Département Informatique

IUT de Saint-Die des Vosges

TD/TP de web GL

Author :
PF Villard

email :
pierre-frederic.villard@
univ-lorraine.fr

13 décembre 2018

Table des matières

1 Fonctionnement des TD/TP

Pour chaque TD/TP, télécharger l'archive `exercicesWebGL.zip` et dézipper la. Elle contient :

— Un fichier `Exercices.html` à ouvrir dans un navigateur pour vérifier l'exécution d'un exercice

— Un dossier `lib` qui contient la bibliothèque `THREE.JS`

— Un dossier `media` qui contient des images et autres données extérieures utilisées par les exercices

— Des dossiers `unit_i` avec `i` qui correspond au cours lié à l'exercice (exemple : unit_1 = exercices sur le cour 1 - *Points, vecteurs et maillage*)

Pour chaque exercice, il faut se rendre dans le dossier correspondant au cours et éditer le fichier JavaScript.

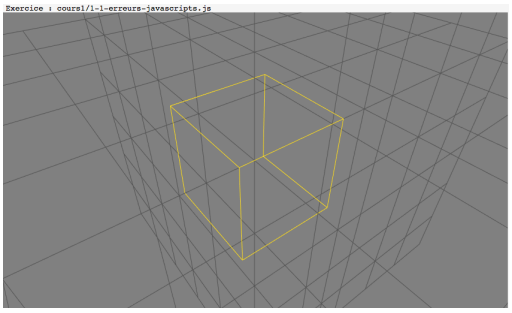
Par exemple, pour l'exercice **1-7, creation d'escalier**, editer le fichier `creation-escalier.js` se trouvant dans le dossier `unit_1` et exécuter le en ouvrant `Exercices.html` dans un navigateur et en sélectionnant `creation-escalier`.

2 Points, vecteurs et maillage

Rappel :
Création de sommet :
vertices.push(new THREE.Vector3(x, y, z));
Création de triangle :
faces.push(new THREE.Face3(vertex1, vertex2, vertex3));

Exercice 1-1 Erreurs javascripts

Corriger les erreurs dans le code JavaScript de façon à obtenir un code sans erreur. Le résultat doit être le suivant :

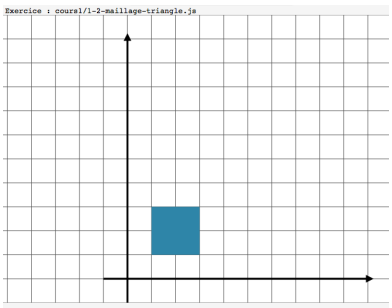


Exercice 1-2 Maillage de triangles

L'objectif est de compléter la fonction square à la ligne 28. La fonction prend 4 paramètres : les coordonées x1, y1, x2, y2 et retourne un objet géométrique (TREE.Geometry()) qui défini un carré au coordonnées voulues.

Pour cela, observer comment le triangle déjà présent a été codé et quels doivent être les coordonnées du nouveau triangle.

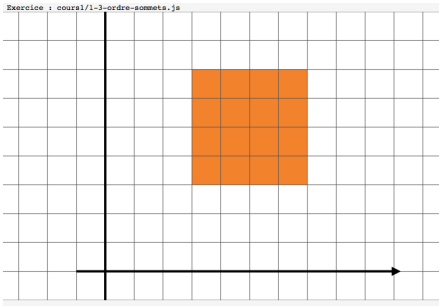
Le résultat doit être le suivant :



Exercice 1-3 Ordre des sommets

Déterminer quel est le problème avec le code et résoudre l'erreur au niveau de l'ordre des sommets. L'erreur se trouve dans la fonction `someObject()` et la correction commence à la ligne 17.

Le résultat doit être le suivant :



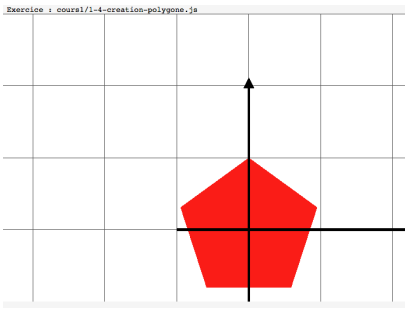
Exercice 1-4 Création de polygone

La tâche ici est de compléter la fonction `PolygonGeometry(sides)` qui prend un paramètre : `sides`, le nombre de coté du polygone. Elle retourne un maillage défini par le nombre minimum de triangles nécessaire pour dessiner un polygone. Le rayon du polygone est 1, le centyre du polygone est (0, 0).

La ligne 29 doit correspondre à l'écriture d'un sommet (voir les codes précédent).

La ligne 33 doit correspondre à l'écriture des face (voir les codes précédent). Essayer d'abord d'afficher les faces une par une avant d'effectuer une boucle `for`.

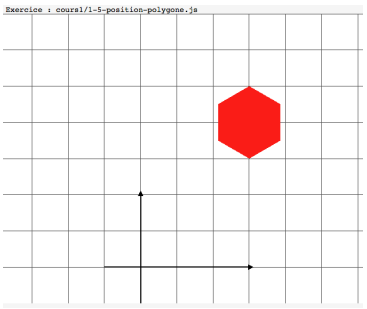
Le résultat doit être le suivant :



Exercice 1-5 Position de polygone

La tâche ici est de compléter la fonction `PolygonGeometry(sides, location)` qui prend deux paramètres : `sides`, le nombre de cotés du polygone et `position`, la position du centre du polygone comme un `THREE.Vector3`. Elle doit retourner le maillage défini par le nombre de triangle minimum nécessaire pour dessiner le polygone.

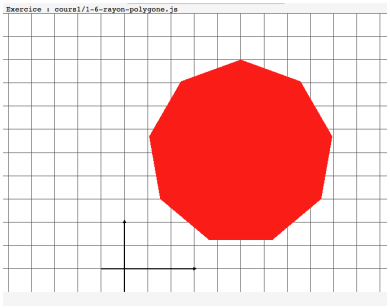
Le résultat doit être le suivant :



Exercice 1-6 Rayon de polygone

La tâche ici est de compléter la fonction `PolygonGeometry(sides, location, radius)` qui prend trois paramètres : `sides`, le nombre de cotés du polygone et `position`, la position du centre du polygone comme un `THREE.Vector3` et `radius`, le rayon du polygone. Elle doit retourner le maillage défini par le nombre de triangle minimum nécessaire pour dessiner le polygone.

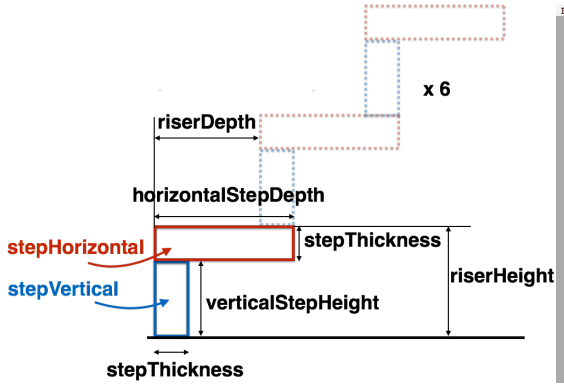
Le résultat doit être le suivant :

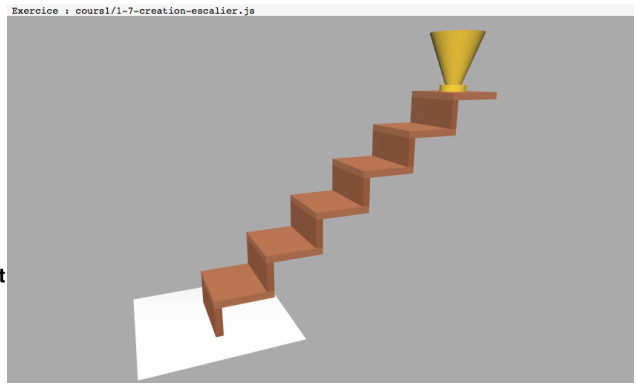


Exercice 1-7 Création d'escalier

La tâche ici est de compléter le modèle 3D d'escalier en utilisant les tailles et couleurs fournies. L'objectif est d'atteindre exactement la coup en or. Il faut bien penser à décaler les objets de type `CubeGeometry`.

