Juego con detección de movimiento a través de una cámara web

Curso 2k19/2k20

Apellidos, nombre	Cantos Agudo, Juan Carlos (juacanag@inf.upv.es) Pla Tortosa, Jordi (jorplato@arq.upv.es) Haba Claramunt, Marcos (marhacla@inf.upv.es)
Titulación	Grado de Ingeniería informática
Fecha	4/2020

UNIVERSIDAD POLITECNICA DE VALENCIA

Índice

UNIVERSIDAD POLITECNICA DE VALENCIA

Índice de figuras

1 Resumen de ideas claves.

En este trabajo se explica, principalmente, la elaboración de un juego sencillo que se basa en la **detección de pose y de movimiento**, capaz de funcionar correctamente en cualquier equipo que este equipado con una cámara web. También se hará un pequeño repaso de diferentes detectores de pose que se estudiaron para el desarrollo de este trabajo.

2 Objetivos

Los objetivos de este trabajo son:

-Encontrar herramientas de **detección de pose y movimiento sencillas**.

-Llevar a cabo una idea de aplicación a esta herramienta que sea **entretenida** y capaz de **captar la atención** de la gente.

3 Introducción

En este artículo se mostrará y explicará el desarrollo de un pequeño juego basado en la **detección de pose y movimiento**. El juego funciona de la siguiente manera: cada cierto tiempo cae de arriba de la pantalla un **círculo rojo**, al que nos referiremos a partir de ahora como "**obstáculo**" y un **círculo azul**, que será un "**tanto**", el objetivo del juego es conseguir todos los tantos que puedas antes de cometer 3 **fallos**, que se pueden obtener:

-Cogiendo un obstáculo.

-Dejando que un tanto llegue al final de la pantalla.

La **puntuación** se obtiene dependiendo de cuánto tardes en conseguir el tanto, o lo que es lo mismo, cuanto más alto consigas obtener el tanto más puntos obtienes y viceversa. Además, el juego tiene diferentes **dificultades**, variando el intervalo de aparición de las bolas.

Para el **desarrollo** de este juego se utilizará **JavaScript**, junto con la biblioteca **p5.js** totalmente gratuita y de código abierto, basado en **Processing**, desde nuestro punto de vista muy útil para dibujar y crear objetos visuales en pantalla. Para detectar la pose y el movimiento utilizaremos **PoseNet** y **ml5.js**, una librería de machine learning para web, muy accesible y fácil de utilizar, está construido sobre **TensorFlow**.

4 Desarrollo

En este apartado se hablará de cómo fue avanzando el proyecto y se comentara todo lo que utilizamos y lo que decidimos no utilizar además del porqué. Todo esto de manera **cronológica**.

4.1 Primeras semanas.

Una vez decidido el foco del proyecto final, nos pusimos a buscar herramientas para detectar poses y movimiento, encontramos dos, **OpenPose** y **DensePose**.

Son detectores de pose que crean un esqueleto sobre el video que recibe y divide la pose que obtiene en diferentes "puntos clave" como son los codos, muñecas, rodillas, caderas, etc. En el caso de **DensePose**, lo divide en "áreas clave" capturando así una imagen del cuerpo entero y no el esqueleto de la pose.

Nada más las encontramos intentamos instalarlas en nuestros equipos, pero debido a sus múltiples dependencias y sus altos requisitos, la instalación se nos dificulto de sobremanera así que decidimos dejarlos de lado. El principal problema era la instalación de **Nvidia CUDA** y de **Caffe**.

Mas adelante encontramos otro detector de poses, este funcionaba vía web y parecía sencillo, es **PoseNet**, además encontramos este video y este que nos enseña a cómo utilizarlo y como capturar la pose que nos fueron de muchísima ayuda para comenzar a entender cómo funciona **PoseNet**, estos videos también nos decían una manera de implementarlo junto con **P5.js** así que al principio trabajamos en el editor de **P5.js** para web, una herramienta sorprendentemente útil para programar aplicaciones web visuales e interactivas.

