2) Mise en place de l'espace de travail

Dans un premier temps, vous devez « planter le décor », c'est-à-dire définir votre repère de travail, le positionnement de la caméra, etc.

2.1) Mise en place du système de coordonnées

2.1.1) Affichage du repère dans l'espace

Pour l'ensemble de ce projet, vous adopterez le repère cartésien 3D suivant :

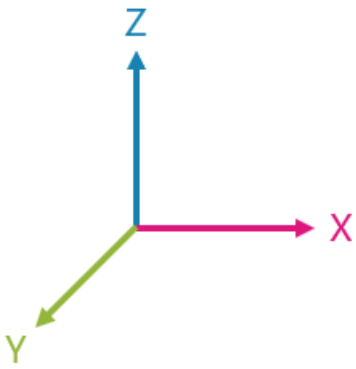

Figure 2 - Repère cartésien du projet

Pour visualiser votre repère, vous devrez créer votre première forme, que l'on appelle généralement un « gizmo » dans les logiciels de modelage 3D.

Cette forme est constituée des 3 lignes du repère, qui matérialisent l'axe des X en Rouge, l'axe des Y en vert et l'axe des Z en bleu. Cette construction s'avèrera utile pour vous aider à vous situer visuellement, lorsque vous changerez de point de vue en bougeant la caméra.

Processing utilise par défaut un repère main gauche avec les Y positifs vers le bas :

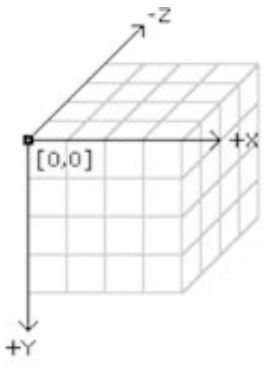

Figure 3 - Repère Processing 3D par défaut

De plus, en 3D, la caméra par défaut ne pointe pas vers l'origine *(0,0,0)* mais en *(width/2,*

height/2, 0), soit au milieu de la fenêtre visible à l'écran[1].

Afin de visualiser le projet en conformité avec le repère choisi *(main gauche avec Z positif vers le haut)*, vous allez déplacer votre caméra à l'aide de l'instruction suivante :

```
// 3D camera (X+ right / Z+ top / Y+ Front)
camera(
  0, 2500, 1000,
  0, 0, 0,
  0, 0, -1
  );
```

Votre « Gizmo » doit désormais s'afficher au centre de l'écran.

La caméra étant placée à une distance de 2500 et une hauteur de 1000 ; vous devez obtenir une ligne rouge dirigée vers la droite (*X+ Right*), une ligne bleue dirigée vers le haut (*Z+ Up*), et une ligne verte dirigée vers vous (*Y+ Front*) en contreplongée :

Figure 4 - Affichage du Gizmo après placement de la caméra

2.1.2) Pré-construction des formes

Au final, votre projet comprendra plusieurs centaines de formes. Afin de préserver de bonnes performances d'affichage, vous allez construire vos formes à l'avance (*c'est ce qu'on appelle le « retained mode » d'OpenGL*).

Pour cela, vous allez définir une variable globale *gizmo* de type *PShape* qui contiendra votre Gizmo.

Puis vous construirez le Gizmo une seule fois dans la fonction *setup* de Processing :

```
// Gizmo
this.gizmo = createShape();
this.gizmo.beginShape(LINES);
this.gizmo.noFill();
```

[1] Les valeurs par défaut sont décrites dans : Processing - méthode camera()

```
this.gizmo.strokeWeight(3.0f);

// Red X
this.gizmo.stroke(0xAAFF3F7F);
// INSÉREZ VOTRE CODE ICI

// Green Y
this.gizmo.stroke(0xAA3FFF7F);
// INSÉREZ VOTRE CODE ICI

// Blue Z
this.gizmo.stroke(0xAA3F7FFF);
// INSÉREZ VOTRE CODE ICI

this.gizmo.endShape();
```

Enfin vous l'afficherez dans la fonction *draw* à l'aide de l'instruction *shape(this.gizmo);*

2.1.3) Création de votre première classe

Afin d'organiser correctement votre projet, toutes vos formes doivent être organisées dans des classes Java séparées.

