Práctica 6 y 7 :
Procesamiento de imagen y video, y síntesis de sonido

David Alberto Medina Medina
Dr. Modesto Fernando Castrillon Santana

6 de abril de 2019

Índice general

Introducción

Processing es un es un IDE *opensource* que utiliza *Java* como lenguaje de programación. Este proyecto está desarrollado y mantenido por la *Processing Foundation* que sirve como soporte de aprendizaje para instruir a estudiantes de todo el mundo en el mundo de la codificación dentro del contexto de las artes visuales.

El objetivo de esta experiencia consite en desarrollar una aplicación de realidad aumentada desde la cual se puede interactuar con objetos virtuales en un espacio tridimensional y generar audio a partir de ellos. El modelo de una guitarra eléctrica es elegido en un intento de ilustrar esta funcionalidad.

En la escena se muestra la estructura de un esqueleto humano resultado de un algoritmo de seguimiento del cuerpo que procesa la *Kinect v1.8* obteniendo como resultado la información de los puntos de articulación del mismo (DART, s.f.).

La *Kinect* solo ofrece la posición de cada punto de articulación en el plano XY por lo que es necesario analizar la información que de los sensores IR si queremos conocer la profundidad de cada uno de los puntos de articulación que conforman el esqueleto.

Método y materiales

2.1. Materiales

El desarrollo de este proyecto se ha llevado a cabo utilizando el IDE de desarrollo de aplicaciones *Java* de *JetBrains*, *IntelliJ*, y las siguientes herramientas:

- Librería *Processing* (Processing, s.f.).
- Librería *Kinect4WinSDK* (Chung, s.f.).
- Librería *Sound* (Foundation, s.f.).
- Modelo *OBJ* de una guitarra eléctrica (stone777, s.f.).
- *Kinect v1.8* (Microsoft, s.f.)

2.2. Método

Las siguientes clases que se definenen en este documente se organizan en los siguientes paquetes:

1. main
2. kinect
3. soundFX
4. object
5. algorithms

2.2.1. Paquete `main`

Clase `KinectMe`

Para poder utilizar las primitivas de *Processing* esta clase debe heredar de `PApplet`. Para iniciar una aplicación de *Processing*, el método estático `main()` debe llamar a la primitiva `PApplet.main()` para indicar el nombre de la clase principal desde la cual se llama y sobrescriben los métodos primitivos: `settings()`, `setup()` y `draw()`.

```
1    public static void main(String[] args) {
2        PApplet.main("main.KinectMe");
3    }
```

El método primitivo `settings()` establece el tamaño de la pantalla llamando al método `size()` y pasándole la marca del renderizador de gráficos 3D (`P3D`).

```
    @Override
    public void settings() {
        super.settings();
        size(640, 480, P3D);
    }
```

El método `setup()` inicializa un objeto de la clase `Kinect` –además, de establecer el radio de acción de las manos– que se utiliza como controlador de la misma. Además, se llama al método `spawnGuitar()` responsable del renderizado y puesta en escena de una guitarra que será el objeto tridimensional por medio del cual el usuario puede interactuar. El método `createFloor()` permite visualizar una rejilla que será utilizada como suelo de la escena.

```
    @Override
    public void setup() {
        super.setup();
        smooth();
        stroke(255);

        kinect = new Kinect(this, null, null, null);
        kinect.setHandRadius(10);

        spawnGuitar();

        createFloor();
    }
```

El método privado `spawnGuitar()` carga el modelo 3D de la guitarra y le asigna una posición en la escena. El método `doDrawInteractionArea()` permite visualizar los volúmenes de interacción asociados a la guitarra si ajustamos el parámetro `DEBUG_AREAS` a `true`.

A continuación, se llama al método privado `addGuitarInteraction()` para generar los ortoedros de interacción a partir de los cuales el usuario puede interacturar con la guitarra. Se generan dos volúmenes de interacción: uno para el cuello y el otro para las cuerdas de la guitarra. Es posible visualizar los vértices de estos ortoedros si se establece el parámetro de configuración `DEBUG_VERTICES` a `true`.

```
    private void spawnGuitar() {
        PShape guitarModel = loadShape("../../data/models/guitar/guitar
    .obj");
        guitar = new Guitar(this, guitarModel, null, null);

        guitar.setPos(new PVector(width / 2.f, 4.f * height / 6.f, 150)
    );
        guitar.setRotation(new PVector(radians(0), radians(0), radians
    (-140)));
        guitar.scale(55.f);

        addGuitarInteraction("NECK", -25, -77, 4,
                radians(0), radians(0), radians(5),
                50, 4, 4);

        addGuitarInteraction("STRINGS", 25, -30, 4,
                radians(0), radians(0), radians(5),
                20, 4, 6);
```

```
16
17          guitar.doDrawInteractionVolume(DEBUG_AREAS);
18      }
19
20      private void addGuitarInteraction(String id, float xOffset, float
        yOffset, float zOffset,
21                                        float xRotationOffset, float
        yRotationOffset, float zRotationOffset,
22                                        float width, float height, float
        depth) {
23          InteractiveVolume volume = new InteractiveVolume(this, id,
        width, height, depth);
24
25          volume.setTranslationOffset(xOffset, yOffset, zOffset);
26          volume.setRotationOffset(xRotationOffset, yRotationOffset,
        zRotationOffset);
27
28          volume.setPos(guitar.getPos().x,
29                  guitar.getPos().y,
30                  guitar.getPos().z);
31          volume.setRotation(guitar.getRotation().x,
32                  guitar.getRotation().y,
33                  guitar.getRotation().z);
34
35          guitar.getInteractions().add(volume);
36
37          volume.setVisualizeVertices(DEBUG_VERTICES);
38      }
```

El método *createFloor()* genera una malla que representará el suelo del mundo.

```
1       private void createFloor() {
2           stroke(255);
3           noFill();
4
5           floor = createShape();
6           for (int z = 0; z < COLS; z++) {
7               floor.beginShape(QUAD_STRIP);
8               for (int x = 0; x < ROWS; x++) {
9                   floor.vertex(x * SCALE, 0, z * SCALE);
10                  floor.vertex(x * SCALE, 0, (z+1) * SCALE);
11              }
12              floor.endShape();
13          }
14      }
```

