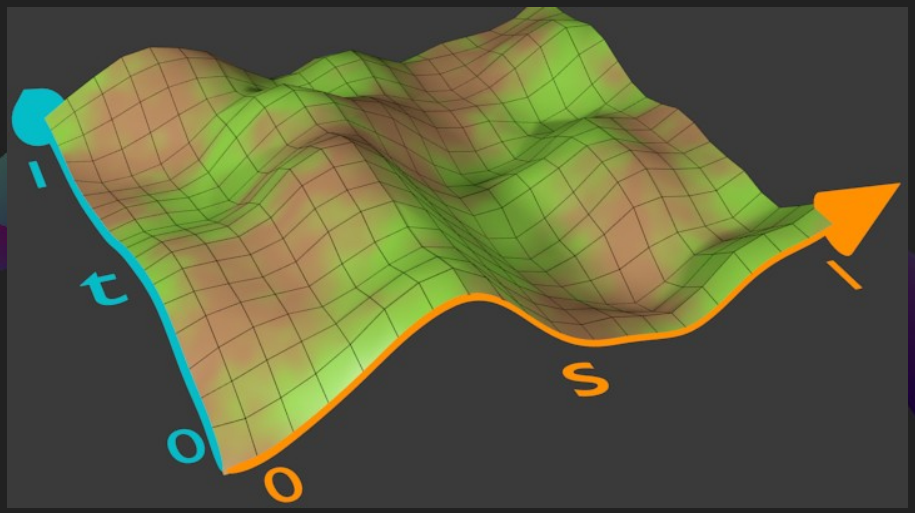
# Partie 5 : Techniques avancées

- Textures
  - Principes
  - Repère de Frenet
- Dessin dans une texture (FBO)
  - Traitement d'images
  - Ombres portées
- Autres effets

# 5.1 - Textures

- Une texture est un « mécanisme » qui fournit la couleur d'un point à la surface d'un objet
  - Fonction mathématique : texture procédurale
  - Image 2D provenant d'un fichier
  - Image issue d'une synthèse précédente (FBO)
- Il faut disposer d'un système de coordonnées à la surface de l'objet
  - Ex : (X, Y), (azimut, hauteur), (longitude, latitude)
  - Plus souvent : ad-hoc, en fonction de l'objet
  - Calculées ou prédéfinies

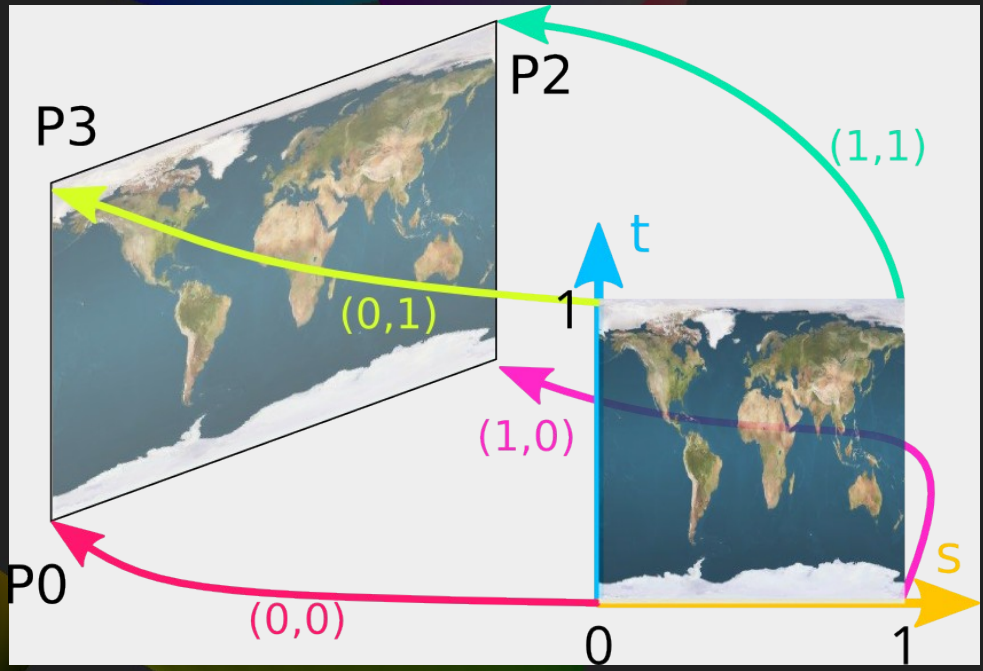
# Coordonnées de texture




- C'est un repère curviligne : plaqué sur la surface
- Ses axes sont généralement nommés (u,v) mais (s,t) en OpenGL (cause composantes xyzw contre uvw)
- Les coordonnées varient entre 0 et 1 et correspondent à l'image qu'on plaque dessus (c'est pour ça que c'est ad-hoc)

# Placage d'une image

- Chaque sommet du maillage porte des coordonnées de texture (en plus des coordonnées xyz, normales...)
- Ces coordonnées de texture permettent d'attribuer les ~~pixels~~ texels de l'image au maillage
  - Ex : P3 = (0,1) donc il reçoit la teinte du coin haut-gauche de l'image
  - Les coordonnées des fragments intermédiaires sont interpolées



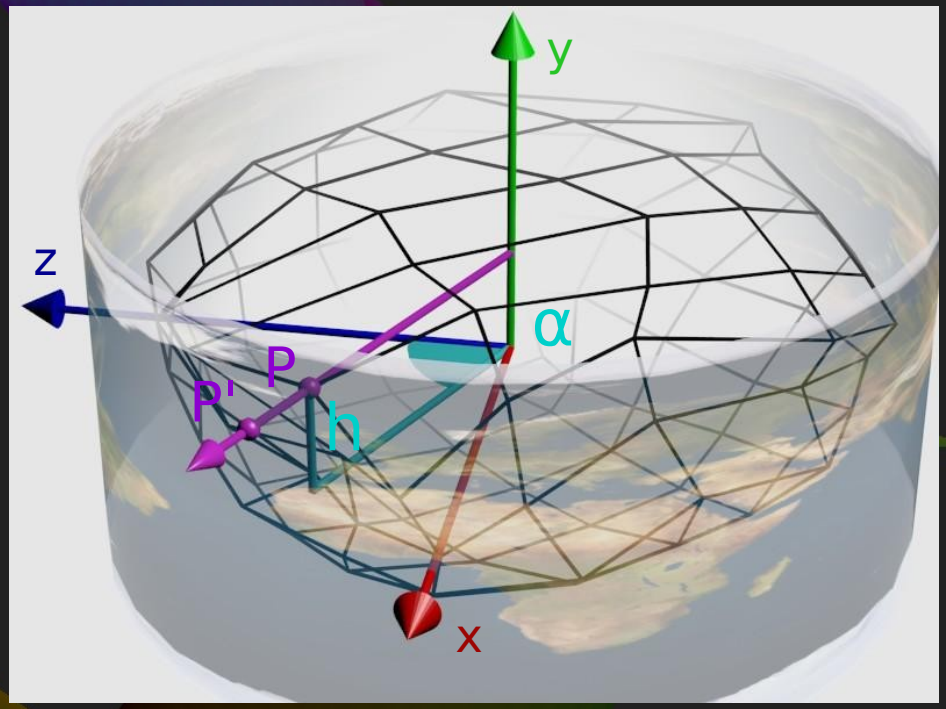
# Attribution des coord. de texture

- Les coordonnées de texture de chaque sommet peuvent être définies (*mapping* en anglais) par :
  - Un scanner 3D
  - Manuellement, avec un modeleur comme Blender
  - Des équations géométriques basées sur des formes simples projetées sur l'objet (on parle alors de *wrapping* = emballage) :
    - Plan ou cube
    - Cylindre
    - Sphère

# Définition cylindrique

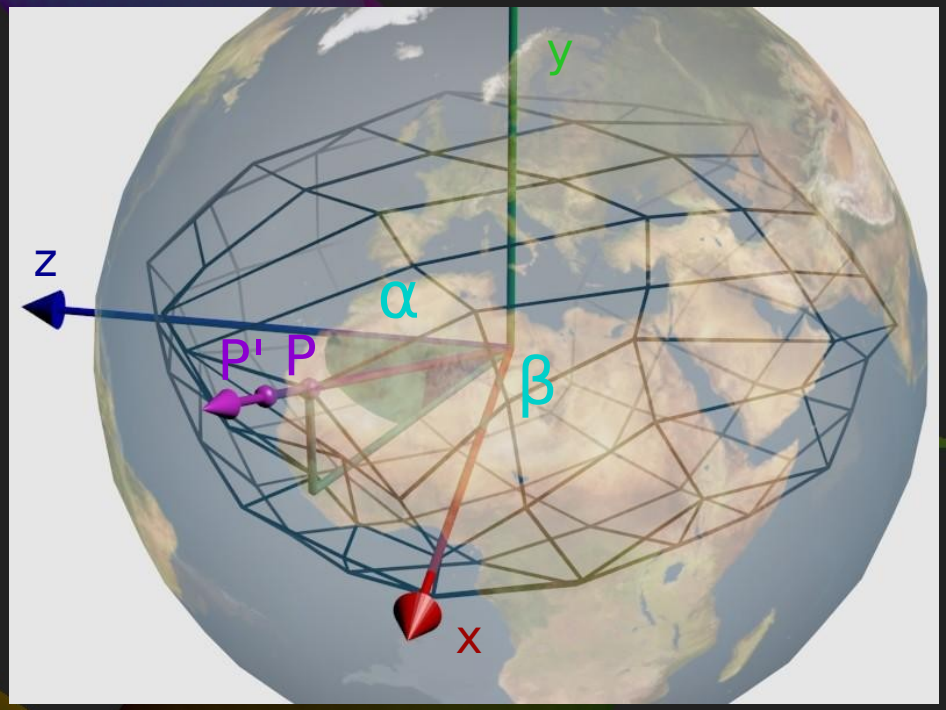
- La projection P' d'un sommet P sur un cylindre englobant au plus serré définit les coordonnées de texture de P