Voici un schéma donnant une description des variables à utiliser ainsi que le résultat attendu :





Exercice 1-8 Drinking bird

La tâche ici est de compléter le modèle 3D du Drinking bird. Les matériaux suivants doivent être utilisés :

— Chapeau (`hat`) and colonne(`body`) : `cylinderMaterial` (bleu)

— Tête (`head` et bas du corps (`support`) : `sphereMaterial` (rouge)

— Le reste du corps (`support`) : `cubeMaterial` (orange)

L'axe X va de la droite vers la gauche, l'oiseau va donc dans la direction de -X dans cet exercice.

Pour la résolution des sphères utiliser : `SphereGeometry( radius, 32, 16 );`

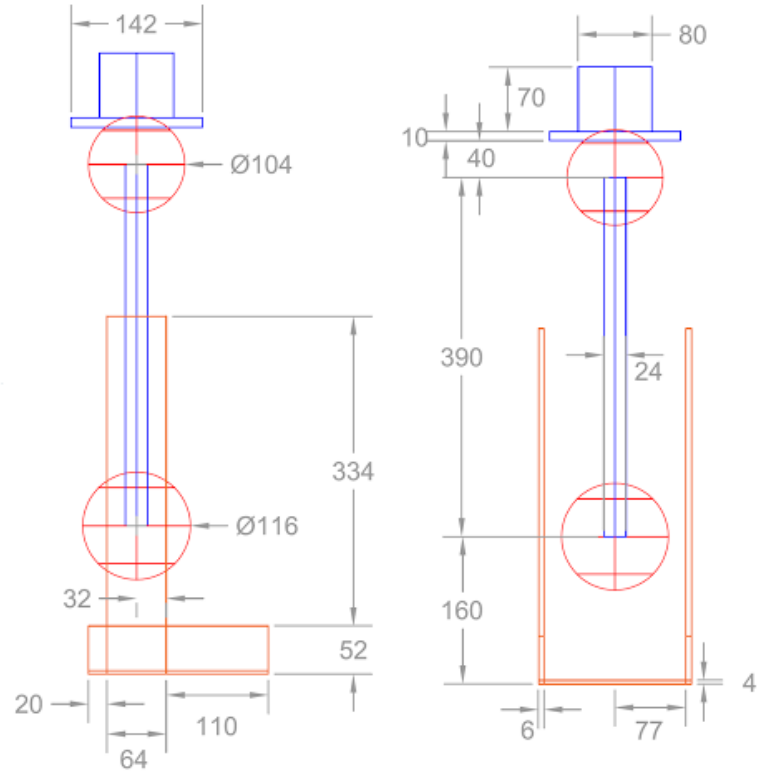
Pour la résolution des sphères utiliser : `CylinderGeometry( radiusTop, radiusBottom, height, 32 );`

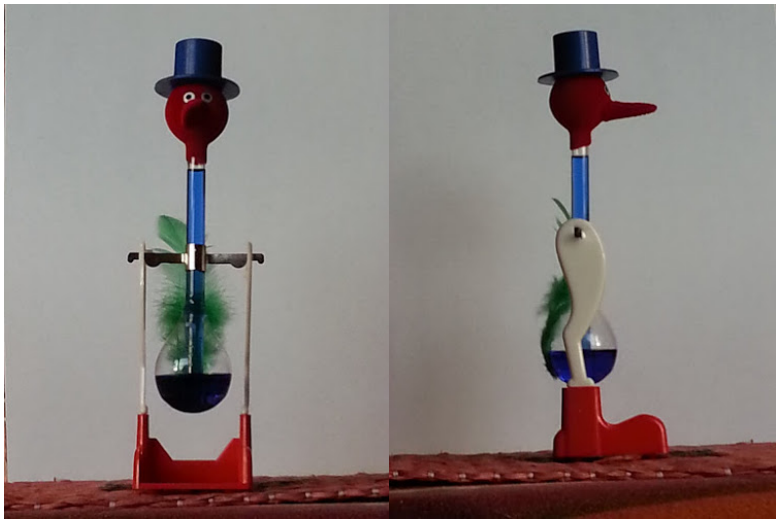
La fonction `createSupport()` sert à modéliser la base, les jambes et les pieds. La base et le pied gauche sont déjà modélisés.

La fonction `createBody(` doit créer le corps de l'oiseau.

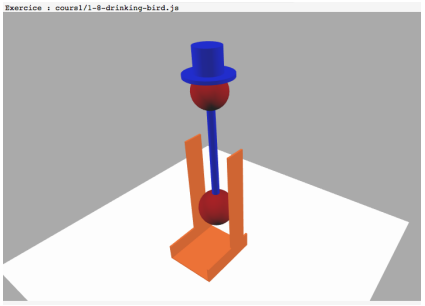
La fonction `createHead()` doit créer la tête et le chapeau.

Les plans du modèle ainsi que des photos sont données ci-dessous :


142
Ø104
334
Ø116
32
52
20
110
64
80
70
10
40
390
24
160
4
6
77



Le résultat doit être le suivant :



3 Les couleurs et matériaux

Exercice 2-1 Vertex attributes

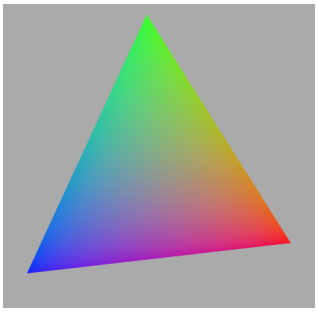
Nous avons vu précédemment qu'un vertex est défini par des coordonnées 3D. On peut attribuer plus de chose à un vertex. Ces données sont les "Vertex attributes". Des couleurs peuvent être ajoutées aux sommets comme vu en cours (http://coursjavavascript.iutsd.univ-lorraine.fr/WebGL/webGL_2.html#/11.

Dans cette exercice dessiner un triangle défini par trois sommets avec les trois couleurs RGB :

1. Ajouter les coordonnées à la géométrie `geometry`
2. Ajouter une face à la géométrie
3. Ajouter une couleur par vertex
4. Créer le triangle avec la geométrie et le matériel `material`
5. Ajouter un triangle à la scène

Vertex	x	y	z	couleur
1	100	0	0	rouge
2	0	100	0	vert
3	0	0	100	bleu

Le résultat doit être le suivant :



Exercice 2-2 Diffuse material

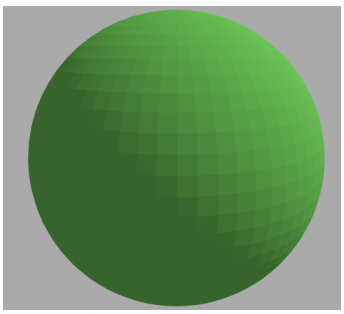
L'objectif de cet exercice est de définir un matériau diffus. Le code par défaut est une sphère qui n'est pas affectée par l'éclairage. La première étape consiste à changer le modèle d'éclairage de matériel basique à Lambert pour avoir le résultat de l'équation 3.1.

$$C = ambient + couleur * \sum (\vec{N}.\vec{L_i}) \tag{3.1}$$

Pour utiliser le modèle Lambertien dans Three.js :

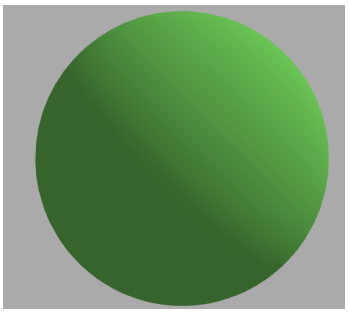
```
material = new THREE.MeshBasicMaterial(
   { color: 0x80fc66, shading: THREE.FlatShading } );
material.color.setRGB( red, green, blue );
var newRed = material.color.r * 0.7;
material.ambient.setRGB( ... );
```

Dans Three.js, la couleur du matériau correspond à la couleur diffuse.
Définir la couleur ambiante telle qu'elle soit égale à la couleur du matériaux*0,4.
Le résultat doit être le suivant :



Exercice 2-3 Smooth Lambert

Enlever le *flat shading* de l'exercice précédent.
Le résultat doit être le suivant :



Exercice 2-4 drinking-bird-shiny.js

L'objectif de cet exercice est de reprendre le projet du *Drinking bird* du TP précédent (exercice 1-10).

