},
```

Rappel WebGL crée les matrices en renseignant les colonnes, les unes après les autres. C'est pour cette raison que notre écriture semble différente de l'écriture mathématique.

☞ **Note importante :** Le répertoire de travail pour cette section est `etape6`.

✎ **A faire :**

- Modifiez le vertex shader et la fonction `updateScene` pour utiliser `createPerspectiveMatrixFromZ`
- Vous devriez avoir le même résultat visuel que l'étape précédente.

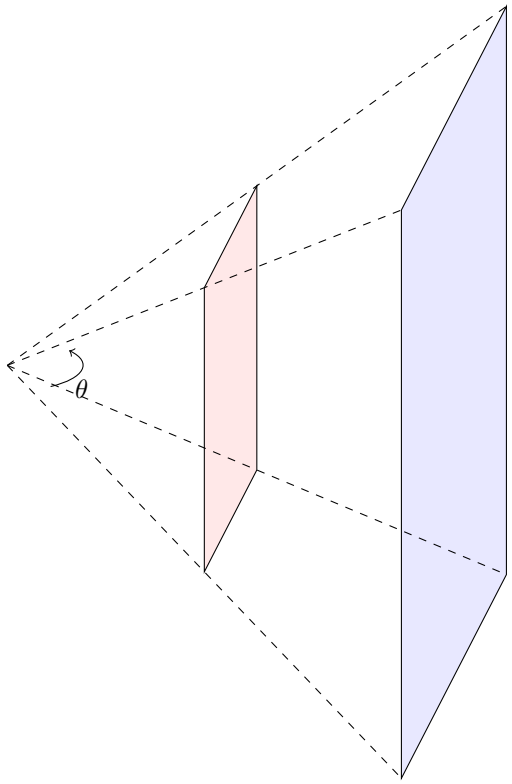


Figure 6: Champ de vision avec ses paramètres : angle de vue, plan proche et plan lointain

La projection perspective avec field of view

Maintenant que nous avons compris le principe de la projection perspective, nous allons présenter une méthode programmatique qui reprend les principes évoqués mais les présente de façon différente.

Le problème reste toujours le même : nous allons réaliser une transformation de nos points 3D écrits en coordonnées homogènes $(x, y, z, 1)$ en des points écrits aussi en coordonnées homogènes $(x', y', z', 1)$.

Nous allons nous intéresser uniquement aux valeurs de x', y' et z' qui sont comprises entre -1 et 1. Ceci va définir pour nous un volume, et tous les points se trouvant dans ce volume seront dessinés et ceux en dehors seront ignorés : la caméra ne peut pas les voir.

Dans le cas de la projection orthographique, le volume de perception était un cube. Nous avons construit ce cube en renseignant les plans proches (near) et lointains (far).

Pour la projection perspective, nous allons spécifier ce volume, appelé le champ de vision, avec les éléments suivants :

- Un angle de vue permettant de déterminer le degré d'ouverture du champ de vision.
- Un plan proche à partir duquel nos objets sont dessinés.
- Un plan lointain au-delà duquel les objets ne sont plus visibles.

Voici la matrice qui réalise la projection perspective dans ce champ de vision :

$$\begin{bmatrix} f/aspect & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & (near + far) \times rangeInv & near \times far \times rangeInv \times 2 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

avec :

$$\begin{aligned} aspect &= \frac{width}{height} \\ f &= tan(\pi/2 - \phi/2) \\ rangeInv &= \frac{1}{near - far} \end{aligned}$$

Le résultat de la multiplication de cette matrice par un vecteur (x, y, z, w) est le suivant:

$$\begin{bmatrix} (f/aspect)x \\ fy \\ (near + far) \times rangeInv \times z + near \times far \times rangeInv \times 2 \\ -z \end{bmatrix}$$

Cela signifie que le vecteur en coordonnées homogènes est :

$$\begin{bmatrix} (f/aspect)x/(-z) \\ fy/(-z) \\ ((near + far) \times rangeInv \times z + near \times far \times rangeInv \times 2)/(-z) \\ 1 \end{bmatrix}$$

Les valeurs de cette matrice font en sorte que les conditions suivantes se réalisent conjointement:

- Les coordonnées des axes $\vec{x}$ et $\vec{y}$ sont mises à une échelle proportionnelle.