1) Déterminer le centre du cylindre ainsi que son extension verticale
2) Calculer x,y,z relatifs au cylindre
3) s = arctan2(z,x)/2π
4) t = y c'est à dire $(y-y_{min})/h$



# Définition sphérique

- La projection P’ d’un sommet P sur une sphère englobant au plus serré définit les coordonnées de texture de P
  1) Déterminer le centre et le rayon de la sphère
  2) Calculer x,y,z relatifs
  3) s = arctan2(z,x)/2π
  4) t = arcsin(y/n)/π+0.5 avec n = norme(x,y,z)



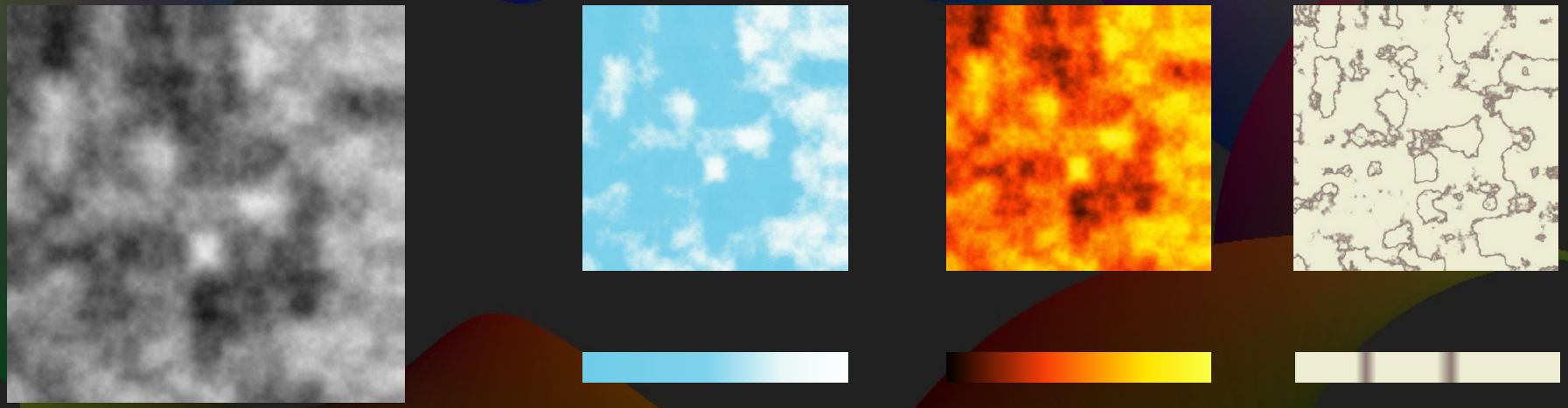
# Texture procédurale

- C'est une fonction qui retourne une couleur dépendant des coordonnées de texture
  - Visualisation de fonctions mathématiques
  - Fonctions utiles : arithmétique, fonctions périodiques, distance
- Elle est calculée dans le fragment shader
- Ex :

```
float d = distance(frgTexCoords, vec2(0.5));
gl_FragColor = vec4(d,d,d, 1.0);
```

# Textures aléatoires (bruit)

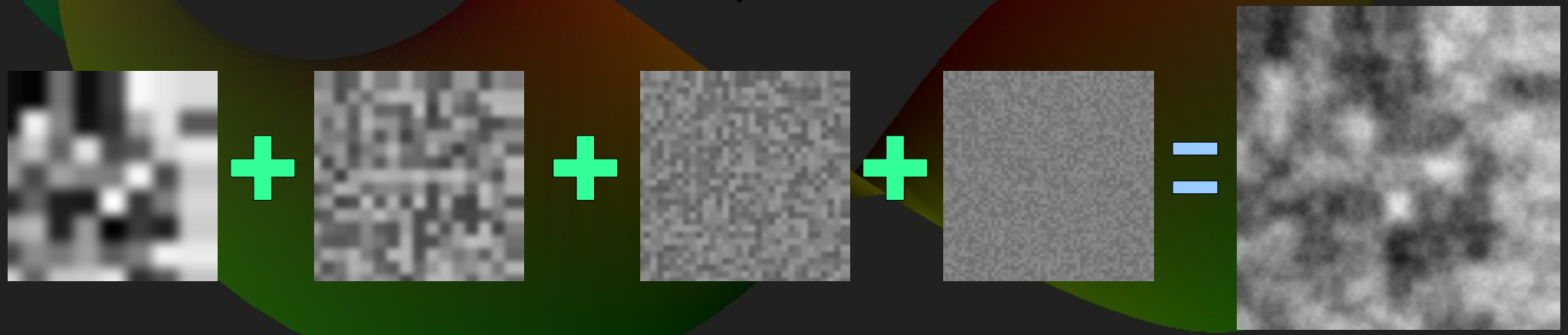
- Ce sont des textures, générées préalablement à l'aide d'un algorithme, pour certains matériaux naturels : bois, nuages, feu, etc.



- La même texture en niveau de gris est colorée par différents gradients pour obtenir ces effets

# Création d'une texture de bruit

- Ken Perlin a proposé en 1985 un algo basé sur l'addition de textures de bruits simples interpolés, et ayant des fréquences spatiales imbriquées en puissance de 2 (comme les octaves en musique)
  - Fréquence spatiale = cycles par unité de distance (pixels)
  - La qualité augmente avec le nombre d'octaves additionnés
  - Nombreuses variantes depuis



# Textures images, principes

- C'est une image 2D qui fournit la couleur : png ou jpg
  - Chargement de l'image par WebGL
- Pour accéder aux ~~pixels~~ texels en fonction des coordonnées de texture dans un shader, on emploie un mécanisme appelé `sampler2D`
  - Une variable `uniform` représente ce `sampler2D`
  - Il est lié à la texture WebGL
  - Une fonction GLSL effectue l'accès à un texel via le sampler2D

# Textures images, chargement

- La classe `Texture2D`, dossier `libs` facilite le chargement d'une image jpg, png...
  - Constructeur : nom complet du fichier, il charge l'image de manière asynchrone
    - Paramètres supplémentaires : configuration filtrage et répétitions
  - Les instances contiennent l'objet WebGL interne représentant la texture
  - Une méthode permet de lier une texture à un shader (transparent suivant)

# Textures images, liaison

- Le fragment shader définit une variable `uniform` de type `sampler2D`
  - Un sampler2D représente une « unité de texture », c'est un dispositif d'interpolation capable d'aller chercher un ~~pixel~~ texel dans une `Texture2D`
  - OpenGL définit au moins 8 unités identifiées par un code : `gl.TEXTURE0`, `gl.TEXTURE1`...
- On doit lier une image à ce sampler
  - Méthode `Texture2D.setTextureUnit(code, loc)`
    - code = identifiant de l'unité
    - loc = emplacement de la variable `uniform sampler2D`

# Accès à la texture par le shader

- Dans le vertex shader :

```
...
in vec2 glTexCoords;   // VBO coords txture
out vec2 frgTexCoords;

void main()
{
    ...
   frgTexCoords = glTexCoords;
}
```

Les coordonnées sont transmises au fragment shader, interpolées d'un sommet à l'autre

# Accès à la texture, suite

- Dans le fragment shader :

```
in vec2 frgTexCoords;
uniform sampler2D txColor;
out vec4 glFragColor;

void main()
{
    glFragColor =
        texture(txColor, frgTexCoords);
}
```

- La fonction GLSL `texture(sampler, coords)` retourne la couleur de l'image gérée par le sampler à ces coordonnées

# Programme JavaScript

- Dans la classe représentant le matériau :
  - Il faut déterminer l'emplacement de la variable `sampler2D`

```
this.m_TextureLoc = gl.getUniformLocation(
    this.m_ShaderId, "txColor");
```

  - Il faut charger l'image et en faire une texture

```
this.m_Texture = new Texture2D("image.jpg");
```

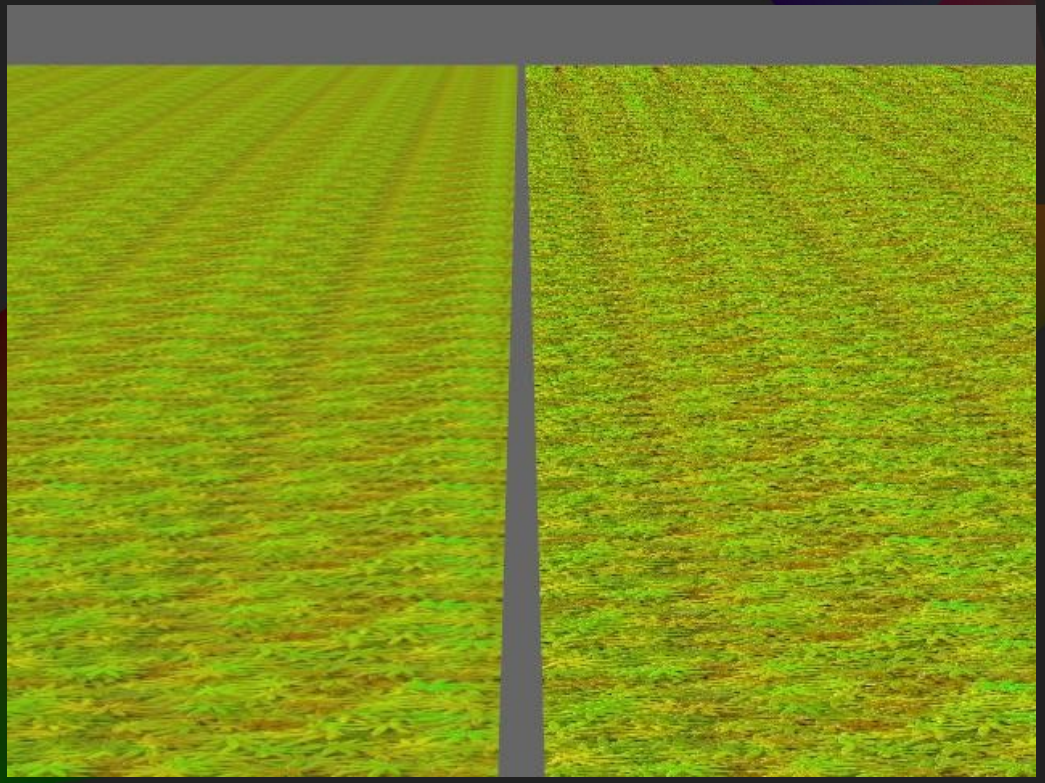
  - Il faut lier le `sampler2D` avec la texture

```
this.m_Texture.setTextureUnit(
        gl.TEXTURE0, this.m_TextureLoc);
```