4.2 Idea.

Ahora que tenemos las herramientas necesitamos una idea para una aplicación que utilice la detección de pose y sea capaz de captar la atención de la gente, rápidamente se nos ocurrió hacer un juego. Pensamos en hacer un juego sencillo y divertido, en el que caerían objetos que tendrías que coger, y obstáculos que deberás esquivar. Esta fue la idea principal, pero fue evolucionando a medida que se desarrollaba el juego.

4.3 Desarrollo.

Primero tratamos de comprender como funciona p5.js, resulta que tiene varios métodos definidos muy útiles, nosotros utilizamos estos dos:

-El primero es el método **setup(),** en el que se inicializa todo lo necesario, en este caso será el canvas donde se dibujará todo, la cámara y las librerías necesarias que serán **ml5.js** y **PoseNet**.

-El segundo es el método **draw(),** que se trata de un tipo de bucle donde pones lo que tiene que dibujar y él lo hará constantemente.

Sabiendo esto, crearemos el canvas e iniciaremos la captura de video, además iniciaremos **PoseNet** en la función de **setup(),** y haremos que dibuje la captura de la cámara web en el canvas, y para ver si la detección de pose funciona dibujaremos un par de cosas en las muñecas, puesto que **PoseNet** no capta las manos, sino las muñecas, y los "huesos" del esqueleto, el código sería algo así:

```
// Si encuentra pose, crea los aspectos visuales para poder reconocer tu pose
if (pose){
  handR = pose.rightWrist;
  handL = pose.leftWrist;
  let er = pose.rightEye;
  let el = pose.leftEye;
  let nose = pose.nose;
  d = dist(er.x,er.y,el.x,el.y); // Mide la distanca entre los dos ojos para saber si te alejas o acercas
  fill(0,255,0);
  ellipse(handR.x,handR.y,(d+15));
  ellipse(handL.x,handL.y,(d+15));
  triangle((nose.x - 10),nose.y,nose.x,(nose.y + 20),(nose.x + 20),nose.y);
  ellipse(er.x,er.y,10);
  ellipse(el.x,el.y,10);


  // Crea los "huesos" del esqueleto
  for (let i = 0; i < skeleton.length; i++) {
    let a = skeleton[i][0];
    let b = skeleton[i][1];
    strokeWeight(2);
    stroke(255);
    line(a.position.x,a.position.y,b.position.x,b.position.y);
  }

}
```

Figura 1. *Código de esqueleto*

De forma que el resultado es este:

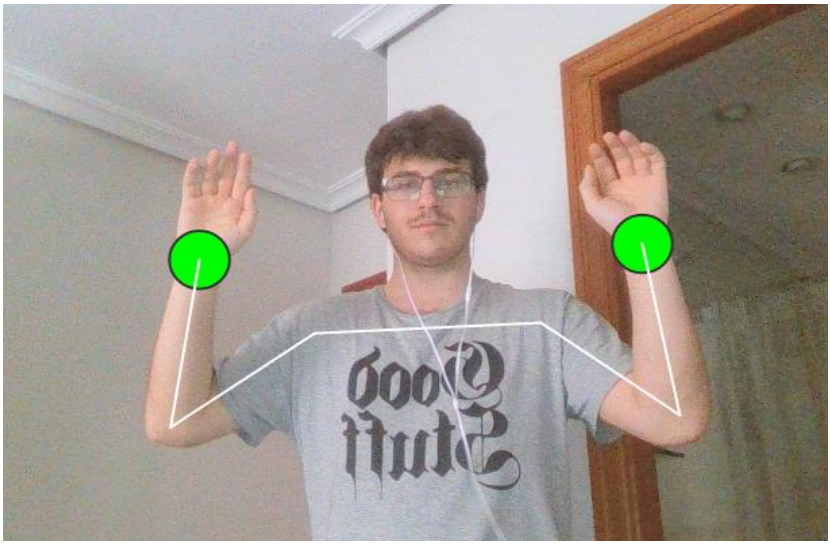

Figura 2. *Imagen cámara + esqueleto*

Cuando obtuvimos la imagen nos dimos cuenta de que estaba en modo espejo, lo que hacía que fuese muy poco intuitivo para el jugador así que invertimos la imagen, haciendo uso del método **scale()**, encontramos como usar este método y muchos otros en la documentación de **p5.js**