- Tous les points se trouvant sur le plan proche (near) avec donc des coordonnées $(x, y, -near)$ vont être projetés vers des points $(x', y', -1, 1)$.
- Tous les points se trouvant sur le plan lointain (far) avec donc des coordonnées $(x, y, -far)$ vont être projetés vers des points $(x', y', +1, 1)$.
- Tous les points se trouvant entre le plan proche (near) et lointain (far) avec donc des coordonnées $(x, y,\ z \in [near, far])$ vont être projetés vers des points $(x', y',\ z' \in [-1, 1])$.
- Tous les points se trouvant dans le plan proche (near) avec une hauteur qui forme un angle de $(\phi/2)$ par rapport à l'origine vont avoir des coordonnées homogènes de la forme $(x', 1, z')$.
- Tous les points se trouvant dans le plan proche (near) avec une hauteur qui forme un angle de $(-\phi/2)$ par rapport à l'origine vont avoir des coordonnées homogènes de la forme $(x', -1, z')$.

Voyons ensemble comment toutes ces propriétés sont réalisées par les valeurs de notre matrice.

Copie de z vers w C'est le composant le plus simple à expliquer. Nous allons simplement copier la valeur de z et la mettre dans w. La transformation inverse le signe pour changer l'orientation de l'axe $\vec{z}$ (c'est une convention).

$$\begin{bmatrix} (1/aspect)f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & (near + far) * rangeInv & near * far * rangeInv * 2 \\ 0 & 0 & \boxed{-1} & 0 \end{bmatrix}$$

Le résultat de la multiplication par un vecteur (x, y, z, w) sera :

$$\begin{bmatrix} (f/aspect)x \\ fy \\ (near + far) \times rangeInv \times z + near \times far \times rangeInv \times 2 \\ \boxed{-z} \end{bmatrix}$$

Normalisation entre largeur et hauteur

$$\begin{bmatrix} \boxed{(1/aspect)} \times f \times x \\ f \times y \\ (near + far) \times rangeInv \times z + near \times far \times rangeInv \times 2 \\ -z \end{bmatrix}$$

Nous allons expliquer la multiplication par $(1/aspect)$. Pour cela imaginons un point se trouvant dans un plan perpendiculaire à l'axe de vision $\vec{z}$ de la caméra

et donc plan paralllèle au plan $(\vec{x}, \vec{y})$. Si ce point appartient à une diagonale qui traverse l'origine de son plan, il aura comme coordonnées dans l'espace (c, c, z_0). En effet, les coordonnées sur $\vec{x}$ et $\vec{y}$ seront égales.

Notre projection se fera sur un plan de dessin qui est un rectangle de largeur $width$ et de hauteur $height$, et les coordonnées dans le cadre de dessin seront exprimées en proportion entre 0 et 1 (en réalité entre -1 et 1 mais cela ne change en rien l'explication donnée ici).

Nous souhaitons, par exemple, que le point haut gauche de notre cadre, qui a pour coordonnées $(width, height)$, corresponde au point de coordonnées $(1, 1)$ dans le cadre de dessin.

Pour répondre à cette contrainte, il suffit de multiplier la première coordonnée par $((height/width) \times x, y)$. Nous pouvons vérifier alors que le point $(width, height)$ va correspondre à $(1, 1)$.

Nous définissons alors par convention que $aspect = width/height$ et nous demandons à ce que les coordonnées de l'axe $\vec{x}$ subissent une dilatation correctrice de valeur $1/aspect$.

