Trabajo Final CIU

Space Invaders.

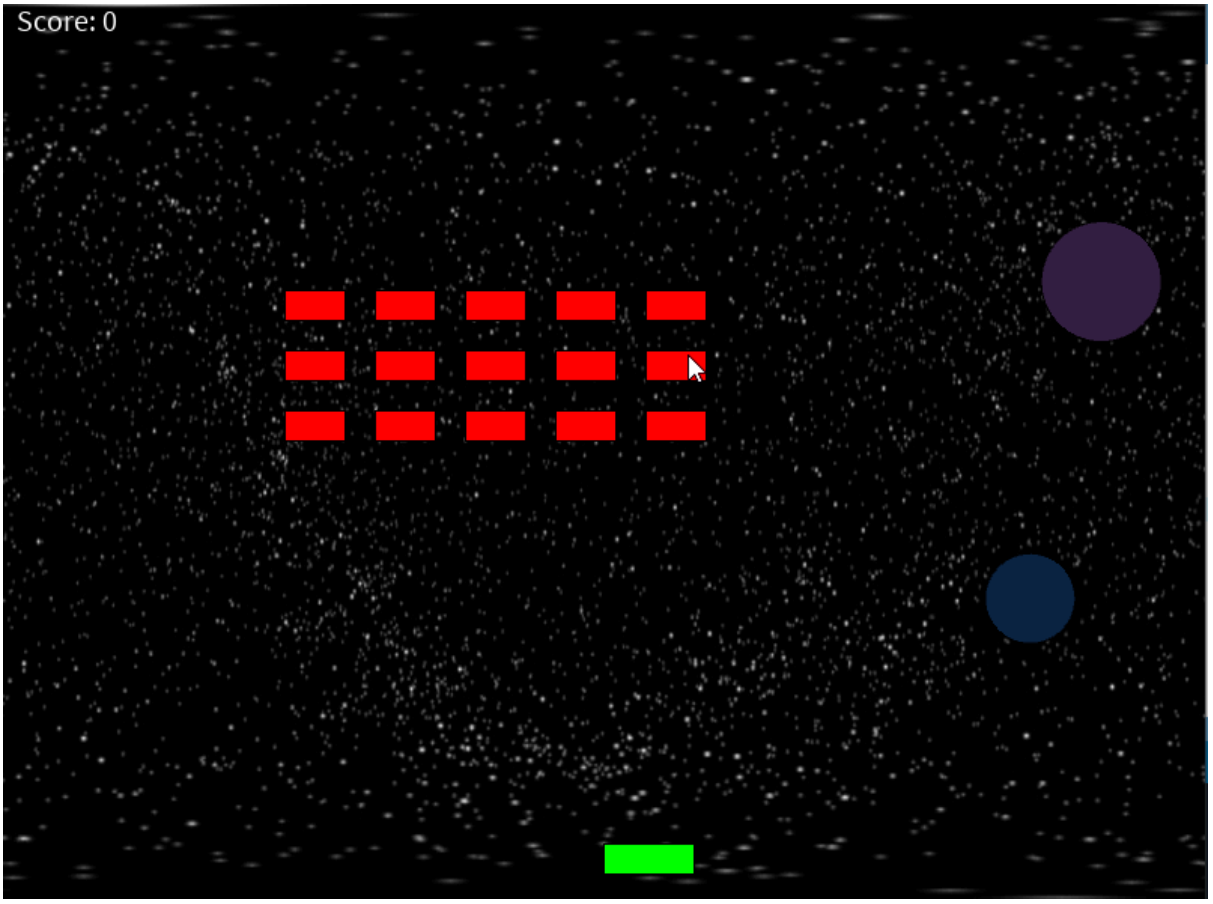
Brian Palmés Gómez

1.Propuesta.