El método primitivo `draw()` refresca los elementos de la escena y son mostrados por pantalla para su visualización. Se habilita al controlador de la *Kinect* mostrar el esqueleto por pantalla y el suelo del mundo.

```
1       public void draw() {
2           background(0);
3
4           setCamera();
5
6           kinect.doSkeleton(true);
7           kinect.refresh(KinectSelector.NONE, true);
8
9           lights();
10
11          guitarInteraction();
```

```
12
13          makeFloor();
14      }
```

La posición de la cámara en función de la posición del punto de articulación de la columna del esqueleto detectado por la *Kinect* al llamar al método `setCamera()`.

```
1       private void setCamera() {
2           PVector spine = kinect.getSkelPos(KinectAnathomy.SPINE);
3
4           if (spine != null) {
5               PVector camPos = new PVector(width/2.f,
6                       height/2.f,
7                       (height/2.f) / tan(PI * 30.f / 180.f));
8
9               camera(camPos.x, camPos.y, camPos.z,
10                      spine.x, spine.y, spine.z,
11                      0,1, 0);
12          }
13      }
```

El estado de la guitarra –resultado de la interacción del usuario con los volúmenes de interacción defnidos para la misma– se actualiza con cada llamada al método `guitarInteraction()`, así como su posición en el mundo.

```
1       private void guitarInteraction() {
2           PVector leftHandPos = kinect.getSkelPos(KinectAnathomy.
        HAND_LEFT);
3           PVector rightHandPos = kinect.getSkelPos(KinectAnathomy.
        HAND_RIGHT);
4
5           guitar.touched(KinectAnathomy.HAND_LEFT.getSkelId(),
6                   leftHandPos,
7                   kinect.getHandRadius());
8
9           guitar.touched(KinectAnathomy.HAND_RIGHT.getSkelId(),
10                  rightHandPos,
11                  kinect.getHandRadius());
12
13          guitar.refresh();
14      }
```

El suelo se muestra por pantalla al llamar al método `makeFloor()`.

```
1       private void makeFloor() {
2           pushMatrix();
3           translate(-ROWS * SCALE / 2.f, 400, -COLS * SCALE /2.f);
4           shape(floor);
5           popMatrix();
6       }
```

Cada vez que aparezca, desaparezca o se mueva algún punto de articulación por la pantalla se procede a llamar a los eventos correspondientes.

```
1       public void appearEvent(SkeletonData _s) {
2           kinect.appearEvent(_s);
3       }
4
5       public void disappearEvent(SkeletonData skel) {
6           kinect.disappearEvent(skel);
7       }
```

```
8
9       public void moveEvent(SkeletonData _b, SkeletonData _a) {
10          kinect.moveEvent(_b, _a);
11      }
```

2.2.2. Paquete `kinect`

Clase `Kinect`

Esta clase es la interfaz que permite interactuar con la *Kinect v1.8*. El constructor acepta los siguientes parámetros:

parent Es una referencia a un objeto de la clase principal `KinectMe`.

pos Establece al posición inicial en el espacio tridimensional de la imágenes obtenidas desde la *Kinect v1.8*.

scale Establece la escala de la imagen.

skeletonRGB Se trata de un vector que ajusta el color del esqueleto que se muestra por pantalla.

```
1       public Kinect(PApplet parent, PVector pos, Float scale, Float[]
        skeletonRGB) {
2           this.parent = parent;
3           kinect = new kinect4WinSDK.Kinect(this.parent);
4           bodies = new ArrayList<SkeletonData>();
5           this.pos = pos;
6           this.scale = scale;
7           this.skeletonRGB = skeletonRGB;
8
9           if (this.pos == null) this.pos = new PVector(0,0,0);
10          if (this.scale == null) this.scale = 1.f;
11          if (this.skeletonRGB == null) this.skeletonRGB = new Float
        []{255.f, 255.f, .0f};
12
13          skelPositions = new HashMap<>();
14          doSkeleton = false;
15      }
```

El método `refresh()` permite refrescar las imágenes que se muestran por pantalla así como visualizar el esqueleto detectado por la *Kinect*. El primer parámetro de este método es un selector de la clase `KinectSelector` permite elegir entre las diferentes opciones de visualización. El segundo parámetro, es una variable booleana que permite visualizar por pantalla las imágenes obtenidas de la *Kinect* cuando se asigna a `true`.

```
1       public void refresh(KinectSelector selector, boolean onScreen) {
2           switch (selector) {
3               case RGB:
4                   img = kinect.GetImage();
5
6                   if (onScreen) {
7                       parent.image(img,
8                               pos.x, pos.y,
9                               640 * scale, 480 * scale);
10                  }
11                  break;
```

```
            case DEPTH:
                img = kinect.GetDepth();

                if (onScreen) {
                    parent.image(img,
                            pos.x, pos.y,
                            640 * scale, 480 * scale);
                }
                break;
            case MASK:
                img = kinect.GetMask();

                if (onScreen) {
                    parent.image(img,
                            pos.x, pos.y,
                            640 * scale, 480 * scale);
                }
                break;
            default:
                break;
        }

        // Calibrates skeleton to Cam
        if (KinectSelector.RGB.equals(selector)) {
            parent.pushMatrix();
            parent.translate(xOffset, yOffset);
        }

        if(doSkeleton) bodyTracking();

        if (KinectSelector.RGB.equals(selector)) parent.popMatrix();
    }
```

El método `bodyTracking()` muestra por pantalla todos los esqueletos detectados por la *Kinect* con cada llamada al método `drawSkeleton()`. Este último, sondea todos los puntos de articulación que conforman el esqueleto detectado e imprime por pantalla cada una de las partes que conforman el esqueleto.

```
    private void bodyTracking() {
        for (int i = 0; i < bodies.size(); i++) {
            drawSkeleton(bodies.get(i));
        }
    }

    private void drawSkeleton(SkeletonData _s) {
        collectPoints(_s);

        // Body
        drawBody();

        // Left Arm
        drawLeftArm();

        // Right Arm
        drawRightArm();

        // Left Leg
        drawLeftLeg();

        // Right Leg
        drawRightLeg();

