Notes TD processing

CHASSAGNOL Rémi

2021-05-01

Les inputs

Toutes les fonctions sont appelée automatiquement au moment de l'évènement.

Le clavier

Ces fonction mettent à jour les variables **key** et **keycode**.

```
void setup() {}
void draw() {}

void keyPressed() {
  switch(key) {
  case 'a': // ...
  case 'b': // ...
  // ...
  default: break;
  }

  switch(keyCode) {
  case LEFT: // ...
  // ...
  default: break;
  }
}

void keyReleased() {
  // ...
}

void keyTyped() {
  // ...
}
```

La souris

```
void mousePressed() {
  // ...
}

void mouseReleased() {
  // ...
}
```

```
// Met à jour pmouseX/Y et mouseX/Y:
void mouseDragged() {
  int anciennePositionX = pmouseX;
  int anciennePositionY = pmousey;
  int positionActuelleX = mouseX;
  int positionActuelleY = mouseY;
}

void mouseWheel(MouseEvent ev) {
  totalMouseWheel += ev.getCount();
}
```

Webcam

Processing video

```
import processing.video.*;
Capture webCam;
String[] cameras;    // Liste textuelle des webCams disponibles

void setup() {
  size(640, 480);
  surface.setTitle("Exemple 4a - WebCam Capture Standard - E. Mesnard / ISIMA");
  cameras = Capture.list();
  if (0 == cameras.length) {
    println("Pas de Webcam sur cet ordinateur !"); exit();
  } else {
    webCam = new Capture(this, 640, 480);
    webCam.start();
  }
}

void draw() {
  if (webCam.available() == true) {
    webCam.read();
    image(webCam, 0, 0);
  }
}
```

- `Capture.list()`: `String[]` -> liste des cameras disponibles
  - permet de tester si il y a des caméras (taille du tableau)
- `webCam = new Capture(this, 640, 480)`: init objet caméra
- `webCam.start()`: démarre la capture
- `webCam.available()`: **true** si frame dispo
- `webCam.read()`: lecture du flux vidéo

A noter que l'on peut aussi copier l'image récupéré dans le flux vidéo de la façon suivante:

```
PImage img;
// ...
void draw() {
  if (webCam.available() == true) {
    webCam.read();
    pImg.copy(webCam, 0, 0, 640, 480, 0, 0, 640, 480);
```

```
    image(pImg, 0, 0);
  }
}
```

Bibliothèque sarxos

```
import com.github.sarxos.webcam.*;
import java.awt.image.BufferedImage;
import java.awt.Dimension;
import java.util.List;

Webcam webCam;
List<Webcam> cameras ;
BufferedImage BImgWebCam;
PImage PImgWebCam;

void setup() {
  size(640, 480);
  surface.setTitle("Exemple 4b - WebCam Sarxos - E. Mesnard / ISIMA");
  colorMode(RGB, 255);
  PImgWebCam = createImage(640, 480, ARGB);
  cameras = Webcam.getWebcams();
  if (cameras.isEmpty()) {
    println("Pas de Webcam sur cet ordinateur !"); exit();
  } else {
    webCam = Webcam.getDefault();
    Dimension ResolWebCam = new Dimension(640, 480);
    webCam.setViewSize(ResolWebCam);
    webCam.open();
  }
}

void draw() {
  if (webCam.isImageNew() && webCam.isOpen()) {
    BImgWebCam = webCam.getImage();
    BImgWebCam.getRGB(0, 0, 640, 480, PImgWebCam.pixels, 0, 640);
    PImgWebCam.updatePixels();
    image(PImgWebCam, 0, 0);
  }
}
```

