TP2

Site: lms.univ-cotedazur.fr
Cours: Realite virtuelle - EIMAD919
Livre: TP2

Imprimé par: Theo bonnet
Date: vendredi 28 février 2020, 14:53

Table des matières

TP2

Mise en place

Le point de départ est le Tuto2 qu'il faut entièrement copier en un répertoire TP2. Renommer le fichier de projet `RVTuto2.pro` en `RVTP2.pro`.
Lancer QtCreator sur ce projet.
Dans le fichier `main.cpp` modifier le titre de la fenêtre.
On a donc dans ce projet les fichiers et classes suivantes :

- `main.cpp` qui est le point d'entrée du programme, qui crée une instance de `MainWindow` qui est la fenêtre principale de l'application.
- Classe `MainWindow` qui est la fenêtre principale de l'application et dont l'interface graphique est défnie dans le fichier `mainwindow.ui` qui est utilisé par QtDesigner (le concepteur d'interface). La variable membre `ui` de cette classe donne accès (par leur nom) à tous les composants de l'interface utilisateur.
- Classe `RVWidget` qui est notre widget (qui hérite du widget QOpenGLWidget de Qt) qui gère le contexte de rendu de OpenGL. Cette classe agit comme un Vue-Controlleur : c'est à dire qu'il s'occupe de l'affichage de la vue OpenGL, et qu'il gère les intéractions avec l'utilisateur et les classes de type Model.
- Classe `RVBody` qui est la classe de base (abstraite) de tous les objets 3D que l'on veut afficher dans le widget. Cette classe va évoluer dans ce TP (et dans les suivants) pour ajouter des propriétés qui peuvent être *factorisées*.
  - `RVPyramid` hérite de `RVBody` et représente un tétrahèdre dont les sommets sont rouge, vert, bleu et blanc. Exemple de base pour voir comment on implémente des classes qui héritent de `RVBody`.
- Classe `RVCamera` qui est la classe de base de toutes les caméras qui sont utilisées par le widget pour faire le rendu des objets. Noter que chaque instance de `RVBody` (ou de ses classes filles) doit contenir un pointeur sur la caméra à utiliser dans le rendu.

Le but de ce TP est de se familiariser avec cette hiérarchie de classes afin de l'enrichir en lui ajoutant des nouvelles fonctionnalités, des nouveaux objets 3D et de nouvelles caméra.

Classe RVCube

La classe `RVCube`, construite sur le même modèle que `RVPyramid`, va encapsuler tout le code que vous avez écrit dans le TP1 pour afficher le cube coloré.
Créer une nouvelle classe avec `Add New...`, choisir `C++ Class` de nom `RVCube` qui hérite de `RVBody`.
Dans `rvcube.h` ajouter

- l'include de `rvbody.h`
- la déclaration des 3 méthodes virtuelles pures de la classe mère à surcharger :
  - void draw()
  - void initializeBuffer()
  - void initializeVAO()

  Normalement dès que l'on commence à écrire le type `void...`, l'éditeur vous propose le nom des méthodes virtuelles à surcharger.

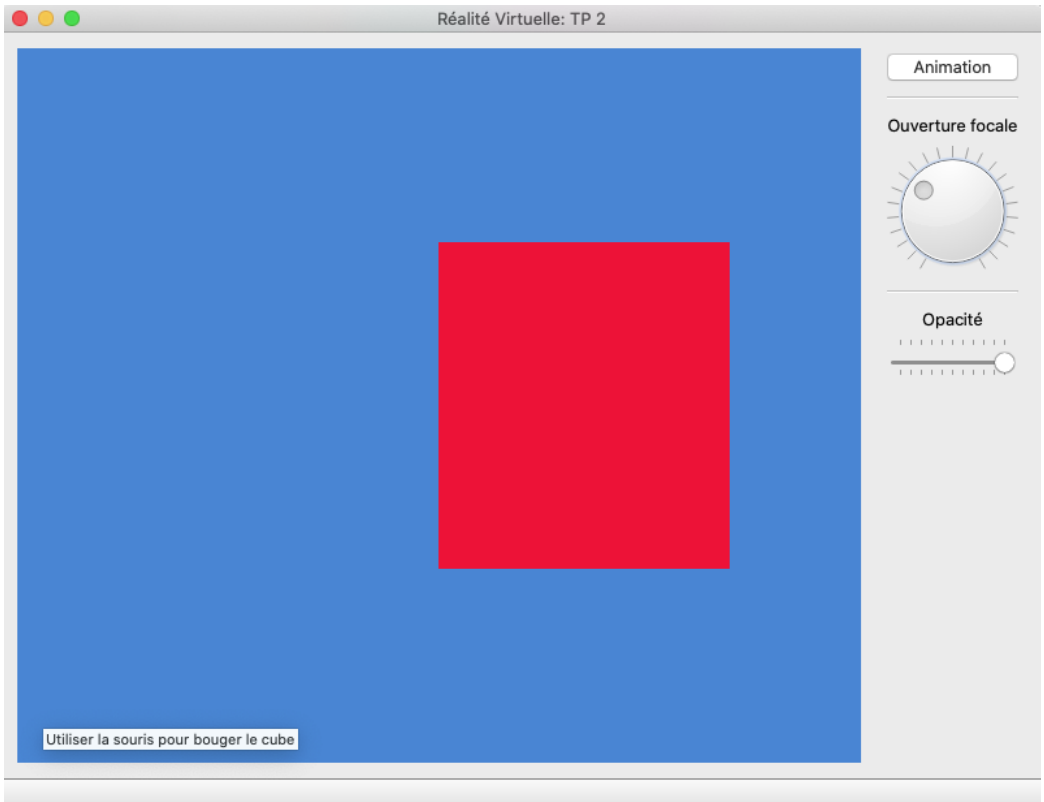
Dans le fichier `rvcube.cpp`

- **Constructeur** : comme dans le constructeur de `RVPyramid` on appelle le constructeur de la classe parent et on définit les noms des fichiers qui contiennent les shaders (les mêmes).
- **méthode** `initializeBuffer()` : on copie, à partir de la classe RVWidget du TP1, le code de initializeBuffer
- **méthode** `initializeVAO()` : on copie, toujours à partir du RVWidget du TP1, la partie de code qui initialise le *vertex array objet* (VAO) qui se trouve à la fin de la méthode `initializeShader()`.
- **méthode** `draw()` : en s'inspirant de la méthode `draw` de `RVPyramid` on intègre les commandes de dessin du cube fait dans le TP1 (en laissant de coté l'opacité dans un premier temps).