```

```
25          drawPosition(_s);
26      }
```

Los puntos son sondeados con la llamada al método `collectPoints()`.

```
1       private void collectPoints(SkeletonData _s) {
2           PImage depthImg = kinect.GetDepth();
3           for (KinectAnathomy ka :
4                   KinectAnathomy.values()) {
5               skelPositions.put(ka, ka.getJointPos(_s, depthImg, parent.
    width, parent.height, xOffset, yOffset, skelPositions.get(ka)));
6           }
7       }
```

Cada una de las partes que componen el esqueleto son construidas llamando al método correspondiente. Cada método realiza una llamada a `DrawBone()` con dos selectores de puntos de articulación (`KinectAnathomy`) como argumentos.

```
1       private void drawRightLeg() {
2           DrawBone(KinectAnathomy.HIP_RIGHT,
3                   KinectAnathomy.KNEE_RIGHT);
4           DrawBone(KinectAnathomy.KNEE_RIGHT,
5                   KinectAnathomy.ANKLE_RIGHT);
6           DrawBone(KinectAnathomy.ANKLE_RIGHT,
7                   KinectAnathomy.FOOT_RIGHT);
8       }
9
10      private void drawLeftLeg() {
11          DrawBone(KinectAnathomy.HIP_LEFT,
12                  KinectAnathomy.KNEE_LEFT);
13          DrawBone(KinectAnathomy.KNEE_LEFT,
14                  KinectAnathomy.ANKLE_LEFT);
15          DrawBone(KinectAnathomy.ANKLE_LEFT,
16                  KinectAnathomy.FOOT_LEFT);
17      }
18
19      private void drawRightArm() {
20          DrawBone(KinectAnathomy.SHOULDER_RIGHT,
21                  KinectAnathomy.ELBOW_RIGHT);
22          DrawBone(KinectAnathomy.ELBOW_RIGHT,
23                  KinectAnathomy.WRIST_RIGHT);
24          DrawBone(KinectAnathomy.WRIST_RIGHT,
25                  KinectAnathomy.HAND_RIGHT);
26      }
27
28      private void drawLeftArm() {
29          DrawBone(KinectAnathomy.SHOULDER_LEFT,
30                  KinectAnathomy.ELBOW_LEFT);
31          DrawBone(KinectAnathomy.ELBOW_LEFT,
32                  KinectAnathomy.WRIST_LEFT);
33          DrawBone(KinectAnathomy.WRIST_LEFT,
34                  KinectAnathomy.HAND_LEFT);
35      }
36
37      private void drawBody() {
38          DrawBone(KinectAnathomy.HEAD,
39                  KinectAnathomy.SHOULDER_CENTER);
40          DrawBone(KinectAnathomy.SHOULDER_CENTER,
41                  KinectAnathomy.SHOULDER_LEFT);
42          DrawBone(KinectAnathomy.SHOULDER_CENTER,
43                  KinectAnathomy.SHOULDER_RIGHT);
44          DrawBone(KinectAnathomy.SHOULDER_CENTER,
```

```
                KinectAnathomy.SPINE);
        DrawBone(KinectAnathomy.SHOULDER_LEFT,
                KinectAnathomy.SPINE);
        DrawBone(KinectAnathomy.SHOULDER_RIGHT,
                KinectAnathomy.SPINE);
        DrawBone(KinectAnathomy.SPINE,
                KinectAnathomy.HIP_CENTER);
        DrawBone(KinectAnathomy.HIP_CENTER,
                KinectAnathomy.HIP_LEFT);
        DrawBone(KinectAnathomy.HIP_CENTER,
                KinectAnathomy.HIP_RIGHT);
        DrawBone(KinectAnathomy.HIP_LEFT,
                KinectAnathomy.HIP_RIGHT);
    }
```

`DrawBone()` toma dos selectores de articulación y pinta una línea entre esos dos puntos originando, de este modo, una articulación.

En el caso de pintar el punto de articulación de las manos, se comprueba si el radio es mayor que 0. Si es así, se pinta una esfera del radio que se ha especificado.

```
    private void DrawBone(KinectAnathomy _j1, KinectAnathomy _j2) {
        parent.pushStyle();
        parent.noFill();
        parent.stroke(skeletonRGB[0], skeletonRGB[1], skeletonRGB[2]);

        int i = 0;
        if (KinectAnathomy.HIP_CENTER.equals(_j1) && KinectAnathomy.
    HIP_LEFT.equals(_j2)) {
            i = 1;
        }

        PVector joint1 = skelPositions.get(_j1);
        PVector joint2 = skelPositions.get(_j2);

        if (joint1 != null && joint2 != null) {
            parent.line(joint1.x, joint1.y, joint1.z,
                    joint2.x, joint2.y, joint2.z);

            if ((KinectAnathomy.HAND_LEFT.equals(_j2) || KinectAnathomy
    .HAND_RIGHT.equals(_j2))
                    && handRadius > 0) {
                parent.pushStyle();
                parent.fill(255,0,0,50);
                parent.pushMatrix();
                parent.translate(joint2.x, joint2.y, joint2.z);
                parent.sphereDetail(15);
                parent.sphere(handRadius);
                parent.popMatrix();
                parent.popStyle();
            }
        }

        parent.popStyle();
    }
```

La gestión de los eventos mencionados al final de la sección 2.2.1 son llevado a cabo por los métodos `appearEvent(), disappearEvent()` y `moveEvent()`.

```
    public void appearEvent(SkeletonData _s) {
        if (_s.trackingState == kinect4WinSDK.Kinect.
    NUI_SKELETON_NOT_TRACKED) {
```

```
            return;
        }
        synchronized(bodies) {
            bodies.add(_s);
        }
    }

    public void disappearEvent(SkeletonData skel) {
        synchronized(bodies) {
            for (int i=bodies.size()-1; i>=0; i--) {
                if (skel.dwTrackingID == bodies.get(i).dwTrackingID) {
                    bodies.remove(i);
                }
            }
        }
    }

    public void moveEvent(SkeletonData _b, SkeletonData _a) {
        if (_a.trackingState == kinect4WinSDK.Kinect.
    NUI_SKELETON_NOT_TRACKED) {
            return;
        }
        synchronized(bodies) {
            for (int i=bodies.size()-1; i>=0; i--) {
                if (_b.dwTrackingID == bodies.get(i).dwTrackingID) {
                    bodies.get(i).copy(_a);
                    break;
                }
            }
        }
    }
```