Con el detector funcionando y una representación visual de nuestro esqueleto plasmado en la captura de video decidimos comenzar a crear los círculos que caerán por la pantalla y deberemos recoger para ganar puntos. Encontramos un código que ya nos daba un **Object.prototype** para una bola que cae, el único problema es que caían todas a la vez y

nosotros necesitábamos que cayesen poco a poco, así que reutilizamos el modo en el que definía el circulo como un **Object.prototype** y los métodos que tenía, como son **display()** y **move(),** utilizados para poder visualizarse en pantalla y mover el objeto hacia abajo creando la caída de la bola respectivamente. La parte de en qué momento caen las bolas la solucionamos creando un array vacío para las bolas e ir rellenándolo de bolas cada cierto tiempo usando el **SetInterval()** y con un bucle en la función **draw()** que revisa el array todo el rato y ejecuta los métodos de cada objeto bola con los atributos del objeto del array, haciendo así que se visualice(**display()**) y se mueva(**move()**), este sería el código:

```
// Itera sobre el array de los tantos para dibujarlos y aplicarles los metodos propios constantemente
for (var i = 0; i < circles.length; i++) {
  if(game == true){ // Solo si el juego ha comenzado
    circles[i].move();
    circles[i].display();
    circles[i].intersects(i);
  }
```

Figura 3. *Bucle inicializador de tantos*

```
// Intervalo de tiempo para lanzar bolas
var intervalPunct = setInterval(function() {

                    // Si el juego se acaba, corta el intervalo
                    if ( game == false){
                      clearInterval(intervalPunct);
                    }

                    // Crea un tanto y lo mete en su array
                    var x = random(width);
                    var y = random(height/10);
                    var d = random(10, 50);
                    var c = color(255, 0,0);
                    var s = random(st, 3);
                    var bola = new DrawCircle(x, y, d, c, s);
                    circles.push(bola);

                    // Crea un obstaculo y lo mete en su array
                    var x = random(width);
                    var y = random(height/10);
                    var d = 40;
                    var c = color(random(255), random(255), 255);
                    var s = random(0.5, 1.5);
                    var obstacle = new DrawObstacle(x, y, d, s);
                    obstacles.push(obstacle);

},inter);
```

Figura 4. *Intervalo de aparición*

```
// Objto prototipo de Tanto //
DrawCircle.prototype = {
  constructor: DrawCircle,

  // *** Metodo: dibuja el elemento en el canvas *** //
  display: function() {
    fill('blue');
    ellipse(this.xPos,this.yPos, this.diameter, this.diameter);
  },

  // *** Metodo: mueve el elemento hacia abajo *** //
  move: function() {
    this.yPos += this.speed;
    // Si el circulo aparece fuera del canvas, lo devuelve dentro
    if (this.yPos - this.diameter/2 > height) {
      this.yPos = -this.diameter/2;
    }
  },
```

Figura 5. *Object prototype de tanto*

¡Y con esto ya tenemos bolas que caigan! Pero poco después de ejecutar esta versión nos dimos cuenta de que necesitamos que las bolas que caen nos den puntos al tocarlas, y que desaparezcan una vez las toquemos.

Ahora necesitamos detectar **colisiones** entre dos objetos. Con un poco de investigación por la red, encontramos este video que nos explicaba como detectar colisiones entre dos objetos circulares, casualmente del mismo chico que nos enseñó a utilizar **PoseNet** y **ml5.js**.

