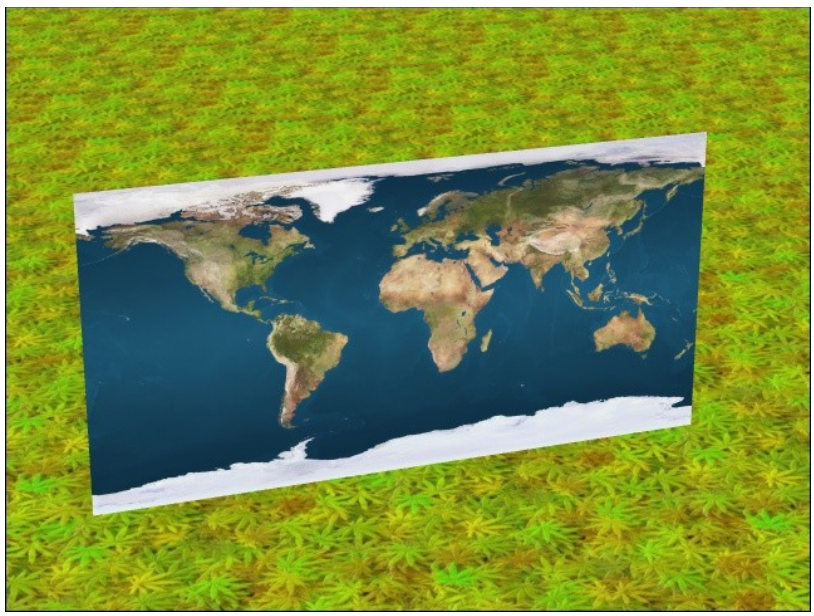
TP n°5 : Textures et effets

Le but du TP est de comprendre comment on utilise les textures pour différents effets temps réel.

Téléchargez l'archive du TP5 qui est sur Moodle et décompressez-la.

Le fait que, sur certains systèmes (Windows), les dossiers `libs` et `data` ne puissent pas être des liens logiques pose un problème au navigateur : le « cross-origin resource sharing » est interdit, voir https://developer.mozilla.org/fr/docs/Web/HTTP/CORS. Pour éviter cette erreur, il y a un script Python à lancer dans chaque projet, `server.py`, et qui consiste en un serveur HTTP sur http://localhost:8000 — cette page correspond à `main.html`, et toutes les références à `libs` et `data` se font dans le dossier au dessus.

1. Application d'une texture : `01—Texture`



Ce projet bac à sable permet de comprendre ce qui concerne l'application d'une image sur un maillage. Chaque sommet définit des coordonnées dites de texture. C'est fait dans la classe `Rectangle`. Ces coordonnées correspondent au repère 2D de l'image. La classe `MaterialRectangle` crée un shader qui applique l'image de la terre. Le vertex shader transmet les coordonnées de texture au fragment shader. Elles sont interpolées au passage entre les deux shaders. Le fragment shader utilise un `sampler2D` associé à l'image. L'association se fait dans la méthode `select`.

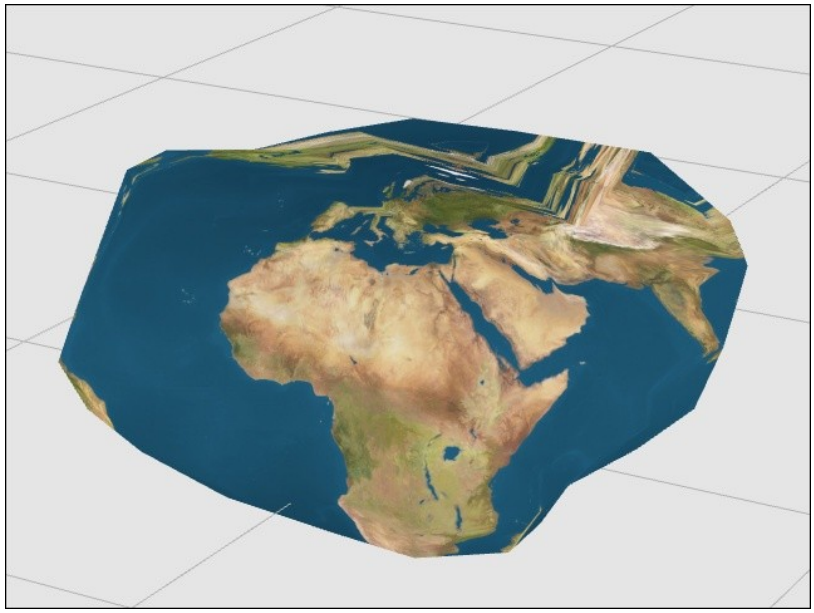
Vous êtes invité(e) à modifier les coordonnées de texture et regarder ce qui se passe. Quelles valeurs faut-il mettre pour inverser la terre verticalement. Quelles valeurs faut-il pour avoir une vue rapprochée de l'Europe et du nord de l'Afrique ?