Dans un nouvel onglet Processing, vous allez créer une première classe appelée « *WorkSpace* ». Cette classe contiendra la variable *PShape* gizmo, ainsi que les méthodes suivantes :

Un constructeur *WorkSpace*, qui sera en charge de créer la *PShape* de votre Gizmo.

Une méthode *update*, qui réalisera l'affichage du Gizmo.

Une méthode *toggle* qui vous permettra d'afficher ou non les formes de cette classe :

```
/**
 * Toggle Grid & Gizmo visibility.
 */
void toggle() {
  this.gizmo.setVisible(!this.gizmo.isVisible());
}
```

Dans le programme principal, vous déclarez une variable globale *WorkSpace workspace* à la place de la *PShape* antérieure.

Puis vous construisez votre forme une seule fois dans la fonction *setup* de Processing en créant un nouvel objet : *this.workspace = new WorkSpace() ;*

Enfin, l'affichage au cours de la fonction *draw* sera désormais réalisé par un appel à la méthode de la classe par *this.workspace.update();*

Vous pouvez maintenant ajouter après la fonction *draw* votre première commande utilisateur, la touche « w » qui permettra d'afficher ou de masquer l'espace de travail à l'écran :

```
void keyPressed() {
  switch (key) {
```

```
  case 'w':
  case 'W':
    // Hide/Show grid & Gizmo
    this.workspace.toggle();
    break;
  }
}
```

2.2) Représentation du sol plan

Dans la classe *WorkSpace*, vous allez matérialiser le sol plan (altitude Z=0) par une nouvelle *PShape* qui représente une grille carrée de *size* x *size* unités *(Nb : dans la suite du projet, une unité vaudra 1 mètre)* centrée sur l'origine (0,0,0), et constituée de 100 x 100 dalles.

Cette seconde *PShape* appelée *grid* sera également implémentée dans la classe *WorkSpace*, de sorte que vous la construirez simultanément à la *PShape* du *gizmo* précédent. La valeur de la variable *size* sera donnée en paramètre lors de la construction de la classe. *(Nb : pour votre projet, une valeur de 25 kms sera suffisante pour matérialiser le sol ainsi que l'horizon, chaque dalle représentera alors un carré de 250 mètres de côté) :*

```
// Grid
this.grid = createShape();
this.grid.beginShape(QUADS);
this.grid.noFill();
this.grid.stroke(0x77836C3D);
this.grid.strokeWeight(0.5f);
// INSÉREZ VOTRE CODE ICI
this.grid.endShape();
```

Les méthodes *update* et *toggle* de la classe seront modifiées pour gérer l'affichage et la visibilité de cette seconde forme.

Pour obtenir un affichage correct de la grille, il est conseillé de paramétrer le niveau d'*antialiasing* le plus élevé possible supporté par votre carte graphique, à l'aide de l'instruction *smooth();* dans la fonction *setup*[2], par exemple :

```
void setup() {

  // Display setup
  fullScreen(P3D);
  smooth(8);
  frameRate(60);

  // Initial drawing
  background(0x40);

  // Prepare local coordinate system grid & gizmo
  this.workspace = new WorkSpace(250*100);

}
```

[2] Voir également : Processing - méthode smooth()

Pour terminer, vous ajouterez 2 lignes « fines » à votre gizmo, pour matérialiser respectivement les couleurs des axes X & Y du repère sur la grille.