Parcours de pixels

Exemple en utilisant processing video:

```
  if (webCam.available() == true) {
    webCam.read();

    // Analyse de l'image
    for (yy = 0; yy < heightCapture; yy++) { // abscisse yy
      yPos = yy * widthCapture;
      for (xx = 0; xx < widthCapture; xx++) { // ordonnees xx
        i = xx + yPos;

       // recuperation couleur
```

```
        currColor = webCam.pixels[i];
        teinte = hue(currColor);

        // Traitement / analyse
    }
    image(webCam, 0, 0);
  }
```

RA: les marqueurs

Pattern markers

```
import processing.video.*;
import jp.nyatla.nyar4psg.*;

MultiMarker sceneMM;

void setup() {
  /* configuration classique (size, ...) */

  /* démarage/initialisation de la webcam */

  // Declaration de la scene de recherche avec parametres par defaut :
  // calibration de camera et systeme de coordonnees
  sceneMM = new MultiMarker(this, widthCapture, heightCapture,
                            "camera_para.dat",
                            NyAR4PsgConfig.CONFIG_PSG);

  // Declaration du marqueur a rechercher, avec sa dimension en mm
  sceneMM.addARMarker("Marqueur_ISIMA.patt", 80);
}

void draw() {
  if (webCam.available() == true) {
    webCam.read();
    sceneMM.detect(webCam);
    webCam.updatePixels();
    background(0);
    image(webCam, 0, 0);

    // Incrustation de l'image virtuelle si marqueur trouve
    if (sceneMM.isExist(0)) { // Marqueur ISIMA
      sceneMM.beginTransform(0);

      // Modification de l'image !!!

      sceneMM.endTransform();
    }

  }
}
```

`sceneMM.isExist(x)`: test si le marqueur numéro **x** est dans l'image. Le numéro du marqueur dépend de l'ordre de chargement des fichier **.dat** dans le setup.

Important: `sceneMM.beginTransform(0)` change le plan, en l'appelant, vous n'êtes plus dans le plan de l'image mais dans le plan 3D du marqueur (c'est le système de double coordonnées de ARToolkit).

NFT markers

```
MultiNft sceneNFT;

void setup() {
  /* configuration classique (size, ...) */

  /* démarage/initialisation de la webcam */

  // Declaration de la scene de recherche avec parametres par defaut :
  // calibration de camera et systeme de coordonnees
  sceneNFT = new MultiNft(this, widthCapture, heightCapture,
                          "camera_para.dat",
                          NyAR4PsgConfig.CONFIG_PSG);

  // Declaration du marqueur a rechercher, avec sa dimension en mm
  sceneNFT.addNftTarget("Image_ISIMA", 80);
}


void draw() {
  if (webCam.available() == true) {

    webCam.read();
    sceneNFT.detect(webCam);
    webCam.updatePixels();
    background(0);
    image(webCam, 0, 0);

    if (sceneNFT.isExist(0)) {
      sceneNFT.beginTransform(0);

      // Transformation de l'image !!!

      sceneNFT.endTransform();
    }
  }

}
```