On va étudier le mode de filtrage. Il est défini lors de la création de la texture à la fin du constructeur de `MaterialRectangle`. Approchez-vous du rectangle avec la touche `Z`. On voit apparaître les pixels. Comparez avec l'image d'origine. Remplacez `gl.NEAREST` par `gl.LINEAR` et regardez si c'est mieux.

Il faudrait maintenant modifier le mode de répétition pour le sol dans `MaterialGround`. C'est le troisième paramètre de la création de la texture (vous pouvez aller voir le source, c'est pas décevant). Remplacez `gl.CLAMP_TO_EDGE` par `gl.REPEAT`. Ce mode est à utiliser dès que les coordonnées de texture sont hors 0..1, voir la classe `Ground`. Le résultat n'est pas terrible à cause du filtrage, même avec `gl.LINEAR` car il y a un repliement de spectre au loin — l'échantillonnage y a une période plus grande que la taille de la texture. La meilleure valeur possible est `gl.LINEAR_MIPMAP_LINEAR`. Elle fait créer des versions réduites et filtrées de la texture afin d'éviter le crénelage au loin.

2. Calcul des coordonnées de texture : `02—Mapping`

Ce projet est optionnel. Il permet de comprendre comment on calcule les coordonnées de texture quand elles ne sont pas définies dans le maillage.



Le but de ce projet est de calculer les coordonnées de texture de manière automatique. Le principe est de transformer les coordonnées 3D des sommets en coordonnées 2D mesurées sur une forme géométrique simple : un cylindre ou une sphère. Cette forme doit englober au plus juste le rocher. Comme le rocher n'est pas régulier, la texture ne sera pas joliment appliquée.

La classe qu'il va falloir compléter est `Rock`. Elle charge un maillage puis appelle la méthode `onRockLoaded`. C'est là qu'on va faire notre tambouille. Une partie est déjà faite : le calcul de la boîte englobante du maillage. C'est tout simplement les bornes min et max sur les 3 axes x, y et z des sommets du maillage. Il y a également le calcul du point central.

Maintenant, c'est à vous : le calcul actuellement présent fait n'importe quoi. Vous devez déterminer l'azimut `s` du sommet relativement au centre et le mettre dans la plage 0..1. C'est l'angle de position relative du sommet par rapport au centre et dans le plan XZ. Vous devez déterminer la hauteur `t` du sommet relativement à la boîte englobante : on sait que `v.m_Coords[1]` est compris entre `min[1]` et `max[1]` ([1] désigne la coordonnée y), il faut donc en déduire la hauteur relative et l'exprimer entre 0 et 1.

Le résultat du « mapping » cylindrique n'est pas terrible sur le rocher. Les pôles sont exagérés. Sauriez-vous modifier (mettez ce qui change en commentaire) le code pour faire un « mapping » sphérique ?

Il reste un problème bizarre : au niveau de la ligne de changement de date ? Pourquoi ? Que faudrait-il faire pour le résoudre (et c'est donc impossible avec ce maillage) ?

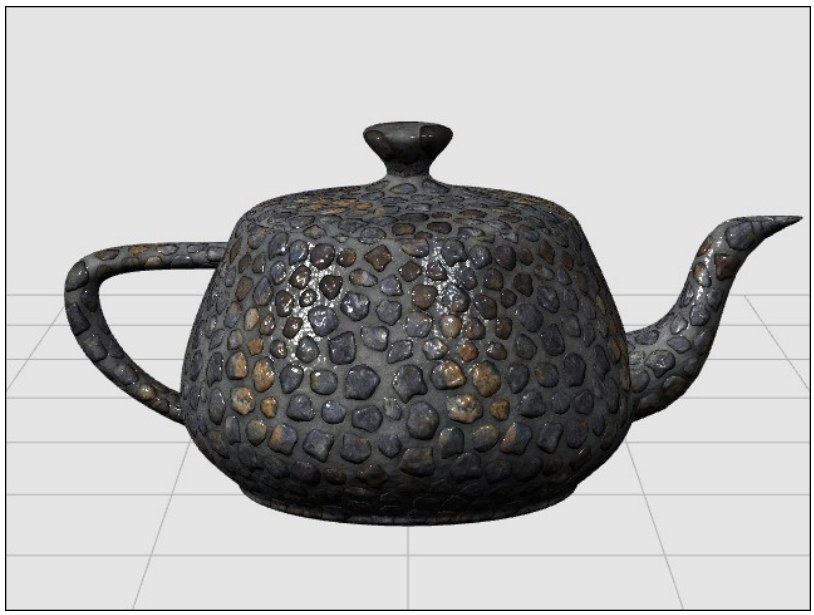
Combien de temps vous faudrait-il pour insérer la sphère par subdivision du précédent TP, lui ajouter des coordonnées de texture et lui plaquer l'image de la Terre ? Par contre, le résultat ne sera toujours pas bon au niveau de la ligne de changement de date. Il faudrait générer la sphère autrement : une sorte de rectangle qu'on replie sur lui-même.