Votre plan de travail est désormais prêt à l'emploi :

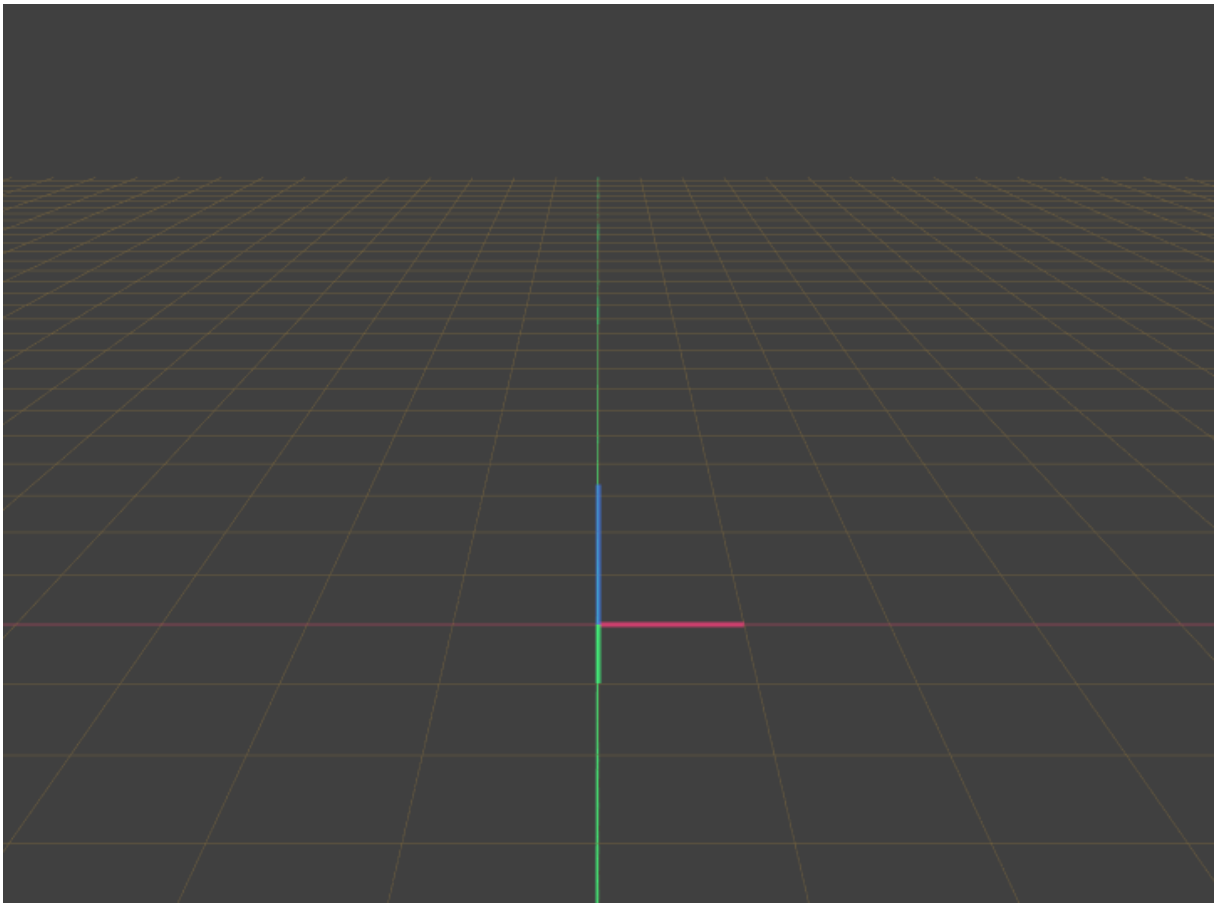

Figure 5- Grille de plan de sol & repères 3D

2.3) Implémentation d'une caméra mobile

2.3.1) Calcul de l'emplacement de la caméra

À ce stade, le plan de travail visualisé est un plan fixe. Cette étape consiste à implémenter une caméra mobile, à même de pivoter autour de l'origine du plan de travail.

Pour cela, vous allez créer une classe *Camera*.