Como trabajo final de la asignatura se ha decidido desarrollar un juego arcade clásico y sencillo como el Space Invaders y añadirle algunos de los temas trabajados en la asignatura.
Como un sensor para disparar o el cambio de color de certos elementos al ritmo de la música

2.Descripción técnica del trabajo.

Nuestro proyecto consiste en la implementación del clásico arcade Space Invaders donde una nave controlada por el jugador debe destruir las demás para hacer puntos sin que estas naves enemigas consigan tocar la del jugador.



El enunciado del trabajo final nos pide integrar con nuestro prototipo diferentes características trabajadas durante la asignatura.

En nuestro caso hemos realizado una versión de Space Invaders a la que le hemos añadido:

-Captura de audio usando la librería dff.minim.

-El uso de un sensor integrado en un arduino que nos permitirá disparar más rápido si lo usamos.

A continuación haremos una explicación de las partes más importantes del código.

Setup()

```
Serial myPort;
int threshold = 190; // Umbral para disparar

void setup() {
  size(800, 600, P2D);
  setupAudio();
  player = new Player();
  bullets = new ArrayList<Bullet>();
  enemies = new ArrayList<Enemy>();
  backgroundTexture = loadImage("space.jpg");

  // Crear planetas con velocidades de traslación
  planet1 = new Planet(width / 2, height / 2, 300, 30, 0.01);
  planet2 = new Planet(width / 2, height / 2, 350, 40, -0.01);



  // Inicializar enemigos
  resetEnemies();

  // Configurar comunicación serie
  String portName = Serial.list()[0]; // Selecciona el primer puerto disponible
  myPort = new Serial(this, portName, 9600);
}
```

En esta zona del programa preparamos todo lo necesario para empezar la partida y establecemos el tamaño de la app.
Luego con la función setupAudio() inicializamos las opciones de audio.
Se crean los jugadores el array de disparos y los enemigos y se establece una imagen de fondo.

Luego se inicializan los shaders y se incia la lógica de los enemigos con resetEnemies()

La última parte inicia la comunicación con el arduino.

Lógica del Juego.

```
void draw() {
  if (gameOver) {
    background(backgroundTexture);
    fill(255, 0, 0);
    textSize(50);
    textAlign(CENTER, CENTER);
    text("Game Over", width / 2, height / 2);
    return;
  }

  background(backgroundTexture);

  detectBeat();

  if (beat.isOnset()) {
    planet1.changeColor();
    planet2.changeColor();
  }

  // Dibujar planetas
  planet1.display();
  planet2.display();

  player.display();
```

En el la zona de draw() es donde tenemos implementada la lógica del juego en la imagen vemos el código de game over en caso de que la partida termine salta una pantalla de fin de partida.
Luego vemos el método detectBeat() que detecta los beats de la canción que se reproducen en el juego y cambia el color de los planetas que orbitan de fondo a golpe de la canción.

```
player.display();

for (Bullet b : bullets) {
  b.update();
  b.display();
}



for (Enemy e : enemies) {
  e.move();
  e.display();
  if (e.hits(player)) {
    gameOver = true;
  }
}



// Control del jugador
if (keyPressed) {
  if (key == 'a' || key == 'A') {
    player.move(-1);
  } else if (key == 'd' || key == 'D') {
    player.move(1);
  }
}
```

Seguimos con el control del array de proyectiles.
Añadimos el movimiento y dibujado de los enemigos controlando que si tocan al jugador acabe la partida y añadimos los botones de movimiento del jugador.

```
// Verificar colisiones
for (Bullet b : bullets) {
  for (Enemy e : enemies) {
    if (b.hits(e)) {
      e.alive = false;
      score += 10;
    }
  }
}

// Eliminar balas inactivas y enemigos muertos
bullets.removeIf(b -> !b.active);
enemies.removeIf(e -> !e.alive);

// Verificar si todos los enemigos han sido eliminados
if (enemies.isEmpty()) {
  resetEnemies();
}

// Dibujar la puntuación
fill(255);
textSize(20);
text("Score: " + score, 10, 20);
```

En esta última imagen vemos la gestión de colisiones de los proyectiles con los enemigos y eliminamos las balas y los enemigos.