Manipulation des textures

```
PShape Cube_Simple;
PShape Cube_ISIMA;
PImage Texture_Cube;

void setup() {
  /* settings de bases */

  Cube_Simple = createShape(BOX, width/3.5);
  // Shapes possibles : ELLIPSE, RECT, ARC, TRIANGLE, SPHERE, BOX, QUAD, LINE
```

```
  // Creation d'un cube avec une texture
  Texture_Cube = loadImage("ISIMA.jpg");
  // Les coordonnees de la texture sont normalisees de 0 a 1 en U et en V:
  textureMode(NORMAL);
  Cube_ISIMA = CreerCubeTexture(Texture_Cube, width/3.5);

}

void draw() {
  background(0);

  // Positionnement des cubes et tracages...
  translate(width/4.0, height/2.0, -100);
  shape(Cube_Simple);
  translate(width/2.0, 0, 0); // Attention : les deplacements sont cumulatifs
  shape(Cube_ISIMA);
}

// Fonction de creation d'un cube, avec une taille et une texture
PShape CreerCubeTexture(PImage tex, float taille) {
  PShape formeCube;
  formeCube = createShape();
  formeCube.beginShape(QUADS);
  // Les geometries possibles sont : POINTS, LINES, TRIANGLES, TRIANGLE_FAN,
  //                                 TRIANGLE_STRIP, QUADS, QUAD_STRIP
  formeCube.noFill();
  formeCube.noStroke();

  formeCube.texture(tex); // A definir des le depart

  // Declaration des listes de points pour les faces

  // vertex(x, y, z, u, v) : (x,y,z) = coordonnees du point
  //                         (u,v) = coordonnees de la texture

  // Face +Z = Avant
  formeCube.vertex(-1,-1, 1, 0, 0);
  formeCube.vertex( 1,-1, 1, 1, 0);
  formeCube.vertex( 1, 1, 1, 1, 1);
  formeCube.vertex(-1, 1, 1, 0, 1);

  // Face -Z = Arriere
  formeCube.vertex( 1,-1,-1, 0, 0);
  formeCube.vertex(-1,-1,-1, 1, 0);
  formeCube.vertex(-1, 1,-1, 1, 1);
  formeCube.vertex( 1, 1,-1, 0, 1);

  // Face +Y = Dessous
  formeCube.vertex(-1, 1, 1, 0, 0);
  formeCube.vertex( 1, 1, 1, 1, 0);
  formeCube.vertex( 1, 1,-1, 1, 1);
  formeCube.vertex(-1, 1,-1, 0, 1);
```

```
  // Face -Y = Dessus
  formeCube.vertex(-1,-1,-1, 0, 0);
  formeCube.vertex( 1,-1,-1, 1, 0);
  formeCube.vertex( 1,-1, 1, 1, 1);
  formeCube.vertex(-1,-1, 1, 0, 1);

   // Face +X = Droite
  formeCube.vertex( 1,-1, 1, 0, 0);
  formeCube.vertex( 1,-1,-1, 1, 0);
  formeCube.vertex( 1, 1,-1, 1, 1);
  formeCube.vertex( 1, 1, 1, 0, 1);

  // Face -X = Gauche
  formeCube.vertex(-1,-1,-1, 0, 0);
  formeCube.vertex(-1,-1, 1, 1, 0);
  formeCube.vertex(-1, 1, 1, 1, 1);
  formeCube.vertex(-1, 1,-1, 0, 1);

  formeCube.endShape();

  // Mise a l'echelle
  formeCube.scale(taille/2.0);

  // Orientation initiale possible
  // formeCube.rotateX(PI/6);
  // formeCube.rotateY(PI/4);
  return formeCube;
}
```

Les OBJ

```
PShape objetOBJ;

void setup() {
  float facteurEchelle;

  /* Settings */

  // Chargement de l'objet utilise
  objetOBJ = loadShape("dragon.obj");

  nombreVertex = objetOBJ.getVertexCount();
  nombreEnfant = objetOBJ.getChildCount();

  // Determination de la taille effective de l'objet
  dimOBJ_X = objetOBJ.getWidth();
  dimOBJ_Y = objetOBJ.getHeight();
  dimOBJ_Z = objetOBJ.getDepth();

  facteurEchelle = min(width/(1.6*dimOBJ_X), height/(1.6*dimOBJ_Y), width/(1.6*dimOBJ_Z));
  objetOBJ.scale(facteurEchelle);

  objetOBJ.translate(decX, decY, decZ);
```

```
}

void draw() {
  background(0);
  translate(width/2, height/2, 0);

  // Affichage des objets
  lights();
  shape(objetOBJ);
}
```