3. Texture procédurale : `03—Procédurale`

Ce projet montre comment calculer une couleur par une fonction mathématique au lieu d'avoir une image à charger. La fonction qui est fournie calcule une teinte définie par trois composantes : le rouge et le bleu dépendent de la position relative à l'objet et le vert dépend de la distance au point (0.5, 0.5) dans l'espace des textures.

Proposez un autre calcul de votre choix pour produire un effet esthétique. Par exemple, des vagues ou des lignes à la surface des objets. Notez qu'il y a des fonctions GLSL pour transformer les couleurs hsv en rgb et réciproquement. On peut s'en servir pour créer un arc-en-ciel sur les objets.

4. Illusion de macro-reliefs : `04—NormalMap`



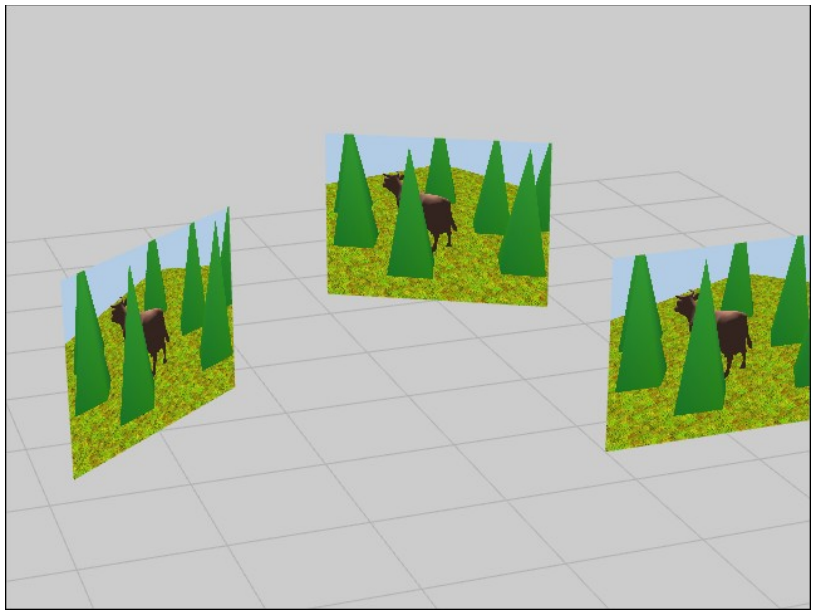
Voici un effet intéressant, très utilisé dans les jeux vidéos, pour donner l'illusion d'un matériau en relief. Tout est dans la définition du vecteur normal. Au lieu d'être calculé géométriquement par la forme des triangles, il est défini par une texture spéciale : ses couleurs indiquent la direction de la normale dans l'espace tangent, aussi appelé repère de Frenet. C'est un repère composé de trois vecteurs (T, B, N) ; T et B sont tangents et N orthogonal à la surface. T et B sont en plus alignés selon les axes s et t des coordonnées de texture. C'est à dire que le vecteur N, calculé par la géométrie, sert à accéder à la texture « normal map » puis il est remplacé par celui de la texture.

Le calcul du vecteur T est déjà réalisé dans la classe `Mesh`, voir la méthode `computeTangents`. Elle ressemble énormément à celle qui calcule les normales. La différence est dans la classe `Triangle`, le calcul du vecteur tangent d'un triangle est spécifique. L'ensemble des vecteurs T est fourni au shader par un VBO créé par `getTangentBufferId`, comme les autres attributs de sommets.

Il vous reste à compléter le fragment shader dans la classe `MaterialTeapot`. Les étapes sont marquées par un ///TODO et expliquées dans les derniers transparents du cours. Pour obtenir un bon résultat, il faut modérer l'inclinaison du vecteur N extrait de la « normal map ». Pour cela, il suffit de multiplier ses coordonnées x et y par 0.5 ou moins. C'est à ajuster en fonction des textures.

Si l'affichage est trop lent, alors changez la théière pour une pomme en décommentant les lignes appropriées dans Teapot.js et Scene.js.