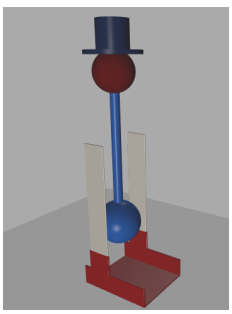
Changer les matériaux des chapeau, corps, jambes et pied pour avoir un modèle **spéculaire** avec les propriétés suivantes :

objet	brillance	couleur spéculaire
Hat, Body	100	0.5, 0.5, 0.5
Leg	4	0.5, 0.5, 0.5
Foot	30	0.5, 0.5, 0.5

Pour utiliser le modèle de Phong dans Three.js :

```
var aMaterial = new THREE.MeshPhongMaterial( { shininess: 10 } );
aMaterial.specular.setRGB( 0.1, 0.2, 0.3 );
```

Le résultat doit être le suivant :



Exercice 2-5 drinking-bird-transparency.js

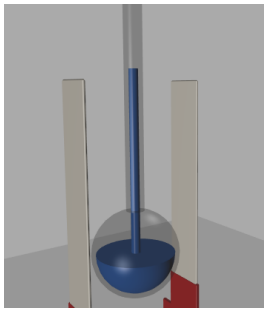
Utiliser le *Drinking bird* de l'exercice précédent pour ajouter de la transparence.

Pour cela, créer un matériel `glassMaterial` définit par une certaine transparence. Appliquer ce matériel au corps du *Drinking bird*. Il doit être : opaque à 30%, de couleur noir et avec une couleur spéculaire blanche.

Afin de créer un liquide bleu à l'intérieur, ajouter un cylindre et une demi-sphère bleus :

```
var sphere = new THREE.Mesh(
    new THREE.SphereGeometry( 104/2, 32, 16, 0, Math.PI * 2, Math.PI/2, Math.PI ),
        bodyMaterial );
```

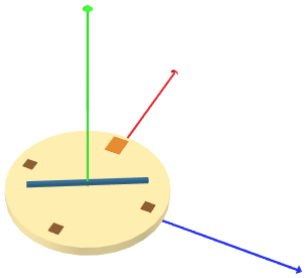
Le résultat doit être le suivant :



4 Les transformations

Exercice 3-1 Rotate a block

L'objectif de cet exercice est de tourner une géométrie pour simuler une aiguille sur une horloge. L'aiguille doit indiquer 2h et 8h.
Le résultat doit être le suivant :

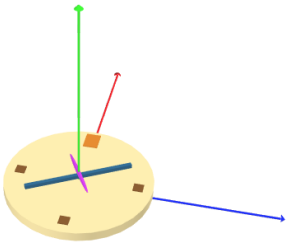


Exercice 3-2 Scale a sphere

L'objectif de cet exercice est de représenter l'aiguille des heures sur l'horloge précédente.
Il y a deux parties :

1. changer l'échelle d'une sphère (attention aux axes) pour avoir une aiguille longue de 60 unités et large et haute de 4 unités ;
2. tourner l'aiguille pour qu'elle indique 11h et 5h.

Le résultat doit être le suivant :

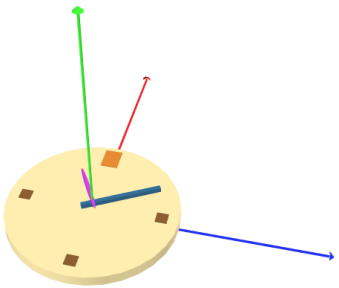


Exercice 3-3 Two clock hands

L'objectif de cet exercice est de déplacer les aiguilles de l'horloge précédente pour qu'elle tourne à partir de l'origine.

La configuration par défaut représente le résultat de la transformation avec une simple translation.

Le résultat doit être le suivant :

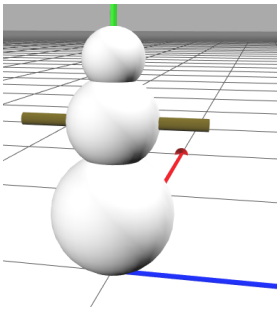


Exercice 3-4 Snowman

L'objectif de cet exercice est de réaliser un bonhomme de neige, et plus spécialement de positionner un bout de bois pour représenter les bras.

Le centre du cylindre doit être positionné à 50 unités au dessus du sol.

Le résultat doit être le suivant :



Exercice 3-5 Extended robot arm

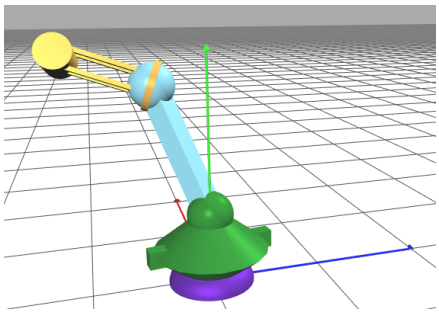
L'objectif de cet exercice est de rajouter une partie mécanique à un bras robotique.

Utiliser la fonction `createRobotBody` déjà créée.

Penser à modifier correctement le graphe de scène.

Dé-commenter les bonnes ligne spour avoir l'interface graphique.

Le résultat doit être le suivant :



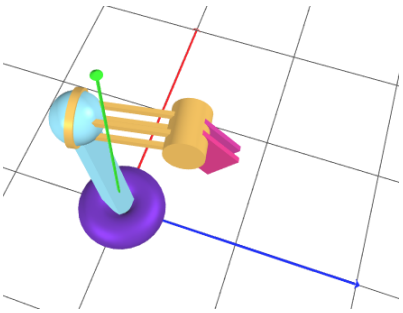
Exercice 3-6 Robot hand

L'objectif de cet exercice est de rajouter une main au bras robotique.

Le code fournit comporte une seule main, définie par la fonction `createRobotGrabber`. Il faut donc créer la deuxième main.

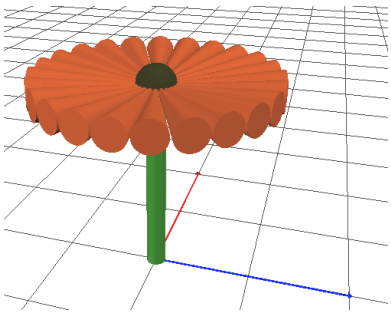
Il y a deux fonctions à modifier : `fillScene()` et `render()`.

Le résultat doit être le suivant :



Exercice 3-7 flower

L'objectif de cet exercice est de créer une fleur.
Au début, elle est constituée d'une seule pétale qui n'est pas encore positionnée.
Mettre 24 pétales sur la fleur avec une boucle `for` en utilisant des angles de rotation.
Le résultat doit être le suivant :

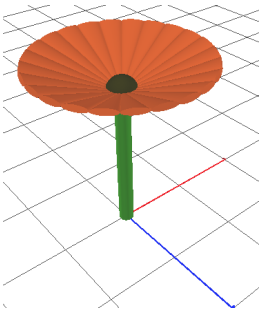


Exercice 3-8 Improved flower

L'objectif de cet exercice est d'améliorer le rendu des pétale de la fleur.
Pour cela, modifier :

1. le facteur des pétales, pour avoir une forme plus aplatie (un quart de la hauteur initiale)
2. l'angle d'inclinaison des pétales (20°)

Le résultat doit être le suivant :



5 Les matrices

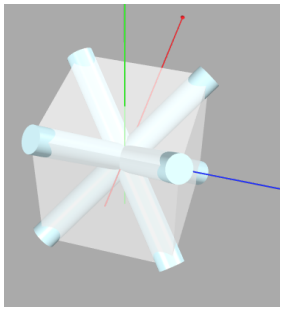
Exercice 4-1 Make an ornament

L'objectif de cet exercice est de créer une géométrie composée de cylindres ayant subit différentes rotations.

Le cylindre de base est donné, ainsi que le code pour effectuer une rotation définit par un axe.

Construire 4 cylindres en utilisant 4 matrices de transformation définies chacune par une rotation, elle-même définie par un axe de rotation et un angle.

Le résultat doit être le suivant :



Exercice 4-2 Cylinder positioning

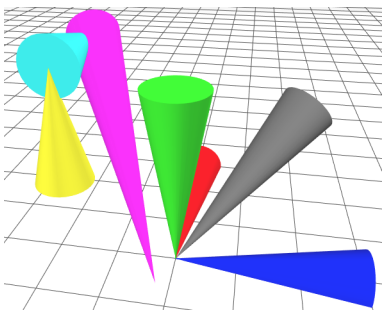
L'objectif de cet exercice est de définir une fonction qui permet de dessiner des cônes définis par :

- `top` le point 3D qui définit le haut du cône
- `bottom` le point 3D qui définit le bas du cône
- `material` le matériel
- `radiusTop` le rayon en haut
- `radiusBottom` le rayon en bas
- `segmentsWidth` la résolution
- `openEnded` un booléen pour dire s'il est fermé ou pas

Le début de la fonction gérant `segmentsWidth` et `openEnded` est déjà codé.