  - Le shader peut maintenant travailler

# Configuration des textures

- L'accès aux texels peut être configuré :
  - Répétitions (pour des coordonnées quelconques)
  - Interpolation linéaire, par mipmaps ou aucune (à droite)

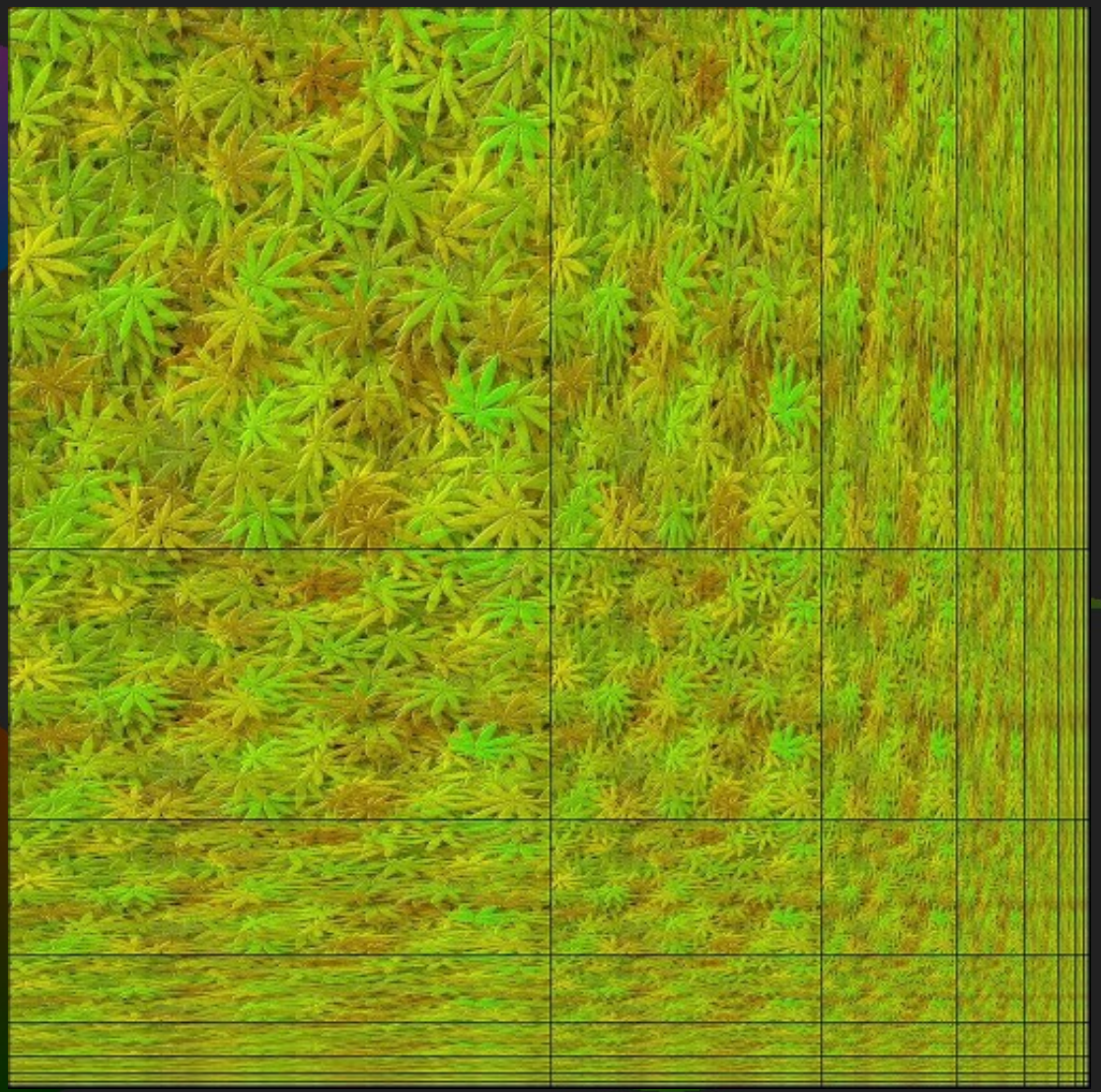


# Filtrage des textures

- Le constructeur de `Texture2D` permet de spécifier comment interpoler :
  - `gl.NEAREST` : aucune interpolation (rapide, défauts)
  - `gl.LINEAR` : interpolation bilinéaire simple
  - `gl.LINEAR_MIPMAP_LINEAR` : interpolation bilinéaire avec mipmaps (lent mais meilleur aspect)
- Un mipmap (textures à résolutions multiples) fournit plusieurs résolutions pour une texture, permettant de l'adapter à celle de l'écran

# MipMap

- *Multum In Parvo*
- L'image est augmentée de ses réductions
  - La place occupée = 4x texture initiale
  - Choix de la sous-image selon la taille affichée

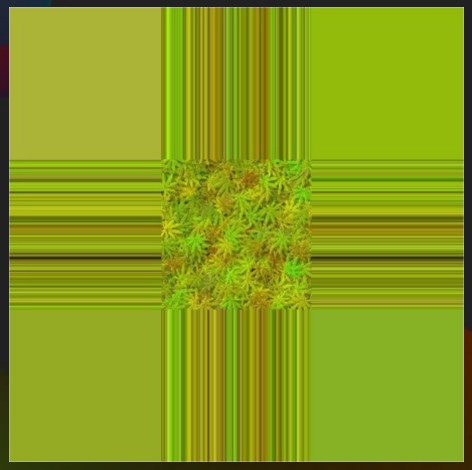


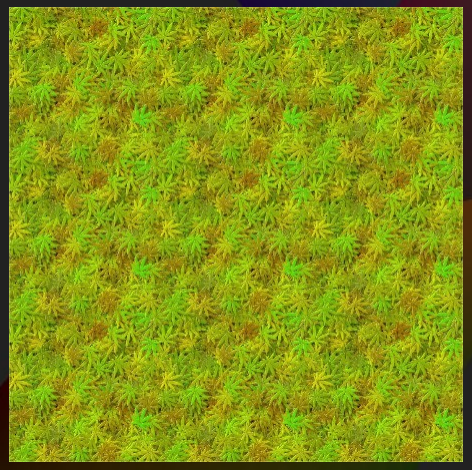
# Textures pavables

- *Tileable* ou *seamless textures* en anglais
- Les textures répétables peuvent paver un plan
  - Cf http://serge.mehl.free.fr/anx/pavages_plans.html
- Chaque bord correspond au bord opposé
  - On peut les créer avec un outil de dessin, voir http://hugin.sourceforge.net/tutorials/tileable-textures/en.shtml

# Modes de répétition des textures

- Si les coordonnées sont hors des bornes [0, 1]
  - `gl.CLAMP_TO_EDGE` : le texel est pris sur le bord le plus près
  - `gl.REPEAT` : partie fractionnaire des coordonnées de texture



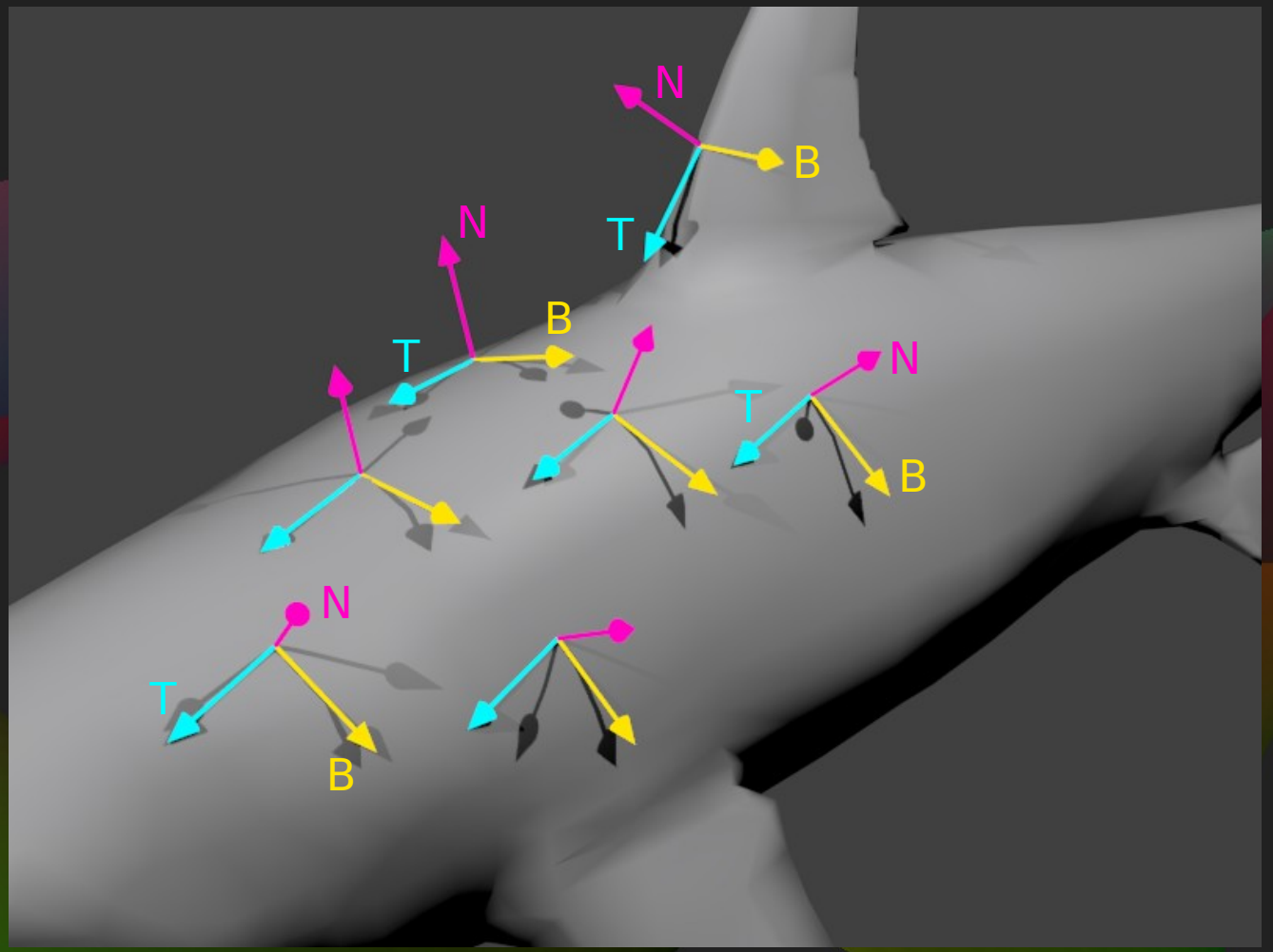


- Autres modes OpenGL pas disponibles sur WebGL (ex : `GL_MIRRORED_REPEAT`)