Ahora que tenemos el algoritmo para detectar colisiones entre dos objetos circulares hay que decidir dónde ponerlo en el código para que surta efecto, después de varios intentos decidimos incluirlo como una función del **Object.prototype** de bola, de forma que creamos el método **intersects()** como se muestra en la siguiente imagen:

```
// *** Metodo: vigila todo el rato que si hay una colision *** //
intersects: function(index) {
      if(pose){
      let distR = dist(handR.x,handR.y,this.xPos,this.yPos);
      let distL = dist(handL.x,handL.y,this.xPos,this.yPos);

          // Colision contra la linea del final
          if(this.yPos >= 440 ){
            circles.splice(index,1);
            fallos += 1;
            var fall = document.getElementById('fallos');
            fall.innerHTML = fallos.toString();

            // Condicional para GAME OVER
            if(fallos == 3){
              game = false;
              circles.length = 0;
              obstacles.length = 0;
              var res = document.getElementById('res');
              res.innerHTML = "GAME OVER"

            }
          }

          // Colision contra las manos
          if((distR < this.radius + d) || (distL < this.radius + d)) {
              circles.splice(index,1);
              st += 0.1;
              console.log(st);
              var p = 1000 - Math.trunc(this.yPos);
              puntuacion += p;
              var punt = document.getElementById('punt');
              punt.innerHTML = puntuacion.toString();

          }
      }
  }
}
```

***Figura 6**. Intersects() de tanto*

Con un simple **if()** controlando que el radio de mis manos sea menor al radio del tanto más la distancia entre estos dos tenemos ya nuestro detector de colisiones, ahora toca decidir qué hacer cuando se detecte una colisión, en este caso decidimos borrarla del array para así borrarla de la pantalla y sumar 100 puntos a una variable global que llevara la puntuación.

Una vez ejecutada esta versión caímos que no hay manera para el jugador saber cuántos puntos lleva ya que no lo mostramos en la pantalla, así que creamos un div en el html para conectarlo con la variable puntuación del js mediante la función **document.getElementById(),** y cada vez que se produjese una colisión se transmitiría la puntuación actualizada.

Con esta versión del juego ya se puede pasar un rato entretenido, pero le faltaba una cosa, dificultad, era muy simple y no tenía un **GAME OVER**, se discutió en si marcar el final de la partida con tiempo o con fallos, y al final ganaron los fallos.

La primera manera de conseguir fallos que se implemento fue dejar pasar un tanto y que llegue al final de la pantalla, para esto creamos una línea abajo de la captura de video y utilizando el método anterior, detectamos la colisión haciendo que cuando un tanto toque la línea, desaparezca el tanto y se sume uno a una variable global que llevara los fallos y que, además,

será visible en el html de la misma forma que la puntuación. también pensamos en poner el límite de fallos a 3 y cuando se llegara a tres fallos saldría un h1 en el html con "GAME OVER", además borraría todos los círculos del array y la pantalla quedaría limpia. Se puede ver claramente en la **Figura 6.**

Okey, ahora ya tenemos un final de la partida, pero el juego sigue siendo fácil, así que añadimos **obstáculos** que se deben esquivar, después de pensar maneras de cómo crear los obstáculos nos decidimos a hacerlo igual que los tantos, crear un **Object.prototype** copiando los métodos, pero con pequeñas variaciones en el constructor y en el método de colisión, puesto que la reacción a la colisión será diferente.

Los obstáculos serán del mismo tamaño y de color rojo, para su fácil distinción de los demás objetos en la pantalla, y serán más lentos para añadir así una capa de dificultad puesto que se irán acumulando poco a poco en la pantalla y cada vez será más difícil moverse sin tocar uno. En cuanto a las consecuencias de coger un obstáculo con las manos, primero desaparecerá ese obstáculo como si de un tanto se tratase y se sumara un fallo al contador y si se llega a 3 pues se vaciarán los dos arrays (obstáculos y tantos) para dejar la pantalla limpia.

```
// Objeto prototipo de Obstaculo //
DrawObstacle.prototype = {
constructor: DrawObstacle,
display: function() {
  fill('red');
  ellipse(this.xPos,this.yPos, this.diameter, this.diameter);
},

move: function() {
  this.yPos += this.speed;
  if (this.yPos - this.diameter/2 > height) {
    this.yPos = -this.diameter/2;
  }
},

  intersects: function(index) {
      if(pose){
      let distR = dist(handR.x,handR.y,this.xPos,this.yPos);
      let distL = dist(handL.x,handL.y,this.xPos,this.yPos);

          // Colision con la linea del final
          if(this.yPos >= 440 ){
            obstacles.splice(index,1);
          }

          // Colision contra las manos
          if((distR < this.radius + d) || (distL < this.radius + d)) {
              obstacles.splice(index,1);
              fallos += 1;
            var fall = document.getElementById('fallos');
            fall.innerHTML = fallos.toString();

            // Condicion para GAME OVER
            if(fallos == 3){
              game = false;
              circles.length = 0;
              obstacles.length = 0;
              var res = document.getElementById('res');
              res.innerHTML = "GAME OVER"

            }
          }
      }
  }
```