Enumerado `KinectAnathomy`

Este enumerado identifica los diferentes puntos de articulación detectados por la *Kinect v1.8*.

```
    HEAD(0, Kinect.NUI_SKELETON_POSITION_HEAD),
    SHOULDER_CENTER(1, Kinect.NUI_SKELETON_POSITION_SHOULDER_CENTER),
    SHOULDER_LEFT(2, Kinect.NUI_SKELETON_POSITION_SHOULDER_LEFT),
    SHOULDER_RIGHT(3, Kinect.NUI_SKELETON_POSITION_SHOULDER_RIGHT),
    SPINE(4, Kinect.NUI_SKELETON_POSITION_SPINE),
    HIP_CENTER(5, Kinect.NUI_SKELETON_POSITION_HIP_CENTER),
    HIP_LEFT(6, Kinect.NUI_SKELETON_POSITION_HIP_LEFT),
    HIP_RIGHT(7, Kinect.NUI_SKELETON_POSITION_HIP_RIGHT),
    ELBOW_LEFT(8, Kinect.NUI_SKELETON_POSITION_ELBOW_LEFT),
    WRIST_LEFT(9, Kinect.NUI_SKELETON_POSITION_WRIST_LEFT),
    HAND_LEFT(10, Kinect.NUI_SKELETON_POSITION_HAND_LEFT),
    ELBOW_RIGHT(11, Kinect.NUI_SKELETON_POSITION_ELBOW_RIGHT),
    WRIST_RIGHT(12, Kinect.NUI_SKELETON_POSITION_WRIST_RIGHT),
    HAND_RIGHT(13, Kinect.NUI_SKELETON_POSITION_HAND_RIGHT),
    KNEE_LEFT(14, Kinect.NUI_SKELETON_POSITION_KNEE_LEFT),
    ANKLE_LEFT(15, Kinect.NUI_SKELETON_POSITION_ANKLE_LEFT),
    FOOT_LEFT(16, Kinect.NUI_SKELETON_POSITION_FOOT_LEFT),
    KNEE_RIGHT(17, Kinect.NUI_SKELETON_POSITION_KNEE_RIGHT),
    ANKLE_RIGHT(18, Kinect.NUI_SKELETON_POSITION_ANKLE_RIGHT),
    FOOT_RIGHT(19, Kinect.NUI_SKELETON_POSITION_FOOT_RIGHT),
    LABEL(20, Kinect.NUI_SKELETON_POSITION_SHOULDER_CENTER),
    NOT_TRACKED(21, Kinect.NUI_SKELETON_POSITION_NOT_TRACKED);

    private int id;
```

```
25      private int skelID;
26
27      KinectAnathomy(int id, int skelID) {
28          this.id = id;
29          this.skelID = skelID;
30      }
31
32      public int getId() {
33          return this.id;
34      }
35
36      public int getSkelId() {
37          return this.skelID;
38      }
```

Las coordenadas de cada punto de articulación –incluida la profundidad– son calculadas en el método `getJointPos()`.

```
1       public PVector getJointPos(SkeletonData _s, PImage depthImg, float
        width, float height, float xOffset, float yOffset, PVector original
        ) {
2           if (_s.skeletonPositionTrackingState[this.skelID] !=
        KinectAnathomy.NOT_TRACKED.skelID) {
3               PVector v = new PVector(_s.skeletonPositions[this.skelID].x
         * width + xOffset,
4                       _s.skeletonPositions[this.skelID].y * height +
        yOffset,
5                       0);
6               return getDepth(v, depthImg, width, height, xOffset,
        yOffset, original);
7           }
8           return null;
9       }
```

El cálculo de la profundidad se obtiene a partir de la imagen en blanco y negro que devuelve la librería `Kinect4WinSDK` tras llamar a la primitiva `GetDepth()`.

Para establecer la profundidad de cada punto de articulación se analiza cada uno de los píxeles de la imagen devuelta por `GetDepth()` y se toma el byte menos significativo. Posteriormente, se comprueba que el valor del byte se encuentre en el rango umbral del intervalo $[100, 250]$. El rango de profundidad se establece en el intervalo $[0, 360]$ y se asigna este valor al punto de articulación correspondiente.

```
1       private PVector getDepth(PVector joint, PImage depthImg, float w,
        float h, float xOffset, float yOffset, PVector original) {
2           PVector j = new PVector(joint.x, joint.y, joint.z);
3           j = Transformation.translate(j, -xOffset, -yOffset, 0);
4
5           int x = (int) j.x;
6           int y = (int) j.y;
7
8           int arrayPos = x + (y * depthImg.width) - 1;
9
10          if (arrayPos < 0 || arrayPos >= depthImg.pixels.length) return
        original;
11
12          int depthData = depthImg.pixels[arrayPos];
13
14          float data = depthData & 0xFF;
```

```
15
16          if (data >= 100 && data <= 250) {
17              float depth = PApplet.map(data, 130, 230, 0, 360);
18              joint.z = depth;
19              return joint;
20          }
21
22          return original;
23      }
```

Enumerado `KinectSelector`

Este enumerado permite seleccionar entre el tipo de imagen que se desea mostrar por pantalla:

- RGB
- DEPTH
- MASK
- NONE

```
1  public enum KinectSelector {
2      RGB,
3      DEPTH,
4      MASK,
5      NONE
6  }
```

2.2.3. Paquete `object`

Clase `Object`

Esta clase abstracta contiene un constructor que toma como parámetros de entrada el modelo del objeto (`PShape`), las texturas (`Texture`) y el material (`Material`).

```
1
2  public abstract class Object {
3      protected PApplet parent;
4
5      protected PVector pos;
6      protected PVector rotation;
7      protected PShape model;
8      protected Texture texture;
9      protected Material material;
10     protected List<InteractiveVolume> interactions;
11
12     public Object(PApplet parent, PShape model, Texture texture,
       Material material) {
13         this.parent = parent;
14
15         this.pos = new PVector(0,0,0);
16         this.rotation = new PVector(0,0,0);
17         this.model = model;
18         this.texture = texture;
19         this.material = material;
```

```
        interactions = new ArrayList<>();
    }
```

