INSTITUTO TECNOLÓGICO SUPERIOR PROGRESO

Carrera:

Ingeniería en sistemas computacionales

Materia:

Graficación

Actividad:

ADA 3.3 - Reporte de prácticas 3d y animación

Docente:

Dr. Holzen Atocha Martínez García

Alumno:

David Ezequiel Caballero González

Fecha de entrega:

Domingo 12 de octubre de 2025

Prácticas de animación y 3D

Introducción

Este reporte presenta de manera detallada una serie de prácticas desarrolladas en el entorno de programación Processing, enfocadas en los temas de animación bidimensional y renderizado tridimensional. El propósito central es documentar de manera clara y estructurada cada ejercicio, desde la lógica planteada hasta su implementación final, explicando los pasos de programación, las estructuras de control empleadas, y los resultados obtenidos. La introducción de conceptos como el movimiento continuo, la interacción mediante eventos, el uso de transformaciones jerárquicas y la construcción de superficies tridimensionales ofrece una visión integral del potencial de Processing como herramienta para la visualización dinámica. El reporte, además, busca destacar la importancia del pensamiento algorítmico y la traducción de modelos matemáticos a representaciones gráficas, mostrando cómo el código se convierte en un medio para generar experiencias visuales interactivas. De esta manera, se establece una base sólida para comprender y extender proyectos más avanzados, no solo en el ámbito académico, sino también en aplicaciones prácticas de simulación, diseño digital y creación artística.

- Los códigos pueden verse en el repositorio en GitHub: Graficación_Prácticas de anicación y 3D

Objetivos

Objetivo general

Aplicar los fundamentos de animación y render 3D de Processing para construir escenas interactivas, superficies y sistemas jerárquicos de transformaciones.

Objetivos específicos

- Simular múltiples partículas rebotando con control de colisiones (Práctica 5-1).
- Dibujar órbitas y extender un sistema planetario con transformaciones anidadas (5-2 y 5-3).
- Implementar interacción 3D con rotación por ratón y zoom por teclado (7-1).
- Renderizar superficies con QUAD_STRIP y iluminación básica (7-2).
- Aplicar gradientes por altura (z) mediante interpolación de color por vértice (7-3).
- Construir una malla 3D a partir del nivel de brillo de una imagen (mapa de alturas) (7-4).

Marco teórico

La animación en Processing se basa en el uso del ciclo setup() y draw(). La función setup() se ejecuta una sola vez al inicio del programa, mientras que draw() se ejecuta de manera continua a una velocidad aproximada de 60 cuadros por segundo. Este esquema permite la actualización constante de las variables y la representación dinámica de los objetos en pantalla. El movimiento se logra modificando las posiciones de los objetos en cada iteración, mientras que las colisiones y reacciones se controlan mediante estructuras condicionales que invierten o modifican la dirección de las velocidades.

Un concepto clave en animación es el uso de arreglos y bucles, ya que permiten escalar fácilmente de un objeto a varios. Por ejemplo, al controlar múltiples partículas, cada una con su propia posición y velocidad, se logra un comportamiento grupal más complejo sin aumentar significativamente el código. Asimismo, el empleo de colores, transparencias y efectos como rastros visuales mejora la experiencia visual y transmite sensaciones de movimiento fluido.

En cuanto a las transformaciones gráficas, Processing ofrece funciones como translate(), rotateX(), rotateY(), rotateZ() y scale(). Estas modifican el sistema de coordenadas y permiten rotar, trasladar o escalar objetos. El uso de pushMatrix() y popMatrix() es esencial para controlar jerarquías de transformación, es decir, asegurar que una rotación o traslación aplicada a un objeto no afecte a todos los demás. Esto es particularmente útil en sistemas jerárquicos como un modelo planetario, donde el movimiento de las lunas depende de su planeta, y este, a su vez, del sol.

Para el trabajo con gráficos tridimensionales, Processing utiliza el renderizador P3D. Con él es posible manipular objetos en el espacio tridimensional, empleando coordenadas (x, y, z). Aquí cobra importancia la iluminación: la función lights() activa un conjunto básico de luces (ambientales y direccionales), lo que otorga realismo a los objetos 3D. También es posible agregar luces específicas como pointLight() o directionalLight(). La iluminación influye en cómo percibimos las superficies y es indispensable para visualizar relieves y profundidad.

La construcción de superficies 3D puede realizarse con estructuras de vértices utilizando beginShape(QUAD_STRIP) y endShape(). Esta técnica permite definir mallas paramétricas o basadas en funciones matemáticas. El color puede aplicarse por superficie completa con fill() o variar por vértice para generar gradientes mediante lerpColor(). Además, la introducción de mapas de alturas a partir de imágenes amplía el potencial de representación al convertir la luminosidad de cada píxel en un valor de elevación, lo que produce modelos topográficos básicos.

Finalmente, la interacción con el usuario mediante teclado y ratón enriquece las prácticas. Con mouseDragged() se implementan rotaciones interactivas, mientras que con keyPressed() se pueden programar acercamientos y alejamientos en la vista 3D. Este componente interactivo transforma los ejercicios en entornos dinámicos y controlables, esenciales para explorar la visualización en tiempo real.