Se reinicia el bloque de enemigos si ya se ha eliminado un bloque de enemigos y se dibuja la puntuación.

Integración con sensor y Arduino.

Se ha integrado a juego la opción de disparar usando un sensor de movimiento integrado en una placa Arduino.
Cuando el jugador quiera disparar más rápido y seguido solo deberá pasar la mano por encima del sensor.

Para comenzar necesitaremos importar la librería de arduino en nuestro proyecto

import processing.serial.*;

Y en el setup añadir:

```
// Configurar comunicación serie
String portName = Serial.list()[0]; // Selecciona el primer puerto disponible
myPort = new Serial(this, portName, 9600);
```

Primero almacenamos en la variable el nombre del primer puerto en serie que devuelve el array con la lista de puertos en serie disponibles el primero y único es nuestra placa arduino.

Luego hacemos la típica inicialización para comunicar nuestro sketch con la placa arduino dónde **this** es el sketch actual **portName** el nombre que recogimos antes y **9600** es una velocidad de transmisión usada para conectar con arduino.

```
// Leer del puerto serie
if (myPort.available() > 0) {
  String inString = myPort.readStringUntil('\n');
  if (inString != null) {
    inString = trim(inString);
    int sensorValue = int(inString);
    if (sensorValue > threshold) {
      bullets.add(new Bullet(player.x + player.w / 2, player.y));
    }
  }
}
```

En el draw() comprobamos si hay información relevante en el puerto en serie y limpiamos la información hasta pasarla a entero y en caso que el valor leído sea mayor que el umbral disparará un proyectil desde la posición actual del jugador.

```
int sensorPin = A0;  // Pin analógico donde está conectado el sensor de distancia
int sensorValue = 0; // Valor leído del sensor

void setup() {
  Serial.begin(9600);  // Inicia la comunicación serial a 9600 baudios
}

void loop() {
  sensorValue = analogRead(sensorPin); // Lee el valor del sensor
  Serial.println(sensorValue);         // Envía el valor leído por el puerto serial
  delay(100);                          // Espera 100 ms antes de la siguiente lectura
}
```

Por último el código que se carga en la placa para leer el valor de proximidad y enviarlo al sketch.

Captura de Audio.

A esta práctica hemos añadido una canción que sonará mientras dure la partida como se hizo en otra práctica usaremos la librería minim para gestionar los beats de sonido de la canción mientras esta se reproduce y estos beats cambiarán el color de los planetas que hemos puestos orbitar de fondo al ritmo de la música.

Primero importamos las librerias necesarias en el sketch principal
import ddf.minim.*;
import ddf.minim.analysis.*;

```
import ddf.minim.*;
import ddf.minim.analysis.*;

Minim minim;
AudioPlayer song;
BeatDetect beat;

void setupAudio() {
  minim = new Minim(this);
  song = minim.loadFile("imperial.mp3", 1024);
  song.play();

  beat = new BeatDetect();
  beat.setSensitivity(300); // Ajusta la sensibilidad según sea necesario
}

void detectBeat() {
  beat.detect(song.mix);
}
```

Tras importar las librerías necesarias para el manejo de los beats declaramos una serie de variables una minim otra tipo AudioPlayer que usaremos para reproducir la canción y una variable beat tipo BeatDetect una clase que se usa para detectar los beats en una pista de audio.

Luego la función setupAudio() que se usará en el setup() inicializamos lo necesario para detectar beats.
Cargamos la canción que vamos a reproducir usando la variable minim y cargando la pista de audio en el objeto tipo Audio Player.
También inicializamos el objeto tipo BeatDetect u establecemos la sensibilidad del detector de beats a más baja más sensible es a los beats de audio.