Se puede acceder a estas propiedades a través de sus `getters`.

```
    public PShape getModel() {
        return model;
    }

    public void setModel(PShape model) {
        this.model = model;
    }

    public Material getMaterial() {
        return material;
    }

    public void setMaterial(Material material) {
        this.material = material;
    }

    public Texture getTexture() {
        return texture;
    }

    public void setTexture(Texture texture) {
        this.texture = texture;
    }
```

La posición y ángulos de rotación de un objeto presentan también sus propios *getters* y *setters*. Cada vez que se actualiza la posición o el ángulo de rotación del objeto, se modifican también los parámetros de posición y rotación de los volumenes de interacción asociados al mismo.

```
    public PVector getPos() {
        return pos;
    }

    public void setPos(PVector pos) {
        this.pos = pos;
        for (InteractiveVolume iv :
                interactions) {
            iv.setPos(pos.x, pos.y, pos.z);
        }
    }

    public PVector getRotation() {
        return rotation;
    }

    public void setRotation(PVector rotation) {
        this.rotation = rotation;
        for (InteractiveVolume iv :
                interactions) {
            iv.setRotation(rotation.x, rotation.y, rotation.z);
        }
    }
```

El objeto puede ser escalado llamando al método `scale()`.

```
    public void scale(float s) {
        model.scale(s);
```

```
3     }
```

Cada objeto dispone de una lista de objetos `InteractiveVolume` que designa los volúmenes de interacción con los que podrá interactuar el usuario. Esta lista dispone a su vez del siguiente *getter* y *setter*.

```
1     public List<InteractiveVolume> getInteractions() {
2         return interactions;
3     }
4
5     public void setInteractions(List<InteractiveVolume> interactions) {
6         this.interactions = interactions;
7     }
```

Se puede actualizar el estado de un objeto `SceneObject` llamando al método `refresh()`. El material, textura y posición de una instancia de esta clase son actualizados con cada iteración de `draw()`. Las traslaciones y rotaciones que se lleven a cabo en el objeto son aplicadas llamando al método `setTransformations()`. El estado del objeto es actualizado llamando al método `updateState()`.

```
1     public void refresh() {
2         if (material != null) material.refresh();
3         if (texture != null) model.setTexture(texture.getTexture());
4         setTransformations();
5         updateState();
6     }
7
8     private void setTransformations() {
9         parent.pushMatrix();
10        parent.translate(pos.x, pos.y, pos.z);
11        parent.rotateX(rotation.x);
12        parent.rotateY(rotation.y);
13        parent.rotateZ(rotation.z);
14        parent.shape(model, 0, 0);
15        parent.popMatrix();
16    }
17
18    public abstract void updateState();
```

Otros métodos abstractos que se declaran en esta clase son:

`updateState()`. Actualizar el estado de los volúmenes de interacción asociados al objeto.

`touched()`. Lanza las respuestas necesarias cuando un volumen de interacción es tocado por un usuario.

doDrawInteractionVolume(). Permite dibujar los volúmenes de interacción cuando se asigna su parámetro a `true`.

```
1     public abstract void updateState();
2
3     public abstract void touched(int id, PVector joint, int inRadius);
4
5     public abstract void doDrawInteractionVolume(boolean b);
```

Clase `Texture`

Esta clase es la responsable de almacenar las texturas de una instancia del objeto `SceneObject`.

```
    private PImage texture;

    public Texture(PImage texture) {
        this.texture = texture;
    }

    public void setTexture(PImage texture) {
        this.texture = texture;
    }

    public PImage getTexture() {
        return texture;
    }
```

Clase `Material`

Esta clase es la responsable de gestionar los parámetros de los materiales que son utilizados por los objetos de la escena. Estos parámetros controlan cómo se comporta cada objeto de la escena ante la luz. El método `refresh()` es el responsable de llamar a las primitivas en cada iteración.

```
        this.parent = parent;
        this.ambient = ambient;
        this.emissive = emissive;
        this.specular = specular;
        this.shininess = shininess;
    }

    public void refresh() {
        parent.pushStyle();

        parent.noStroke();
        parent.ambient(ambient.x, ambient.y, ambient.z);
        parent.emissive(emissive.x, emissive.y, emissive.z);
        parent.specular(specular.x, specular.y, specular.z);
        parent.shininess(shininess);

        parent.popStyle();
    }

    public void setAmbient(PVector ambient) {
        this.ambient = ambient;
    }

    public void setEmissive(PVector emissive) {
        this.emissive = emissive;
    }

    public void setSpecular(PVector specular) {
        this.specular = specular;
    }

    public void setShininess(float shininess) {
        this.shininess = shininess;
    }

```

```
36      public PVector getAmbient() {
37          return ambient;
38      }
39
40      public PVector getEmissive() {
41          return emissive;
42      }
43
44      public PVector getSpecular() {
45          return specular;
46      }
47
48      public float getShininess() {
49          return shininess;
50      }
```

Clase `InteractionVolume`

Los volúmenes de interacción asociados a un objeto son definidos en esta clase. Su constructor toma los siguientes argumentos:

parent. Es una referencia a un objeto de la clase principal `KinectMe`.

id. Una *string* que identifica el tipo de volumen de interacción que se va a generar.

width. Número de tipo `float` que asigna el ancho del volumen de interacción.

height. Número de tipo `float` que asigna el alto del volumen de interacción.

depth. Número de tipo `float` que asigna la profundidad del volumen de interacción.

```
1       public InteractiveVolume(PApplet parent, String id, float width,
        float height, float depth) {
2           this.parent = parent;
3           this.id = id;
4
5           pos = new PVector();
6           rotation = new PVector();
7
8           this.width = width;
9           this.height = height;
10          this.depth = depth;
11
12          translationOffset = new PVector();
13          rotationOffset = new PVector();
14
15          generateBoxVolume();
16          getCentroid();
17
18          contactPoint = new PVector();
19      }
```