Import de fichiers MD2

```
import MD2Importer.*;

MD2_Loader chargeurOiseau;
MD2_Model Oiseau;
MD2_ModelState[] animation;


void setup() {

  /* Settings */

  // Creation du chargeur pour acceder au fichier MD2
  chargeurOiseau = new MD2_Loader(this);

  // Chargement effectif du modele texture
  Oiseau = chargeurOiseau.loadModel("Oiseau.md2", "Oiseau.jpg");

  if (Oiseau == null) {
    println("Probleme de chargement de ce fichier MD2");
    exit();
  } else {
    chargeurOiseau = null; // Liberation de l'espace memoire
    // Recuperation de toutes les animations disponibles
    animation = Oiseau.getModelStates();
    Oiseau.centreModel();
    // Mise a l'echelle souhaitee de l'objet MD2
    Vector3 dimensionOiseau = new Vector3();
    dimensionOiseau = Oiseau.getModSize();
    Oiseau.scaleModel((0.8*height)/dimensionOiseau.y);
    // Oiseau.scaleModel(0.1);

    // Selection de l'animation (par exemple, la derniere !)
    Oiseau.setState(animation.length-1);

    // Calcul (eventuel) d'une vitesse d'evolution dans les animations
    pourcentAnimation = constrain(4/(frameRate + 0.01), 0.01, 1);
  }
}

void draw() {
```

```
  background(fondEcran);
  lights();
  // Positionnement et orientation de l'objet dans la scene
  translate(width/2,height/2);
  rotateX(radian(70));
  rotateZ(radian(15));
  // Trace de l'objet en faisant evoluer son animation
  Oiseau.update(pourcentAnimation);
  Oiseau.render();
}
```

Quelques méthodes pour debug:

- `chargeurOiseau.displayHeader()`: infos sur le modèle chargé
- `chargeurOiseau.displayGLcommands()`: affichage des commandes **opengl** dispo
- `chargeurOiseau.displayModelStates()`: noms + frames des animations
- `chargeurOiseau.displayFrameNames()`: noms des frames

Les shaders

Il faut activer la fonction `lights()` dans le setup.

Rapel du cours

Réflexion ambiante + réflexion diffuse + réflexion spéculaire = modèle d'illumination de phong.

Réflexion ambiante

- coloriage uniforme
- on distingue la silhouette de l'objet éclairé
- pas d'ombres ou de reliefs

Réflexion diffuse

- diffucion uniforme dans l'espace
- on distingue les reliefs de l'objet éclairé
- dépend de la position de la source mais **pas** de l'observateur

Réflexion spéculaire

- diffucion non uniforme
- dépend du point de vue de l'observateur
- lumière = taches brillante sur l'objet
- la couleur de la tache dépend de la couleur de la lumière et non pas de celle de l'objet

Les types d'éclairages

Processing

Important: dans les exemples ci-dessous, on se place dans le cas ou le mode de couleur choisi est le mode `RGB`.

```
ambientLight(r, g, b);
ambientLight(r, g, b, x, y, z)

// nx, ny, et nz sont compris entre -1 et 1 (c'est une direction et non pas des coordonnees)
directionalLight(r, g, b, nx, ny, nz);
```

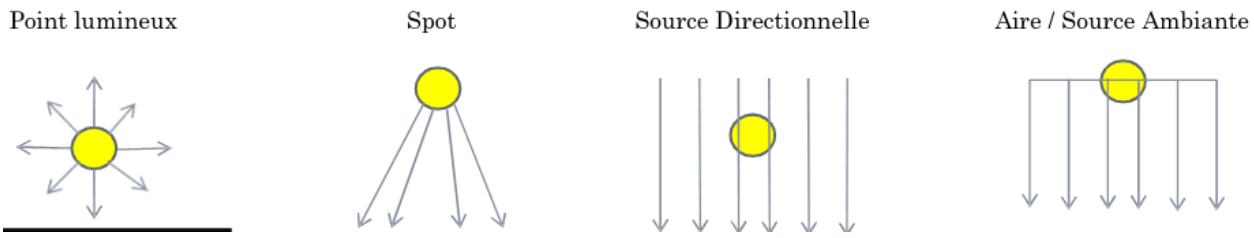

Figure 1: les types d'éclairages

Les fonctions ci-dessous possède trois variantes, qui sont les même que celles de `ambient`:

```
ambient(rgb);
ambient(gray);
ambient(v1, v2, v3);

emissive();

specular();
```

Lorsqu'on utilise les fonctions ci-dessus, on peut ajout un effet de **gloss** en utilisant la fonction `shininess(x)` qui prend en paramètre `x` compris entre 0 et 1.