Il suffit donc de calculer les paramètres de la fonction `makeLengthAngleAxisTransform( cyl, cylAxis, center );`

Le résultat doit être le suivant :



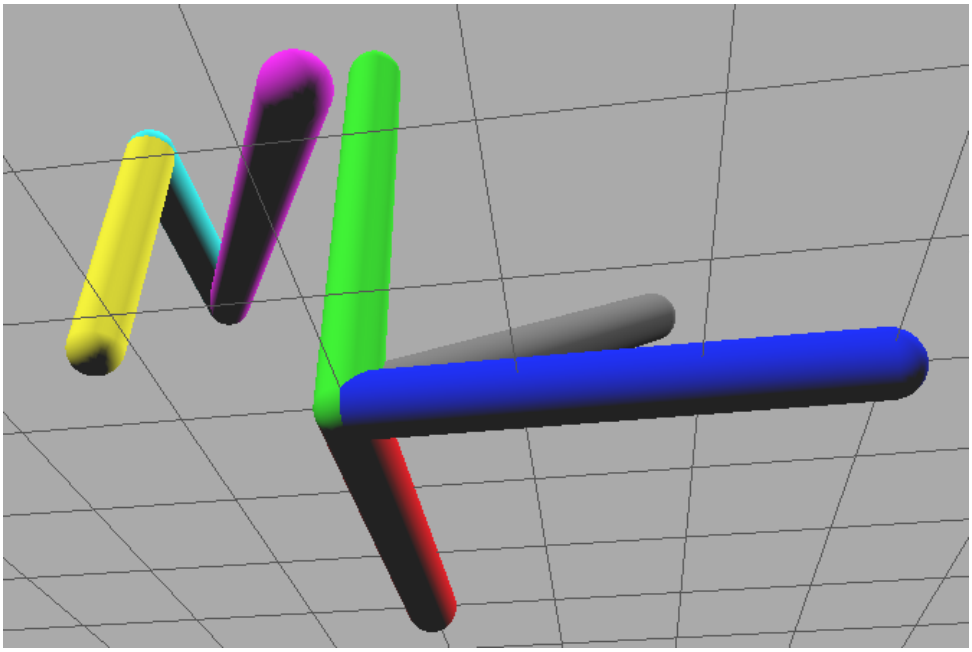
Exercice 4-3 Capsule

L'objectif de cet exercice est de créer des *capsules*, assemblage de cylindre avec deux sphères aux extrémités.

Modifier la fin de la fonction `createCapsule` pour créer ces capsules.

Les cylindres existent déjà et ont été correctement positionnés. Il ne reste qu'à positionner les sphères.

Le résultat doit être le suivant :

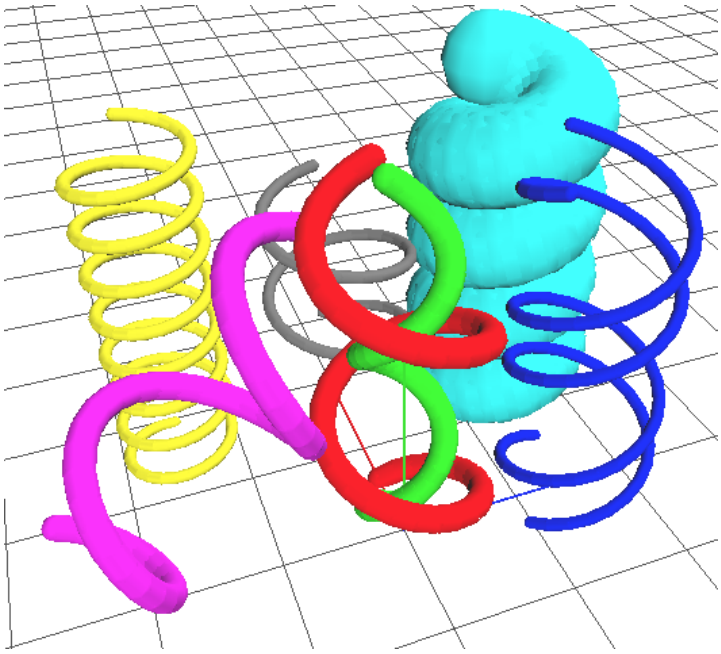


Exercice 4-4 Helices

L'objectif de cet exercice est de réaliser des hélices continues en utilisant les capsules.

Pour que les capsules se touchent correctement, il faut générer qu'une sphère, sauf dans le cas des extrémités.

Le résultat doit être le suivant :

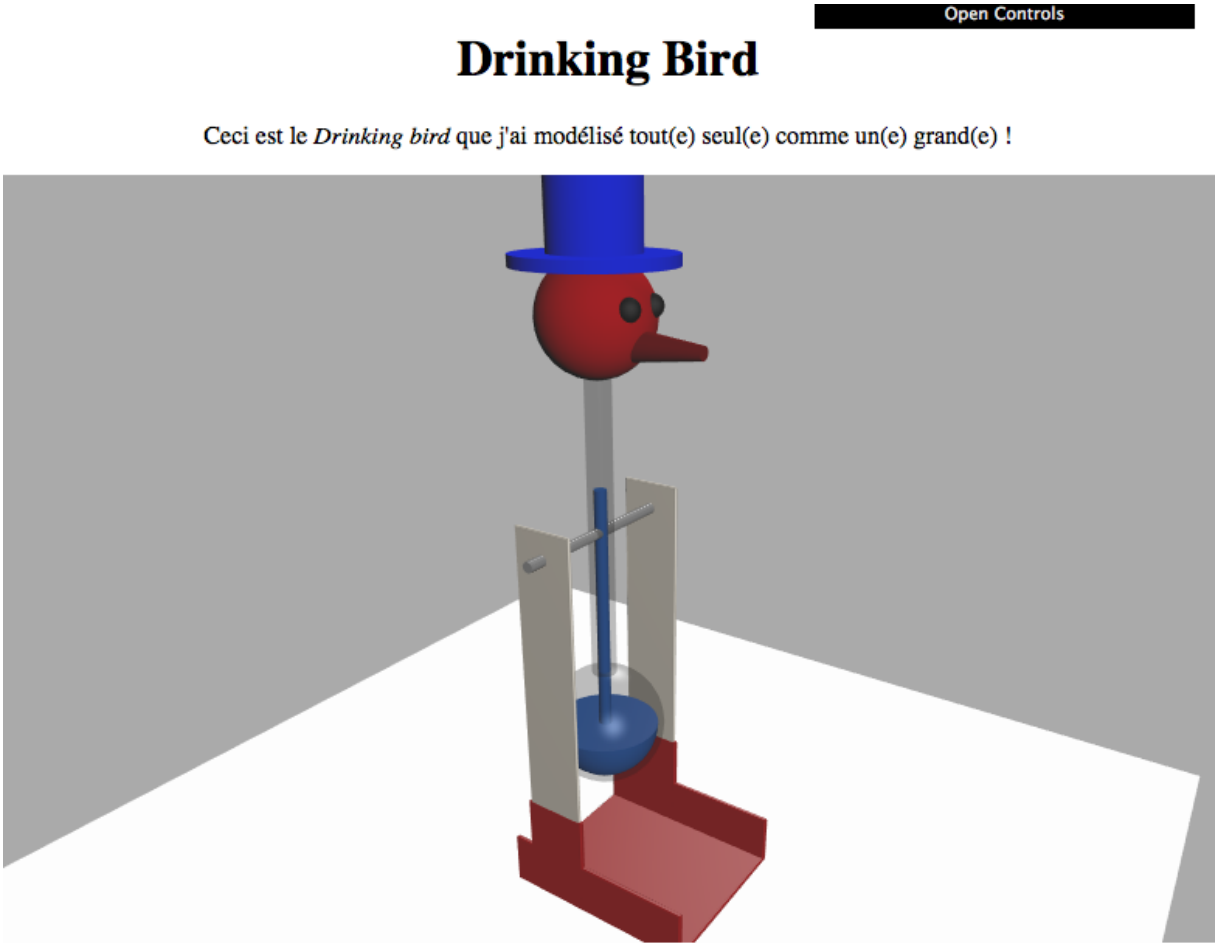


Exercice 4-5 Drinking bird face

L'objectif de cet exercice est de continuer la page web avec le *Drinking bird*, en ajoutant les yeux, le nez et la barre pour attacher le corps au support.