Pour tester si votre classe `RVCube` fonctionne, il faut ajouter l'include de `rvcube.h` dans le fichier `rvwidget.h` et de changer une ligne dans `RVWidget::initializeGL()` : à la place de la création d'une instance de RVPyramid on met :

```
m_body = new RVCube();
```

Tout le reste est identique et donne ça :

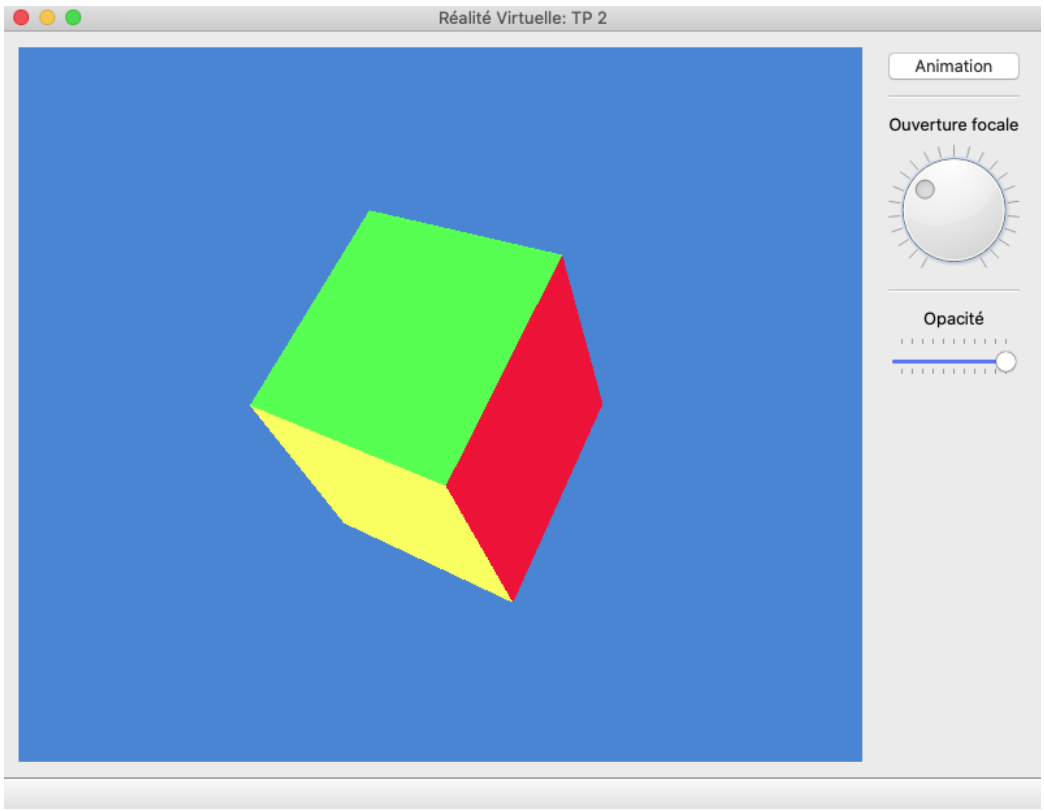


Mais si l'on essaie de le déplacer ou si on active l'animation, on s'aperçoit que maintenant les rotations ne se font plus par rapport au centre du cube mais par rapport à un axe qui passe par un sommet. Il manque la translation de vecteur (-0.5, -0.5, -0.5) qui mettait le centre du cube à l'origine. Il faut donc surcharger la méthode `modelMatrix` de `RVBody` de façon à ce que l'on fasse cette

translation en premier (c'est à dire à droite dans le produit matriciel) avant les transformations de RVBody telles qu'elles sont définies par `m_position` et `m_orientation`.

```
QMatrix4x4 RVCube::modelMatrix()
{
    QMatrix4x4 model;
    model.translate(-0.5f, -0.5f, -0.5f);
    return RVBody::modelMatrix() * model;
}
```

Enfin pour que le cube soit bien au centre de l'image, il faut changer sa position en (0, 0, -4) lors de sa création.



Transparence et autres améliorations pour RVBody

Transparence

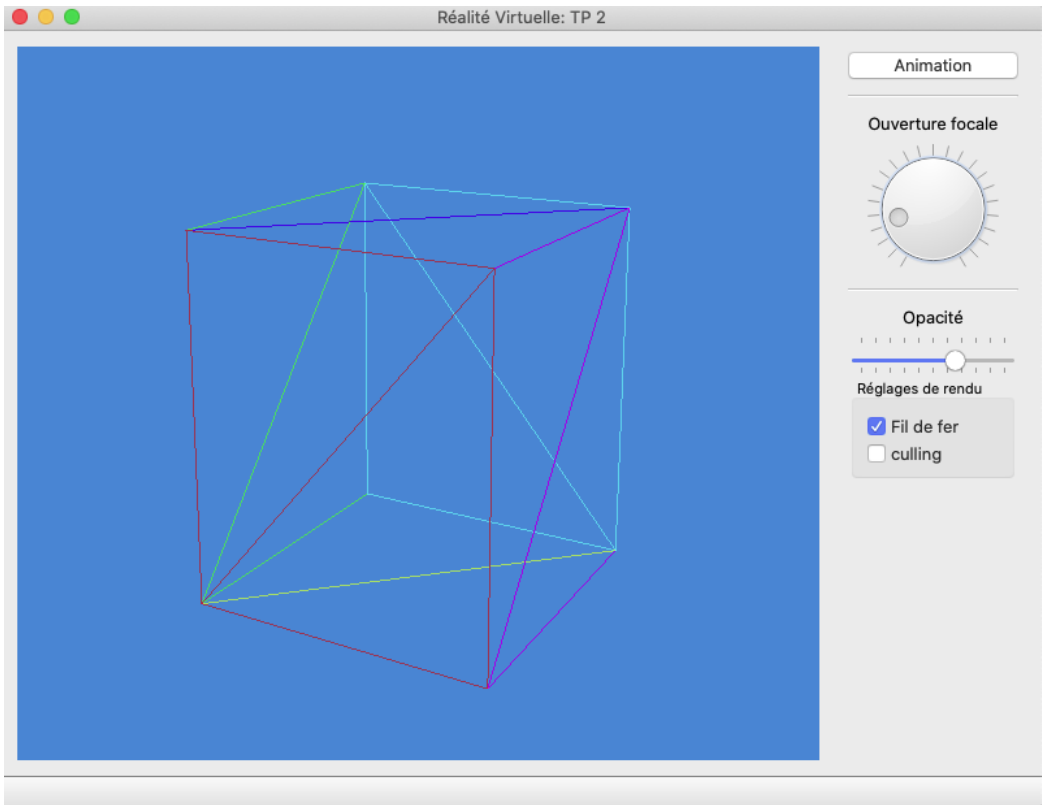
Pour intégrer la transparence comme on avait fait dans le TP1, il faudrait intégrer les modifications du vertex shader, la variable membre `m_opacity` etc.. au RVCube....
Mais on pourrait aussi se dire que cela vaut la peine que toutes les classes qui héritent de `RVBody` devraient avoir la possibilité d'avoir leur transparence modifiée. Donc on va faire les modifications au niveau de la classe de base.
Donc dans la version 2.0 de la classe `RVBody` on :

- ajoute une variable membre `m_opacity` de type float avec accesseur et mutateur, initialisé à 1 dans le constructeur.
- on intègre dans le vertex shader la variable uniforme `u_opacity` comme dans le TP1
- on ajoute dans la méthode `paint()` de `RVCube` le passage de la valeur de `m_opacity`
- on ajoute à `RVWidget` le slot `changeOpacity(int)` qui modifie l'opacité de `m_body` via son accesseur.