5. Dessin dans une texture : `05—FBO`

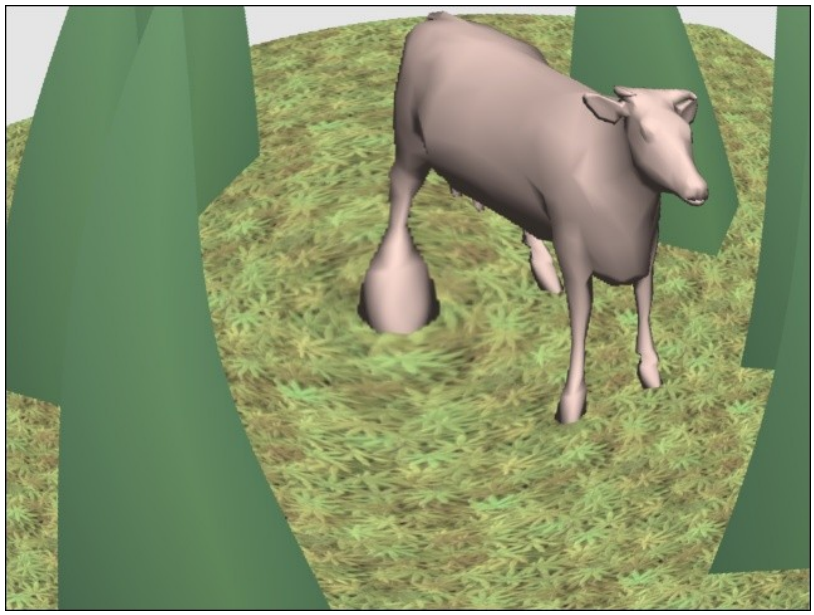


Le but de ce projet bac à sable est de comprendre le fonctionnement des FBO : frame buffer object. Cela ressemble à une texture dans laquelle on peut dessiner. L'écran lui-même est un FBO. Un FBO comprend généralement un buffer pour les pixels (color buffer) et un depth buffer. D'autres buffers peuvent être rajoutés selon les besoins (ex : stencil buffer).

Pour commencer, il se peut qu'il y ait un problème d'affichage de cette scène sur votre PC. Il y a deux versions de WebGL : WebGL1 et WebGL2. Les concepteurs ont fait des erreurs dans les deux. En effet, pour utiliser les FBO dans WebGL1, il faut activer certaines extensions, c'est à dire des modules internes d'OpenGL. Elles fournissent des fonctions comme drawBuffersWEBGL, et des constantes, comme gl.FLOAT. WebGL2 rend certaines de ces extensions actives par défaut, mais pas toutes, et pas avec les mêmes constantes. Du coup, pour essayer de faire quelque chose qui passe partout, on écrit du code spaghetti.

S'il marche, ce projet dessine une première scène comprenant une vache et des pyramides vertes dans un FBO. Ensuite, une seconde scène comprenant plusieurs rectangles sur une grille. Chaque rectangle possède un matériau qui applique le color buffer du FBO.

6. Traitement d’images : `06-Traitement`

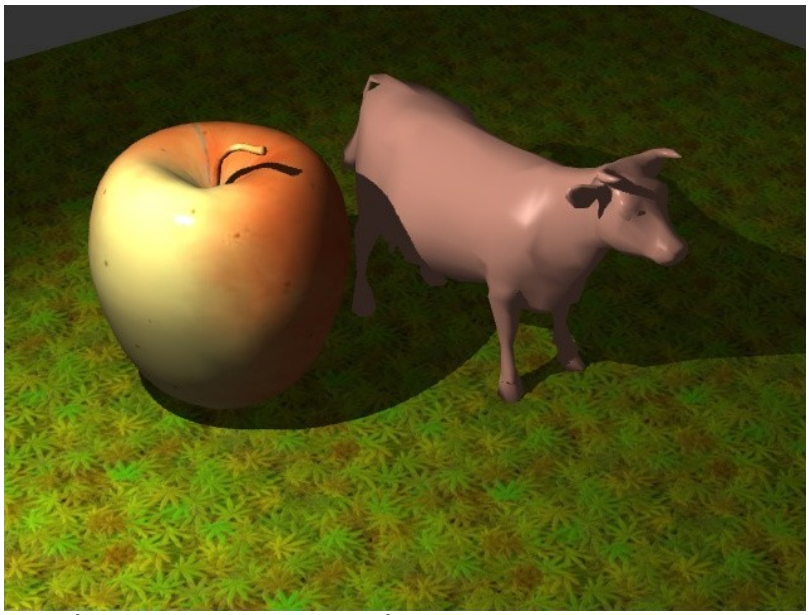


C’est quasiment le même projet que le précédent, mais dans la seconde scène, le rectangle n’est dessiné qu’une seule fois et il occupe la totalité de l’écran. De fait, il n’y a aucune matrice de transformation : ni projection ni matrice de vue.

Ce qui est intéressant, c’est qu’en recopiant l’image du FBO sur le rectangle, on peut appliquer des transformations à la fois sur les coordonnées de texture et sur les couleurs pour faire un effet appelé « post-traitement ».

Le projet proposé montre deux traitements simples : un agrandissement de l’image à partir de son centre et une exagération des couleurs. Il vous est demandé de modéliser un effet de loupe permettant de grossir le centre de l’image, ou alors une sorte de vague comme si on avait jeté un caillou au centre de l’image, voir les démos en séance.