Procedimientos

Práctica 5-1

En la parte superior del código, se deben declarar la cantidad de bolas (n), los arreglos de las posiciones (px[], py[]), de las velocidades (vx[], vy[]) y el diámetro de cada bola (diam).

```
int n = 25;             // cantidad de bolas
float[] px, py;         // posiciones
float[] vx, vy;         // velocidades
int diam = 20;          // diametro de cada bola
```

En setup(), se define el tamaño del lienzo con size(640,360), se desactiva el trazo con noStroke() y se prepara fill(255).

```
void setup() {
  size(640, 360);
  noStroke();
  fill(255);
```

Ahí mismo en setup(), se inicializan los arreglos con posiciones aleatorias dentro de la ventana (respetando el radio) y con velocidades aleatorias en (−4,4). Evitando velocidades cercanas a cero para que todas las bolas se muevan.

```
void setup() {
  size(640, 360);
  noStroke();
  fill(255);

  px = new float[n];
  py = new float[n];
  vx = new float[n];
  vy = new float[n];
  for (int i = 0; i < n; i++) {
    px[i] = random(diam/2.0, width - diam/2.0);
    py[i] = random(diam/2.0, height - diam/2.0);
    vx[i] = random(-4, 4);
    vy[i] = random(-4, 4);
    if (abs(vx[i]) < 0.5) vx[i] = (random(1) < 0.5) ? -1 : 1;
    if (abs(vy[i]) < 0.5) vy[i] = (random(1) < 0.5) ? -1 : 1;
  }
}
```

Ahora en draw(), se pinta un rectángulo semitransparente para el "rastro" de las bolas y luego, para cada bola, se actualiza la posición, se detectan colisiones con los bordes y se invierte la velocidad del eje correspondiente, corrigiendo la penetración.

```
void draw() {
  // efecto de rastro
  fill(0, 30);
  rect(0, 0, width, height);
  fill(255);

  for (int i = 0; i < n; i++) {
    // actualizar
    px[i] += vx[i];
    py[i] += vy[i];

    // colisiones con paredes
    float r = diam/2.0;
    if (px[i] + r > width)  { px[i] = width - r;  vx[i] *= -1; }
    if (px[i] - r < 0)      { px[i] = r;          vx[i] *= -1; }
    if (py[i] + r > height) { py[i] = height - r; vy[i] *= -1; }
    if (py[i] - r < 0)      { py[i] = r;          vy[i] *= -1; }
```

Al final de draw(), se dibuja cada bola con ellipse(px[i], py[i], diam, diam).

```
    // dibujar
    ellipse(px[i], py[i], diam, diam);
  }
}
```

Finalmente, se ejecuta el código y se ajusta la cantidad de bolas (n) o su diámetro (diam) si se desea más o menos densidad visual.

Práctica 5-2

Primero se deben declarar las variables para ángulos y velocidades de los planetas y de las lunas:

```
float angP1=0, angP2=PI/3, angP3=2*PI/3, angL1=0, angL2=PI;
float velP1=0.10, velP2=0.05, velP3=0.025, velL1=0.10, velL2=0.05;
```

En setup(), se establece el tamaño del lienzo con size(500,500) y se configuran las órbitas con stroke(180) y con noFill() para no rellenarlas.

```
void setup(){
  size(500, 500);
  stroke(180);
  noFill();
}
```

En draw(), se limpia el lienzo con background(0) y se centra el sistema con translate(width/2, height/2).

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

Luego se dibuja el sol en el centro de esta manera:

```
// --- Sol ---
noStroke();                    // Quita el contorno para que el sol sea sólido
fill(241,250,3);               // Color del sol
ellipse(0,0, 30,30);           // Dibuja el sol en el centro
stroke(80);                    // Restaura el contorno gris
noFill();                      // Las órbitas son solo círculos vacíos
```

Después se definen los radios orbitales: r1, r2, r3. Y se dibujan sus órbitas con ellipse(0,0,2*r,2*r).

```
// --- Radios orbitales de los planetas ---
float r1 = width*0.12, r2 = width*0.22, r3 = width*0.32;

// --- Dibujar órbitas de referencia ---
ellipse(0,0, r1*2, r1*2); // órbita del planeta 1
ellipse(0,0, r2*2, r2*2); // órbita del planeta 2
ellipse(0,0, r3*2, r3*2); // órbita del planeta 3
```

Finalmente, se dibujan los planetas:

- Para cada planeta se debe guardar el sistema de coordenadas actual con pushMatrix(), se rota alrededor del centro según el angulo con rotate(angP1 += velP1);, se desplaza hasta el radio de su órbita con translate(r1, 0), se define su color y se quita el contorno con noStroke() y fill(5,250,3), se dibuja el planeta con ellipse(0,0, 16,16) y por ultimo se restaura las coordenadas originales con popMatrix().

```
// --- Planeta 1 ---
pushMatrix();                  // Guarda sistema de coordenadas actual
rotate(angP1 += velP1);        // Rota alrededor del centro según ángulo
translate(r1, 0);              // Desplaza hasta el radio de su órbita
noStroke(); fill(5,250,3);     // Color verde brillante
ellipse(0,0, 16,16);           // Planeta como círculo sólido
popMatrix();                   // Restaura coordenadas originales
```

```
  // --- Planeta 3 ---
  pushMatrix();
  rotate(angP3 += velP3);          // Rota en su órbita
  translate(r3, 0);                // Se coloca en su radio
  noStroke(); fill(7,88,6);        // Color verde oscuro
  ellipse(0,0, 16,16);             // Dibuja planeta 3
  popMatrix();
}
```

- Para el planeta con lunas se repite el paso anterior añadiendo dentro de su pushMatrix() las órbitas locales (círculos pequeños) de las lunas.

```
// --- Planeta 2 con lunas ---
pushMatrix();
rotate(angP2 += velP2);          // Rota en su órbita
translate(r2, 0);                // Coloca planeta 2 en su radio
noStroke(); fill(11,160,10);     // Color verde más oscuro
ellipse(0,0, 16,16);             // Dibuja planeta 2

// órbitas locales de sus lunas
stroke(100); noFill();
ellipse(0,0, 36,36); // órbita de luna 1
ellipse(0,0, 60,60); // órbita de luna 2
```