# 5.2 - Géométrie de surface

- Pour le calcul des composantes diffuses et spéculaires de certains matériaux ou d'autres effets, on a besoin d'un système de repères locaux continus : Repère de Frenet
  - Vous connaissez l'un des vecteurs directeurs : N
  - On y rajoute deux vecteurs tangents, T et B, orthogonaux entre eux
    - T = vecteur tangent
    - B = vecteur bi-tangent (ou parfois nommé bi-normal)
  - Ce repère est défini en chaque point et présente une continuité d'un point à l'autre

# Repère de Frenet

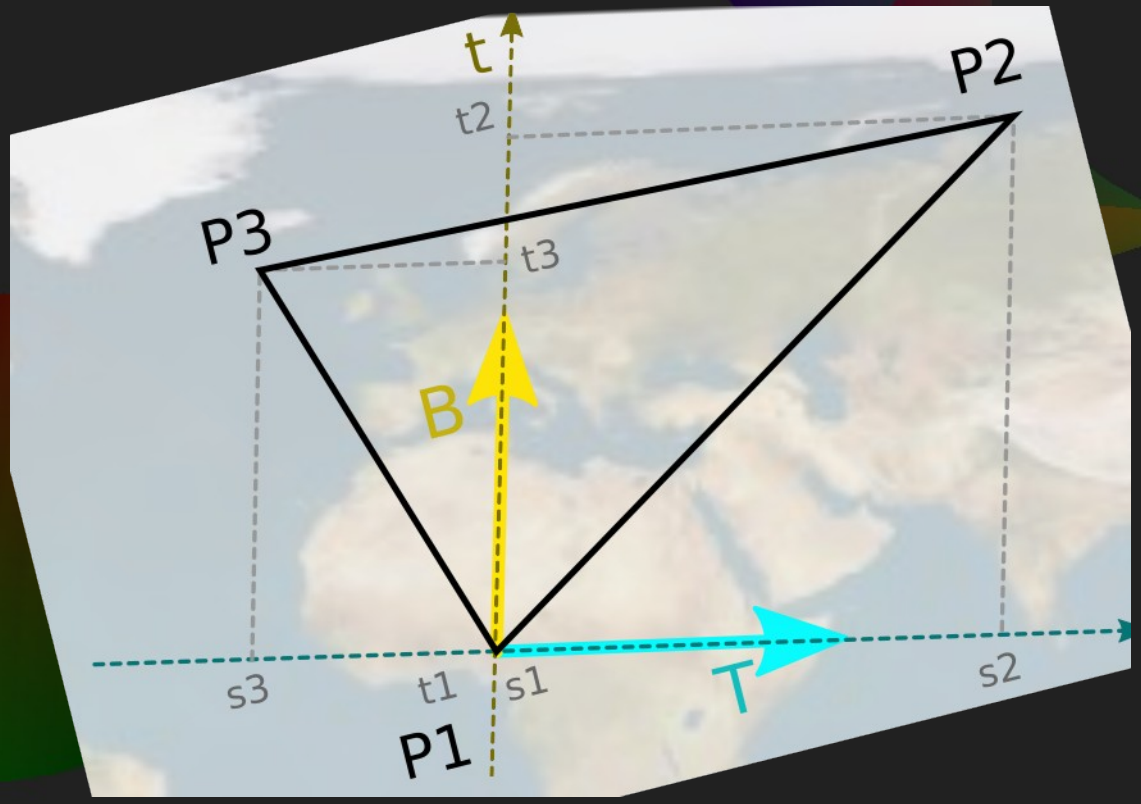



- Le repère (T,B,N) est appelé repère de Frenet

# Construction du repère TBN

- Il est construit en utilisant les coordonnées de texture afin de garantir la continuité à la surface d'un sommet à l'autre
  - On montre que T = (t3-t1).P1P2 – (t2-t1).P1P3

    et B = N ∧ T



# Repère TBN dans un shader

- Le vertex shader reçoit deux vecteurs : N (dans `glNormal`) et T (dans `glTangent`), il les transmet au fragment shader via deux variables varying : `frgNormal` et `frgTangent`

```
frgNormal  = matN * glNormal;
frgTangent = matN * glTangent;
```

- Le fragment shader reconstruit B à la volée

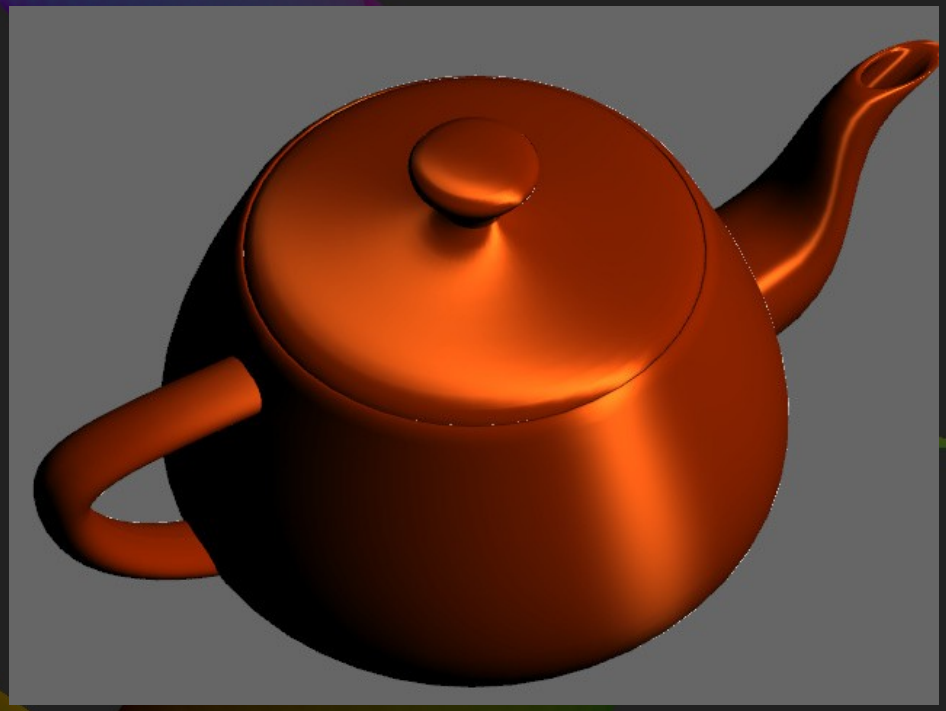
```
vec3 N = normalize(frgNormal);
vec3 T = normalize(frgTangent);
vec3 B = cross(N, T);
```

# 5.3 - Rôles du repère TBN

- Certains matériaux sont anisotropes : leur comportement dépend de la direction d'éclairage et/ou de la vue
  - Ex : métal brossé
  - Ex : textures spéciales
- Le repère TBN permet de s'orienter à la surface
  - Il suffit de construire la matrice [T,B,N] constituée des trois vecteurs en colonne, elle réalise un changement du repère de Frenet vers le local

# Modèle de Ward

- Ce modèle restitue l'aspect « brossé » de certains matériaux
  - Soie, métal poncé
  - Le reflet spéculaire S dépend des directions du regard et de la lumière
  - Il utilise le repère de Frenet (T, B, N)



# Modèle de Ward, calcul

- Voici les calculs, alpha est un vec2 qui caractérise le matériau dans les directions s et t :
  - ```
    float dotNL = clamp(dot(N, L), 0.0, 1.0);
    float dotNV = clamp(dot(N, V), 0.0, 1.0);
    ```
  - ```
    vec3 H = normalize(L + V);
    float dotNH = clamp(dot(N, H), 0.0, 1.0);
    ```
  - ```
    float hts = dot(H, T) / alpha.s;
    float hbs = dot(H, B) / alpha.t;
    float beta =
        -2.0*(hts*hts+hbs*hbs)/(1.0+dotNH);
    ```
  - ```
    float S = exp(beta) /
    (4.0*pi*alpha.s*alpha.t*sqrt(dotNL*dotNV));
    ```