***Figura 7**. Object prototype de Obstacle*

Con el objeto obstáculo hecho, nuestro siguiente paso es meterlos en pantalla y decidir su intervalo, por motivos de dificultad y facilidad de uso elegimos hacer que los obstáculos aparecieran a la vez que los tantos y los pusimos en el mismo **SetInterval(),** esto se puede ver en la **Figura 4**, para que se visualicen en la pantalla hacemos lo mismo que con los tantos, un bucle que recorra el array donde están guardados aplicando los métodos, se muestra en la **Figura 8**.

```
// Itera sobre el array de los obstaculos para dibujarlos y aplicarles los metodos propios constantemente
for (var j = 0; j < obstacles.length; j++) {
  if(game == true){ // Solo si el juego ha comenzado


    obstacles[j].move();
    obstacles[j].display();
    obstacles[j].intersects(j);
  }
}
```

***Figura 8**. Bucle inicializador de obstáculos*

.

Con tantos objetos en pantalla nos era un poco difícil distinguir los tantos y los obstáculos debido al pequeño tamaño del **lienzo** utilizado para plasmar la captura de video, así que decidimos aumentar su tamaño, pero debido a los problemas que nos causaba, decidimos no capturar video y poner un fondo negro para distinguir bien los colores, fue muy sencillo, cambiar la línea que ponía el video en el lienzo por un fondo negro (de todas formas comentamos la línea del modo video por si alguien quiere probarla), además añadimos ojos y nariz puesto que al no ver tu cuerpo es más difícil ver que posición tienes, así que añadiendo unos cuantos elementos de la cara se hace mas intuitivo.

Aquí una pequeña comparación:

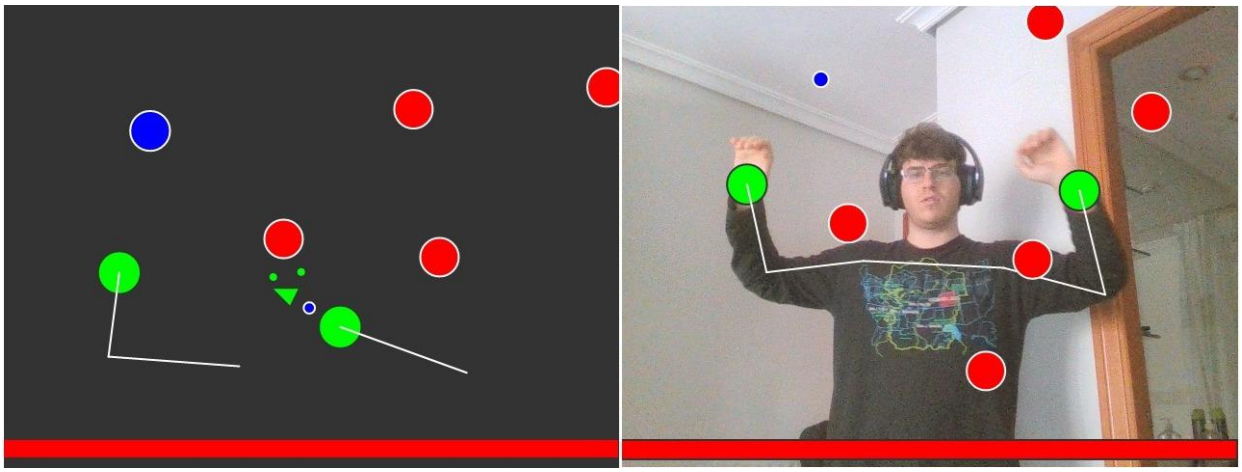

***Figura 9**. Comparación modos de visualización*

Esto ya va tomando forma, ya es un juego bastante **entretenido**, incluso hay **competitividad** entre los compañeros del grupo por ver quien saca la mayor puntuación, pero hay un detalle bastante molesto y es que para volver a iniciar una partida hay que recargar la página y tarda un rato, así que lo siguiente que toca es hacer que con un clic podamos volver a jugar rápidamente. Fácil, necesitamos algo que nos diga cuando comienza el juego y cuando acaba, pues bien primero pondremos un botón en el html que inicie el juego, en el js lo enlazaremos y haremos que cuando sea apretado cambie a **true** una variable que indicara si el juego se inicia o se termina, y esta variable la utilizaremos en la guarda de los condicionales que contendrán las partes de que inician el juego, como la caída de las bolas o el contador de puntos, se puede ver en la **Figura 4**.

Con esto hecho le pasamos el juego a algunos amigos para que lo probasen y nos dijesen sus primeras impresiones, el **feedback** fue que era demasiado difícil, así que pusimos tres botones en el **html** que cambiarían la dificultad a **fácil**, **medio** o **difícil**. El cambio de dificultad se basa en el intervalo de caída de las bolas y los botones del **html** cambian esto, siendo **fácil** 4 segundos, **medio** 3 segundos y **difícil** 2 segundos. Además de la dificultad, nos dijeron que no lograban llegar al borde izquierdo de la pantalla, al investigar este fallo nos dimos cuenta de que en sus ordenadores el canvas, donde se plasma el esqueleto de la pose y donde caen, cambia de dimensiones de 640x480 a 1200x900, esto crea unos bordes inalcanzables en la izquierda y abajo.

5 Conclusión.

En resumen, estamos bastante contentos con el resultado final del proyecto, cumple la finalidad que definimos al principio, ser entretenido, capturar la atención de la gente y crear un entorno "competitivo" entre los jugadores. Pero, todo sea dicho, quedaron varios errores sin arreglar ya sea por falta de tiempo o falta de experiencia, por ejemplo:

- Al ejecutar el juego en un ordenador diferente al nuestro, el canvas cambia de dimensión y crea unos bordes donde siguen cayendo bolas inalcanzables que arruinan el juego. Esto creemos que puede ser por **Windows**, nosotros utilizando **kubuntu** funciona bien.

-Pensamos en hacer un **ranking** guardando las puntuaciones y un ID de las ultimas partidas, pero tras varios intentos fallidos lo dejamos de lado y nos pusimos con otros aspectos más relevantes del juego.

-Todo lo que ponemos en el html se coloca justo encima del **canvas** aunque lo pongamos encima de el en el archivo. Esto nos molestó mucho puesto que no pudimos crear la interfaz que teníamos pensada, y tuvimos que dejarlo así.

-Pensamos en aumentar el tamaño del canvas, pues si te vas a alejar para jugar, la pantalla puede ser pequeña y se hace difícil ver las bolas, pero aun que lo intentamos de muchas maneras diferentes, nos fue imposible por diferentes problemas como conflictos de dimensiones de la cámara web y el canvas.

De todas formas, este proyecto nos ha aportado mucho, sobre todo experiencia sobre la que, en proyectos futuros, nos podremos apoyar, por ejemplo, resolución de problemas, planificación e investigación.

6 Bibliografía.

[1] "ml5.js PoseNet"
https://learn.ml5js.org/docs/#/reference/posenet

[2] "Pose Estimation with TensorFlow, Overview"
https://www.tensorflow.org/lite/models/pose_estimation/overview

[3] "COCO Dataset Explorer"
http://cocodataset.org/#explore

[4] "Real-Time Pose Estimation in the Browser with TensorFlow.js"
https://medium.com/tensorflow/real-time-human-pose-estimation-in-the-browser-with-tensorflow-js-7dd0bc881cd5

[5] "Humans of AI" Online Exhibition
https://humans-of.ai/editorial

[6] "DensePose"
http://densepose.org/

[7] "OpenPose GitHub"
https://github.com/CMU-Perceptual-Computing-Lab/openpose

[8] "OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields" *Coronell University*
https://arxiv.org/abs/1812.08008

[9] "p5.js documentation"
https://p5js.org/reference/