El campo `id` tiene su propio *getter*.

```
1       public String getId() {
2           return id;
3       }
```

La posición y ángulo de rotación del volumen interactivo tienen definidos sus propios *getters* y *setters*. Los *setters* añaden siempre el espaciado necesario para ubicar el volumen en la posición que le corresponde en el objeto.

```
    public PVector getPos() {
        return pos;
    }

    public void setPos(float x, float y, float z) {
        pos = new PVector(x + translationOffset.x,
                y + translationOffset.y,
                z + translationOffset.z);
    }

    public PVector getRotation() {
        return rotation;
    }

    public void setRotation(float thetaX, float thetaY, float thetaZ) {
        rotation = new PVector(thetaX + rotationOffset.x,
                thetaY + rotationOffset.y,
                thetaZ + rotationOffset.z);
    }
```

El punto de contacto del usuario con el volumen de interacción dispone de sus propios *getter* y *setter*.

```
    public PVector getContactPoint() {
        return contactPoint;
    }

    public void setContactPoint(PVector contactPoint) {
        this.contactPoint = contactPoint;
    }
```

Los métodos `isTouchedLeft()` e `isTouchedRight()` indican si un volumen de interacción ha sido tocado con la mano izquierda o derecha, respectivamente.

```
    public boolean isTouchedLeft() {
        return isTouchedLeft;
    }

    public boolean isTouchedRight() {
        return isTouchedRight;
    }
```

El centroide de un volumen de interacción se obtiene a partir del método `getCentroid()`.

```
    private PVector getCentroid() {
        float x = 0;
        float y = 0;
        float z = 0;

        int total = volume.getVertexCount();
        for (int i = 0; i < total; i++) {
            PVector v = volume.getVertex(i);
            x += v.x;
            y += v.y;
            z += v.z;
        }
        centroid = new PVector(x / total, y / total, z / total);
```

```
14
15          Transformation.translate(centroid, pos.x, pos.y, pos.z);
16          return centroid;
17      }
```

El volumen de interacción es generado gracias al método `generateBoxVolume()`.

```
1       private void generateBoxVolume() {
2           parent.rectMode(PConstants.CENTER);
3           volume = parent.createShape();
4
5           volume.beginShape(PConstants.QUAD);
6           volume.vertex(-1 * width,  1 * height,  1 * depth);
7           volume.vertex(1 * width,  1 * height,  1 * depth);
8           volume.vertex(1 * width, -1 * height,  1 * depth);
9           volume.vertex(-1 * width, -1 * height,  1 * depth);
10
11          volume.vertex(1 * width,  1 * height,  1 * depth);
12          volume.vertex(1 * width,  1 * height, -1 * depth);
13          volume.vertex(1 * width, -1 * height, -1 * depth);
14          volume.vertex(1 * width, -1 * height,  1 * depth);
15
16          volume.vertex(1 * width,  1 * height, -1 * depth);
17          volume.vertex(-1 * width,  1 * height, -1 * depth);
18          volume.vertex(-1 * width, -1 * height, -1 * depth);
19          volume.vertex( 1 * width, -1 * height, -1 * depth);
20
21          volume.vertex(-1 * width,  1 * height, -1 * depth);
22          volume.vertex(-1 * width,  1 * height,  1 * depth);
23          volume.vertex(-1 * width, -1 * height,  1 * depth);
24          volume.vertex(-1 * width, -1 * height, -1 * depth);
25
26          volume.vertex(-1 * width,  1 * height, -1 * depth);
27          volume.vertex(1 * width,  1 * height, -1 * depth);
28          volume.vertex(1 * width,  1 * height,  1 * depth);
29          volume.vertex(-1 * width,  1 * height,  1 * depth);
30
31          volume.vertex(-1 * width, -1 * height, -1 * depth);
32          volume.vertex(1 * width, -1 * height, -1 * depth);
33          volume.vertex(1 * width, -1 * height,  1 * depth);
34          volume.vertex(-1 * width, -1 * height,  1 * depth);
35
36          volume.endShape();
37
38          parent.rectMode(PConstants.CORNER);
39      }
```

El método `isColliding()` identifica si un punto de articulación del esqueleto a tocado o colisionado con el volumen de interacción. La colisión es detectada siguiendo el algoritmo de colisiones de cajas delimitadores (ashkatchap, s.f.).

```
1       public boolean isColliding(int id, PVector joint, int inRadius) {
2           if (joint == null || volume.getVertexCount() <= 0) return
        isTouched(id, false);
3
4           int totalVertices = volume.getVertexCount();
5           float[] x = new float[totalVertices];
6           float[] y = new float[totalVertices];
7           float[] z = new float[totalVertices];
8
9           float xmax, xmin, ymax, ymin, zmax, zmin;
10
```

```
        for (int i = 0; i < totalVertices; i++) {
            PVector v = volume.getVertex(i).copy();

            centroid = getCentroid();

            v = Transformation.translate(v, pos.x, pos.y, pos.z);
            v = Transformation.superRotation(v.x, v.y, v.z, centroid.x,
    centroid.y, centroid.z, 0, 0, 1, rotation.z);

            parent.pushStyle();
            parent.strokeWeight(10);
            parent.stroke(255, 0, 0);
            parent.point(centroid.x, centroid.y, centroid.z);

            if (watchVertices) visualizeVertexAndHand(joint, v);

            parent.popStyle();

            x[i] = v.x;
            y[i] = v.y;
            z[i] = v.z;
        }

        xmin = PApplet.min(x);
        xmax = PApplet.max(x);
        ymin = PApplet.min(y);
        ymax = PApplet.max(y);
        zmin = PApplet.min(z);
        zmax = PApplet.max(z);

        return isTouched(id,
                (joint.x >= xmin && joint.x <= xmax) &&
                (joint.y >= ymin && joint.y <= ymax) &&
                (joint.z >= zmin && joint.z <= zmax));
    }
```