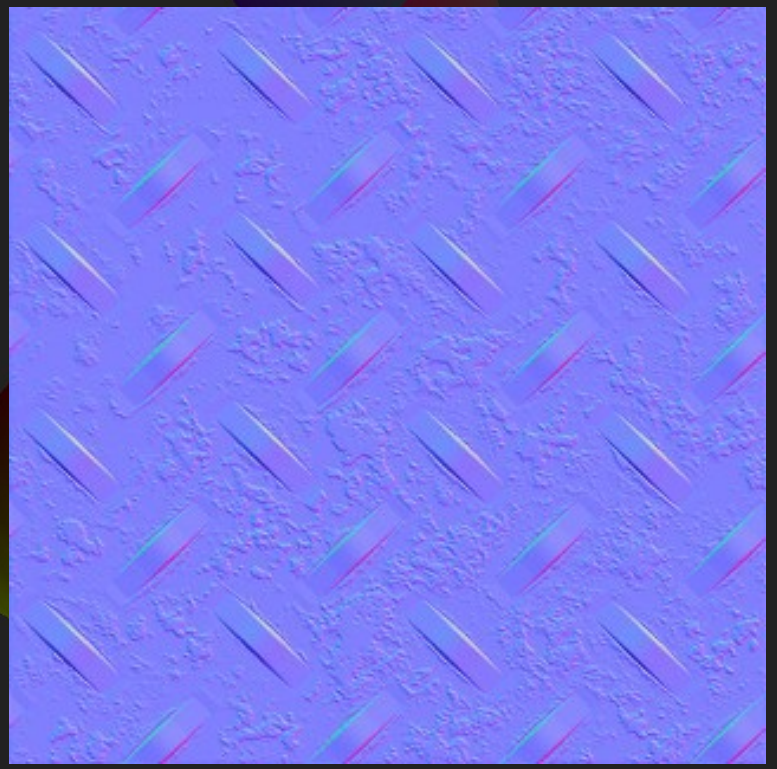
# Relief de surface

- Il est possible de simuler (illusion) un relief macroscopique sans créer de facettes

# Simulation d'un relief

- On utilise une texture spéciale « normal map » : au lieu de couleurs, elle spécifie la direction de la normale dans le repère TBN
  - Rouge => N.x
  - Vert => N.y
  - Bleu => N.z
  - Composantes 0..1 à ramener dans -1..+1

    xyz = rgb*2 - 1



# Calculs

- On extrait le texel de la texture normale, on convertit ses composantes dans la plage -1..+1

```
vec3 Nmod =
    texture(txNormal, frgTexCoord).xyz;
Nmod = Nmod * 2.0 – 1.0;
```

- Ensuite, on exprime cette normale modifiée dans le repère TBN à l'aide d'une matrice :

```
mat3 TBN = mat3(T, B, N);
Nmod = normalize(TBN * Nmod);
```

- Ensuite, on applique le modèle d'éclairement...

# 5.4 – Dessin dans une texture

- Un Framebuffer Object (FBO) permet de dessiner hors écran. L'écran est lui-même une sorte de FBO.
- Un FBO regroupe plusieurs buffers, selon les besoins :
  - Color buffer pour contenir les couleurs des fragments : c'est l'image visible en sortie
  - Depth buffer pour les distances projetées z des fragments
  - Autres buffers pour des traitements spécifiques

# Création d'un FBO

- Comme c'est très complexe, il y a une classe FrameBufferObject dans libs
- Il suffit d'appeler son constructeur avec la taille (largeur, hauteur) souhaitée, ainsi que des codes pour choisir les buffers voulus :
  - Color buffer : type `gl.TEXTURE_2D` utilisable en tant que texture, ou `gl.RENDERBUFFER` pas utilisable extérieurement ou `gl.NONE` pour aucun
  - Depth buffer : idem
- Certaines combinaisons sont interdites

# Autres méthodes

- Méthodes de la classe FrameBufferObject :
  - `enable()` active le FBO, tous les prochains dessins sont faits dans le FBO
  - `disable()` désactive le FBO, les prochains dessins reviennent à l'écran
  - `setTextureUnit(code, loc)` relie un `sampler2D` au color buffer du FBO exactement comme pour une `Texture2D`,
    - rajouter un 3e paramètre : `FBO.getDepthBuffer()` si on veut lier le depth buffer du FBO au sampler

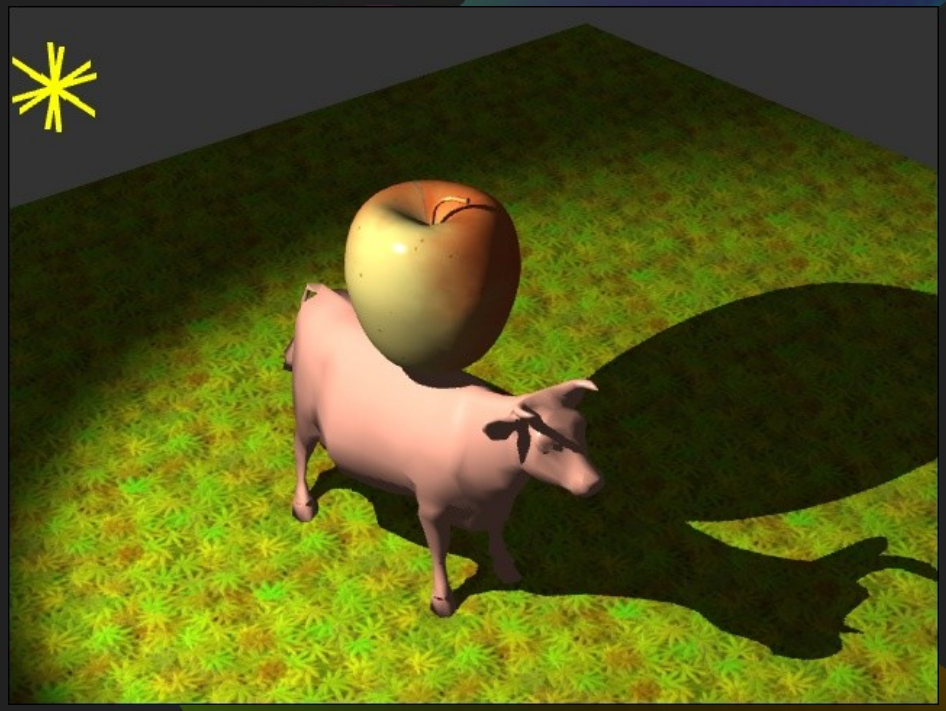
# Emploi d'un FBO

- Un FBO permet de récupérer une scène dessinée en tant que texture
  - Activer le FBO
  - Dessiner la scène
  - Désactiver le FBO
  - Dessiner une forme en utilisant le FBO en tant que texture
- Applications
  - Traitement d'images (post-traitement)
  - Dessin des ombres portées

# Traitement d'images

- Le principe est de dessiner le FBO sur un rectangle plein écran en modifiant (au choix) :
  - Les coordonnées de texture pour déformer l'image, ici loupe
  - Les couleurs de l'image, ici désaturation

# 5.5 – Calculs d'ombres portées

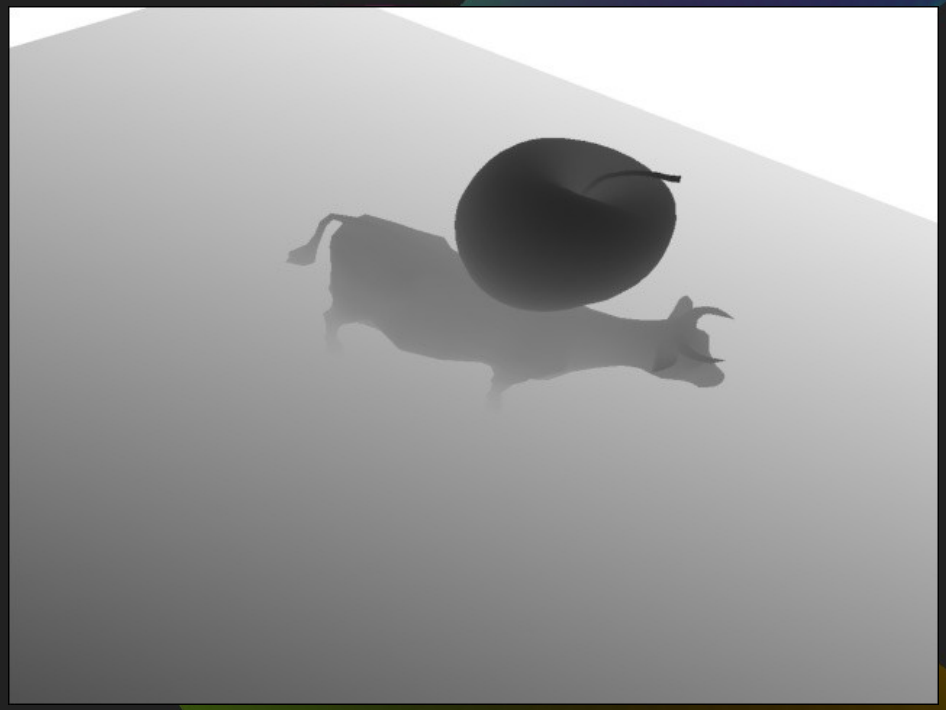


- Les ombres portées ne peuvent pas être calculées comme en lancer de rayons
  - Mode de dessin incompatible
- C'est une couleur calculée par le fragment shader

# Calcul d'ombre

- L'algo du fragment shader est simple :
  - Si le fragment est visible de la lampe, il est éclairé, sinon il est à l'ombre
- Tout le problème est de savoir si un fragment est visible de la lampe
  - On utilise une image de la scène vue de la lampe
    - Lampe de type spot : position, direction, angles
    - On positionne la caméra là où est la lampe
    - L'image RGB est sans intérêt
    - On s'appuie sur le depth buffer de cette vue