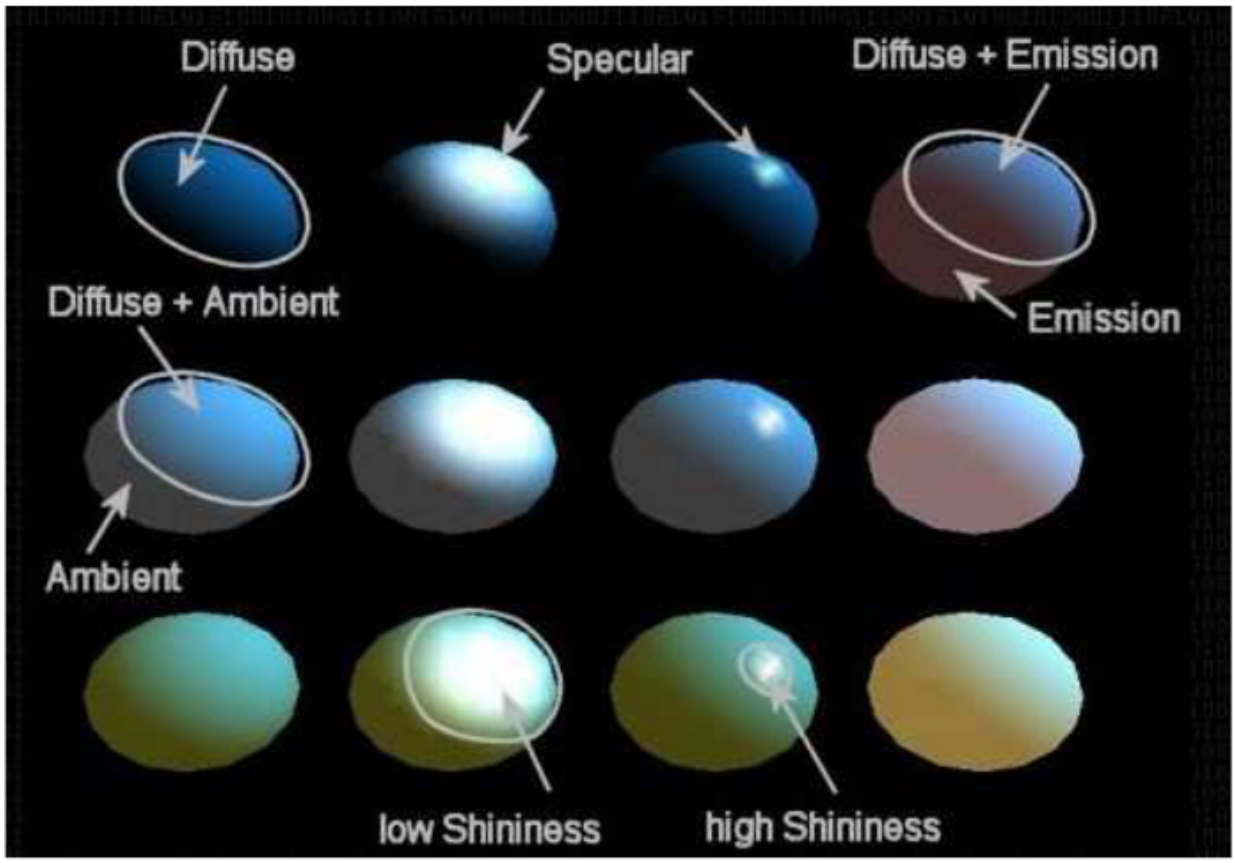

Figure 2: Les différences en image

Android

Important: ne pas oublier de passer processing en mode android.