Y luego se dibujan las lunas de igual forma que los planetas. Despues de dibujar las lunas se usa un popMatrix(), quedando de la siguiente manera:

```
// --- Planeta 2 con lunas ---
pushMatrix();
rotate(angP2 += velP2);       // Rota en su órbita
translate(r2, 0);             // Coloca planeta 2 en su radio
noStroke(); fill(11,160,10);  // Color verde más oscuro
ellipse(0,0, 16,16);          // Dibuja planeta 2

// órbitas locales de sus lunas
stroke(100); noFill();
ellipse(0,0, 36,36); // órbita de luna 1
ellipse(0,0, 60,60); // órbita de luna 2

// --- Luna 1 ---
pushMatrix();
rotate(angL1 += velL1);       // Rota alrededor del planeta
translate(18,0);              // Se aleja 18 píxeles del planeta
noStroke(); fill(8,228,255);  // Color celeste
ellipse(0,0, 6,6);            // Dibuja luna 1
popMatrix();

// --- Luna 2 ---
pushMatrix();
rotate(angL2 += velL2);       // Rota alrededor del planeta
translate(30,0);              // Se aleja 30 píxeles del planeta
noStroke(); fill(17,137,152); // Color azul verdoso
ellipse(0,0, 6,6);            // Dibuja luna 2
popMatrix();

popMatrix();                  // Fin planeta 2 y sus lunas
```


Boulevard Tecnológico de Progreso SN x 62, Progreso, Yucatán, C.P. 97320 Tel. 969 934 3023 E-mail: dir01_dprogreso@tecnm.mx www.progreso.tecnm.mx

Práctica 5-3

Retomando el código de la práctica 5-2, primero se declara el ángulo, la velocidad y el radio de la orbita del sol, junto con las variables de los planetas y las lunas.

```
// Ángulo, velocidad y radio de la órbita del SOL
float angSol = 0;      // ángulo actual del sol
float velSol = 0.01;   // velocidad angular del sol
float rSol  = 40;      // radio de la órbita del sol (respecto al centro de la pantalla)

// Variables de ángulos iniciales de planetas y lunas
float angP1=0, angP2=PI/3, angP3=2*PI/3, angL1=0, angL2=PI;
// Velocidades angulares de cada planeta/luna
float velP1=0.10, velP2=0.05, velP3=0.025, velL1=0.10, velL2=0.05;
```

Luego en void setup }, se elimina stroke(180) y noFill(), dejando solo size(500, 500)

```
void setup(){
  size(500, 500);       // Tamaño de la ventana
}
```

Ahora en draw() y después de background(0) y translate(width/2,height/2), se dibuja la orbita del sol (esta parte se puede omitir si solo se desea ver el movimiento del sol sin su órbita).

```
// --- Dibuja la órbita del sol ---
stroke(80);           // Contorno gris tenue para la órbita
noFill();             // Solo contorno, sin rellenar
ellipse(0,0, rSol*2, rSol*2); // Circunferencia con radio rSol
```

Debajo de la configuración de la orbita del sol, se debe guardar el sistema de coordenadas global para que todos los planetas giren alrededor del sol utilizando pushMatrix().

```
// --- SISTEMA RELATIVO AL SOL ---
pushMatrix();                 // Guarda el sistema de coordenadas global
rotate(angSol += velSol);     // Gira alrededor del centro (avance de ángulo del sol)
translate(rSol, 0);           // Traslada al sol hasta su posición en la órbita
```

Para terminar con el sol, se actualiza su configuración de la practica anterior, simplemente quitando el stroke(80) y noFill() (selecciónalos y oprime ctrl+x) dejando solo noStroke(), fill(241,250,3) y ellipse(0,0, 30,30).

```
// --- Sol ---
noStroke();                   // Sin contorno para disco sólido
fill(241,250,3);              // Amarillo brillante para el sol
ellipse(0,0, 26,26);          // Dibuja el sol en su posición actual
```

Finalmente, se debe actualizar la configuración de las orbitas de los planetas sin usar el tamaño del lienzo (width) y se pone el stroke(80) y noFill() previamente eliminado (oprime ctrl+v).

```
// --- Radios orbitales de los planetas ---
float r1 = 70, r2 = 120, r3 = 170;
stroke(80);                     // Restaura el contorno gris
noFill();                       // Las órbitas son solo círculos vacíos
```

Luego se retoma (copia y pega) la configuración de las orbitas y las características de cada planeta, agregando al final un popMatrix() para cerrar el sistema relativo al sol.

```
  // --- Planeta 3 ---
  pushMatrix();
  rotate(angP3 += velP3);       // Rota en su órbita
  translate(r3, 0);             // Se coloca en su radio
  noStroke(); fill(7,88,6);     // Color verde oscuro
  ellipse(0,0, 16,16);          // Dibuja planeta 3
  popMatrix();

  popMatrix();                   // <- MUY IMPORTANTE: cerramos el sistema relativo al sol
  // A partir de aquí se vuelve al sistema centrado en pantalla,
  // sin las transformaciones del sol y sus planetas.
}
```

Practica 7-1

En el código original, se añade una nueva variable al principio del código llamada zCam que guarda la posición de la cámara en Z.

```
float rotX=0, rotY=0, distX=0, distY=0; // variables para rotación
int lastX, lastY;                      // almacenan la última posición del ratón
float zCam = 0;                        // posición de la cámara en Z (0 cerca, -500 lejos)
```

Luego en void draw(), se agrega la variable zCam en el translate(width/2, height/2).

```
void draw(){
  background(0);                 // Fondo negro
  lights();                      // Luces básicas para dar volumen
  translate(width/2, height/2, zCam); // centra el cubo y aplica zoom con zCam
  rotateX(rotX + distY);         // rota en eje X (vertical)
  rotateY(rotY + distX);         // rota en eje Y (horizontal)
  box(100,100,100);              // dibuja cubo de 100 px
}
```