Ainsi, comme dans le TP1, on peut modifier la transparence du cube coloré avec un slider.
Bilan, on a exactement le même résultat que dans le TP1 mais avec du code complètement transformé et mieux structuré !
Remarque : les classes filles de `RVBody` ne sont pas obligées de tenir compte de cette variable membre `m_opacity`. Elles peuvent l'ignorer totalement et dans ce cas le changement de valeur de cette variable n'aura aucune influence sur le rendu.

Mode filaire

On continue à enrichir la classe `RVBody` pour pouvoir choisir un rendu en mode filaire à la place du rendu faces pleines. La encore c'est une possibilité que l'on ajoute à la classe de base et que certaines classes filles peuvent vouloir implémenter.
Il faut ajouter à la classe `RVBody` un booléen `m_wireframe` initialisé à `false` dans le constructeur avec ses accesseur et mutateur.
On modifie du coup la classe `RVCube` pour qu'elle tienne compte de cette possibilité : il suffit dans la méthode `draw` de modifier, selon la valeur du booléen, les réglages `glPolygonMode` du contexte de rendu.
En mode filaire, et même lorsqu'on active la transparence, on peut vouloir désactiver le *culling* c'est à dire l'élimination des triangles qui ne font pas face à l'observateur : pour cela une nouvelle variable membre `m_culling` avec ses accesseurs et mutateur et le tour est joué.
Après pour en tenir compte dans l'application, il faut modifier l'IHM, ajouter des slots et on peut changer interactivement le type de rendu de l'objet.



Homothétie

Actuellement la matrice `modelMatrix` de `RVBody` qui place l'objet sur la scène est la composition d'une translation et d'une orientation, donc une isométrie. Mais il peut arriver que l'on veuille modifier la taille des objets lorsqu'on les place sur la scène par rapport à leur taille originale (s'ils sont trop gros ou trop petits par rapport aux autres éléments qui composent la scène).
Pour cela nous allons composer l'isométrie originale par une *homothétie* de rapport `m_scale` (qui est donc une nouvelle variable membre de la classe `RVBody` de type *float*).
Donc dans la méthode `modelMatrix` on ajoute (après les autres appel) l'appel de la méthode `Qmatrix4x4::scale(float k)`

Je rappelle (encore une fois) que cela correspond à multiplier la matrice précédente *à droite* par la matrice d'une homothétie de centre O et de rapport k (qui a donc une diagonale de k). Puisque cette matrice est à droite ce sera la première à être appliquée aux sommets du cube original, puis la rotation et à la fin la translation.
La encore on peut ajouter un slider pour modifier la taille du cube de façon interactive. Toutefois comme le cube est pour l'instant le seul objet de la scène, on ne voit pas vraiment de différence avec le changement de focale de la caméra.

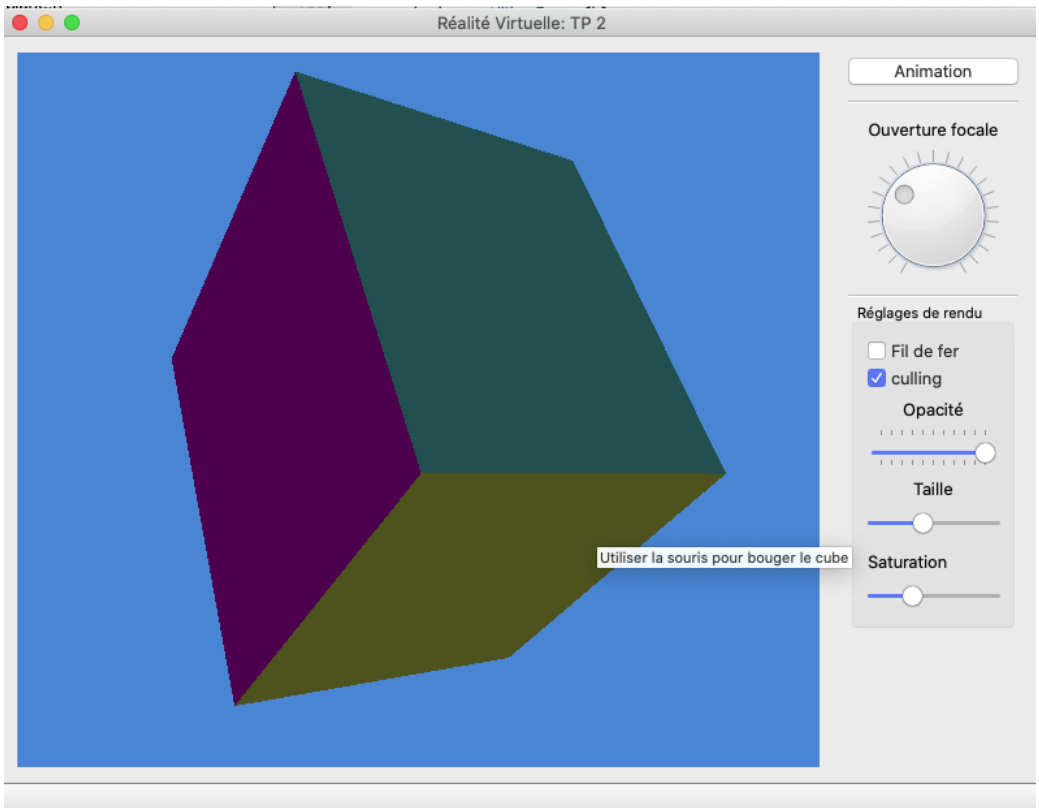
Couleur globale

Dans le cas de notre cube, les couleurs sont intégrées dans le modèle, puisqu'elles sont écrites dans le vertex buffer, c'est à dire dans la mémoire vidéo. Mais dans certains cas la couleur peut venir d'une texture appliquée aux faces de l'objet (c'est ce que nous ferons dans la session 3) mais il peut aussi arriver d'avoir un objet 3D défini par un vertex buffer qui ne contient que des informations sur les positions des sommets et c'est tout.
Pour que cet objet aussi soit colorable, vous allez ajouter une couleur globale à `RVBody` sous la forme d'un `Qvector3D` appelé `m_globalColor` que l'on peut initialiser à blanc (1, 1, 1) dans le constructeur (et toujours accesseur et mutateur).
Comment utiliser cette donnée ?
Les objets concernés devront ajouter une variable globale à leur vertex shader qui donnera la couleur finale `outColor` (c'est ce que vous allez faire dans la prochaine partie).
Mais actuellement, pour le cube, nous pouvons utiliser cette couleur globale pour assombrir les couleurs naturelles du cube :