Bases et intéractions

```
import ketai.ui.*;

KetaiGesture gesture;

void setup() {
  fullScreen();
  orientation(LANDSCAPE);
  // Mise en place de la gestion des mouvements sur ecran
  gesture = new KetaiGesture(this);
  imageMode(CENTER);

  // ...
}

void draw() {
  /* ... */
}

/* Fonctions d'intéractions avec le téléphone */

void onDoubleTap(float x, float y) {
  /* ... */
}

void onTap(float x, float y) {
  /* ... */
}

void onLongPress(float x, float y) {
  /* ... */
}

void onFlick( float x, float y, float px, float py, float v) {
  /* ... */
}

void onPinch(float x, float y, float d) {
  /* ... */
}

void onRotate(float x, float y, float angle) {
  /* ... */
}
```

Les fonctions d'intéraction sont appelées **automatiquement**, on y utilise les variables en paramètres pour modifier les variables globales.

Figure 3: Mouvement correspondant aux fonctions ci dessus

Caméras et texte

```
import ketai.camera.*;

KetaiCamera cam;
PImage ImageCourante;

void setup() {
  fullScreen(P2D);
  orientation(LANDSCAPE);
  // Gestion du mode d'affichage
  imageMode(CENTER);
  textAlign(CENTER, CENTER);
  textSize(displayDensity * 25);
  // Ouverture du systeme de gestion des cameras a 24 fps
  cam = new KetaiCamera(this, 1280, 720, 24);
}

void draw() {
  if (cam != null && cam.isStarted()) {
    image(cam, width/2, height/2, width, height);
  } else {
    background(#D16363);
    // Affichage de texte sur l'écran:
    text("!! Camera eteinte !!", width/2, height/2);
  }
}

// Fonction appelée automatiquement
void onCameraPreviewEvent() {
  cam.read(); // Lecture d'une image
}
```

L'immersif

```
import processing.vr.*;

// The Setup script is only run 1 time at the initialization of the program
void setup() {
  // Set the screen to STEREO whihc means the side by side view used in the google cardboard
  fullScreen(STEREO);
  /* Chargement des OBJ + rescale */
}
```

```
void draw() {
  background(0);

  // Include lights in the scene
  pointLight(180, 180, 180, 0, 600, 0);
  directionalLight(100, 100, 100, -1, 0, 0);

  /* Dessins de toutes les textures une par une
   * au bonnes coordonnees (cf: translate)
   * avec la bonne rotation (cf: rotate)
   *
   * Si on fait un pushMatrix(), les translate() sont cumulatifs jusqu'au prochain
   * popMatrix(). On commence par faire un pushMatrix() de manière à pouvoir revenir
   * dans le plan de départ un utilisant popMatrix() (on sauvegarde le plan).
   */

  // Exemple:
  pushMatrix();
  translate(x, y, z);
  rotateY(anle);
  shape(OBJ);
  popMatrix();
}
```