Por último, se crea la función llamada keyPressed() para que se pueda acercar y alejar el cubo, definiendo que se aleje o se acerque 10 en 10.

```
// Con UP/DOWN modificamos zCam para acercar/alejar la cámara*****
void keyPressed(){
  if (key == CODED){
    if (keyCode == UP)   zCam = min(0, zCam + 10);      // acercar
    if (keyCode == DOWN) zCam = max(-500, zCam - 10);   // alejar
  }
}
```

Práctica 7-2

Con el código original, en las variables se agregan unas nuevas para la interacción con el mose.

```
// Dibujo de una función 3D
import processing.opengl.*;          // Soporte OpenGL (P3D usa OpenGL por debajo)

// ---------- Variables de interacción (rotación con el ratón) ----------
float rotX = 0.0, rotY = 0.0;        // rotación acumulada en X e Y
int lastX, lastY;                    // última posición del ratón presionado
float distX = 0.0, distY = 0.0;      // delta de arrastre (en radianes)

// ---------- Resolución y escalas ----------
int steps = 50;                      // resolución de la malla (cuantos quads por lado)
float scaleZ = 200.0;                // escala del relieve (altura Z)
float zoomZ  = -300.0;               // alejamiento de la "cámara" (translate en Z)
float gX = 500.0, gY = 500.0;        // tamaño "en píxeles" de la malla en X e Y
```

Luego en setup(), en size() se cambia OPENGL por P3D, ya que OPENGL es obsoleta y ahora se usa P3D. Tambien se debe eliminar el noFill() y en cambio agregar smooth(8) para el antialiasing y noStroke() para que la superficie se dibuje sin contorno.

```
void setup() {
  size(500, 500, P3D);               // renderer P3D (3D)
  smooth(8);                         // antialiasing
  noStroke();                        // la superficie se dibuja sin contorno; lo activamos sólo para ejes
}
```

Tambien se debe agregar la función de la superficie:

```
// ---------- Función de la superficie (x,y ∈ [0,1]) ----------
float funcion(float x, float y) {
  return x*x*x + y*y*y;              // curva suave; puedes cambiarla libremente
}
```

Ahora en draw(), se configura la iluminación ambiente como se ve en la imagen:

```
// --- Iluminación: ambient + dos direccionales para sombreado suave ---
ambientLight(90, 90, 90);          // luz ambiental (suaviza sombras)
directionalLight(215,215,215, -0.6, -0.5, -1); // luz principal (arriba-izquierda)
directionalLight(60,60,60,     0.4,  0.2,  1); // luz de relleno
```

Seguidamente se pone la configuración de encuadre y de vista del código original pero sin el scale(gX, gY, scaleZ), (se usara mas adelante).

```
// --- Iluminación: ambient + dos direccionales para sombreado suave ---
ambientLight(90, 90, 90);         // luz ambiental (suaviza sombras)
directionalLight(215,215,215, -0.6, -0.5, -1); // luz principal (arriba-izquierda)
directionalLight(60,60,60,     0.4,  0.2,  1); // luz de relleno

// --- Encadre y controles de vista ---
translate(gX/2, gY/2, zoomZ);     // centra la escena y aplica "zoom" z
rotateY(rotY + distX);            // rotación interactiva horizontal
rotateX(rotX + distY);            // rotación interactiva vertical
translate(-gX/2, -gY/2);          // vuelve a poner el (0,0) en la esquina superior-izquierda
```

Despues se configura la superficie usando pushMatrix y adentro de eso ponemos el scale(gX, gY, scaleZ) que se quito anteriormente, se configura la superficie y se llama la función dibujarFuncion(). Al final se pone popMatrix().

```
// ================== SUPERFICIE ==================
pushMatrix();
  // Pasar de coordenadas normalizadas [0..1] a píxeles (gX,gY) y altura (scaleZ)
  scale(gX, gY, scaleZ);

  // Material: gris mate con un poco de componente especular
  noStroke();
  fill(175);                      // color base gris
  specular(40);                   // leve brillo
  shininess(6);                   // tamaño del "highlight"

  dibujarFuncion();               // emite la malla con QUAD_STRIP
popMatrix();
```

Después de eso, se debe de reconfigurar los ejes ya que en el código original son demasiado grandes y además hace que se distorsionen o que no se vean bien, por eso, se realiza la siguiente configuración:

```
  // ================== EJES FINOS ANCLADOS ==================
  // Dibujamos los ejes SIN scale(...) para que strokeWeight sea en píxeles reales.
  // Anclaje en la esquina SUPERIOR-IZQUIERDA de la malla: (ax=0, ay=0)
  float ax = 0.0, ay = 0.0;
  float axPix = ax * gX;            // convertir a coordenadas de pantalla (porque no estamos escalando)
  float ayPix = ay * gY;
  float azPix = funcion(ax, ay) * scaleZ;

  strokeWeight(2);                  // grosor real (en píxeles de pantalla)

  // Eje X (rojo) — hacia +X (derecha). Lo hiciste largo (900 px) a propósito.
  stroke(255, 0, 0);
  line(axPix, ayPix, azPix,  axPix + 900, ayPix, azPix);

  // Eje Z (azul) — hacia +Z (hacia el observador). Largo (900 px).
  stroke(0, 90, 255);
  line(axPix, ayPix, azPix,  axPix, ayPix, azPix + 900);

  // Eje Y (verde) — hacia +Y (hacia abajo en pantalla). Largo (500 px).
  stroke(0, 255, 0);
  line(axPix, ayPix, azPix,  axPix, ayPix + 500, azPix);
}
```