- modifier le vertex shader du cube en ajoutant une variable uniforme `vec3 u_color` qui sera multipliée par la couleur `rvColor` qui provient du Vertex Buffer. Dans GLSL le produit de deux vec3 se fait *terme à terme* donc si `u_color` est un gris (g, g, g) avec g plus petit que un, le résultat sera un obscurcissement de chaque couleur.
- modfier le `draw()` du cube pour renseigner cette nouvelle variable uniforme à partir de la variable membre `m_globalColor` qui est (255, 255, 255) donc ne donnera aucun effet au rendu tant qu'elle restera comme ça.

Remarquez que la fonction `setUniformValue` va convertir le `QColor` qui contient des données entières entre 0 et 255 en un `vec4` de `float` entre 0 et 1.

- modifier l'IHM pour ajouter un nouveau slider appelé "saturation"
- lier ce slider à un slot `changeSaturation(int g)` qui utilise g pour modifier la variable `m_globalColor` de `m_body`.



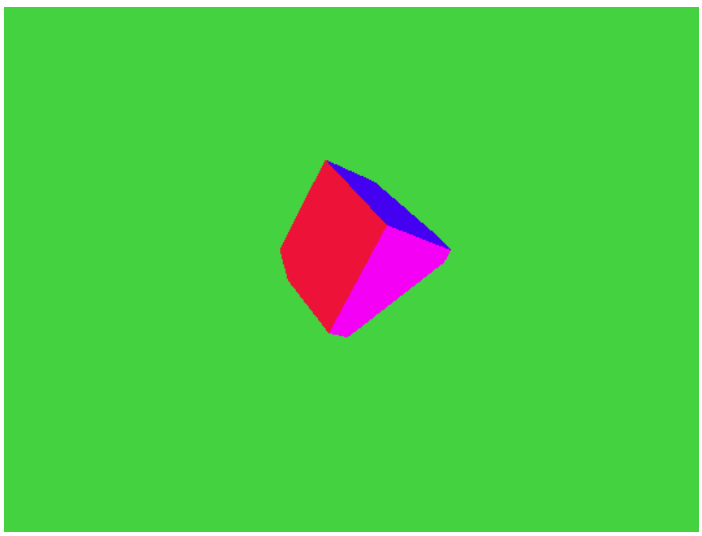
Classe RVPlane

`RVPlane` est une classe qui hérite de `RVBody` qui représente un rectangle horizontal défini par quatre sommets coplanaires.

Spécifications :

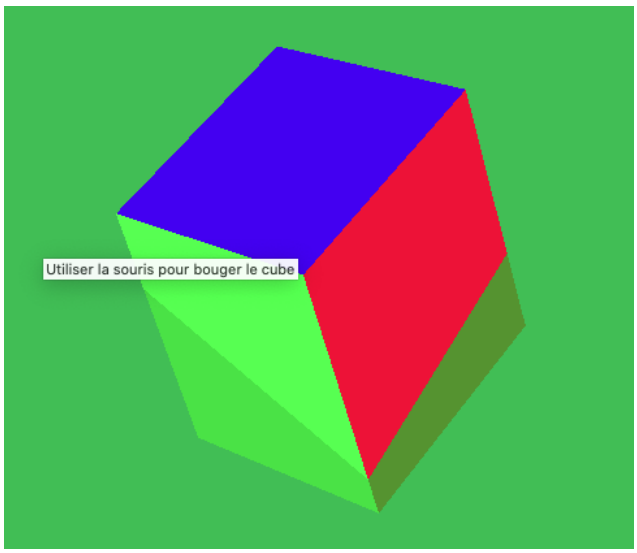
1. Le **constructeur** de cette classe doit pouvoir définir sa longueur (le long de l'axe x) et sa largeur (le long de l'axe z). Ces deux variables membres doivent avoir des valeurs par défaut (par exemple 10x10).
2. Le centre du rectangle (intersection des diagonales) est en (0, 0, 0).
3. Le vertex buffer ne contient que les coordonnées des quatre sommets, pas de couleurs. Il n'y a pas non plus besoin d'utiliser les indices (comme dans RVPyramid). La couleur du plan sera donnée par `m_globalColor` que l'on a rajouté à `RVBody` .
4. Il faut modifier le vertex shader en une version plus simple `VS_simpler.vsh` :
   - plus d'attribut `rv_Color`
   - une variable uniforme `u_color` utilisé dans le `main()` en lieu de `rv_Color` pour générer la couleur finale du vertex.
5. dans la méthode `draw()`, il faut associer la variable membre `m_globalColor` à la variable uniforme `u_color`.

Si on ajoute le plan à notre `RVWidget` en plus du cube, en mettant ces deux objets à leur position par défaut (0, 0, 0), alors il faut éloigner et *élever* l'observateur en plaçant la caméra à (0, 7, 4) et sa cible en (0, 0, 0). Le résultat obtenu est (avec un plan vert) :

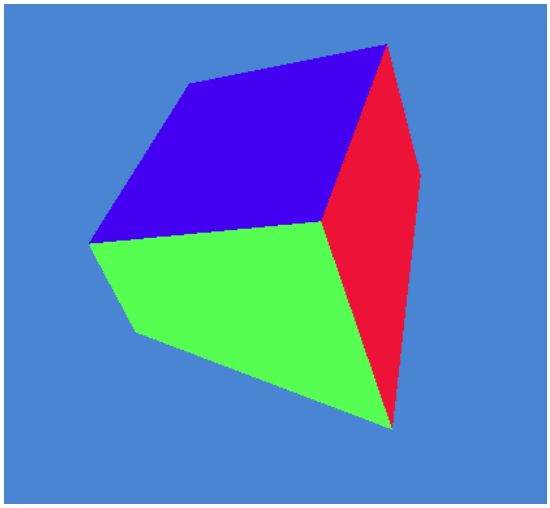


Évidemment tous les contrôles (opacité, taille, saturation) s'appliquent sur le cube et non sur le plan.
Mais si l'on choisit d'appliquer l'opacité au plan au lieu du cube, on va obtenir des résultats surprenants selon l'ordre dans lequel on a lancé les commandes de rendu dans la méthode `RVWidget::paintGL()`.