7. Ombres portées : `07—Ombres`



Ce projet permet de découvrir les calculs nécessaires pour dessiner les ombres portées. Il vous suffit de compléter le calcul de la matrice d'ombre pour obtenir un premier résultat. Cette matrice permet de transformer les coordonnées des fragments : du repère caméra, on doit les amener dans le repère de la « shadow map » afin d'extraire la distance z. La shadow map est simplement le depth buffer d'un FBO qui contient la scène dessinée à partir de la position et orientation de la lampe.

Pour faire disparaître l'acné, vous avez plusieurs possibilités. Soit vous rajoutez un décalage de polygones lors du dessin ; c'est une configuration d'OpenGL qui consiste à ajouter un epsilon (tout petit nombre déterminé automatiquement par OpenGL) à la coordonnée z au moment d'écrire dans le depth buffer. Vous pouvez aussi rajouter cet epsilon (0.0001 par exemple) dans la fonction `isIlluminated` du shader. Le principe est de faire en sorte qu'un polygone ne soit pas détecté en tant qu'occultation de la lumière pour lui-même.

Un autre défaut peut apparaître : l'effet de Peter Pan : le décollage des ombres au pied des objets. Ce sont des sortes d'ombres flottantes dans les angles. L'ombre est écartée des objets. Malheureusement, faire disparaître ce défaut génère de l'acné et inversement. On voit donc souvent ce problème dans les jeux vidéos.

8. Transparence : `08-Blending`

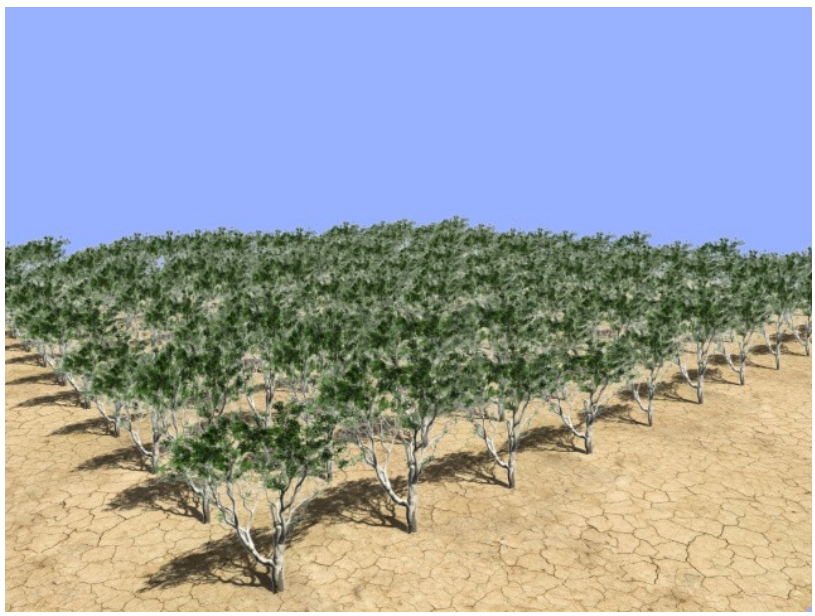
C'est un projet « bac à sable » pour comprendre le mécanisme de mélange d'OpenGL.

La scène est composée de quatre rectangles superposés. Vous commencerez par vérifier dans quel ordre ils sont dessinés par rapport à leur distance à la vue. La souris permet de changer le point de vue.

Ensuite, à vous de jouer : dans la classe `Scene`, décommentez certaines des lignes qui configurent le mélange afin de bien comprendre comment ça marche et quelles sont les possibilités.

9. Ensemble de panneaux : `09-Foret`

Dans ce projet, la scène est composée d'un grand nombre de panneaux. Ce sont chacun de simples rectangles portant une texture partiellement transparente et ici, tous ont la même texture. Le but est d'afficher correctement l'ensemble de ces panneaux.



Les exercices sont en plusieurs étapes : d'abord dessiner un seul de ces arbres, ensuite faire en sorte qu'on voie une image différente quand on tourne autour, ensuite rajouter une ombre portée, ensuite afficher tout un ensemble d'arbres.

9.1. Étude du programme

Commencez par regarder comment fonctionne le programme. Dans la classe `Scene`, le constructeur crée les ressources : textures et billboards. Les textures sont d'un type un peu particulier, `Texture360`. Ce n'est pas une sous-classe de `Texture2D`, mais une sorte de tableaux de textures. Les textures de l'image ci-dessus proviennent du dossier `data/textures/arbres`.

L'idée est d'avoir plusieurs textures disponibles, mais seulement l'une d'elles est affichée, celle qui correspond à l'angle de vision. C'est la méthode `setAngle` qui définit la texture qui sera employée. Ces textures à 360° sont mises en œuvre dans un matériau appelé `Texture360Material`.