Si efectivamente se ha colisionado con el volumen de interacción el método privado `isTouched()` indica qué mano lo ha tocado y se ajusta el color de relleno del volumen a verde. Si no ha habido ninguna colisión el color del volumen se asigna a un color rojo.

```
    private boolean isTouched(int id, boolean isColliding) {
        switch (id) {
            case SkeletonData.NUI_SKELETON_POSITION_HAND_LEFT:
                isTouchedLeft = isColliding;
                break;
            case SkeletonData.NUI_SKELETON_POSITION_HAND_RIGHT:
                isTouchedRight = isColliding;
                break;
            default:
                break;
        }

        if (isTouchedLeft || isTouchedRight) rgb = new int[]{ 0, 255, 0
    };
        else rgb = new int[]{ 255, 0, 0 };

        return isColliding;
    }
```

Si se desea visualizar los vértices y el centroide del volumen de interacción y el punto de articulación de las manos se llama al método privado `visualizeVertexAndHand()`.

```
    private void visualizeVertexAndHand(PVector joint, PVector v) {
        parent.stroke(0, 0, 255);
        parent.point(v.x, v.y, v.z);
        parent.point(centroid.x, centroid.y, centroid.z);
        // Hands
        if (joint != null) parent.point(joint.x, joint.y, joint.z);
    }
```

Las traslaciones y rotaciones del volumen de interacción poseen sus propios *getters* y *setters*.

```
    public void setTranslationOffset(float x, float y, float z) {
        translationOffset = new PVector(x, y, z);
    }

    public PVector getTranslationOffset() {
        return translationOffset;
    }

    public  void setRotationOffset(float x, float y, float z) {
        rotationOffset = new PVector(x, y, z);
    }

    public PVector getRotationOffset() {
        return rotationOffset;
    }
```

El método `refresh()` coloca el volumen de interacción en el espacio asignando las rotaciones y traslaciones pertinentes.

```
    public void refresh() {
        parent.pushMatrix();
        parent.translate(pos.x, pos.y, pos.z);
        parent.rotateX(rotation.x);
        parent.rotateY(rotation.y);
        parent.rotateZ(rotation.z);

        if (rgb == null) rgb = new int[]{ 255, 0, 0 };

        volume.setFill(parent.color(rgb[0], rgb[1], rgb[2], 100));
        parent.shape(volume);
        parent.popMatrix();
    }
```

Clase `Guitar`

Se trata de una subclase de `Object` que sobrescribe los métodos `toched()` y `updateStates()`. Esta clase se ubica dentro del paquete `object.intrument`.

El método `touched()` detecta por cada volumen de interacción asociado a la guitarra –el cuello y las cuerdas de la guitarra– si ha habido alguna colisión de las manos izquierda o derecha.

Si ha ocurrido una colisión con la caja de interacción del cuello de la guitarra, se asigna a las coordenadas del punto de contacto del volumen de interacción las del punto de articulación de la mano que colisionó con él.

Si la colisión ocurre en la caja de interacción de las cuerdas de la guitarra entonces se llama al método `play()` de la clase `SoundFX` para reproducir un sonido generado por un oscilador a elección.

Si por el contrario, no ha habido ninguna colisión de las manos con los volúmenes del cuello de la guitarra, entonces se moverá la guitarra en bloque junto con sus volúmenes de interacción asociados a la coordenada que indique la mano con la que colisionó.

En el caso de que se deje de interactuar con el volumen de las cuerdas, se detendrá la reproducción del sonido generado por la guitarra.

```
1   public void touched(int id, PVector joint, int inRadius) {
2       for (InteractiveVolume iv :
3               interactions) {
4           if (iv.isColliding(id, joint, inRadius)) {
5
6               if ("NECK".equals(iv.getId())) {
7                   if (iv.isTouchedLeft() && id == SkeletonData.
    NUI_SKELETON_POSITION_HAND_LEFT) {
8                       iv.setContactPoint(joint.copy());
9                       takeLeft = true;
10                  }
11                  else if (iv.isTouchedRight() && id == SkeletonData.
    NUI_SKELETON_POSITION_HAND_RIGHT) {
12                      iv.setContactPoint(joint.copy());
13                      takeRight = true;
14                  }
15
16              } else if ("STRINGS".equals(iv.getId())) {
17                  soundFX.play(OscillatorSelector.TRIANGULAR, (float)
     Math.sqrt(joint.x * joint.x + joint.y * joint.y + joint.z * joint.
    z));
18              }
19
20          } else {
21              if ("NECK".equals(iv.getId()) && joint != null) {
22
23                  if (takeLeft && id == SkeletonData.
    NUI_SKELETON_POSITION_HAND_LEFT) {
24                      if (takeRight)
25                          takeRight = false;
26
27                      this.joint = joint;
28                      moveToJoint(iv, joint);
29
30                  } else if (takeRight && id == SkeletonData.
    NUI_SKELETON_POSITION_HAND_RIGHT) {
31                      if (takeLeft)
32                          takeLeft = false;
33                      this.joint = joint;
34                      moveToJoint(iv, joint);
35                  }
36              } else if ("STRINGS".equals(iv.getId())) {
37                  soundFX.stop();
38              }
39          }
40      }
41  }
```

El método privado que permite mover la guitarra en bloque hacia el punto de articulación de la mano cuando se deja de tocar el cuello de la guitarra es `moveToJoint()`.

```
1   private void moveToJoint(InteractiveVolume iv, PVector joint) {
2       PVector dist = joint.copy().sub(iv.getContactPoint());
```

```
        this.setPos(pos.add(dist));
        iv.setContactPoint(joint);
    }
```

El método público `updateState()` permite actualizar la posición y rotación de los volúmenes de interacción asociados a la guitarra.

```
    @Override
    public void updateState() {
        if (doDrawInteractions) {
            for (InteractiveVolume iv :
                    interactions) {
                iv.refresh();
            }
        }
    }
```

2.2.4. Paquete `sounFX`

Clase `SoundFX`

Esta clase define el generador de audio asociado a la guitarra eléctrica usando diferentes osciladores.