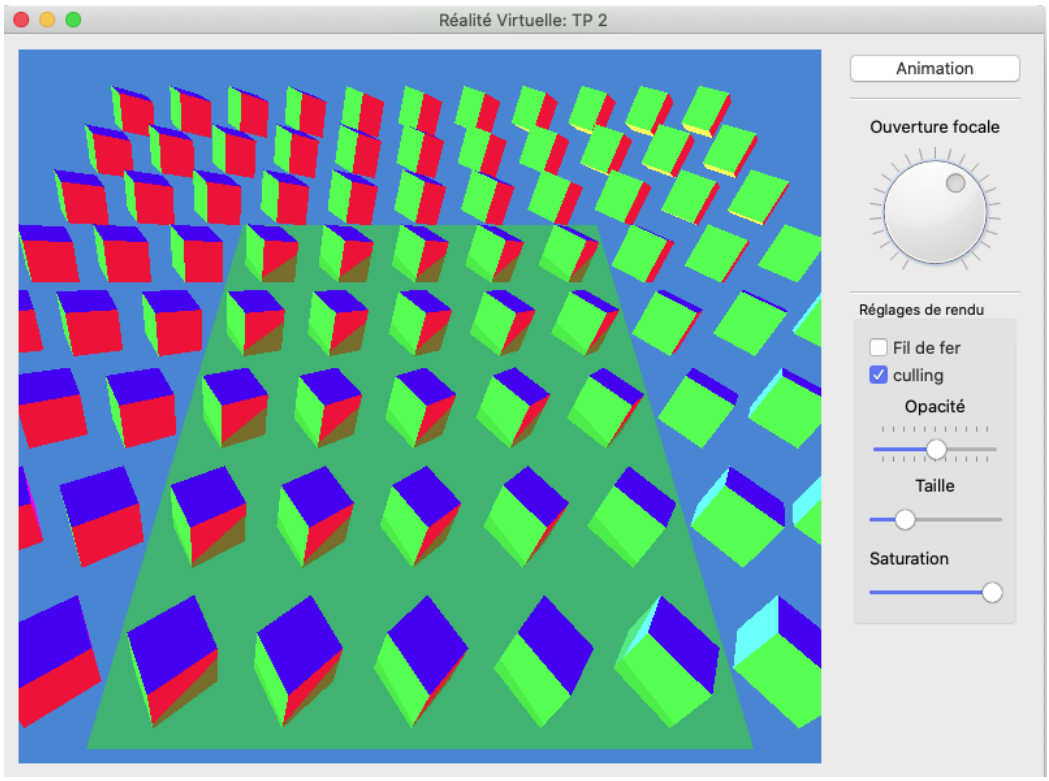
- Si le cube est affiché **avant** le plan, si on baisse l'opacité du plan, on voit apparaître sous le plan, en transparence, le partie inférieure du cube (teintée de vert). Le plan physiquement se comporte comme si c'était un verre vert.



- Si le cube est affiché **après** le plan, avec une opacité égale à 0, toute la partie du cube sous le plan a disparu. C'est le résultat de l'algorithme du *Z-buffer* : lors du rendu, tous les fragments qui ont une profondeur (z) supérieure au fragment déjà enregistré dans le buffer d'image sont ignorés. Ici, lors du traitement des fragments du cube, derrière le plan (même si le plan est transparent).



En bref, la gestion de la transparence avec la composante *alpha* de la couleur ne fonctionne que si l'objet potentiellement non opaque est le dernier à être envoyé au moteur de rendu OpenGL.
Pour tirer profit de tout la travail fait sur les classes dans cette session, on peut afficher une centaine de cubes au lieu d'un seul en ajoutant une double boucle `for` dans `RVWidget::paintGL` et en modifiant la position de cube avant de lancer le rendu.



Shader par défaut

La simplicité du vertex shader `VS_simpler.vsh` qui n'utilise que la position des sommets comme attribut et utilise une couleur globale donnée par une couleur uniforme en fait un bon candidat pour être (avec `FS_simple.fsh`) les **shaders par défaut** de la classe mère `RVBody` .
Ainsi, pour tous les objets 3D avec une teinte unie, les étapes de création seraient beaucoup plus lègères ; par exemple dans la classe `RVCurve` dans le cours sur les graphes et les surfaces.
Un autre avantage serait de donner une version par défaut de la méthode `initializeVAO()` : au lieu d'en faire une méthode virtuelle pure, nous lui donnons une implémentation ar défaut basée sur le shader par défaut ; seulement les filles de RVBody qui utilisent un vertex shader plus sophistiqué auront besoin de surcharger cette méthode.

Consignes

- dans le constructeur de `RVBody` on définit les deux shaders par défaut ;

```
    m_VSFileName = ":/shaders/VS_simpler.vsh";
    m_FSFileName = ":/shaders/FS_simple.fsh";
```

- `initializeVAO` n'est plus une méthode virtuelle pure. Elle a un code qui est en fait celui de `RVPlane` :

```
void RVBody::initializeVAO()
{
    //Initialisation du VAO
    m_vao.create();
    m_vao.bind();
    m_vbo.bind();
    m_ibo.bind();

    //Définition du seul attribut position (la couleur est uniforme avec le VS_simpler)
    m_program.setAttributeBuffer("rv_Position", GL_FLOAT, 0, 3);
    m_program.enableAttributeArray("rv_Position");

    //Libération
    m_vao.release();
    m_program.release();
}
```

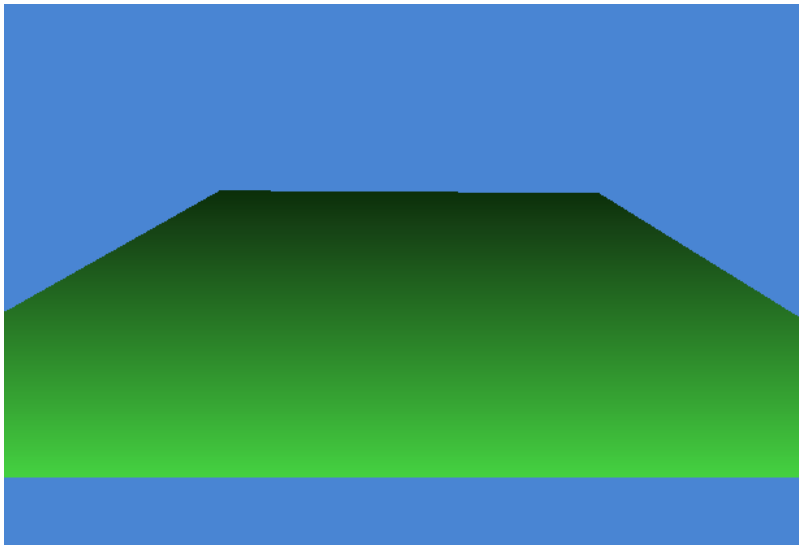
Donc maintenant toute classe fille de `RVBody` qui n'utilise que la couleur globale, le constructeur n'a **plus** à définir quels shaders utiliser, ni il n'est plus nécessaire de donner le code de `initializeVAO()` mais seulement de `initializeBuffer()` et `draw()`.