La Kinect

RGB et Depth

```
import edu.ufl.digitalworlds.j4k.*;

PKinect kinect;
byte[] colorMap;
short[] depthMap;

void setup() {
  /* Settings de bases */

  // Initialisation Objet Kinect
  kinect = new PKinect(this);

  // Ouverture des flux "COLOR" et "DEPTH"
  if (kinect.start(PKinect.DEPTH|PKinect.COLOR) == false) {
    exit();
    return;
  } else if (kinect.isInitialized()) {
    // récupération des tailles des images RGB et Depth
    colorW = kinect.getColorWidth();
    colorH = kinect.getColorHeight();
    depthW = kinect.getDepthWidth();
    depthH = kinect.getDepthHeight();
  } else {
    exit();
    return;
  }
```

```
  // Creation des objets Color et Depth
  colorMap = new byte[colorW*colorH*4];
  colorImage = createImage(colorW, colorH, RGB);
  depthMap = new short[depthW*depthH];
  depthImage = createImage(depthW, depthH, RGB);
}

void draw() {
  int i, j; // Indices des boucles
  int ValZ; // Composante distance Z pour le point considere
  int mini, maxi; // Valeur Z mini non nulle et maximum

  // Recuperation d'eventuelles donnees sur la kinect...
  colorMap = kinect.getColorFrame(); // Flux "COLOR"
  depthMap = kinect.getDepthFrame(); // Flux "DEPTH"

  // Traitement du Flux "COLOR"

  // Traitement du Flux "DEPTH"

  // Effacement de la fenetre
  background(0);
  // Affichage des deux images, cote a cote
  image(depthImage, 0, 0, width/2, height);
  image(colorImage, width/2, 0, width/2, height);

}
```

Traitement de l'image couleur

Le flux doit être convertit en tableau de pixels.

```
  if (colorMap!=null) {
    // Conversion du tableau en une image en couleur
    colorImage.loadPixels();
    j = 0;
    for (i = 0; i < colorMap.length; i+=4) {
      colorImage.pixels[j] = (colorMap[i+2]&0x0000FF)<<16 |
        (colorMap[i+1]&0x0000FF)<<8  |
        (colorMap[i]&0x0000FF);
      j++;
    }
    colorImage.updatePixels();
  }
```

Traitement de l'image profondeur

```
  if (depthMap!=null) {
    // Conversion du tableau en une image de profondeur
    depthImage.loadPixels();
    for (i = 0; i < depthH*depthW; i++) {
      if (depthMap[i] == 0) {
        // hors champs ou trop proche
        depthImage.pixels[i] = 0;
```

```
      } else {
        ValZ = (int) map((float)depthMap[i], mini, maxi, 255, 0);
        depthImage.pixels[i] = color(ValZ, ValZ, ValZ);
      }
    }
    depthImage.updatePixels();
  }
```

Squelette

```
import edu.ufl.digitalworlds.j4k.*;

PKinect kinect;
Skeleton[] s;
int sMax;

void setup() {
  /* Settings de bases */

  // Initialisation Objet Kinect
  kinect = new PKinect(this);

  // Ouverture du flux "SKELETON"
  if (kinect.start(PKinect.SKELETON) == false) {
    exit();
    return;
  } else if (kinect.isInitialized()) {
    sMax = kinect.getSkeletonCountLimit();
  } else {
    exit();
    return;
  }
}

void draw() {
  background(0);

  // Recuperation d'eventuelles donnees sur la kinect...
  s = kinect.getSkeletons();

  // Traitement du Flux "Skeletons"
  for (int i = 0; i < sMax; i++) {
    if (s[i]!=null) {
      if (s[i].isTracked()==true) {
        traceSquelette(i);
      }
    }
  }
}
```

Traçage du squelette

```
void traceSquelette(int userId) {
  // Tete
```

```
  dessinMembre(userId, Skeleton.HEAD, Skeleton.NECK);

  // Bras Gauche
  dessinMembre(userId, Skeleton.NECK, Skeleton.SHOULDER_LEFT);
  dessinMembre(userId, Skeleton.SHOULDER_LEFT, Skeleton.ELBOW_LEFT);
  dessinMembre(userId, Skeleton.ELBOW_LEFT, Skeleton.WRIST_LEFT);
  dessinMembre(userId, Skeleton.WRIST_LEFT, Skeleton.HAND_LEFT);

  // ...
}

void dessinMembre(int userId, int jointType1, int jointType2) {
  int[] jointPos1; // Coordonnees des membres
  int[] jointPos2;

  // Verification de la presence effective du membre
  if ( (s[userId].isJointTracked(jointType1)==true) &&
    (s[userId].isJointTracked(jointType2)==true) ) {

    // Recuperation des coordonnees 2D, proportionnelles a la taille de la fenetre
    jointPos1 = s[userId].get2DJoint(jointType1, width, height);
    jointPos2 = s[userId].get2DJoint(jointType2, width, height);

    // Trace du trait
    line(jointPos1[0], jointPos1[1], jointPos2[0], jointPos2[1]);
  }
}
```