Ceci est réalisé dans la matrice de projection par le composant encadré:

$$\begin{bmatrix} \boxed{1/aspect} \times f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & (near + far) \times rangeInv & near \times far \times rangeInv \times 2 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Angle de vue

$$\begin{bmatrix} (1/aspect)\boxed{f}x \\ \boxed{f}y \\ (near + far) * rangeInv * z + near * far * rangeInv * 2 \\ -z \end{bmatrix}$$

Nous souhaitons limiter notre perception à un angle de vue de $\phi/2$ entre l'axe $\vec{z}$ en direction positive de $\vec{y}$, et $-\phi/2$ entre l'axe $\vec{z}$ en direction négative de $\vec{y}$.

Pour cela, nous allons faire en sorte que les points d'intersection du plan incluant l'origine et réalisant un angle de $\phi/2$ avec le plan horizontal $(\vec{x}, \vec{z})$, et le plan perpendiculaire à $\vec{z}$, aient comme coordonnées $(x', y' = 1)$. Pour les valeurs $y' > 1$, WebGL ne va pas dessiner ces points.

Pour les points se trouvant sur cette droite, nous savons par définition de la tangente que :

$$y = \tan(\frac{\phi}{2}) * z$$

Nous savons également que la projection perspective va définir $y' = y/z$.

$$y' = y/z = \tan(\frac{\phi}{2})$$

Si nous souhaitons que $y' = 1$, il suffit de multiplier y par le facteur $1/\tan(\phi/2)$:

$$y' = \left(\frac{1}{\tan(\frac{\phi}{2})} * y\right)/z = 1$$

$1/\tan(\frac{\phi}{2})$ est la définition de la co-tangente et nous pouvons écrire :

$$y' = \left(\cot(\frac{\phi}{2}) * y\right)/z = 1$$

Par ailleurs, nous savons grâce aux tables de conversion trigonométrique que :

$$\cot(\frac{\phi}{2}) = \tan(\frac{\pi}{2} - \frac{\phi}{2})$$

En posant $f = \tan(\pi/2 - \frac{\phi}{2})$, nous arrivons finalement à la relation :

$$y' = (f * y)/z$$

Nous pouvons tenir exactement le même raisonnement pour les coordonnées de l'axe $\vec{x}$ et nous arrivons à :

$$x' = (f * x)/z$$

Conclusion, nous devons multiplier nos coordonnées (x, y) par $f = \tan(\pi/2 - \frac{\phi}{2})$ pour arriver à observer uniquement le volume, c'est une pyramide, délimité par les quatre plans qui réalisent des angles de $\phi/2$ et $-\phi/2$ avec les plans $(\vec{x}, \vec{z})$ et $(\vec{y}, \vec{z})$.

Ceci est matérialisé par les composantes encadrées de notre matrice :

$$\begin{bmatrix} 1/aspect\boxed{f} & 0 & 0 & 0 \\ 0 & \boxed{f} & 0 & 0 \\ 0 & 0 & (near + far) * rangeInv & near * far * rangeInv * 2 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Les plans near et far

$$\begin{bmatrix} (1/aspect)fx \\ fy \\ \boxed{(near + far) * rangeInv * z + near * far * rangeInv * 2} \\ -z \end{bmatrix}$$

Intéressons-nous maintenant aux valeurs encadrées.

Nous souhaitons limiter notre perception aux objets se trouvant entre le plan $near$ et le plan far. Pour cela, il suffit d'affecter une coordonnée z' de -1 aux points se trouvant sur $near$ et +1 aux points se trouvant sur far. En deçà et au-delà de ces valeurs de -1 et +1, WebGL va ignorer ces points.

Voyons comment cela est réalisé.

Premièrement, écrivons la coordonnée homogène z' qui est le résultat de la division par $w = -z$:

$$z' = ((near + far) * rangeInv * z + near * far * rangeInv * 2) / (-z)$$

Mettons-nous sur le plan $near$, ce qui revient à substituer $z = -near$ (nous avons inversé la direction dans $\vec{z}$) dans l'expression et rappelons que $rangeInv = 1/(near - far)$.