Modfier `RVPlane` dans ce sens.

Dégradé de couleur

On peut modifier la taille de ce plan avec `setScale()` et donc le rendre très grand, se rapprochant à l'horizon apparent ; à un certain moment, il sera de toute façon tronqué à cause des valeurs `m_zMin` (en avant) et `m_zMax` (à l'arrière).
Mais en augmentant la taille du plan, on peut être gênés par le fait que la couleur reste la même jusqu'au fond... Ce n'est pas très réaliste. Du coup, on aimerait que la teinte du plan s'assombrisse lorsqu'on va vers l'horizon.
On peut faire ça avec un *fragment shader* grâce à la variable `gl_FragCoord` : il s'agit d'un `vec4` qui contient les coordonnées normalisées (entre -1 et 1) du fragment que l'on est en train de traiter. Donc la coordonnée z représente la profondeur du fragment, c'est à dire son éloignement par rapport à l'observateur (plus c'est proche de 1 plus on s'approche de la valeur limite `m_zMax`).

- copier le fragment shader `FS_simple.fsh` en `FS_plan.fsh`;
- intégrer ce changement de fragment shader dans le constructeur de `RVPlane` ;
- ajouter ce fragment shader dans les ressources du projet
- édition de `FS_plan.fsh` : les composantes rgb de la variable de sortie `gl_FragColor` doivent être multipliées par `(1 - gl_FragCoord.z)`. En revanche, la composante *alpha* ne doit pas être modifiée (pour conserver les effets de transparence) et doit reprendre celle de `out_Color`. Ainsi quand la profondeur du fragment s'approche de 1, la couleur de sortie va tendre vers 0 donc va s'assombrir.
- enfin, pour que l'effet soit plus visible, il faut mettre la variable `m_zMin` de la caméra à 10. Ainsi la valeur minimale de la profondeur (z = -1 en coordonnées normalisées) correspond à la distance 10 vis à vis de la caméra.



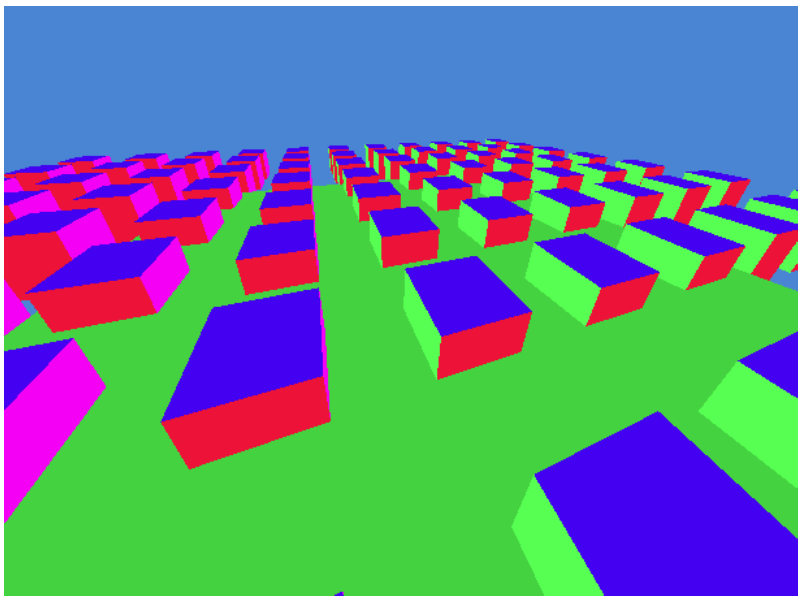
Projection orthogonale

Déplacement de la caméra

Actuellement la caméra ne se déplace pas. Ainsi avant d'ajouter à `RVCamera` la possibilité d'utiliser une projection orthogonale au lieu d'une projection perspective, vous allez utiliser le clavier pour déplacer la caméra : touches `Left-Right` pour le déplacement en x, touches `Up-Down` pour le déplacement en y.

- La méthode de `Qwidget` qui réagit aux événements liés au clavier est `void keyPressEvent(QKeyEvent* event)` ;
- il faut ajouter l'include de `QKeyEvent` à `RVWidget` ;
- il faut déclarer que notre widget réagit aux événements claviers avec un appel de la méthode `grabKeyboard()` dans son constructeur;
- dans la méthode `keyPressEvent` on peut comparer la valeur de `event->key()` avec des constantes prédéfinies par Qt pour chaque touche clavier (d'une façon indépendante du périphérique) `Qt::Left` , `Qt::Right` ...etc...

Ainsi en baissant la caméra et en augmentant la focale, on produit une image où la déformation perspective est visible et importante. En particulier sont visibles les deux points de fuite principaux.



Projection parallèle orthogonale

Vous allez ajouter à la classe `RVCamera` une variable membre `m_isOrthogonal` de type `bool` (fausse par défaut dans le constructeur), ainsi que ses accesseurs. C'est cette variable qui doit être utilisée dans `projectionMatrix()` pour savoir quel type de projection utiliser.
Si ce *flag*> est `false` on a déjà le code à utiliser, alors que s'il est `true` on doit utiliser une autre méthode de `QMatrix4x4` .
La méthode

```
void QMatrix4x4::ortho(float left, float right, float bottom, float top, float nearPlane, float farPlane)
```

construit justement une matrice correspondant à une projection parallèle orthogonale selon l'axe z sur le plan (x,y) réduite à un volume de vue parallélépipédique