Se utiliza tal cual la función dibujarFuncion() del código original:

```
// ---------- Construcción de la malla con tiras de cuadros ----------
void dibujarFuncion() {
  float x, y;
  int i, j;
  float in_steps = 1.0 / steps;     // paso uniforme en [0..1]

  // Precalcular las alturas z en una matriz (evita recalcular funcion(x,y) en cada vertex)
  float[][] matriz = new float[steps+1][steps+1];

  for (y = 0.0, j = 0; y <= 1.0; y += in_steps, j++) {
    for (x = 0.0, i = 0; x <= 1.0; x += in_steps, i++) {
      matriz[i][j] = funcion(x, y);
    }
  }

  // Emitir la superficie por franjas (QUAD_STRIP conecta pares de vértices)
  for (j = 0, y = 0.0; j < steps; j++, y += in_steps) {
    beginShape(QUAD_STRIP);
    for (i = 0, x = 0.0; i <= steps; i++, x += in_steps) {
      vertex(x, y,            matriz[i][j]);     // punto en fila j
      vertex(x, y + in_steps, matriz[i][j+1]);   // punto en fila j+1
    }
    endShape();
  }
}
```

Y finalmente, se configuran los controles de rotación con el mouse:

```
// ---------- Controles de rotación con el ratón ----------
void mousePressed() {
  lastX = mouseX; lastY = mouseY;   // almacena dónde empezó el arrastre
}
void mouseDragged() {
  // delta en píxeles convertido a radianes para rotación suave
  distX = radians(mouseX - lastX);
  distY = radians(lastY - mouseY);
}
void mouseReleased() {
  // acumula rotaciones y reinicia deltas
  rotX += distY;
  rotY += distX;
  distX = distY = 0.0;
}
```

Práctica 7-3

Tomando el código de la practica 7-2, se debe agregar las variables minZ y maxZ para el rango de alturas de los colores. Tambien se saca la función de la superficie de setup() y se pone después de las variables.

```
// ---------- Resolución y escalas ----------
int steps = 50;                 // densidad de la malla
float scaleZ = 200;             // relieve (alto en Z)
float zoomZ  = -300;            // "acercar/alejar" en Z
float gX=500, gY=500;           // tamaño en píxeles de X e Y

// ---------- Rango de alturas para normalizar colores ----------
float minZ, maxZ;

// Función de la superficie en [0,1]x[0,1]
float funcion(float x, float y){
  return x*x*x + y*y*y;
}

void setup(){
  size(500,500,P3D);
```

Luego en la función setup(), se agrega el llamado de la función calcMinMax(), que se encarga de precalcular el minZ y el maxZ.

```
void setup(){
  size(500,500,P3D);
  smooth(8);
  noStroke();                   // sin contorno para la malla
  // precalcula minZ y maxZ en el dominio [0,1]
  calcMinMax();
}
```

Despues se rediseña la función dibujarFuncion() del código anterior y ahora se le llamara dibujarFuncionGradiente() con la siguiente configuración:

```
// Emite la malla con color por vértice según altura normalizada
void dibujarFuncionGradiente(){
  float in_steps = 1.0/steps;

  // Recorremos por franjas en Y usando QUAD_STRIP
  for (int j=0; j<steps; j++){
    float y  = j*in_steps;
    float y2 = (j+1)*in_steps;

    beginShape(QUAD_STRIP);
    for (int i=0; i<=steps; i++){
      float x = i*in_steps;

      // Vértice 1 (fila j)
      float z1 = funcion(x, y);
      float t1 = map(z1, minZ, maxZ, 0, 1);  // normaliza a 0..1
      fill( lerpColor(color(255,0,0), color(255,255,0), t1) );
      vertex(x, y, z1);

      // Vértice 2 (fila j+1)
      float z2 = funcion(x, y2);
      float t2 = map(z2, minZ, maxZ, 0, 1);
      fill( lerpColor(color(255,0,0), color(255,255,0), t2) );
      vertex(x, y2, z2);
    }
    endShape();
  }
}
```

Finalmente, se configura la función calcMinMax():

```
// Calcula minZ y maxZ de la función en una rejilla (para normalizar color)
void calcMinMax(){
  minZ =  1e9;
  maxZ = -1e9;
  float in_steps = 1.0/steps;
  for (int j=0; j<=steps; j++){
    float y = j*in_steps;
    for (int i=0; i<=steps; i++){
      float x = i*in_steps;
      float z = funcion(x,y);
      if (z<minZ) minZ=z;
      if (z>maxZ) maxZ=z;
    }
  }
}
```

Práctica 7-4

Tomando como base el código de la practica anterior, primero se eliminan los apartados de "Resolución y escalas", "Rango de alturas para normalizar colores" y "Función de la superficie en [0,1]x[0,1]", dejando solo las variables de la interacción y añadiendo nuevas variables que son solo para la malla.

```
import processing.opengl.*;

// ----- Interacción -----
float rotX = 0.0, rotY = 0.0;       // rotación acumulada en X e Y
int lastX, lastY;                   // última posición del ratón presionado
float distX = 0.0, distY = 0.0;     // delta de arrastre (en radianes)

// ----- Configuración de la malla y su resolución -----
float zoom=-50;          // alejar cámara
float alturaZ=100;         // altura de los relieves
int steps=100;           //resolución
PImage img;              //imagen que se usara
```

Luego en la función setup, se elimina el smooth(8), el noStroke() y el llamado de la función calcMinMax(). Em cambio se añadira la función stroke(70, 200, 70), la función noFill() y también se debe carga la imagen a utilizar ajustando su tamaño al de la malla.

```
void setup(){
  size(500,500,P3D);
  stroke(70, 200, 70);
  noFill();

  // --- Carga y preproceso de imagen ---
  img = loadImage("Ejemplo.jpg");      //cargar imagen
  img.resize(steps, steps);            //ajustarlo al tamaño de la malla
}
```

En la función draw() eliminamos los apartados "Superficie con color por vértice", Opcional) Ejes largos anclados en la esquina sup-izq. Despues de eso, centramos la imagen.