Definimos el método beatDetect() que usaremos en el draw() usamos el método detect() de la clase BeatDetect para leer los beats en la pista de audio y le pasamos song.mix que es la mezcla estéreo del audio cargado.

Gráficos.

Presentamos un breve resumen de los elementos gráficos del juego por un lado la clase player.

```
class Player {
  float x, y;
  float w, h;
  color c;

  Player() {
    x = width / 2;
    y = height - 40;
    w = 60;
    h = 20;
    c = color(0, 255, 0);
  }

  void display() {
    fill(c);
    rect(x, y, w, h);
  }

  void move(float dir) {
    x += dir * 5;
    x = constrain(x, 0, width - w);
  }

  void changeColor() {
    c = color(random(255), random(255), random(255));
```

En esta clase se establece el movimiento del nave (rectángulo) controlada por el jugador.

La clase Planet

```
class Planet {
  float centerX, centerY;
  float orbitRadius;
  float angle; // Ángulo de la órbita
  float rotationSpeed; // Velocidad de la órbita
  float radius;
  color c;

  Planet(float centerX, float centerY, float orbitRadius, float radius, float rotationSpeed) {
    this.centerX = centerX;
    this.centerY = centerY;
    this.orbitRadius = orbitRadius;
    this.radius = radius;
    this.angle = 0;
    this.rotationSpeed = rotationSpeed;
    this.c = color(255);
  }

  void display() {
    float x = centerX + cos(angle) * orbitRadius;
    float y = centerY + sin(angle) * orbitRadius;
    fill(c);
    ellipse(x, y, radius*2, radius*2);
    angle += rotationSpeed;
  }

  void changeColor() {
```

Una elipse con una órbita que se ven por encima de la imagen de fondo y que cambiarán de color con los beats de la música.

La clase Enemy

```
class Enemy {
  float x, y;
  float w, h;
  boolean alive = true;
  color c;

  Enemy(float startX, float startY) {
    x = startX;
    y = startY;
    w = 40;
    h = 20;
    c = color(255, 0, 0);
  }

  void display() {
    if (alive) {
      fill(c);
      rect(x, y, w, h);
    }
  }

  void move() {
    y += 0.7;
    if (y > height) {
      alive = false;
```

Otro rectángulo que será generado en una matriz de enemigos. Y aquí su collider

```
  void changeColor() {
    c = color(random(255), random(255), random(255));
  }

  boolean hits(Player p) {
    return alive && x < p.x + p.w && x + w > p.x && y < p.y + p.h && y + h > p.y;
  }
}
```

Comprueba sus coordenadas respecto a las del jugador comparando sus bordes con los del jugador y si se solapan.
Si todas las condiciones son verdaderas significa que hay una colisión del jugador con un enemigo.

3.Fuentes y tecnologías.

Cómo fuente principal se ha usado el material de la asignatura recogido en el github del profesor

https://github.com/otsedom/otsedom.github.io/tree/main/CIU

Para la implementación de Space Invader originalmente se intento un proyecto como el de estos chicos pero al final se ha ido cambiado y descartando este código.

https://gist.github.com/ihavenonickname/5cc5b9b1d9b912f704061a241bc096ad

Y estos canales de youtube

https://www.youtube.com/watch?v=9nv8NoxLWcw

https://www.youtube.com/watch?v=NFIUnssR65g

Y luego mirando en google en general.

Tecnologías se han usado processing y Arduino.

4.Participación

El único integrante de este grupo es Brian Palmés Gómez.

5.Código.

https://github.com/bpalmes/CIU24/tree/main/Trabajo%20Final/spaceinvaders

6.Conclusiones y propuesta de ampliación.

Como conclusión saco que debería haber invertido más tiempo e integrar otras temáticas dadas en la asignatura como los shaders

En cuanto al juego es bastante mejorable se podrían añadir varios tipos de disparo.
Varias hordas de enemigos simultáneas.
Enemigos diferentes.
Que los enemigos fueran más rápidos a medida que se avanza.
Un menú principal.