$$left \leq x \leq right$$

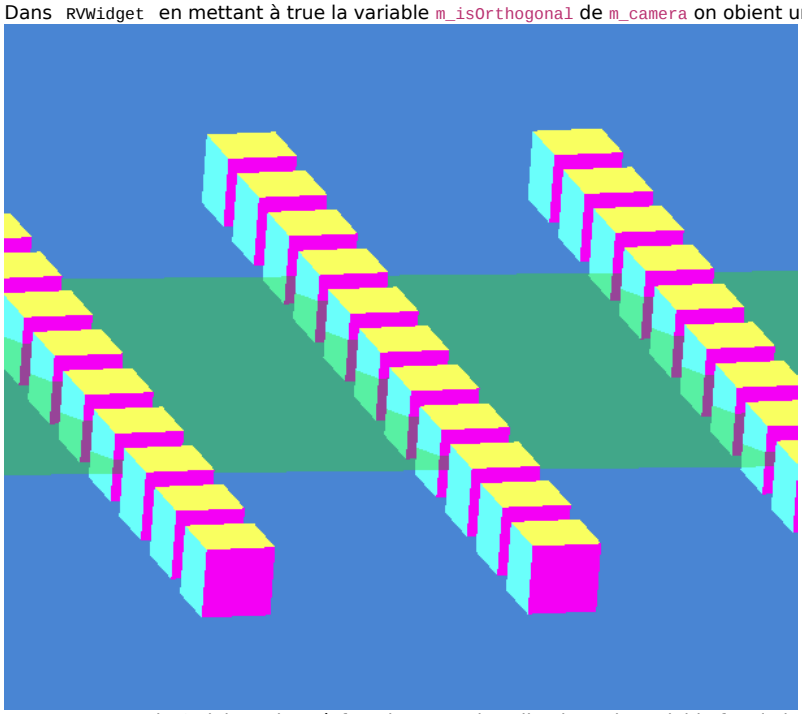
$$bottom \leq y \leq top$$

$$nearPlane \leq z \leq farPlane$$

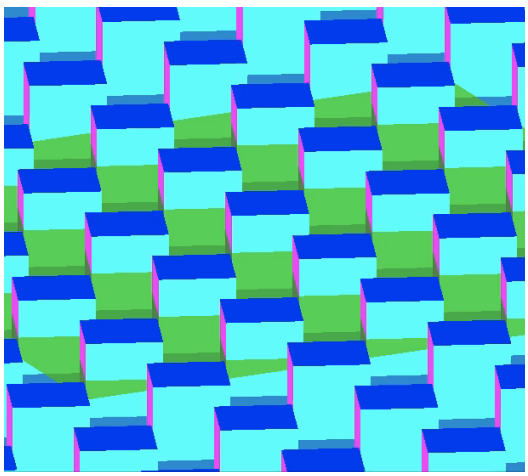
. Il faut donc lui fournir ces 6 arguments en fonction de ce qu'on a déjà dans la classe :

- nearPlane et farPlane sont `m_zMin` et `m_zMax`.
- pour bottom et top on va utiliser `-m_fov/2` et `+m_fov/2` ; c'est à dire qu'on va transformer l'angle vertical (en degrés) en dimension verticale de la fenêtre. Et si c'est trop grand on pourra diviser par plus de 2.
- pour left et right, on prend les valeurs de bottom et top multipliées par `m_aspect` pour respecter les proportions des objets (les cubes ne doivent pas apparaître étirés ou écrasés).

Dans `RVWidget` en mettant à true la variable `m_isOrthogonal` de `m_camera` on obient une image de ce type.



Remarquer que le QDial continue à fonctionner puisqu'il agit sur la variable fov de la caméra.



Bonus : SplitView avec 4 caméras

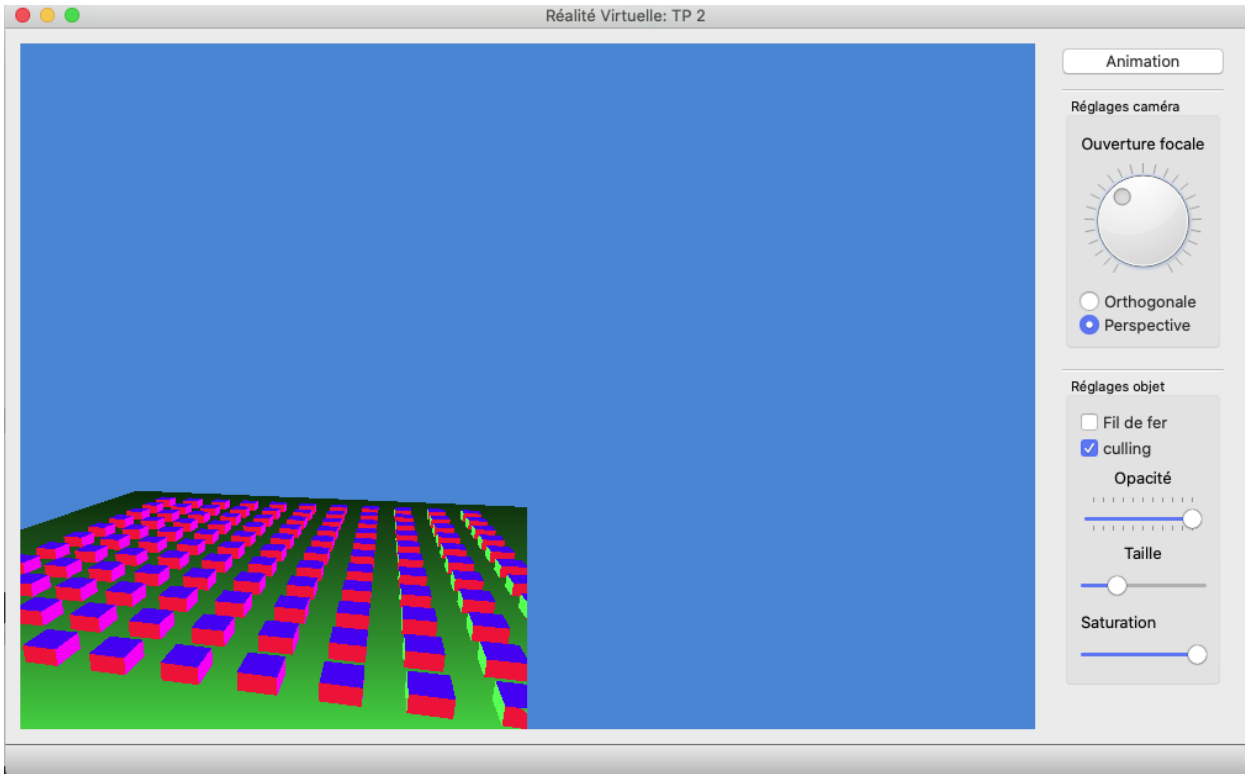
La transformation de `viewport` est la dernière étape du *pipeline de rendu* : le *frame buffer* de OpenGL qui est la zone mémoire qui contient tous les fragments issus du *fragment shader* va être copié sur la mémoire d'écran et le format de pixel du contexte de rendu OpenGL va être converti au format de pixe l de l'écran.
Jusqu'à maintenant la commande était :

```
glViewport(0, 0, w, h);
```