# Scène vue de la lampe

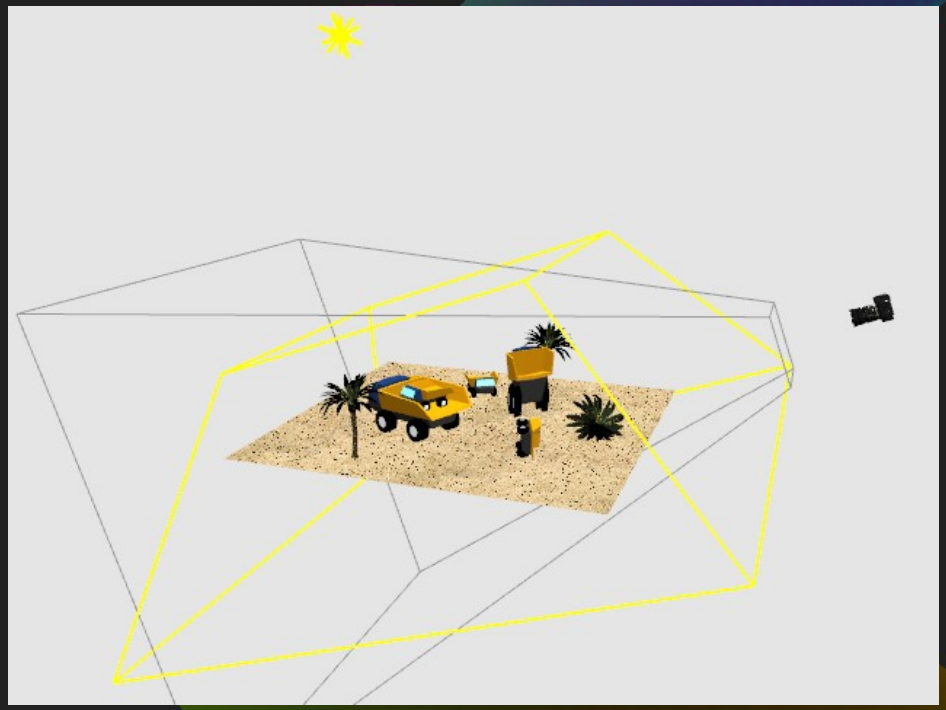


- Voici le depth buffer de la scène vue de la lampe
  - Sombre = proche
  - Clair = éloigné
- Le champ est incliné et distordu à cause des transformations géométriques

# Rôle du depth buffer

- Ce depth buffer indique la distance entre la lampe et l'objet le plus proche, en chaque fragment
  - Fragment éclairé : « visible » dans le depth buffer
  - Fragment à l'ombre : « absent » du depth buffer
  - « Visible ou absent » = comparaison de distances
- Problème : dans ce depth buffer, la scène est vue de la lampe, or les fragments vus de la lampe ne sont pas ceux vus de la caméra
  - Comparer les fragments dans deux repères

# Repères de dessin



- Il faut considérer deux repères :
  - Repère caméra
  - Repère lampe
    - Relatif à la scène
  - Chacun a ses transformations
- Quelle est la matrice de changement entre eux ?

# Rappel matrices P, V et M

- Les maillages contiennent les coordonnées locales, `glVertex` dans le shader
- La matrice M les transforme dans le repère scène (position et orientation du maillage)

  cette matrice M est spécifique à chaque objet

- La matrice V les transforme dans le repère caméra (mouvements souris et clavier)
- La matrice P les transforme dans le repère écran ([-1, +1] sur les trois axes)

# Repères caméra et lampe

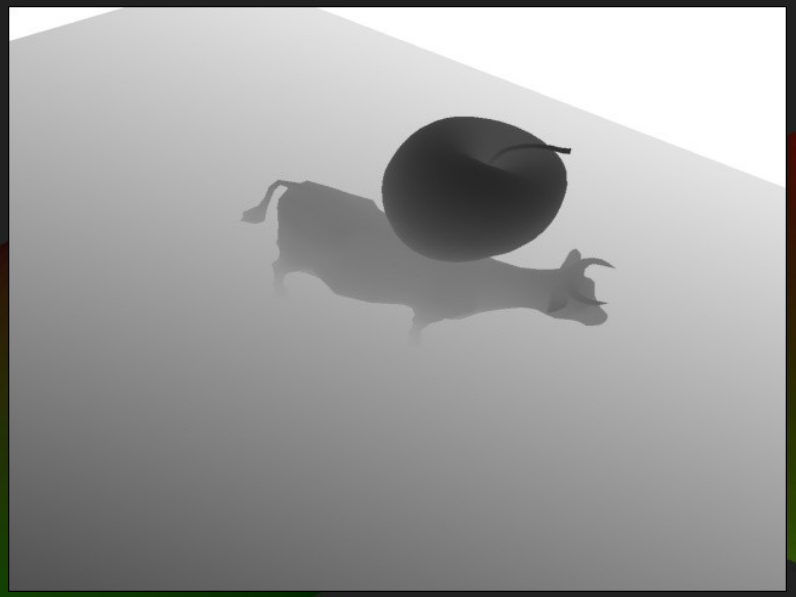
- On doit distinguer deux points de vues :
  - Caméra : M, $V_{caméra}$, $P_{caméra}$
  - Lampe : M, $V_{lampe}$, $P_{lampe}$
  - La matrice M est commune aux deux (position et orientation de l'objet par rapport à la scène)
- Dans le vertex shader, on décompose :

  frgPosition = $V_{caméra}$ * M * glVertex

  gl_Position = $P_{caméra}$ * frgPosition

# Scène vue de la lampe

- C'est pareil que lors du dessin « normal », sauf qu'on dessine dans le depth buffer :
  - Pos. depth buffer = $P_{lampe}$ * $V_{lampe}$ * M * glVertex



# Repère caméra vers lampe

- Maintenant, on dessine la scène vue de la caméra et on veut savoir si un fragment est éclairé à l'aide du depth buffer de la lampe
- Il faut déterminer ses coordonnées de texture (projetées) dans le depth buffer à partir des coordonnées caméra frgPosition
  - On construit la matrice

    $matShadow = P_{lampe} * V_{lampe} * V_{caméra}^{-1}$
  - On a alors :

    Pos. depth buffer = matShadow * frgPosition

# Mise en œuvre

- Plusieurs étapes
  1) Créer un FBO qui ne contient qu'un depth buffer
  2) Dessin de la scène vue de la lampe dans le FBO
     a) Préparation des matrices $P_{lampe}$ et $V_{lampe}$
     b) Préparation de la matrice matShadow
     c) Dessin de la scène dans le FBO
  3) Dessin de la scène vue de la caméra
     a) Fournitude du depth buffer et de matShadow à tous les matériaux
     b) Dessin de la scène : chaque fragment shader teste la visibilité de la lampe

# Préparation de la matrice $P_{lampe}$

- On utilise l'angle de vue maximal de la lampe (spot) et on rajoute near et far :

```
mat4.perspective(
        matProjectionLight,
        Utils.radians(MaxAngle)*2.0, 1.0,
        near, far);
```

# Préparation de la matrice $V_{lampe}$

- Les caractéristiques de la lampe Spot :

```
let target = vec3.create();
vec3.add(target,
     this.m_PositionCamera,
     this.m_DirectionCamera);
mat4.lookAt(matViewLight,
     this.m_PositionCamera,
     target,
     vec3.fromValues(0,1,0));
mat4.multiply(matViewLight,
     matViewLight, matViewCamera);
```

# Matrice d'ombre

- La matrice matShadow est calculée par :

```
mat4.invert(matShadow, matViewCamera);
mat4.multiply(matShadow,
    matViewLight, matShadow);
mat4.multiply(matShadow,
    matProjectionLight, matShadow);
```

- Il reste un dernier calcul pour ramener les coordonnées de -1..+1 à 0..1

# Matrice d'ombre, suite

- On termine la préparation de matShadow par :

```
mat4.multiply(matShadow, matBias, matShadow);
```

- Avec la matrice matBias qui décale les coordonnées projetées -1..+1 vers 0..1 :

```
let matBias = mat4.create();
mat4.translate(matBias, matBias,
    vec3.fromValues(0.5, 0.5, 0.5));
mat4.scale(matBias, matBias,
    vec3.fromValues(0.5, 0.5, 0.5));
```

# Dessin des ombres

- Dans le fragment shader des matériaux, il suffit de rajouter un test de visibilité supplémentaire après celui de la présence dans le cône d'illumination de la lampe spot :

```
visibility *= isIlluminated();
```

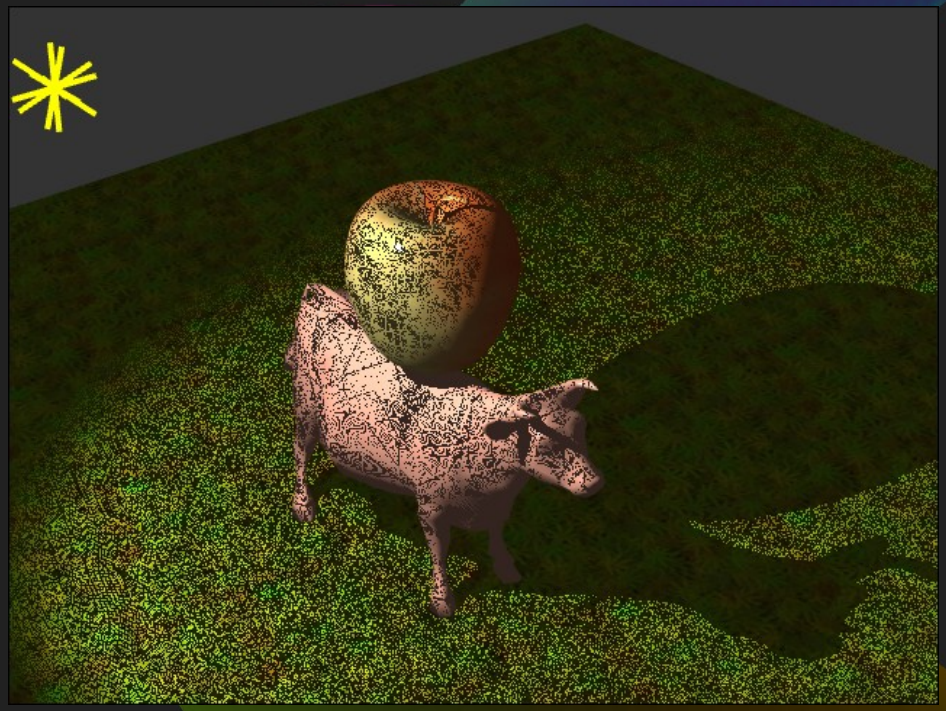
- La fonction `isIlluminated` retourne 0.0 si la lampe n'est pas visible du fragment