En remplaçant les valeurs de z et $rangeInv$ nous avons:

$$\begin{aligned}
\frac{((\text{near} + \text{far}) \times \frac{1}{\text{near}-\text{far}} \times -\text{near} + \text{near} \times \text{far} \times \frac{1}{\text{near}-\text{far}} \times 2)}{-(-\text{near})} &= \frac{((\text{near} + \text{far}) \times -\text{near} + 2 \times \text{near} \times \text{far})}{\text{near} \times (\text{near} - \text{far})} \\
&= \frac{-\text{near}^2 - \text{near} \times \text{far} + 2 \times \text{near} \times \text{far})}{\text{near} \times (\text{near} - \text{far})} \\
&= \frac{-\text{near}^2 + \text{near} \times \text{far}}{\text{near} \times (\text{near} - \text{far})} \\
&= \frac{-\text{near} + \text{far}}{\text{near} - \text{far}} \\
&= -1
\end{aligned}$$

Nous avons bien $z' = -1$ pour les points avec $z = -near$.

Nous pouvons également réaliser la substitution avec $z = -far$ et nous aurons :

$$\begin{aligned}\frac{((\text{near}+\text{far})\times\frac{1}{\text{near}-\text{far}}\times-\text{far}+\text{near}\times\text{far}\times\frac{1}{\text{near}-\text{far}}\times 2)}{-(-\text{far})} &= \frac{(-\text{near}\times\text{far}-\text{far}^2+2\times\text{near}\times\text{far})}{\text{far}\times(\text{near}-\text{far})}\\ &= \frac{-\text{far}^2+\text{near}\times\text{far}}{\text{far}\times(\text{near}-\text{far})}\\ &= \frac{-\text{far}+\text{near}}{\text{near}-\text{far}}\\ &= \frac{\text{near}-\text{far}}{\text{near}-\text{far}}\\ &= 1\end{aligned}$$

Voici les composantes encadrées de la matrice qui réalisent notre transformation :

$$\begin{bmatrix}(1/aspect)f & 0 & 0 & 0\\ 0 & f & 0 & 0\\ 0 & 0 & \boxed{(near+far)*rangeInv} & \boxed{near*far*rangeInv*2}\\ 0 & 0 & -1 & 0\end{bmatrix}$$

Réalisation de la projection par frustum

☞ **Note importante :** Le répertoire de travail pour cette section est `step7`.

✎ **A faire :**

- Réalisez la projection perspective par champs de vision en utilisant la fonction `m4.perspective(phi, aspect, near, far)`
- Testez votre application

Identification de la caméra comme objet

Principe de placement de la caméra

Dans la section précédente, nous avons calculé la matrice de projection afin de dessiner une scène 3D telle qu'elle serait vue depuis l'origine avec un axe de vue correspondant à l'axe $\vec{z}$.

Finalement, nous avons dessiné une scène telle qu'elle serait vue par une caméra si elle était placée au centre, de coordonnées $(0, 0, 0)$ et pointant vers l'axe $\vec{z}$.

Que se passe-t-il si nous déplaçons la caméra par des translations ou des rotations ?

Nous pourrions refaire nos calculs de projection pour obtenir une nouvelle matrice et dessiner la scène. Cela serait complexe et surtout il faudrait le refaire à chaque mouvement de la position et de la direction de la caméra.

Une autre solution consiste à considérer que la caméra sera toujours à l'origine et pointera vers l'axe $\vec{z}$ – nous savons dans ce cas comment calculer la matrice de projection perspective – et à faire bouger tous les objets de la scène pour qu'ils puissent prendre des positions relatives à la caméra correctes.

Mathématiquement, pour réaliser un placement relatif à une caméra qui a subi une transformation de matrice M, nous devons appliquer la transformation M^{-1} aux objets de la scène. La matrice M^{-1} désigne la matrice inverse de M de sorte que le produit des deux matrices donne l'identité :

$$M^{-1} \cdot M = M \cdot M^{-1} = 1_{id}$$

☞ **Note importante :** Le répertoire de travail pour cette section est `etape8`.

✎ **A faire :**

- Visualisez la scène de la page `index.html`; vous remarquez que la caméra est positionnée devant les `T` qui sont positionnés en cercle.