La caméra est positionnée à la surface d'une sphère d'origine *(0,0,0)* et de rayon *radius* variable.

La position est définie en coordonnées polaires par 2 angles : la longitude et la colatitude *(nb : vous pouvez au choix utiliser la latitude qui vaut π/2 – colatitude)* telles que définies

habituellement[3].

Lors de la construction d'un objet Camera, vous déterminerez des valeurs de variables *longitude*, *colatitude* et *radius* par défaut, de manière à obtenir un angle de vue similaire à votre plan fixe antérieur.

En fonction des angles et du rayon, vous convertirez les coordonnées polaires en coordonnées cartésiennes *x*, *y* & *z*, qui seront utilisées dans la méthode *update();* pour positionner la caméra sur la sphère virtuelle :

```
// 3D camera (X+ right / Z+ top / Y+ Front)
camera(
  this.x, -this.y, this.z,
  0, 0, 0,
  0, 0, -1
  );
```

2.3.2) Calcul des mouvements de caméra

Pour rendre la caméra mobile, votre classe implémentera les méthodes *adjustRadius(float offset);* pour ajuster le zoom (en mètres), *adjustLongitude(float delta);* pour se déplacer vers la droite ou la gauche et *adjustColatitude(float delta);* pour définir l'angle de vue plongeante (en radians).

Vous veillerez à limiter les valeurs minimales et maximales des réglages possibles, afin de conserver en permanence une bonne visibilité de l'espace de travail, par exemple :

- Rayon entre *width * 0.5* et *width * 3.0*
- Longitude entre *-3*π/2* et *+π/2* (*nb : le zéro « visuel » est sur l'axe Y+, soit à* -π/2 *)*
- Colatitude entre *epsilon*[4] et *+π/2* (*on ne regarde pas « par dessous »).*

2.3.3) Pilotage de la caméra avec le clavier

Avec ces fonctions, complétez l'exemple ci-après pour piloter la caméra à l'aide des touches de curseur, ainsi que des touches « Plus » ou « Moins » du clavier :

```
void keyPressed() {

  if (key == CODED) {
    switch (keyCode) {
    case UP:
      // INSÉREZ VOTRE CODE ICI
      break;
    case DOWN:
      // INSÉREZ VOTRE CODE ICI
      break;
```

[3] Coordonnées sphériques (Wikipédia)
[4] Pour le type *Float*, on peut prendre *epsilon* = 10^{-6}

```
    case LEFT:
      // INSÉREZ VOTRE CODE ICI
     break;
    case RIGHT:
      // INSÉREZ VOTRE CODE ICI
      break;
    }
  } else {
    switch (key) {
    case '+':
      // INSÉREZ VOTRE CODE ICI
      break;
    case '-':
      // INSÉREZ VOTRE CODE ICI
      break;
    }
  }
}
```

Pour autoriser la répétition automatique des touches afin de faciliter les mouvements de la caméra, vous devrez également ajouter ce réglage à la fin de la fonction *setup* :

```
  // Make camera move easier
  hint(ENABLE_KEY_REPEAT);
```

2.3.4) Pilotage de la caméra avec la souris

Si votre souris dispose d'une molette centrale, vous pourrez également piloter votre caméra par exemple comme ceci :

```
void mouseWheel(MouseEvent event) {
  float ec = event.getCount();
  // INSÉREZ VOTRE CODE ICI
}

void mouseDragged() {
  if (mouseButton == CENTER) {
    // Camera Horizontal
    float dx = mouseX - pmouseX;
    // INSÉREZ VOTRE CODE ICI
    // Camera Vertical
    float dy = mouseY - pmouseY;
    // INSÉREZ VOTRE CODE ICI
  }
}
```

2.4) Afficher des informations dynamiques à l'écran

Les formes que vous avez créées s'affichent dans un espace à 3 dimensions.

Afin d'afficher du texte à l'écran, vous utiliserez l'espace par défaut de Processing (*il s'agit du plan z=0 avant modification de la caméra*).