Rajouter une fonction pour créer ces trois parties. N'oubliez pas d'utiliser des `object3D` pour garantir la bonne matrice transformation.

Le résultat doit être le suivant :

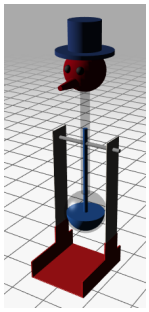


6 L'ÉCLAIRAGE

Exercice 5-1 set-a-directional-light.js

Dans cet exercice, il n'y a plus de lumière par défaut. L'objectif de cet exercice est de rajouter une lumière directionnelle dans `fillScene`. La direction de la lumière doit passer par l'origine et par le point $(-200, 200, -400)$. La lumière doit être blanche et l'intensité de 1.5.

Le résultat doit être le suivant :



Exercice 5-2 head-light.js

Dans cet exercice, il n'y a que de la lumière ambiante par défaut. L'objectif de cet exercice est d'enlever la lumière ambiante et de créer une source de lumière ponctuelle pour imiter une source de lumière type "lampe frontale".

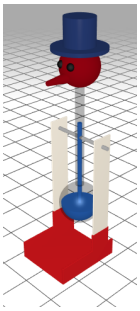
Pour cela, rechercher le mots anglais équivalent (*headlight*). Il y a deux choses à modifier :

1. Ajouter la bonne lumière dans `fillScene` en utilisant la variable `headlight`. La lumière doit être de couleur blanche et d'intensité 1.
2. Faire en sorte que la lumière suive le point de vue de la caméra à chaque pas de temps. Pour cela, modifier la fonction `render()`.

```
function render() {
var delta = clock.getDelta();
cameraControls.update(delta); renderer.render(scene, camera);
}
```

Cette méthode est appelée à chaque fois qu'une image est rendue. Vous pouvez ignorer `delta` et `cameraControls`. Il faut que la position de la source de lumière corresponde à celle de la caméra. `headlight` et `camera` sont des `Object3D` et partagent donc les mêmes attributs de **transformation** (vu en cours).

Le résultat doit être le suivant :

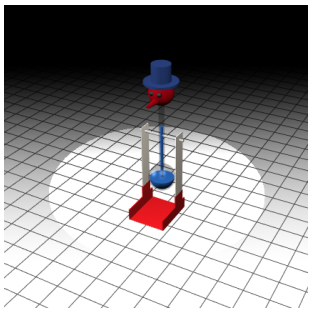


Exercice 5-3 spot-light.js

L'objectif de cet exercice est de créer une source de lumière de type *spot*. Remplacer la source de lumière directionnelle par une lumière spot ayant les caractéristiques suivantes :

couleur	blanche
intensité	1.5
position	-400,1200,300
angle	20°
exposant	1
position de la cible	0,200,0

Le résultat doit être le suivant :



Exercice 5-4 shadows.js

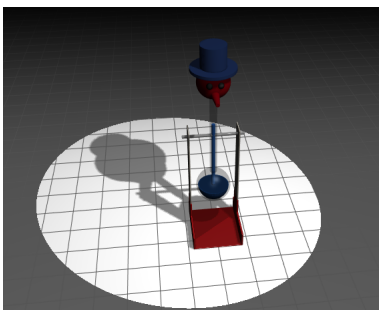
Il est possible dans Three.js de projeter ou non l'ombre de chaque objet, comme il est possible de recevoir ou non l'ombre d'un autre objet :

```
renderer.shadowMapEnabled = true;

spotlight.castShadow = true;

cube.castShadow = true;
cube.receiveShadow = true;
```

L'objectif de cet exercice est de permettre d'avoir l'ombre du *Drinking bird* projeté sur le sol.
Le résultat doit être le suivant :

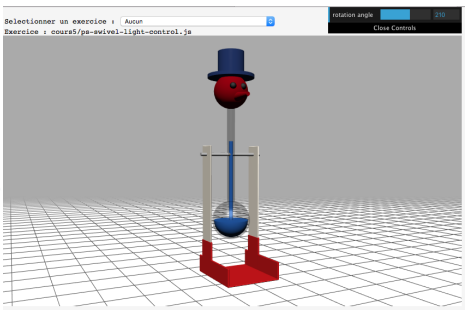


Exercice 5-5 swivel-light-control.js

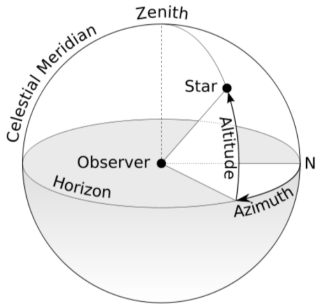
Dans cet exercice, il y a par défaut une lumière directionnelle qui pointe à l'arrière du *Drinking bird*.

L'objectif de cet exercice est de rajouter un slider dans l'interface pour faire varier la direction en **X** et **Z** par le *cosinus* et le *sinus* de l'angle. Le code doit être mis dans la méthode `render()`

Le résultat doit être le suivant :



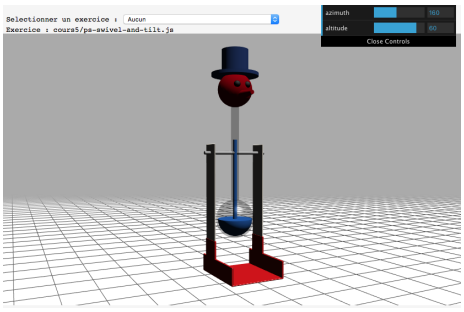
Exercice 5-6 swivel-and-tilt.js



Cet exercice est directement la suite du précédent. L'objectif de cet exercice est de rajouter un deuxième slider pour faire varier l'altitude de la lumière. **X** et **Z** doivent diminuer lorsque **Y** augmente, telle que la direction de la lumière soit *normalisée*. Soit :

$$X^2 + Y^2 + Z^2 = 1 \tag{6.1}$$

Une fois fini, la lumière doit se manipuler avec l'altitude et les angles azimuthes (comme sur la terre). Le résultat doit être le suivant :



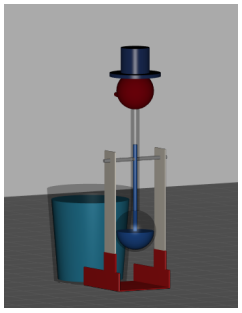
7 Le point de vue

Exercice 6-1 lookat.js

Cet exercice utilise encore l'exemple du *Drinking bird* même s'il n'est pas visible par défaut. *Où est-il passé ?*

L'objectif de cet exercice est de définir la cible (*target*) telle qu'on puisse voir le *Drinking bird*. Il ne faut ni changer la position de la caméra ni la taille de l'image.

Le résultat doit être le suivant :



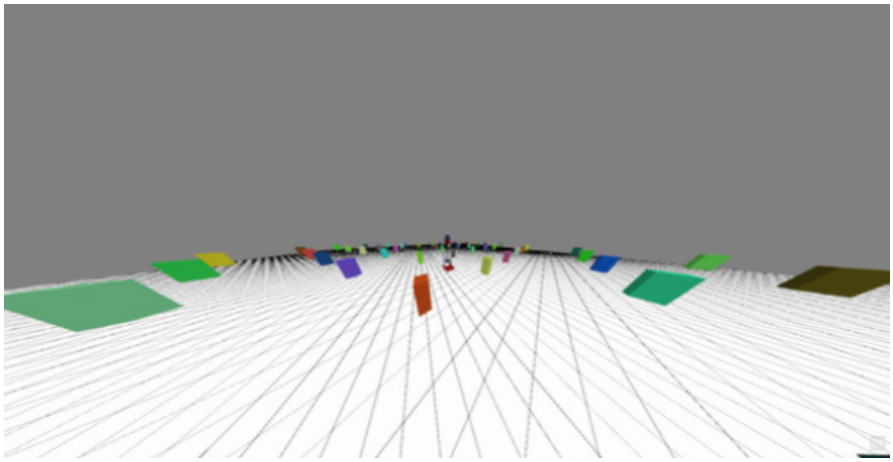
Exercice 6-2 fov-slider.js

L'objectif de cet exercice est de rajouter un slider pour contrôler les paramètres de vue de la projection perspective. Le nom du slider doit être "*field of view*". L'intervalle du slider doit être de 1 à 179 et commencer à 40°.