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

  // Iluminación básica (sombreado suave)
  ambientLight(90,90,90);
  directionalLight(215,215,215, -0.6,-0.5,-1);
  directionalLight(60,60,60,     0.4, 0.2, 1);

  translate(width/2, height/2, zoom);
  rotateX(rotX + distY);
  rotateY(rotY + distX);

  // Centramos el dibujo
  translate(-img.width*2, -img.height*2);
```

Finalmente, eliminamos la función dibujarFuncionGradiente() y la función calcMinMax() y en cambio ponemos la siguiente configuración que es para dibujar la superficie a partir de la imgen:

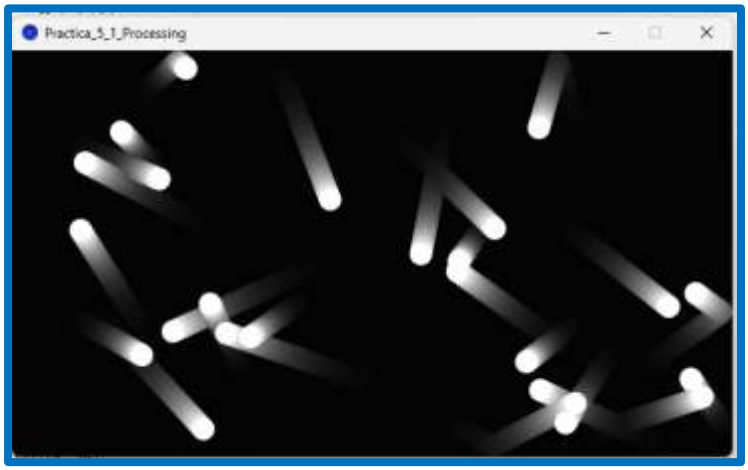
```
  // Dibujamos la superficie a partir de la imagen
  for (int y = 0; y < img.height-1; y++) {
    beginShape(TRIANGLE_STRIP);
    for (int x = 0; x < img.width; x++) {
      // Obtenemos el brillo del píxel actual y el de la fila de abajo
      float brillo1 = brightness(img.get(x, y));
      float brillo2 = brightness(img.get(x, y+1));

      // Mapeamos el brillo (0-255) a la altura en Z
      float z1 = map(brillo1, 0, 255, -alturaZ, alturaZ);
      float z2 = map(brillo2, 0, 255, -alturaZ, alturaZ);