Pour cela vous allez ajouter la classe *Hud* (*Head up display*[5]) dans un nouvel onglet :

[5] Voir également : Wikipédia – Affichage tête haute

```
class Hud {

  private PMatrix3D hud;

  Hud() {
    // Should be constructed just after P3D size() or fullScreen()
    this.hud = g.getMatrix((PMatrix3D) null);
  }

 private void begin() {
    g.noLights();
    g.pushMatrix();
    g.hint(PConstants.DISABLE_DEPTH_TEST);
    g.resetMatrix();
    g.applyMatrix(this.hud);
  }

 private void end() {
    g.hint(PConstants.ENABLE_DEPTH_TEST);
    g.popMatrix();
  }
}
```

Cette classe va vous permettre de mémoriser la matrice de transformation initiale (*avant repositionnement de la caméra*), d'inhiber les éclairages avant tout affichage à l'écran, et d'inhiber le test de profondeur du pipeline *OpenGL* de sorte que les informations soient toujours visibles en avant plan.

Pour l'utiliser, vous déclarerez une variable globale *Hud hud;* que vous initialiserez juste après avoir créé le contexte graphique *OpenGL* de *Processing* via l'instruction *size(…,P3D);* ou *fullScreen(…,P3D);* :

```
  // Setup Head Up Display
  this.hud = new Hud();
```

Ajoutez à la classe la méthode suivante, qui permet d'afficher dynamiquement le nombre de *frames par secondes*[6] supportées par votre carte graphique, en bas à gauche de votre fenêtre.

```
  private void displayFPS() {
    // Bottom left area
    noStroke();
    fill(96);
    rectMode(CORNER);
    rect(10, height-30, 60, 20, 5, 5, 5, 5);
    // Value
    fill(0xF0);
    textMode(SHAPE);
    textSize(14);
    textAlign(CENTER, CENTER);
    text(String.valueOf((int)frameRate) + " fps", 40, height-20);
  }
```

Créez une méthode publique *update* qui sera appelée à la fin de la fonction *draw*, qui enchaînera

[6] L'affichage est considéré « *fluide* » tant que ce nombre reste supérieur à 30 FPS.

les méthodes *this.begin()*, *this.displayFPS()* et *this.end()*.

Enfin, ajoutez une autre méthode *displayCamera(Camera camera);* qui permettra d'afficher dynamiquement la position de la caméra en haut à gauche de l'écran :

Figure 6 - Affichage d'informations dynamiques

Félicitations, vous disposez d'un plan de travail 3D à même de recevoir vos futures constructions !

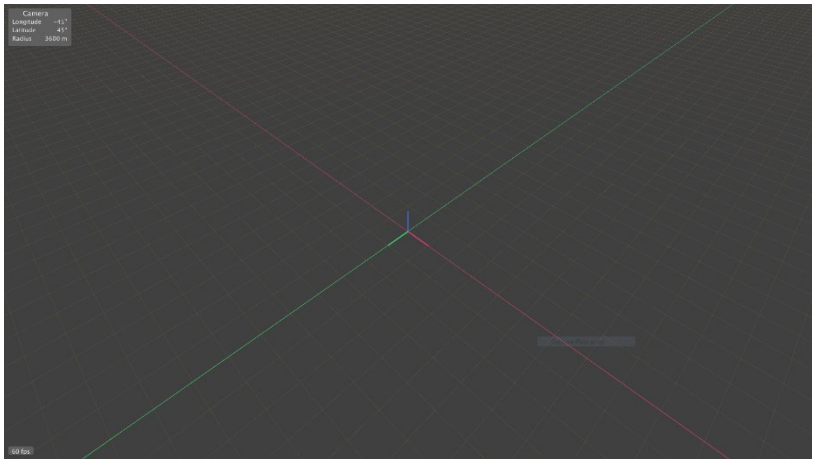

Figure 7 - Espace de travail 3D finalisé