Pour cela, utiliser la documentation de la bibliothèque `dat.gui`, ou tout simplement, s'inspirer des exercices précédents.

N'oublier pas de prendre en compte la nouvelle valeur du `fov` de la caméra dans `render`.

Le résultat doit être le suivant :

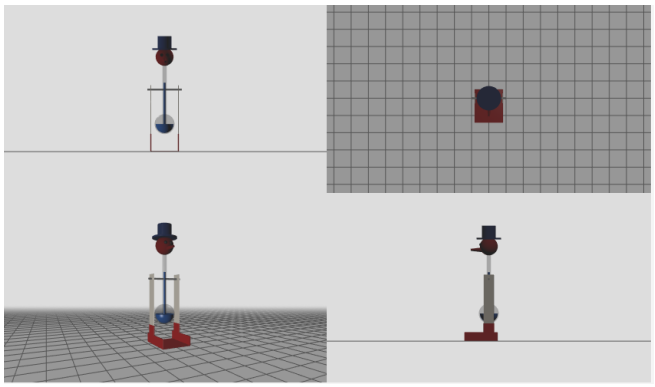


Exercice 6-3 four-viewports.js

Par défaut, la scène de cet exercice comprend deux points de vue (perspective et vue du dessus orthographique).

L'objectif de cet exercice est de rajouter deux points de vue supplémentaire : une vue de face (suivant **+X**) `frontCam` et une vue de coté (suivant **-Z**) `sideCam`. Les deux vues doivent suivre le *Drinking bird*.

Le résultat doit être le suivant :



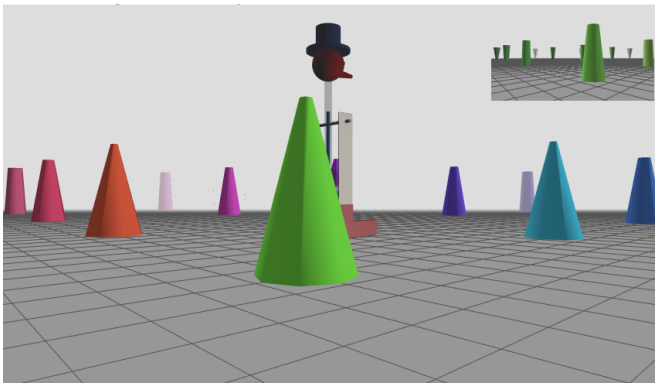
Exercice 6-4 rear-view-camera.js

Dans la scène par défaut, un point de vue en haut à droite la caméra pointe sur le centre de l'objet. La position de la caméra et la cible ne changent pas.

L'objectif de cet exercice est de concevoir une caméra du type rétroviseur de voiture en visualisant dans le point de vue, l'arrière de ce que la caméra principale visualise.

La position de la caméra de ce point de vue doit être la même que celle de la caméra principale. Utiliser la méthode `lookAt()` et des méthodes issues de la classe `Vector3`. Le code doit être mis dans la méthode `render()`.

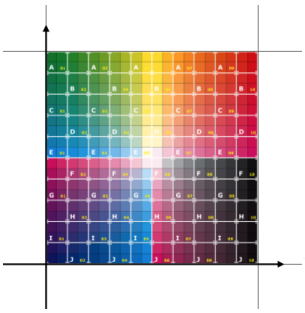
Le résultat doit être le suivant :



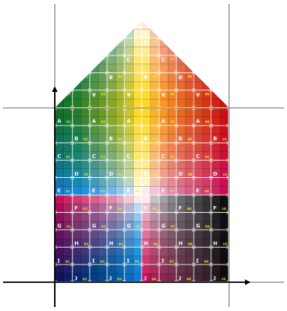
8 Les textures

Exercice 7-1 textured-square.js

1. L'objectif de cet exercice est d'appliquer une texture carrée sur un carré. Pour cela, une texture plaquée sur un triangle est donnée
   Le résultat doit être le suivant :

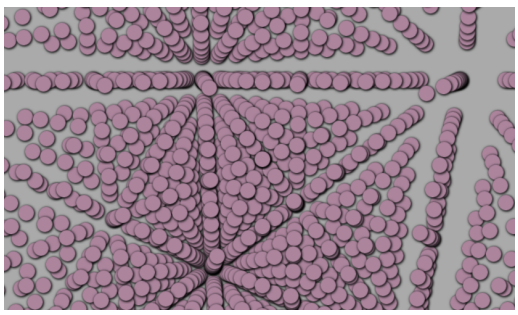


2. Sur le même exercice, construire un nouveau triangle et appliquer une texture pour avoir la forme d'une maison comme ci-dessous :



Exercice 7-2 particle-grid.js

1. L'objectif de cet exercice est de modifier la configuration du système de particule vu en cours. A la place de 8000 particules placées aléatoirement, il faut créer 9261 particules structurées suivant une grille. Refaire la partie du code créant un point à chaque 100 unité du cube $2000 \times 2000 \times 2000$ centré à l'origine. Par exemple, il y aura un point à (-1000,-1000,-1000), un point à (-900, -1000,-1000), etc. jusqu'au point (1000, 1000, 1000). Il y aura 21 points par ligne, soit $21 \times 21 \times 21 = 9261$.
   Le résultat doit être le suivant (avec un fond gris) :



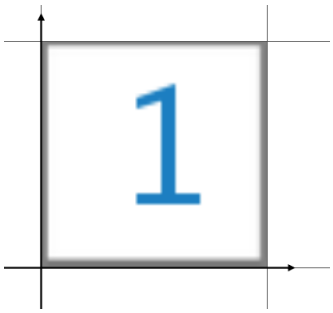
2. L'objectif maintenant est de faire bouger les particules aléatoirement. Chaque cellule doit avoir un déplacement différent avec des valeurs pour XYZ comprises entre -1 et 1.
   Aide Pour parcourir tous les sommets de la géométrie `geometry`, vous pouvez utiliser :

```
for (var i = geometry.vertices.length - 1; i >= 0; i--) {
  // modification de geometry.vertices[i].x
  // modification de geometry.vertices[i].y
  // modification de geometry.vertices[i].z
}
```

   Bien réfléchir où mettre ce code sachant qu'il doit être appelé à chaque frame.

Exercice 7-3 pick-a-letter.js

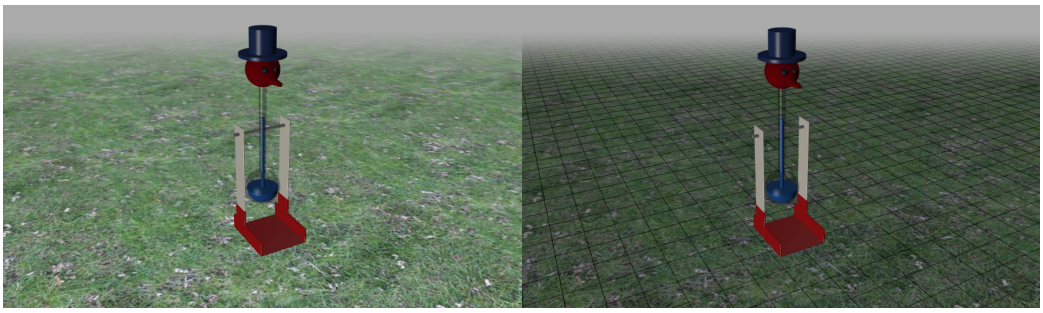
1. L'objectif de cet exercice est d'afficher uniquement la lettre 1 de la grille.
   Le résultat doit être le suivant :



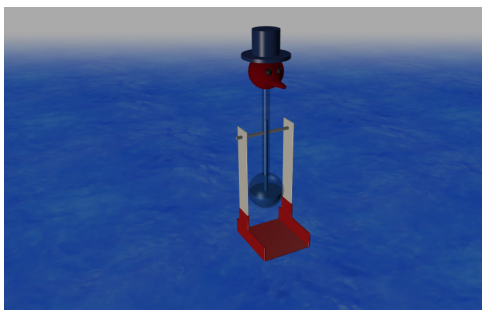
2. Changer le code de façon à ce que la partie de la grille affichée change aléatoirement à chaque refresh. Vous pouvez convertir un nombre réel en entier avec `Math.round()`.