- Réalisez un mouvement de la caméra grâce à l'angle `cameraAngle` pour faire en sorte qu'elle tourne autour des `T`.
- Testez votre application.

La fonction lookat de la caméra

Positionner la caméra devant dans la scène et calculer sa matrice de transformation pour ensuite la prendre comme origine se révèle parfois fastidieux. Une solution pratique serait de simplement donner un point de position de la caméra et un vecteur de direction pour indiquer ce qu'elle observe. À partir de ces informations, nous pouvons déduire une nouvelle base 3D qui sera utilisée pour réécrire les coordonnées de nos objets.

Dans la suite, nous allons donner les étapes de construction de la matrice de la caméra à partir d'un point et d'une direction de visionnage.

Positionner la caméra et déduire la direction de visionnage

La première étape est simple, nous allons :

- donner un point de position pour la caméra `positionCamera` qui correspond au vecteur $\vec{t_c}$,
- donner un point cible dans la scène que la caméra pointe, appelons-le `target`.

Le vecteur de direction de la caméra est simplement $target - \vec{t_c}$. Il va donner la direction de notre axe $\vec{z}$ dans le repère de la caméra. Nous avons juste besoin de le normaliser avant de le considérer comme le vecteur de notre base :

$$\vec{z_c} = \frac{1}{||\vec{target} - \vec{t_c}||}(\vec{target} - \vec{t_c})$$

Construction des 2 autres vecteurs de la base

Nous avons un premier vecteur et nous devons maintenant le compléter avec deux autres vecteurs pour constituer une base.

En mathématiques, il existe un produit vectoriel : il s'agit du produit de vecteurs qui donne un nouveau vecteur !

Dans le cas des espaces vectoriels de dimension 3, le produit vectoriel a une propriété très intéressante : le résultat du produit vectoriel de deux vecteurs $\vec{v}$ et $\vec{w}$ est un vecteur $\vec{k} = \vec{v} \times \vec{w}$ qui est orthogonal aux deux vecteurs.

Nous allons donner un vecteur indicatif pour désigner la direction du haut $\vec{up}$ à la caméra et nous allons déduire le vecteur directeur de l'axe $\vec{x_c}$ par le produit vectoriel $\vec{x_c} = \vec{up} \times \vec{z_c}$.

La dernière étape consiste à construire le dernier vecteur pour compléter notre base en utilisant encore une fois le produit vectoriel entre les deux vecteurs à notre disposition : $\vec{y_c} = \vec{z_c} \times \vec{x_c}$.

Nous avons maintenant les trois vecteurs ainsi que la translation pour placer la caméra. Ces informations vont construire la matrice de transformation pour le changement de repère par rapport à la caméra :

$$\begin{bmatrix} x_c[0] & y_c[0] & z_c[0] & 0 \\ x_c[1] & y_c[1] & z_c[1] & 0 \\ x_c[2] & y_c[2] & z_c[2] & 0 \\ t_c[0] & t_c[1] & t_c[2] & 1 \end{bmatrix}$$

Voici le code de notre objet `m4` qui reprend exactement les étapes que nous venons de décrire :

```
lookAt: function(cameraPosition, target, up) {
        var zAxis = normalize(
            subtractVectors(cameraPosition, target));
        var xAxis = normalize(cross(up, zAxis));
        var yAxis = normalize(cross(zAxis, xAxis));

        return [
           xAxis[0], xAxis[1], xAxis[2], 0,
           yAxis[0], yAxis[1], yAxis[2], 0,
           zAxis[0], zAxis[1], zAxis[2], 0,
           cameraPosition[0],
           cameraPosition[1],
           cameraPosition[2],
           1,
        ];
      }
```

☞ **Note importante :** Le répertoire de travail pour cette section est `etape9`.

✎ **A faire :**

- Repérez les changements dans la fonction `updateScene` pour créer la matrice de la caméra à partir de la fonction `lookAt`.
- Modifiez l'angle de la caméra et observez que la caméra sera toujours orientée vers une lettre T en particulier.