Si se quiere generar un sonido, se debe pasar al método `play()` el selector de oscilador que se desea utilizar para reproducir el sonido así como un número `float` que inidique la frecuencia del mismo. Cada vez que se llama a este método se asigna a 0 el valor de `sustain` que permite que la nota se siga sonando después de un cierto tiempo tras llamar al método `stop()`.

```
    public void play(OscillatorSelector os, float frequency) {
        this.os = os;

        switch (os) {
            case PULSE:
                pulse.freq(frequency);
                pulse.play();
                break;
            case SIN:
                sinOsc.freq(frequency);
                sinOsc.play();
                break;
            case SAW:
                sawOsc.freq(frequency);
                sawOsc.play();
                break;
            case SQUARE:
                sqrOsc.freq(frequency);
                sqrOsc.play();
                break;
            case TRIANGULAR:
                triOsc.freq(frequency);
                triOsc.play();
                break;
            default:
                break;
        }

        sustain = 0;
    }
```

El método `stop()` permite detener el sonido una vez campo `sustain` haya alcanzado el umbral definido por `SUSTAIN_TIME`.

```
    public void stop() {
        sustain += 1;
        if (os != null && sustain >= SUSTAIN_TIME) {
            switch (os) {
                case PULSE:
                    if (pulse.isPlaying()) pulse.stop();
                    break;
                case SIN:
                    if (sinOsc.isPlaying()) sinOsc.stop();
                    break;
                case SAW:
                    if (sawOsc.isPlaying()) sawOsc.stop();
                    break;
                case SQUARE:
                    if (sqrOsc.isPlaying()) sqrOsc.stop();
                    break;
                case TRIANGULAR:
                    if (triOsc.isPlaying()) triOsc.stop();
                    break;
                default:
                    break;
            }

            sustain = 0;
        }
    }
```

Enumerado `OscillatorSelector`

Este enumerado permite seleccionar entre los diferentes tipos de osciladores disponibles por la librería `Sound` para generar sonido.

```
public enum OscillatorSelector {
    PULSE,
    SIN,
    SAW,
    SQUARE,
    TRIANGULAR
}
```

Resultados y Discusión

El algoritmo integrado de seguimiento de cuerpo que dispone la `Kinect v1.8` ha permitido establecer una nube de puntos de referencia a partir de los cuales ha sido posible seguir la posición en 2D de los puntos de articulación del cuerpo.

Ha sido posible establecer un correlación de los valores de profundidad de cada punto de articulación a partir de la imagen en blanco y negro generado por la primitiva `GetDepth()` de la librería `Kinect4WinSDK` generados a partir de los sensores IR de la *Kinect v1.8*.

El resultado final es un prototipo de realidad aumentada que permite interaccionar con un objeto para moverlo por un espacio virtual y generar sonido a partir de él.

Su utilidad podría expandirse para generar una nube de puntos en un espacio tridimensional gracias a sus sensores IR y generar un modelo 3D a partir de ellos.

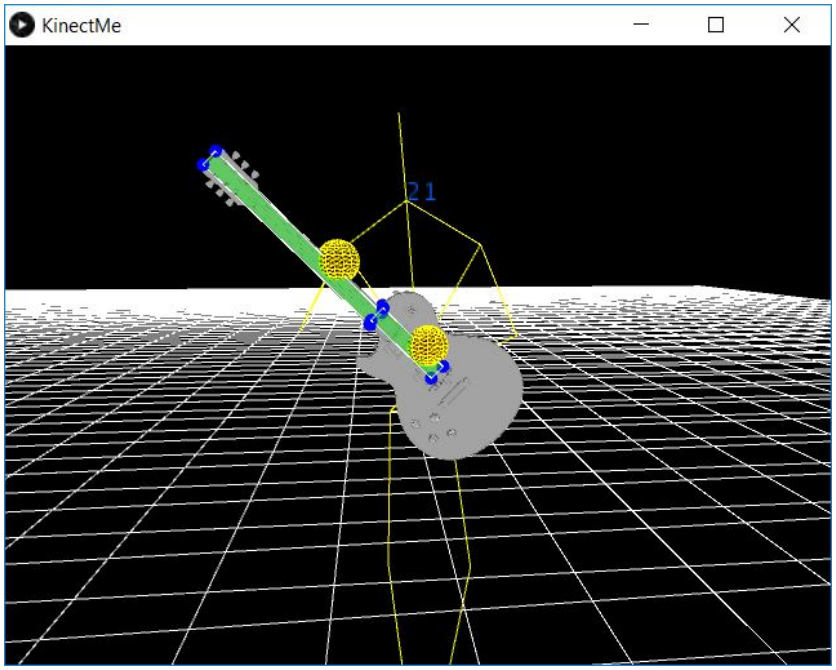

Figura 3.1: Espacio virtual desde el que se puede interaccionar utilizando el algoritmo de detección de cuerpos de la *Kinect*

Conclusiones

La *Kinect* es una herramienta muy útil para realizar el seguimiento de objetos es un espacio tridimensional así como la estimación de profundidad de cualquier cuerpo que se encuentre dentro del rango de acción de los sensores IR.

Estas características convierten a la *Kinect* en un dispositivo económico para realizar funciones de seguimiento del movimiento o definición de nubes de puntos de cualquier cuerpo tridimensional en industrias como la animación.

Referencias

ashkatchap. (s.f.). *3D Collision Detection.* https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection. Mozilla Developer. (Accessed: 2019-04-06)

Chung, B. (s.f.). *Kinect For Processing Library.* http://www.magicandlove.com/blog/research/kinect-for-processing-library/. Magic & Love Interactive. (Accessed: 2019-04-06)

DART. (s.f.). *Kinect Controller.* http://dartecne.wikidot.com/kinect-controller. DART: Dialogs between Art and Technology. (Accessed: 2019-04-06)

Foundation, P. (s.f.). *Sound.* https://processing.org/reference/libraries/sound/index.html. Autor. (Accessed: 2019-04-06)

Microsoft. (s.f.). *Kinect for Windows SDK v1.8.* https://www.microsoft.com/en-us/download/details.aspx?id=40278. Autor. (Accessed: 2019-04-06)

Processing. (s.f.). *Processing: Javadoc.* https://processing.github.io/processing-javadocs/core/index.html. Autor. (Accessed: 2019-02-19)

stone777. (s.f.). *White guitar 3D model.* https://www.turbosquid.com/FullPreview/Index.cfm/ID/1199062. TurboSquid. (Accessed: 2019-04-06)