Ce matériau a été modifié pour permettre de noircir les fragments – voir la méthode `setCoefficients` et les deux variables uniform `colorCoefficient` et `alphaCoefficient`, et choisir la texture à afficher. Ce noircissement permet de dessiner les ombres portées avec la même texture.

La classe `Billboard` est une sous-classe de `Mesh`. Elle dessine un simple rectangle vertical portant une texture. Une partie du problème consiste à calculer l'angle sous lequel on voit ce billboard pour savoir quelle texture afficher. Il y a trois méthodes de dessin dans cette classe : `onDraw` permet de dessiner une forêt, voir plus loin, et `onDrawSimple` ainsi que `onDrawVerySimple` permettent de dessiner un seul billboard.

Voyons maintenant ce que vous avez à faire. C'est repéré par `/// PARTIE 9.`*N*

9.2. Blending

Dans les méthodes `onDrawVerySimple`, `onDrawSimple` et `onDraw` de la classe `Billboard`, rajoutez les appels OpenGL qui permettent d'activer et désactiver le blending. La formule de blending est définie dans le constructeur de la scène car elle ne change jamais.

9.3. Orientation du billboard selon la vue

Vous pouvez maintenant supprimer la méthode `onDrawVerySimple` qui ne sert que pour comprendre les concepts élémentaires.

Le principe est de faire pivoter le billboard à l'inverse de la souris, de manière à ce qu'il soit toujours face à la caméra. Il faudrait appliquer une rotation qui soit l'exacte inverse de celles que la caméra applique à la scène (au moins deux angles + des translations). Ça peut être un peu compliqué, même avec une inversion de matrice, alors on a une autre idée : on va modifier manuellement la matrice ModelView et annuler manuellement toute rotation dedans. Cette matrice arrive dans le paramètre `matVM` de `onDrawSimple`.

Alors voici ce qu'il faut faire. La matrice matVM est clonée, et on la trafique à la main : on place des 1 dans sa diagonale et des 0 ailleurs dans son coin 3x3 haut gauche. Ainsi, même si la caméra tourne autour, le panneau semblera toujours de face.

En fait, il y a deux approches possibles : soit vous annulez toute rotation dans le coin 3x3, soit vous annulez seulement les rotations en Y. Dans ce second cas, vous n'avez pas autant de valeurs à annuler, il suffit seulement de mettre l'identité dans les colonnes X et Z et laisser le vecteur Y tranquille.

Au lieu de mettre 1 dans la diagonale, il faut faire en sorte qu'il y ait une homothétie définie par `this.m_SizeX` et `this.m_SizeY`. C'est parce que chaque billboard a une taille spécifique.

9.4. Images différentes selon l'orientation : Texture360

Il s'agit d'une petite amélioration. On se rend compte rapidement que c'est toujours la même image qu'on voit. L'idée est qu'en faisant le tour, on change d'image, par exemple tous les 45° s'il y a 8 images. Décommentez le code qui dessine le `m_BbChiffre` dans la classe `Scene` et regardez ce qui se passe quand on tourne autour de l'objet.

Ça se passe dans la classe `Billboard`, méhode `onDrawSimple`. Encore une fois, on examine la matrice ModelView pour extraire l'information sur l'orientation de l'objet. Quand on appelle cette méthode, cette matrice contient la rotation demandée par la caméra. Où se trouvent les informations de rotation, en particulier ce qui concerne l'axe Y ? Le principe consiste à extraire le cosinus et le sinus de cette rotation et à en déduire l'angle. Cet angle est ensuite ramené dans la plage 0..1 pour appeler `setAngle` sur la texture 360°.

9.5. Ombres portées

Encore une amélioration. On veut que chaque billboard projette une ombre sur le sol. On se dit qu'il suffit de dessiner le même billboard avec du noir à la place de la couleur – c'est fait dans le shader, allez voir le rôle des paramètres `coefTeinte` et `coefAlpha` – et il faut également mettre le billboard à plat par terre, à l'aide d'une rotation de 90°. Complétez le code dans `onDrawSimple` et `onDraw` afin de dessiner l'ombre. Outre la rotation de 90°, on réduit aussi la hauteur du billboard pour simuler un éclairage assez haut dans le ciel.

C'est donc une astuce par rapport à l'emploi d'une ShadowMap. Mais, question : serait-il possible d'utiliser une ShadowMap ? Est-ce que les ombres pourraient s'additionner en cas de transparence partielle ?