      // Creamos los dos vértices para la tira de triángulos/cuadriláteros
      vertex(x*4, y*4, z1);
      vertex(x*4, (y+1)*4, z2);
    }
    endShape();
  }
}
```

Materiales y equipos utilizados

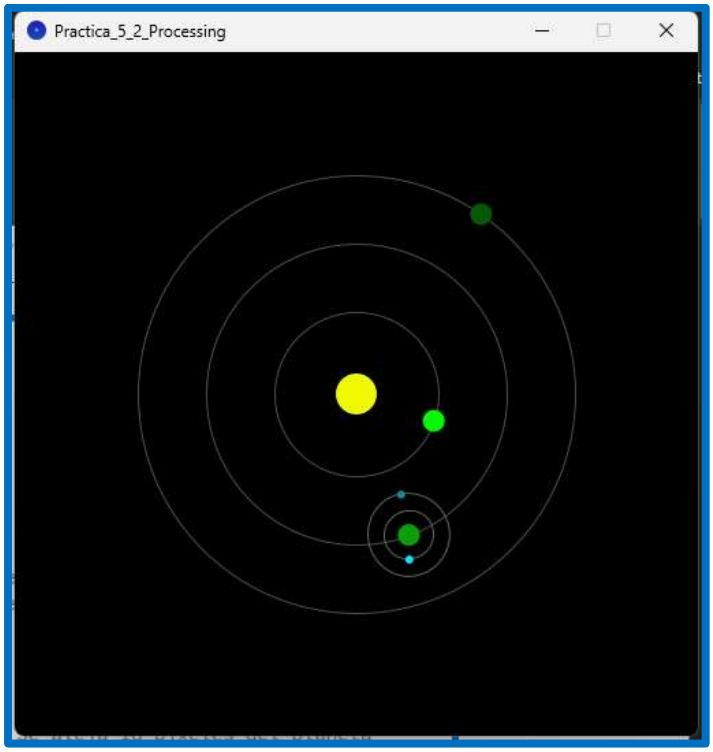
- Laptop
- Processing
- Archivo PDF del profesor con información del tema
- Pagina oficial de Processing

Resultados

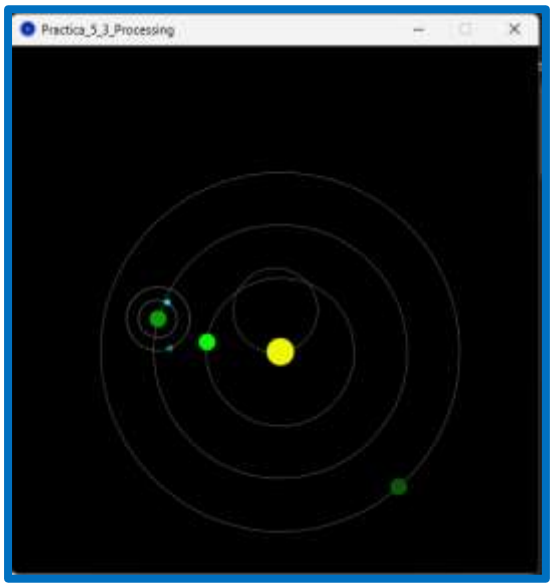
En la práctica 5-1 fue posible observar el movimiento simultáneo de múltiples esferas que rebotaban en las paredes de la ventana, generando un efecto visual dinámico y continuo.



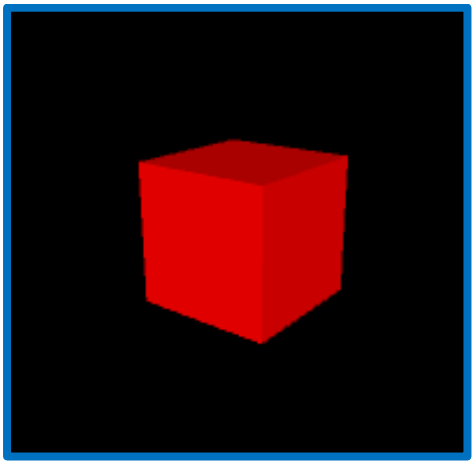
En la práctica 5-2, los planetas y lunas se desplazaban siguiendo trayectorias circulares, con órbitas claramente visibles.



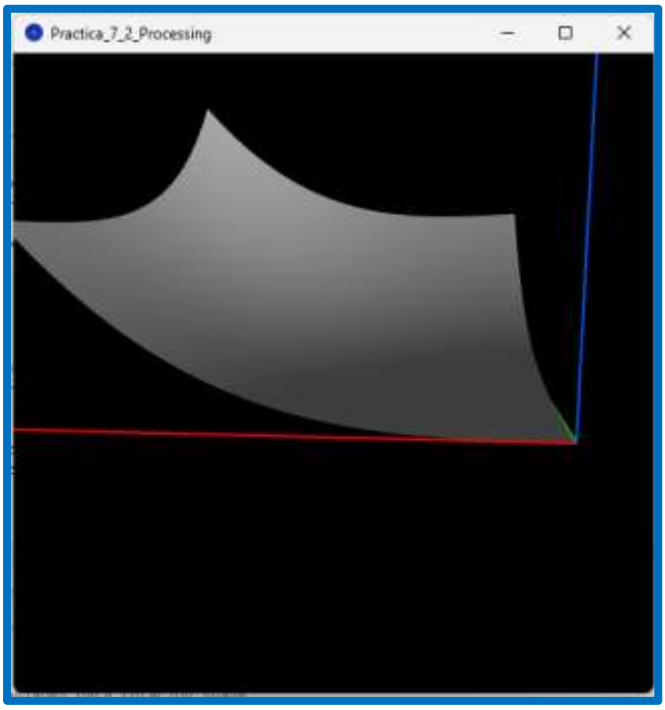
La práctica 5-3 permitió visualizar cómo el sol mismo recorría una órbita y arrastraba consigo a todo el sistema planetario, confirmando la efectividad del uso de transformaciones jerárquicas.



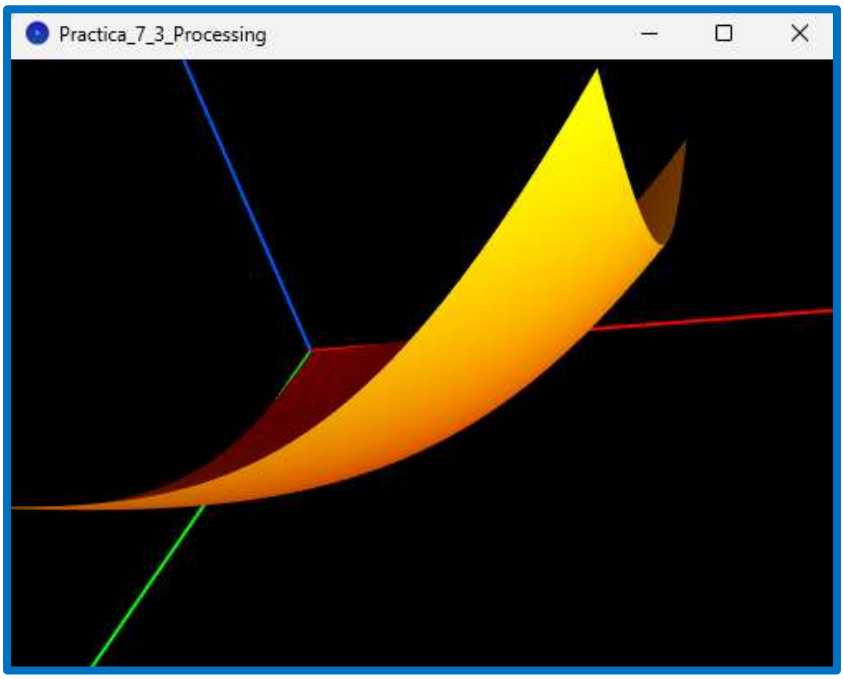
La práctica 7-1 mostró un cubo que podía rotar con el ratón y modificarse en profundidad mediante las teclas de dirección, lo que otorgaba una experiencia interactiva.



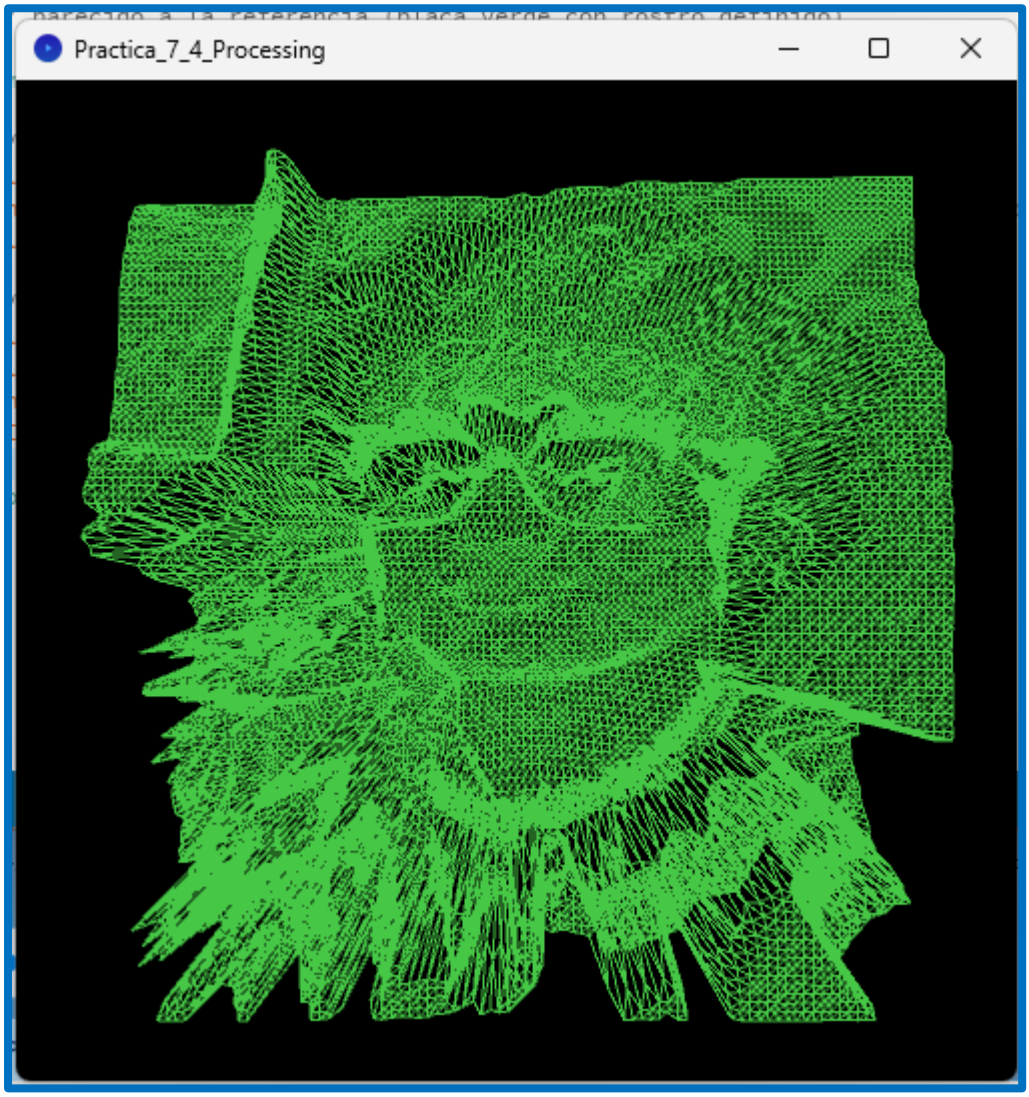
En la práctica 7-2 se logró representar una superficie sólida iluminada, donde las irregularidades matemáticas de la función eran visibles gracias a la luz aplicada.



La práctica 7-3 enriqueció el modelo anterior aplicando colores según la altura de la superficie, de modo que los tonos rojos correspondían a los niveles más bajos y los amarillos a los más altos.



En la práctica 7-4, el uso de un mapa de alturas permitió generar una malla tridimensional cuyo relieve reproducía fielmente las variaciones de luminosidad de la imagen de entrada.



Análisis

El análisis de los resultados muestra cómo los conceptos teóricos se aplican de manera efectiva en Processing. En el caso de las animaciones, el manejo de arreglos permitió escalar fácilmente de un objeto individual a varios, manteniendo un control eficiente sobre las colisiones y movimientos. Asimismo, la incorporación de órbitas y jerarquías de transformación evidenció la importancia de organizar correctamente las matrices de referencia para lograr movimientos dependientes y coherentes.

En los ejercicios de 3D, el empleo de QUAD_STRIP fue fundamental para la construcción de superficies, mientras que la iluminación y el color permitieron añadir una dimensión perceptiva adicional. La práctica 7-3 demostró que un simple gradiente de color puede mejorar significativamente la interpretación de los datos en un modelo, y la práctica 7-4 validó la versatilidad del entorno al traducir una imagen en un terreno virtual. La interacción implementada en la práctica 7-1 refuerza la utilidad de los eventos de teclado y ratón para construir experiencias más inmersivas.