Exercice 7-4 grassy-plain.js

1. L'objectif de cet exercice est de rajouter une texture de terrain sur le sol du *drinking bird*. Etant donné la largeur du plan, répéter la texture 10 fois dans chaque direction.
   Le résultat doit être le suivant (sans et avec grille) :



2. Remplacer maintenant la texture d'herbe par la texture d'eau (water.jpg). Simuler l'écoulement de l'eau en remplaçant l'attribut `offset` dans la boucle de rendu (`render()`).
   Le résultat doit être le suivant :



Exercice 7-5 specular-mapping.js

1. L'objectif de cet exercice est d'appliquer une carte de spécularité (*specular map*) sur la théière. Pour savoir comment se définit une *specular map* sur un matériel de type `MeshPhongMaterial`, regarder la documentation en ligne sur `https://threejs.org/docs/`.

   Le résultat doit être le suivant :

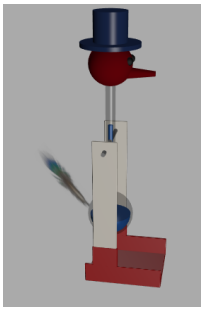
2. Pour la deuxième étape, appliquer une normal map à partir de `http://opengameart.org/textures/5654`.

   Voici un résultat que vous pouvez obtenir :

Exercice 7-6 db-tail.js

L'objectif de cet exercice est de rajouter une queue au *drinking bird*. Le polygone pour recevoir la texture a déjà été créé.

Le résultat doit être le suivant :

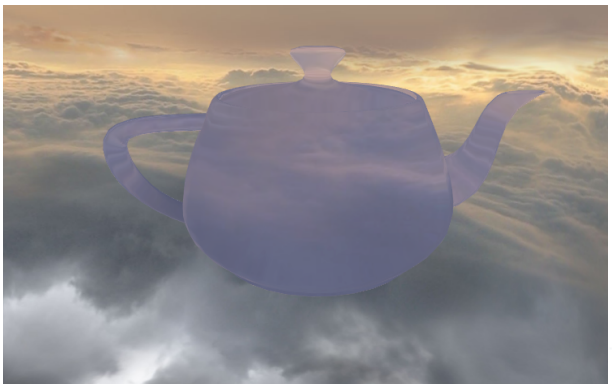


Exercice 7-7 reflection-mapping.js

1. L'objectif de cet exercice est de faire en sorte que la *skybox* se reflète sur la théière. Pour cela, consulter la documentation du matériel de la théière.

   Le résultat doit être le suivant :

2. L'objectif de cet exercice est de faire en sorte que la *skybox* soit utilisé pour la réfraction. Pour cela, utiliser l'exemple vu en cours.
   Le résultat doit être le suivant :



3. L'idée maintenant est de reprendre la *reflective map* de la question 1 pour faire une *glossy map*. Pour cela, faire une copie des 6 images de la *skybox*, effectuer un floutage gaussien dessus et utiliser la nouvelle texture cubique pour avoir l'effet lustré comme ci-dessous :

sans flou (question 1)

avec flou

   Pour calculer l'image floutée, il faut utiliser le script python vu en traitement d'images (soit en utilisant une fonction de convolution en traitant les 4 composantes (R,G,B,A) soit avec la méthode filters.gaussian de la bibliothèque skimage).

Très important : les nombres étant souvent des réels, ils doivent contenir un ".".
Par exemple : `float variable=1.;` et non `float variable=1;`

Le shader est codé dans le fichier `index.html`.

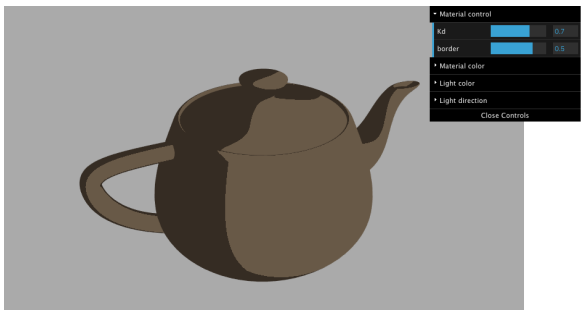
Exercice 8-1 Two-tone-shading

1. L'objectif de cet exercice est de coder un *Fragment Shader* qui calcule la lumière diffuse de chaque pixel. Il y a déjà une variable *uniform* appelée `uBorder` qui vient de l'interface graphique. Changer le programme pour avoir une composante diffuse tel que :

```
Si le produit scalaire est plus grand que uBorder
  alors la composante diffuse est 1.0
  sinon la composante diffuse est 0.5
```

   Ne pas oublier d'utiliser des valeurs réelles.

   Le résultat doit être le suivant :



2. L'objectif de cet exercice est d'utiliser 2 valeurs de seuils sur la composante diffuse : 0.6 et -0.2 pour séparer la valeur en 1.0, 0.7 et 0.3.

   Le résultat doit être le suivant :

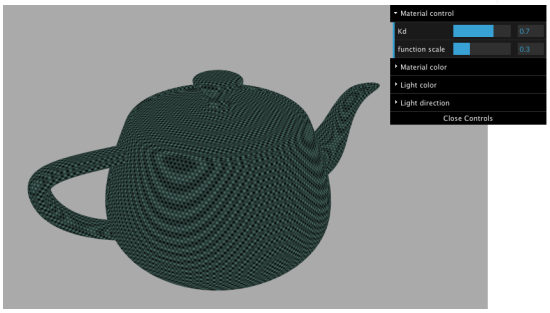
Exercice 8-2 Procedural-texturing

1. L'objectif de cet exercice est de réaliser une texture procédurale 3D. La fonction de cette texture est donnée par l'équation :

$$\frac{1}{2} + \frac{1}{2} \times \sin(uScale \times x) \times \sin(uScale \times y) \times \sin(uScale \times z) \tag{9.1}$$

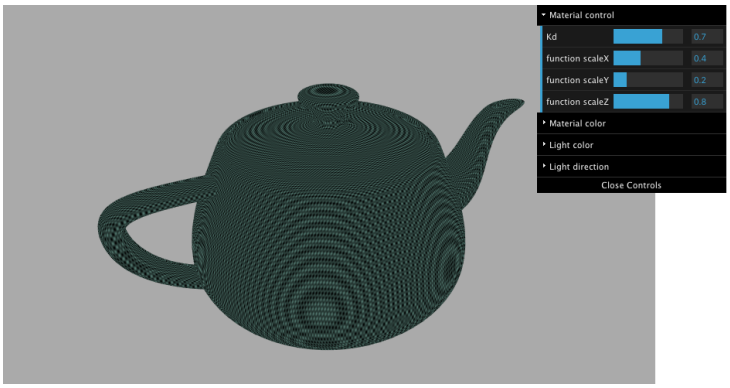
   Les coordonnées de l'objet 3D sont données par `vModelPosition`. Cette fonction effectue des vagues dans les trois directions. La première partie de l'équation sert à avoir un résultat entre 0 et 1.

   Le résultat doit être le suivant :



2. L'objectif de cet exercice est de remplacer le slider `Scale` par 3 sliders `XScale`, `YScale` et `ZScale` pour avoir un comportement de l'équation anisotropique dans les directions **X**, **Y** et **Z**.

   Le résultat doit être le suivant :



Exercice 8-3 Debugging Shaders

Le calcul de la composante diffuse est donné par défaut par :

```
gl_FragColor = vec4( uKd * uMaterialColor * uDirLightColor * diffuse, 1.0 );
```

1. L'objectif de cet exercice est de changer le shader d'une texture procédurale donnée. L'idée est d'utiliser le calcul du shader pour debugger. Il est en effet impossible de débugger un shader avec les outils classiques comme les points d'arrêt.