9.6. Plusieurs billboards

On en arrive au plus intéressant mais au plus compliqué : comment dessiner plusieurs billboards quelle que soit la position de la caméra. Décommentez la partie finale de la fonction `Scene.onDrawFrame`, commentez le reste et testez. On constate que quand on circule autour du paysage, les billboards se mangent parfois entre eux, alors que ça ne devrait pas arriver…

La solution est de les dessiner du plus lointain au plus proche. Il faut donc trier les billboards, mais cela avant de les dessiner. Donc ça sera en deux temps : d'abord il faut savoir où sont les arbres en fonction de la caméra : méthode `Billboard.setModelView`. Elle en déduit la distance du billboard car cette distance est en réalité la composante z d'une translation présente dans la matrice ModelView. Avec cette distance, il suffit de trier la liste des arbres pour les afficher dans le bon ordre. Il vous reste un peu de travail dans `setModelView`. Le reste est fait.

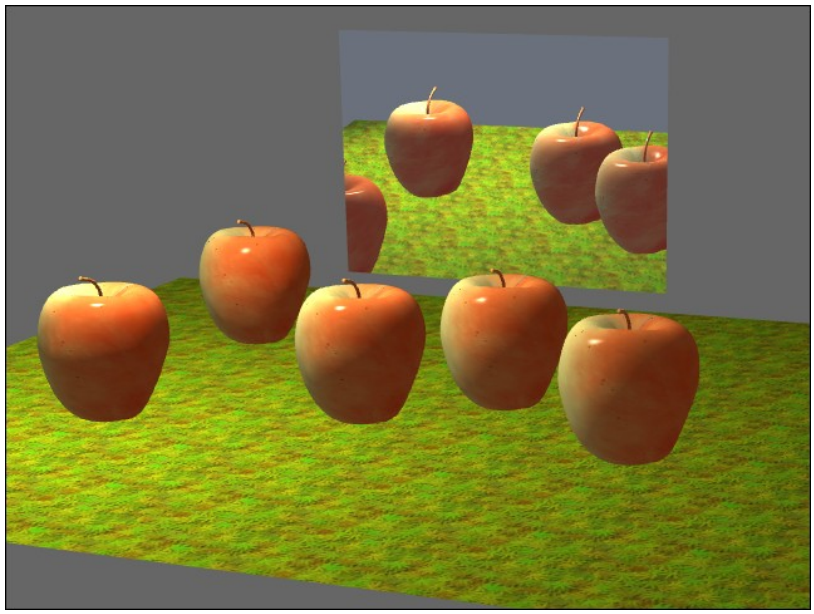
Quand tout marchera, vous pourrez supprimer les méthodes `onDrawSimple` et `onDrawVerySimple`.

Ceux qui auront un peu de temps pourront reprendre la gestion caméra type « première personne » afin de placer le spectateur dans cette forêt. Notez que cette technique des panneaux n'est pas du tout adaptée aux objets proches. Dans les jeux, ils sont dessinés à

l'aide de polygones. Les objets lointains sont dessinés une seule fois dans des FBO, et mis à jour s'il y a de grands déplacements.

10. Reflets : `10-reflets`

Avec cet exercice, on touche à ce qui est le plus complexe dans OpenGL : le séquencement de nombreuses opérations demandant des configurations spécifiques temporaires. C'est le cas pour dessiner des reflets.



Le rectangle en haut à droite semble réfléchir la scène située devant. Le principe est de dessiner cette scène deux fois, dont l'une à l'envers dans le cadre du miroir. Alors d'une part, ce reflet doit être inversé de gauche à droite, et d'autre part, il ne doit pas dépasser du cadre du miroir. Et pour être un peu plus réaliste, il faut l'assombrir un peu.

Remarque : ce double dessin est très coûteux, d'autant qu'il est spécifique à chaque surface réfléchissante. Il impose par exemple de dessiner également les ombres deux fois. C'est pour cela que certains jeux proposent une simplification des reflets, seuls les objets lointains dessinés par un billboard sont reflétés.

Pour dessiner dans la zone du miroir, on fait appel à un *stencil*, c'est à dire un pochoir, un masque qui bloque les dessins en dehors. L'emploi du stencil est simple avec la classe `Stencil.js`. Il faut commencer par l'initialiser dans le constructeur de la scène. Puis à chaque image à dessiner, 1) on découpe une zone, 2) on dessine uniquement dans cette zone, 3) on cesse d'utiliser le stencil. Ce qui est magique, c'est que la zone à découper est définie par un dessin. Ici, on dessine le miroir et ça définit le masque du stencil.

Donc, dans l'algorithme de `onDrawFrame`, on va avoir cet ordonnancement :

1. Mettre le stencil en mode découpe : `Stencil.create();`
2. Dessiner le miroir, mais on ne le verra pas dans l'image finale
3. Mettre le stencil en mode utilisation : `Stencil.use();`
4. Dessiner la scène à l'envers (lampes comprises)
5. Libérer le stencil : `Stencil.disable();`
6. Dessiner la scène à l'endroit

La scène est dessinée deux fois, avec les lampes. Notez que le miroir n'en fait pas partie. Il est commode de mettre les instructions dans une méthode à part : `onDraw(matP, matV)`. La matrice *V* contient la transformation scène vers caméra. C'est la matrice habituelle pour dessine la scène à l'endroit, mais vous devinez avec justesse que ce sera une matrice assez particulière pour dessiner le scène à l'envers. Essayons de la définir.