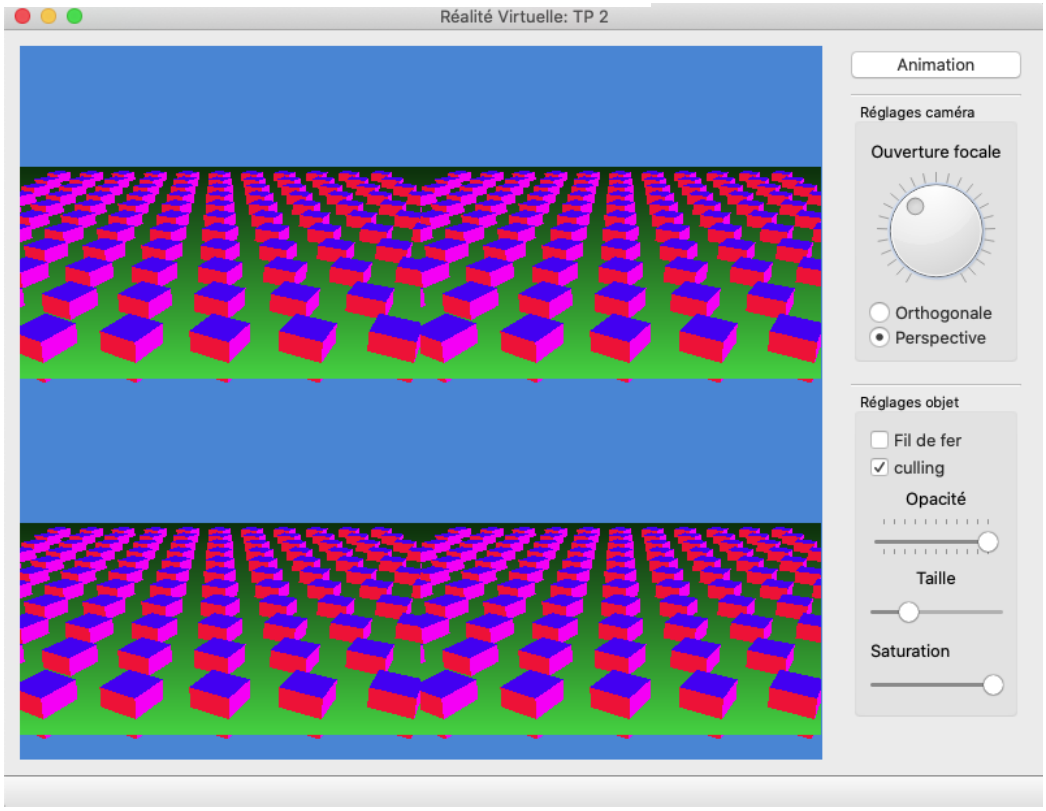
Cela signifie que l'image produite par le moteur de rendu OpenGL va couvrir la totalité de la zone du widget ((0, 0) correspond au point en bas à gauche, w est la largeur du widget et h est sa hauteur).
Si on remplace par

```
glViewport(0, 0, w/2, h/2);
```

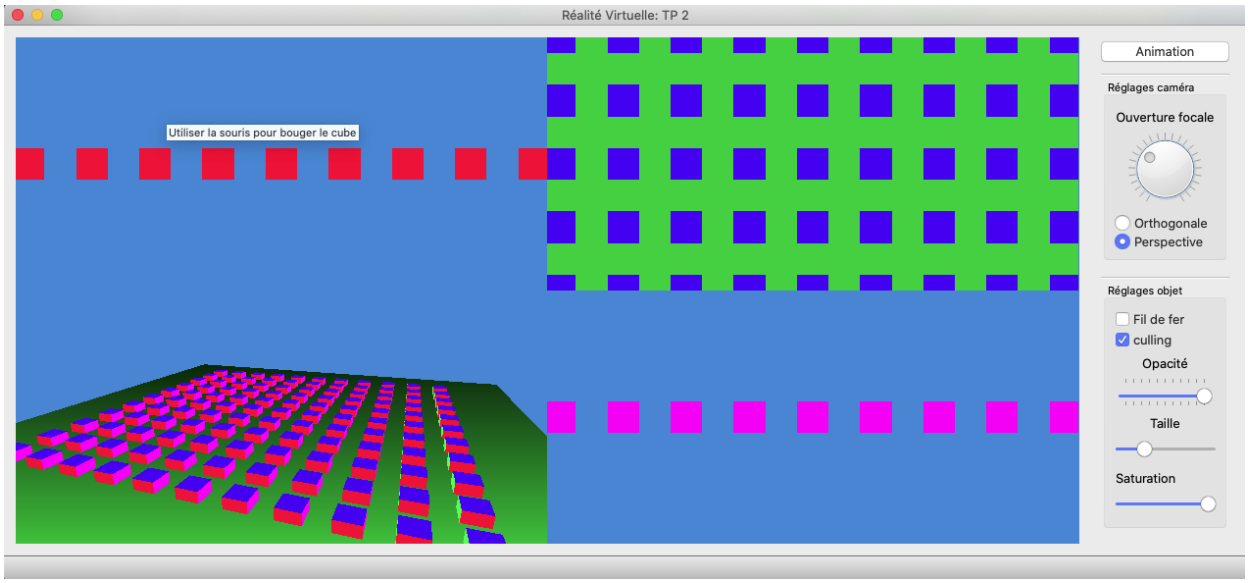
on obtient :

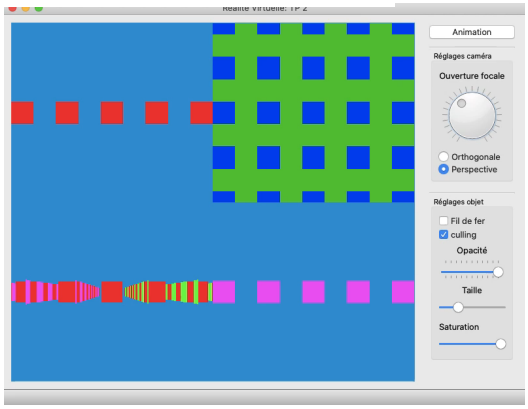


On peut donc dans `glPaint()` changer 4 fois le viewport et répéter chaque fois les commandes de renu pour diviser l'écran en 4 sous-écrans.



Mais l'intérêt du procédé consiste à utiliser 4 caméras différentes, une pour chaque sous-écran, par exemple pour produire une interface de type "dessin technique" avec une vue perspective, une vue orthogonale *de façade*, une vue orthogonale de *élévation* et une vue orthogonale *de plan*.




Animation
Réglages caméra
Ouverture focale
Orthogonale
Perspective
Réglages objet
Fil de fer
culling
Opacité
Taille
Saturation