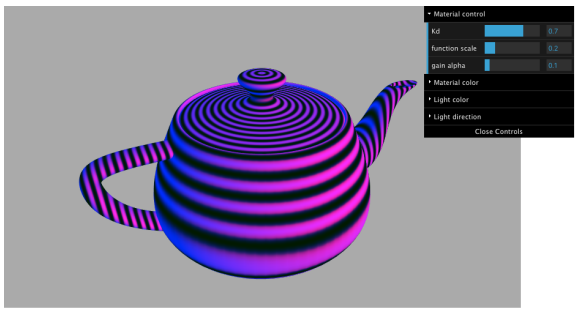
   Changer la sortie du shader pour afficher :

   — la composante diffuse sur la canal rouge

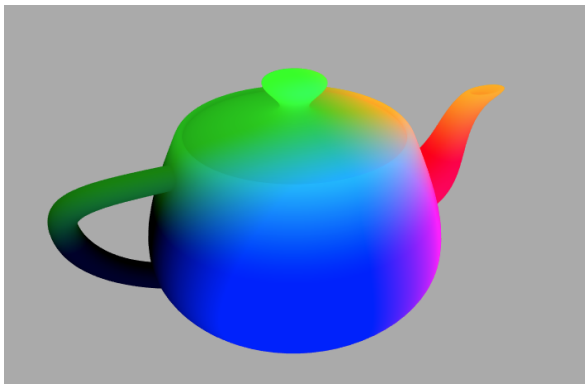
   — `uGainAlpha` sur la canal vert

   — l'atténuation sur la canal bleu

   Le résultat doit être le suivant :



2. L'objectif de cet exercice est d'afficher maintenant l'intersection entre le *color cube* vu dans le cours sur les couleurs et la théière.
   Le résultat doit être le suivant :



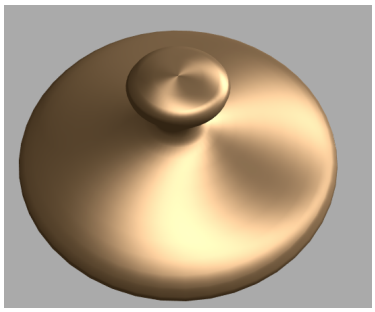
Exercice 8-4 Anisotropic Material

Certains matériaux ont une BRDF anisotrope comme le velours ou l'aluminium. Lorsqu'on tourne ces matériaux, la lumière réfléchit de façon différente. Ces matériaux sont donc appelés anisotrope (contrairement à isotrope pour la plupart des matériaux).

Une façon simple de faire un matériau anisotrope est de donner 2 normales à la place de 1. Cet exercice part d'une forme de l'équation de Blinn-Phong équilibrée.

Les deux normales doivent être calculées dans une boucle `for` déjà présente.

Le résultat doit être le suivant :



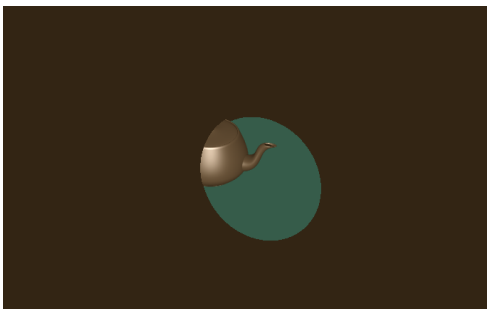
Exercice 8-5 Moving Flashlight

L'objectif de cet exercice est de modéliser le mouvement d'une lampe torche et son influence sur la scène. Le code par défaut contient :

```
if ( length( vViewPosition.xy ) > uFlashRadius ) {
  return;
}
```

Il permet de modifier le rayon du spot mais pas la cible. Pour déplacer le rayon, utiliser la variable `uFlashOffset` qui est un vecteur 2D et qui représente le centre de la zone à éclairer.

Le résultat doit être le suivant :



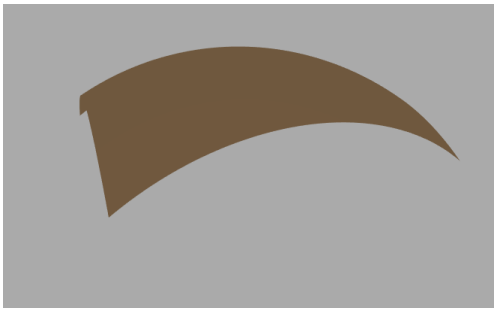
Exercice 8-6 Model Deformation

L'objet par défaut est un plan en XY décomposé en petit carrés. Cette grille va nous permettre de déformer la surface.

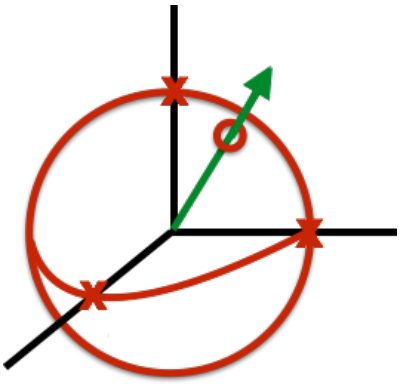
1. L'objectif de cet exercice est de modifier la géométrie des points du plan par un shader. La valeur à changer est la position des vertex. Pour cela, il faudra passer par une variable de position temporaire.
   La formule à utiliser est la suivante :

$$z = \sqrt{(uSphereRadius2 - x^2 - y^2)} - \sqrt{uSphereRadius2} \tag{9.2}$$

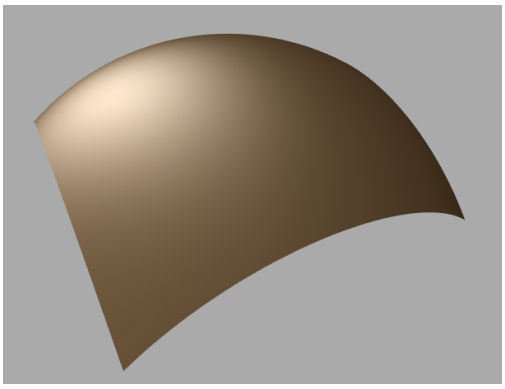
   Le résultat doit être le suivant :



2. L'objectif de cet exercice est de changer les normales du modèle précédent pour avoir un calcul d'illumination correct. Pour une sphère centrée à l'origine, la normale et la position du point pointe dans la même direction. Ce n'est pas utile de normaliser le vecteur car le *fragment shader* le fera plus tard.
   Le résultat doit être le suivant :



Explication du calcul

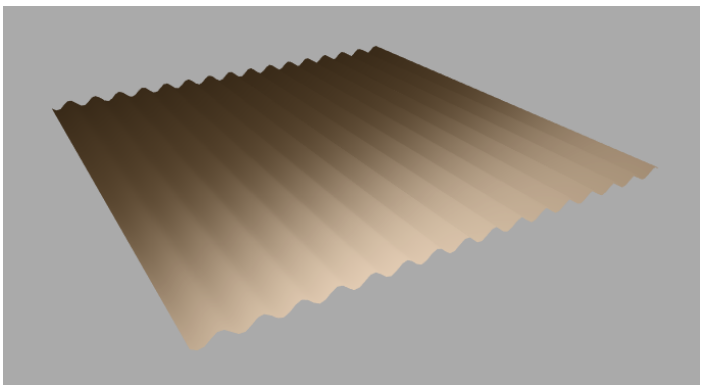


Résultat

3. L'objectif de cet exercice est de déformer le plan pour créer une tôle ondulée. Pour cela, utiliser la formule suivante :

$$z = 0.01 * sin(uSphereRadius2 * y) \tag{9.3}$$

   Le résultat doit être le suivant :

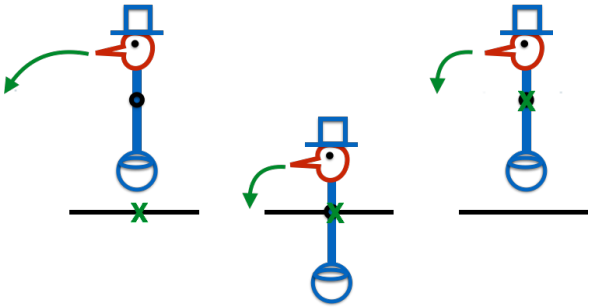


Exercice 9-1 Set the pivot

L'objectif de cet exercice est de changer les paramètres du pivot du *Drinking bird*. Il faut positionner le pivot pour que le corps et la tête soit autour du bras. Si ce n'est pas le cas, ils vont tourner de manière étrange. Etant donné qu'ils sont déjà bien placés, comment les translater ?

Aide : Si le *drinking bird* était construit différemment et que le centre de la rotation serait à l'origine, tout tournerait correctement. Pour cela, un `Object3D` regroupant la tête et le corps a été créé et il s'appelle `bodyhead`. Le script par défaut permet de tourner `bodyhead` autour de Z.

Il faut utiliser la variable `pivotHeight` et le réponse est en 3 lignes de code.



Le résultat doit être le suivant :