Détection de la main droite du joueur 0

Dans ce cas, on suppose que la kinect ne détecte **qu'un joueur** et le met dans le **case 0** du tableau, cependant, ça n'est pas forcément le cas et il est **fortement** recomendé de parcourir les cases du tableau `s` pour trouver le premier joueur traqué par la caméra (ou le plus proche si on a la `depthMap`).

```
void draw() {
  background(0);

  // Recuperation d'eventuelles donnees sur la kinect...
  s = kinect.getSkeletons();

  // Récupération des coordonnees de la main droite du joueur numéro 0:
  int[] rightHandPos = s[0].get2DJoint(Skeleton.HAND_RIGHT, width, height);

  // Tracage d'un ellipse sur ces coordonnees:
  ellipse(rightHandPos[0], rightHandPos[1], 20, 20);
}
```

Les articulations

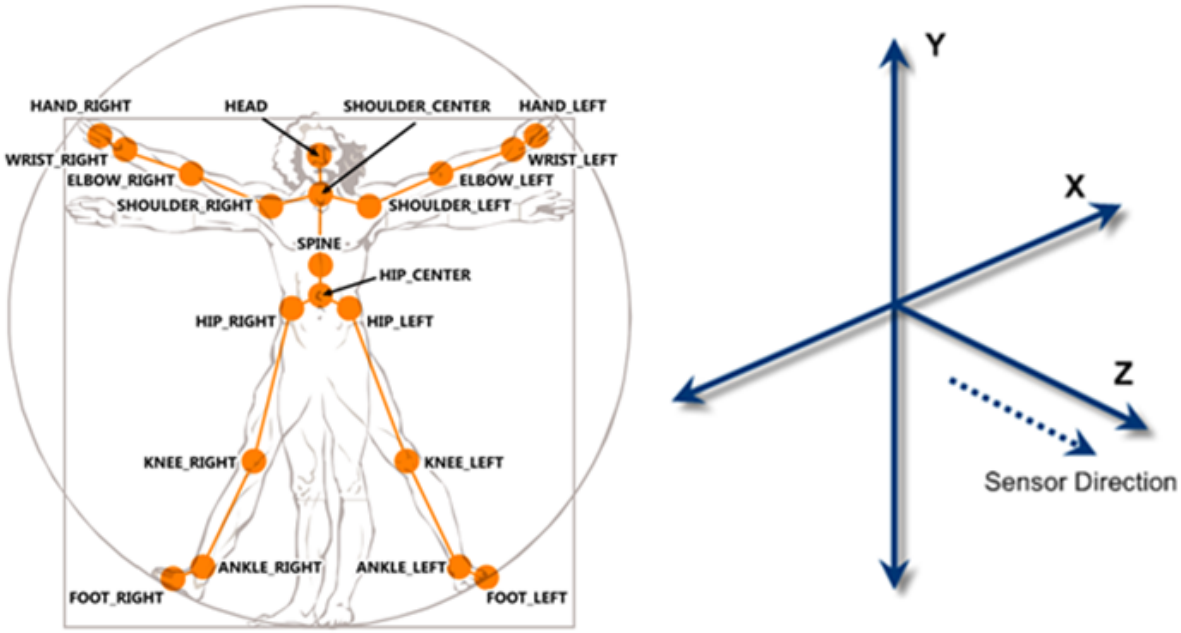

Figure 4: Les articulations sur le squelette