# Test de visibilité

- La fonction `isIlluminated` compare la distance du fragment dans le repère lampe à celle qui est dans la shadow map :

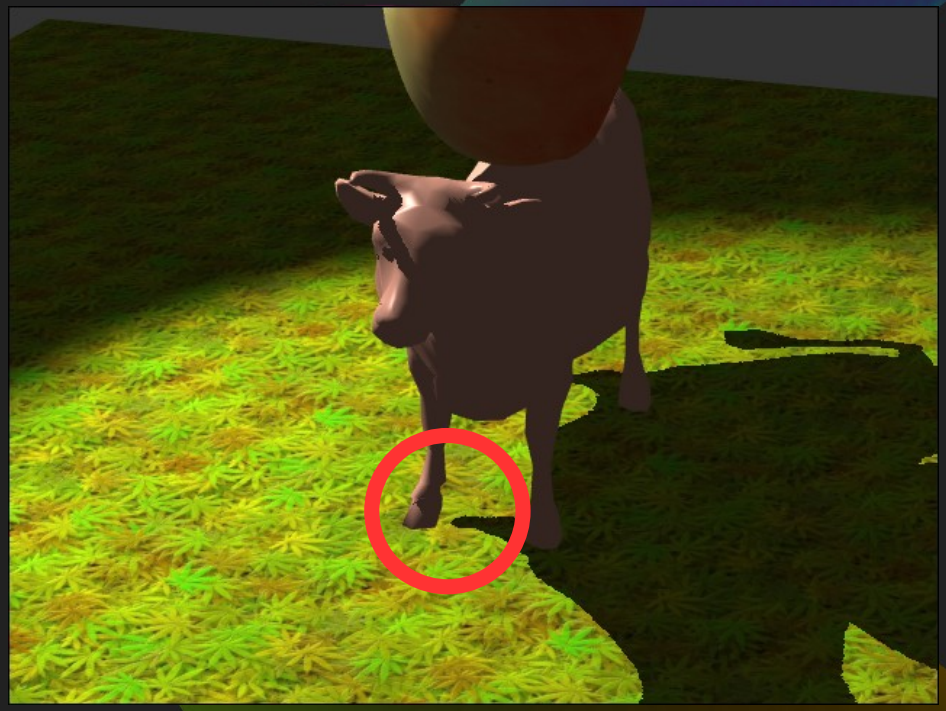
```
float isIlluminated() {
  vec4 posshadow = matShadow * frgPosition;
  posshadow /= posshadow.w;
  float distancePointLight = posshadow.z;
  float distanceObstacleLight =
    texture(ShadowMap, posshadow.xy).r;
  return step(
    distancePointLight,
    distanceObstacleLight);
}
```

# Défauts et remèdes



- À cause du manque de précision, les objets peuvent s'auto-ombrer : « acné de surface »
  - Décaler les polygones ou ajouter un epsilon aux comparaisons de distance
  - Dessiner le dos des objets

# Ombres volantes



- Le défaut de Peter Pan :
  - Quand on décale trop les polygones pour réduire l'acné, les ombres se détachent des objets
  - Impossible d'améliorer les deux simultanément (pb de précision des nombres dans le depth buffer)

# 5.6 - Transparence

- L'objet situé devant laisse voir une partie de ce qui est derrière lui
- Utilisation du « canal alpha » = opacité

# Mélange (blending)

- Le principe du « mélange en avant » est de
  1) Dessiner ce qui est derrière
  2) Dessiner l'objet transparent avec un mode spécial
  3) Dessiner ce qui est devant l'objet transparent
- Le mode spécial consiste à mélanger la couleur des nouveaux fragments avec celle des fragments déjà dessinés
  - Le mélange est paramétré par une 4$^{e}$ composante appelée « canal alpha »

# Exemple

```
onDrawFrame()
{
    gl.clear(gl.COLOR_BUFFER_BIT);
    this.m_RedTriangle.onDraw();
    gl.enable(gl.BLEND);
    this.m_GreenTriangle.onDraw();
    gl.disable(gl.BLEND);
}
```

- Avec le fragment shader du triangle vert :

```
glFragColor=vec4(0.40, 0.75, 0.11, 0.5);
```

# Canal alpha

- C’est un réel entre 0 : transparent, 1 : opaque
- Inclus dans certaines images (GIF et PNG)

# Équation de mélange

- Le mélange se fait entre la couleur de fond (appelée **destination**) et la couleur sortant du Fragment Shader (appelée **source**)
- OpenGL peut faire toutes sortes de mélanges, dont ceux-ci les plus utiles :
  - résultat = source + destination
  - résultat = 0.5*source + 0.5*destination
  - résultat = source.alpha*source + (1-source.alpha)*destination

# Configuration du mélange

- Dans sa forme générale, c'est :

  résultat = *opérateur*(*sfactor* * src, *dfactor* * dest)

- Avec opérateur = `gl.FUNC_ADD`, `gl.FUNC_SUBTRACT`, `gl.MIN`, `gl.MAX` fourni à `gl.blendEquation(operateur)`
- *sfactor* et *dfactor* sont des coefficients comme `gl.ONE`, `gl.SRC_ALPHA`, `gl.ONE_MINUS_SRC_ALPHA` (il y en a d'autres) à fournir à `gl.blendFunc(sfactor, dfactor)`