(Rappel) Quand vous dessinez un objet, trois matrices sont employées : *P*, *V* et *M*. C'est *M* qui indique où est cet objet par rapport à la scène. *V* indique où est la scène par rapport à la caméra. *P* n'est pas concernée par l'effet miroir. Le dessin inversé consiste à seulement modifier *M* : à y insérer une symétrie correspondant à la position et orientation du miroir.

Mettons d'abord le miroir dans une position très simple : vertical, au dessus du (0,0,0) de la scène et face à *z* positif (face à la caméra). Soit *Sz* la matrice représentant cette symétrie en *z* — c'est à dire par rapport au plan *xy*. Par exemple *Sz * (1,2,3) = (1,2,-3)*. Pour dessiner la scène, on utilisera *P*V*Sz*M*. Ainsi tous les objets seront en miroir. En fait, on fournira simplement *V*Sz* en second paramètre à `onDraw`, sachant que *M* est spécifique à chaque objet et définie juste avant leur dessin.

Actuellement, le source `Scene.js` initialise les matrices dans ce cas trivial : une symétrie de plan *Oxy*. Voir la méthode `onDrawFrame` dans laquelle tout va se passer. Ah non, la symétrie n'est pas encore programmée ! Quelle est l'opération matricielle simple qui prend un point *P(x,y,z)* quelconque et qui le transforme en *P(x,y,-z)* ? Elle est à mettre juste après `mat4.identity(this.m_MatVReflected);` et elle doit prendre la matrice `this.m_MatV` en entrée. Le résultat est `this.m_MatVReflected`, la matrice à fournir à `onDraw` pour dessiner la scène à l'envers.

Regardez où `this.m_MatVReflected` est employée. C'est encadré par deux appels à `gl.frontFace`. C'est une instruction OpenGL qui indique quel est le sens de rotation pour définir les faces avant et arrière : CW=clockwise, CCW=counter clockwise. C'est le sens de rotation des sommets d'un triangle pour considérer qu'il est vu de face. Ça agit avec `gl.enable(gl.CULL_FACE);` Pourquoi faut-il l'inverser ?

Dans le cas général, peu importe la position et orientation du miroir, c'est le même principe. Toute la complexité est dans le calcul de la matrice S dans le cas d'un miroir placé n'importe où et orienté n'importe comment. Ce qu'il faut pour dessiner, c'est la matrice `this.m_MatVReflected` contenant la symétrie correspondant au miroir.

Initialement le miroir est un rectangle dans le plan *Oxy*, voir `Miroir.js`. Si vous voulez le bouger et le tourner, agissez sur `this.m_MatVmirror`. Le problème alors, c'est que la matrice `this.m_MatVReflected` ne sera plus aussi simple. La symétrie précédente est correcte en soi, mais elle est exprimée dans le repère local du miroir, qui dans ce cas très simple est aussi celui de la scène. Alors quand le miroir bouge, peut-on insérer un changement de repère scène → miroir, puis appliquer la symétrie, puis faire le changement de repère inverse pour revenir dans celui de la scène ? Il y a donc trois matrices à multiplier correctement pour construire `this.m_MatVReflected` dans le cas général.

Pour finir, le miroir est dessiné avec du mélange afin de teinter légèrement le reflet. En effet, son shader applique une sorte de bleu foncé très transparent. Si on dessine ce rectangle par dessus de l'existant en activant le blending, alors ça assombrira la scène. Remarquez comment on reconfigure OpenGL pour dessiner le dos d'un rectangle. Ce dos doit évidemment disparaître si on voit le miroir de face, c'est pour ça qu'on active quand même la suppression des faces cachées.

L’image du corrigé ci-dessus ne montre pas les ombres portées. Comment faudrait-il procéder pour les voir à la fois dans l’image réelle et dans le reflet ? Là, ça va être réellement difficile. Vous constaterez qu’il y a à la fois des considérations mathématiques (matrices), des dessins spéciaux à faire (stencil, shadow map) et des configurations spécifiques (blending, orientation des polygones, décalage des polygones) qui semblent être des détails mais qui sont cruciales, et pire : leur ordre aussi est crucial.

11. Remise du TP

Déposer un fichier zip contenant votre dossier du TP5, avec chacun des projets dans lequel vous avez travaillé.

Important : faites extrêmement attention aux chemins des dossiers `libs` et `data` présents dans l’archive si vous les avez modifiés. Vous devez faire en sorte que votre projet fonctionne tel quel, sorti de l’archive, sur une autre machine et système que le vôtre. Le correcteur ne peut pas consacrer 10 minutes à éditer chacun de vos sources pour qu’il fonctionne.