Discusión

Los resultados obtenidos reflejan no solo el cumplimiento de los objetivos planteados, sino también la oportunidad de identificar mejoras y extensiones. En animación, la simulación de partículas podría enriquecerse con efectos físicos como gravedad o fricción, lo que daría mayor realismo. En el sistema planetario, incorporar variaciones en las velocidades o trayectorias elípticas ampliaría la complejidad del modelo. En cuanto a las superficies 3D, un siguiente paso sería la implementación de normales calculadas por vértice para optimizar el sombreado y ofrecer una iluminación más realista. Además, en la práctica 7-4 podría integrarse el uso de texturas para complementar el relieve con información visual adicional, generando un entorno aún más cercano a la realidad. Estos aspectos señalan el potencial de Processing no solo como herramienta educativa, sino como plataforma de experimentación creativa.

Conclusión

Las prácticas consolidan habilidades esenciales en Processing: control del ciclo de animación, gestión de eventos de teclado y ratón, composición con transformaciones y construcción de mallas tridimensionales. El desarrollo detallado permitió constatar cómo la teoría se traduce en visualizaciones concretas y dinámicas, resaltando la relación entre algoritmos y representación gráfica. Los resultados muestran que, con una base bien estructurada, es posible escalar hacia proyectos más complejos, integrando física, texturas y modelos de iluminación avanzados. En suma, este conjunto de prácticas constituye un paso fundamental en la formación hacia la programación visual y la exploración creativa de entornos digitales.

Referencias

- Processing. (s. f.). P3D: A 3D rendering mode in Processing. Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/tutorials/p3d
- Processing. (s. f.). keyCode. Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/keyCode.html
- Processing. (s. f.). PImage. Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/PImage.html
- Processing. (s. f.). beginShape(). Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/beginShape_.html
- Processing. (s. f.). lights(). Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/lights_.html
- Processing. (s. f.). translate(). Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/translate_.html
- Processing. (s. f.). size(). Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/size_.html
- Processing. (s. f.). keyPressed(). Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/keyPressed_.html
- Processing. (s. f.). frameRate(). Processing. Recuperado el 12 de octubre de 2025, de https://processing.org/reference/frameRate_.html