# Exemple de mélange

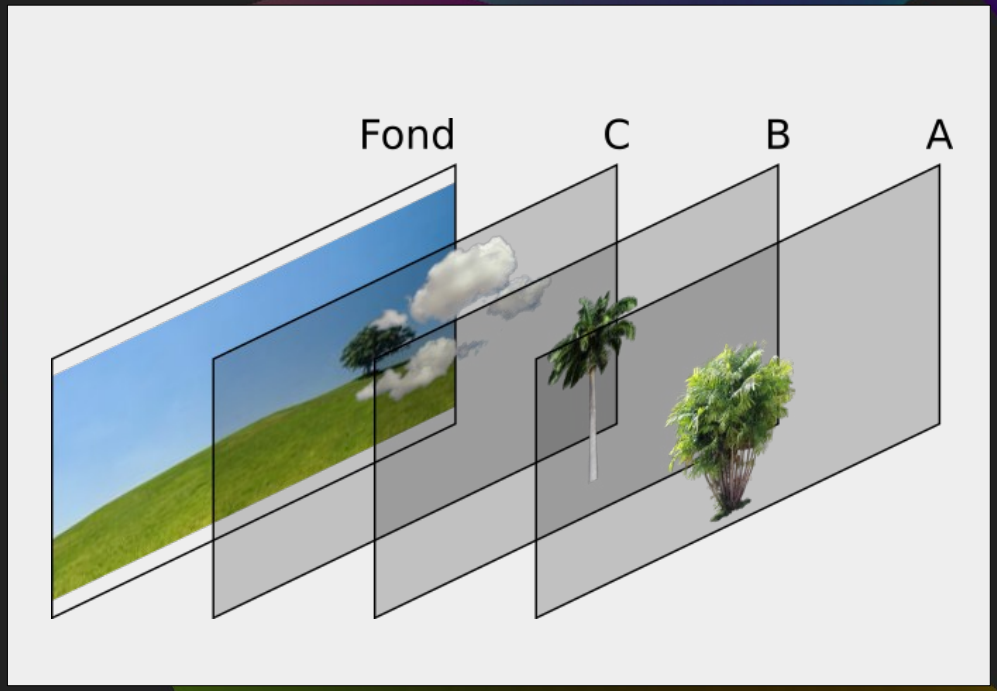
```
gl.blendEquation(gl.FUNC_ADD);
gl.blendFunc(
      gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

- Met en place la formule :

  résultat = source*source.alpha +
  destination*(1-source.alpha)

# Applications

- Superposer des objets




- Dessiner fond puis C, puis B, puis A.

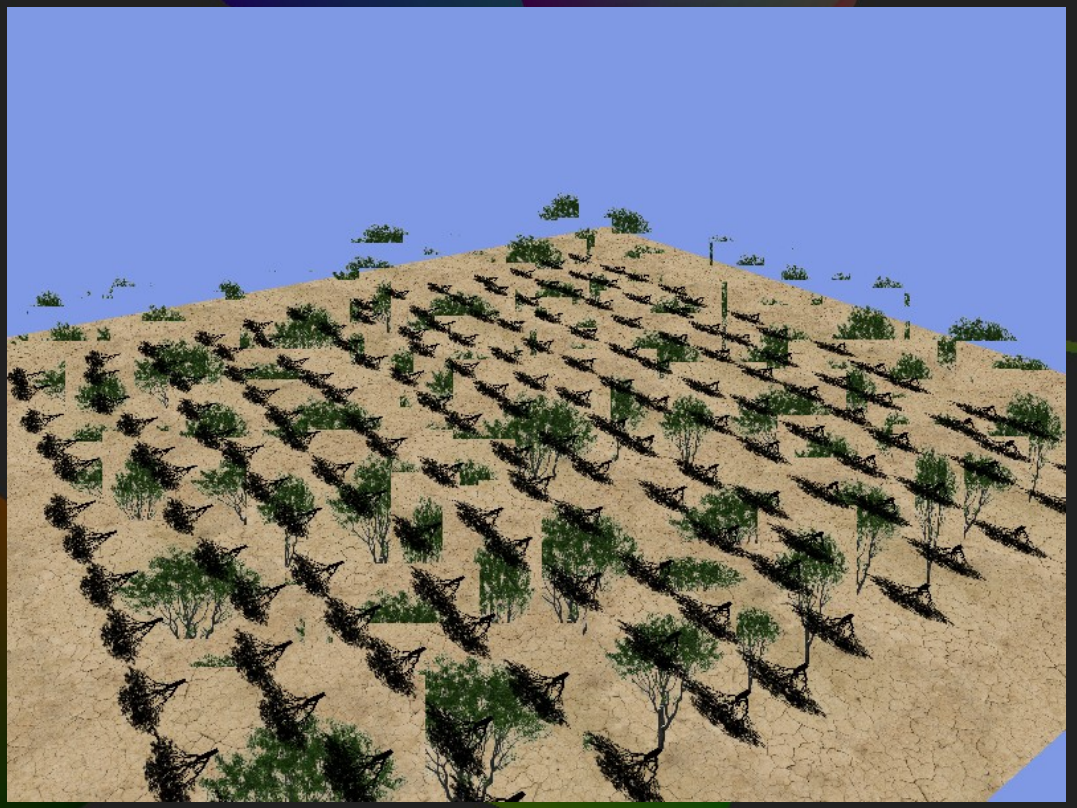
# Panneaux (billboards)

- L'une des applications de la transparence :

- Les ombres sont dessinées de la même manière

# Contrainte du dessin avec mélange

- Il faut dessiner du fond vers l'avant car les dessins se superposent successivement
  - => Tri selon Z
  - Algo du Peintre !
- Sinon :

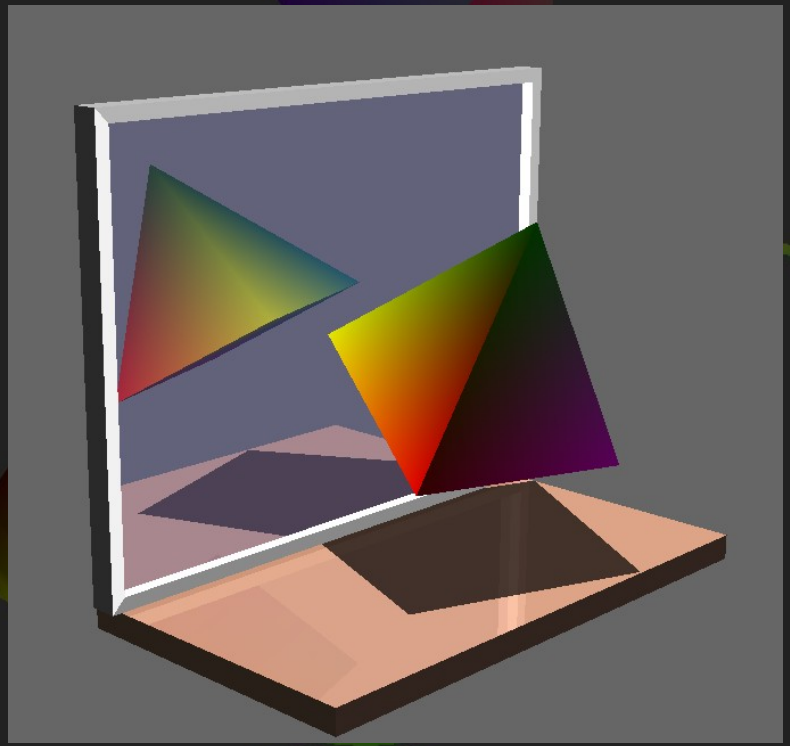


# Toujours face à la caméra

- Dans certains cas, il est nécessaire que le panneau reste toujours face à la caméra
  - => altération de la matrice caméra avant de le dessiner
  - On annule toute rotation dans cette matrice en forçant sa sous-matrice 3x3 haut gauche à l'identité

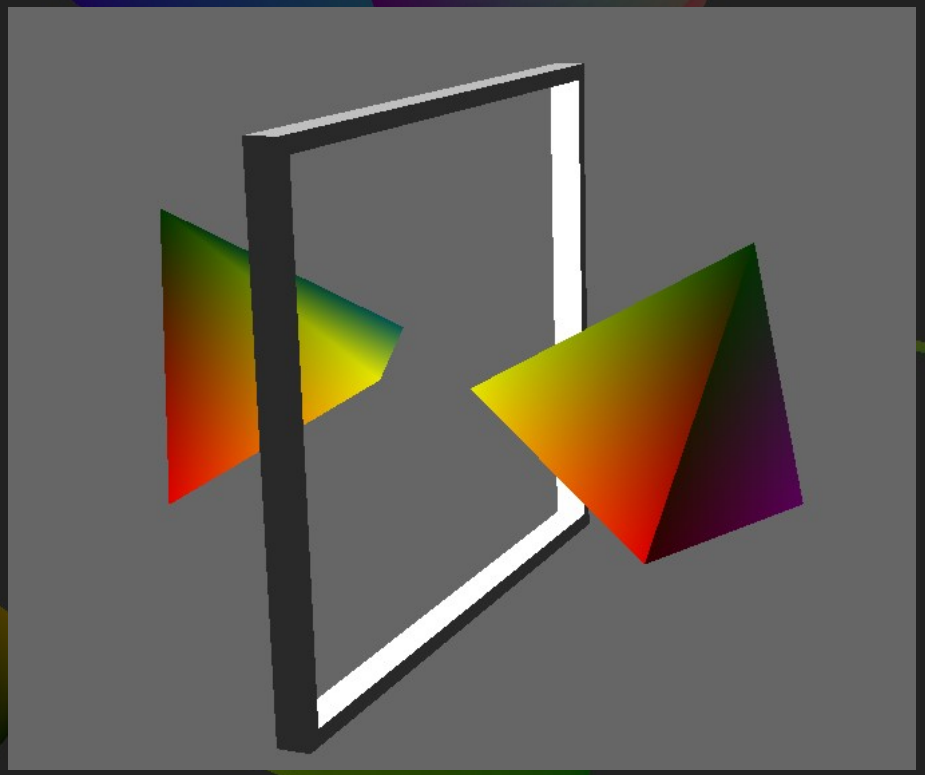
# 5.7 - Reflets

- Le calcul des rayons réfléchis style « lancer de rayons » est incompatible avec OpenGL
- Solution : les objets sont dessinés deux fois
  - Reflet dessiné inversé (matrice de symétrie)
  - Faire en sorte qu'il ne déborde pas du cadre
    - Ajout d'un peu de blending (atténuer le reflet)



# Reflet

- Dessiner en miroir :
  - ModelView multipliée par une « matrice miroir »
  - Inverser les normales

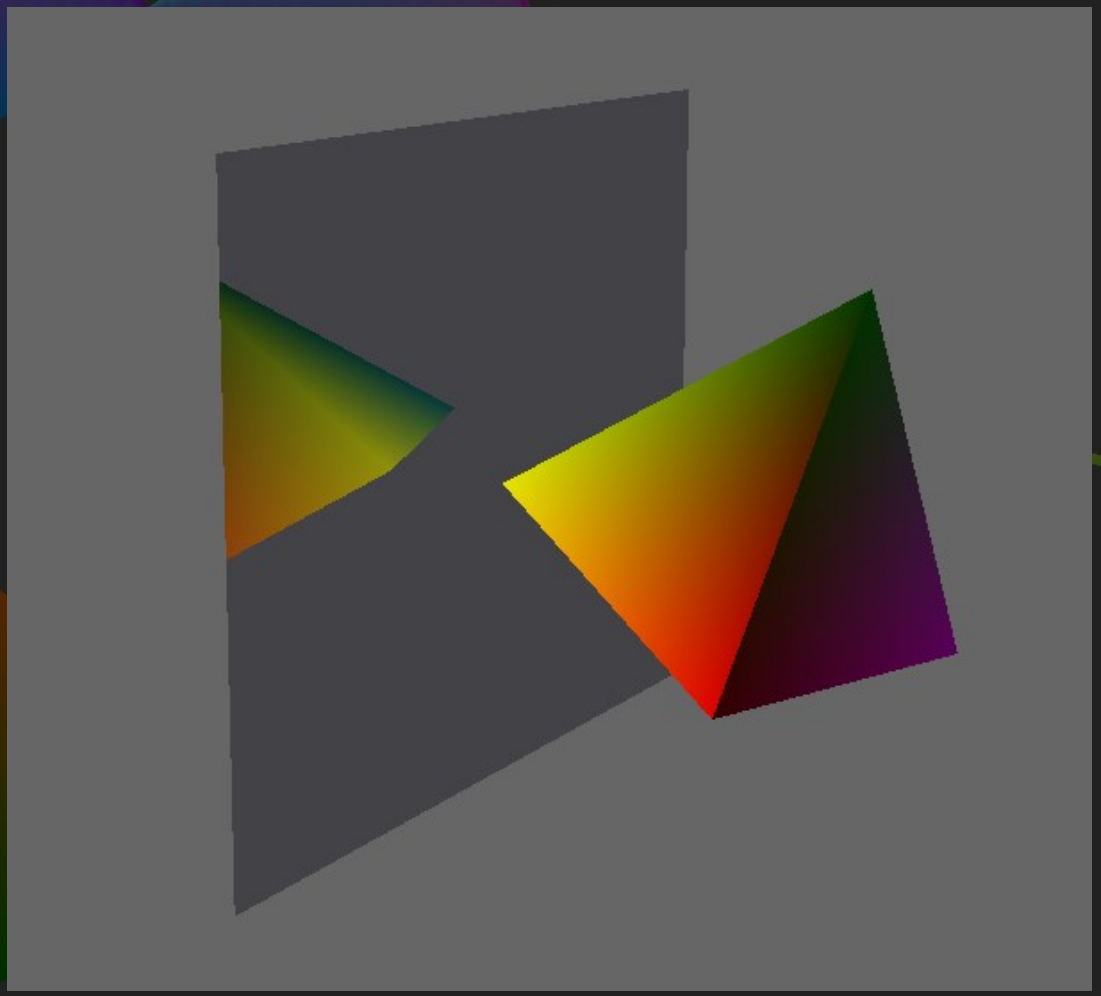


- PB : ça déborde...

  => Utiliser un Stencil

# Stencil OpenGL

- Sorte de masque pour ne dessiner que dans certains endroits
- Ex : le miroir



- Le mot français est « pochoir »

# Mode d'emploi du Stencil

1) Activer le mode création de stencil
2) Dessiner la « zone autorisée » (ex : miroir)
3) Passer en mode utilisation du stencil
4) Dessiner ce qui sera tronqué par le stencil (ex : reflet de l'objet)
5) Revenir au mode sans stencil

- Plus facile à faire avec la classe Stencil dans libs/