```
float angle = 0; // Ángulo de rotación global

// Constantes para configuración
final int WINDOW_WIDTH = 800;
final int WINDOW_HEIGHT = 600;
final int SHAPE_SPACING = 100; // Valor ajustado para reducir la distancia entre figuras
final float ROTATION_SPEED = 0.01;
final int NUM_SHAPE_TYPES = 13; // Total de figuras

// Variables para indicar si se deben mostrar los grupos de figuras
boolean showKeyShapes = false;
boolean showMouseShapes = false;

void setup() {
  size(1000, 1000, P3D);
  noFill(); // Inicialmente sin relleno para ver mejor la estructura
}

void draw() {
  setupScene();

  drawContainerBox(); // Dibuja el contenedor alrededor de las figuras

  // Activa los grupos de figuras según las interacciones
  if (showKeyShapes) {
    drawShapes(true);  // Figuras activadas por keyPressed
  }
  if (showMouseShapes) {
    drawShapes(false); // Figuras activadas por mousePressed
```

```
  }

  angle += ROTATION_SPEED;
}

void setupScene() {
  background(200);
  translate(width/2, height/2, 0);
  rotateY(angle);
  rotateX(angle * 0.5);
}

// Función para dibujar las figuras según el grupo
void drawShapes(boolean byKey) {
  int numFigures = NUM_SHAPE_TYPES / 2; // Mitad para cada grupo

  for (int i = 0; i < numFigures; i++) {
    pushMatrix();

    // Posición en cruz: figuras de keyPressed a la izquierda, mousePressed a la derecha
    if (byKey) {
      translate(-SHAPE_SPACING, i * SHAPE_SPACING - (numFigures / 2) * SHAPE_SPACING, 0);
      drawSelectedShape(i); // Dibuja la figura correspondiente a keyPressed
    } else {
      translate(SHAPE_SPACING, i * SHAPE_SPACING - (numFigures / 2) * SHAPE_SPACING, 0);
      drawSelectedShape(i + numFigures); // Dibuja la figura correspondiente a mousePressed
    }

    popMatrix();
```

```
  }
}

void drawSelectedShape(int index) {
  pushMatrix(); // Guardar la transformación actual

  // Dibuja un cubo contenedor alrededor de la figura
  drawContainerBox();

  // Configuración de color para cada figura
  float r = random(255);
  float g = random(255);
  float b = random(255);
  fill(r, g, b, 300); // Añadido algo de transparencia
  stroke(0);

  // Dibuja la figura seleccionada
  switch (index % NUM_SHAPE_TYPES) {
    case 0:
      sphere(40);
      break;
    case 1:
      box(40);
      break;
    case 2:
      cylinder(25, 60);
      break;
    case 3:
      pyramid(40, 60);
```

```
    break;
  case 4:
    octahedron(40);
    break;
  case 5:
    dodecahedron(40);
    break;
  case 6:
    torus(30, 10);
    break;
  case 7:
    pentagon(40);
    break;
  case 8:
    hexagon(40);
    break;
  case 9:
    rectangle2D(60, 40);
    break;
  case 10:
    rectangle3D(60, 40);
    break;
  case 11:
    triangularPrism(40, 60);
    break;
  case 12:
    cone(30, 60);
    break;
}
```

```
  popMatrix(); // Restaurar la transformación
}

// Función para dibujar el cubo contenedor
void drawContainerBox() {
  stroke(0, 128);
  noFill(); // Sin relleno para el cubo contenedor
  box(100); // Tamaño del cubo contenedor
  translate(0, -50, 0); // Ajuste de posición si es necesario
}



// Detecta la tecla presionada para activar las figuras de keyPressed
void keyPressed() {
  showKeyShapes = true;
}

// Detecta el clic del ratón para activar las figuras de mousePressed
void mousePressed() {
  showMouseShapes = true;
}

// Aquí van las funciones para crear cada tipo de figura...
// cylinder(), pyramid(), octahedron(), dodecahedron(), torus(), pentagon(), hexagon(),
rectangle2D(), rectangle3D(), triangularPrism(), cone()

// Función mejorada para crear un cilindro
void cylinder(float radius, float height) {
```

```
  int sides = 24;
  float angleStep = TWO_PI / sides;

  // Caras laterales
  beginShape(TRIANGLE_STRIP);
  for (int i = 0; i <= sides; i++) {
    float angle = i * angleStep;
    float x = radius * cos(angle);
    float z = radius * sin(angle);
    vertex(x, -height/2, z);
    vertex(x, height/2, z);
  }
  endShape();

  // Tapas superior e inferior
  for (int y = -1; y <= 1; y += 2) {
    beginShape(TRIANGLE_FAN);
    vertex(0, y * height/2, 0);
    for (int i = 0; i <= sides; i++) {
      float angle = i * angleStep;
      vertex(radius * cos(angle), y * height/2, radius * sin(angle));
    }
    endShape();
  }
}

// Función mejorada para crear una pirámide
void pyramid(float baseSize, float height) {
  float halfBase = baseSize/2;
```

```
  // Base
  beginShape();
  vertex(-halfBase, 0, -halfBase);
  vertex(halfBase, 0, -halfBase);
  vertex(halfBase, 0, halfBase);
  vertex(-halfBase, 0, halfBase);
  endShape(CLOSE);

  // Caras laterales
  beginShape(TRIANGLES);
  // Frente
  vertex(0, -height, 0);
  vertex(-halfBase, 0, halfBase);
  vertex(halfBase, 0, halfBase);
  // Derecha
  vertex(0, -height, 0);
  vertex(halfBase, 0, halfBase);
  vertex(halfBase, 0, -halfBase);
  // Atrás
  vertex(0, -height, 0);
  vertex(halfBase, 0, -halfBase);
  vertex(-halfBase, 0, -halfBase);
  // Izquierda
  vertex(0, -height, 0);
  vertex(-halfBase, 0, -halfBase);
  vertex(-halfBase, 0, halfBase);
  endShape();
}
```

```
// Función mejorada para crear un octaedro
void octahedron(float size) {
  float halfSize = size/2;

  beginShape(TRIANGLES);
  // Parte superior
  for (int i = 0; i < 4; i++) {
    float angle = i * PI/2;
    float nextAngle = (i + 1) * PI/2;
    vertex(0, -size, 0);
    vertex(halfSize * cos(angle), 0, halfSize * sin(angle));
    vertex(halfSize * cos(nextAngle), 0, halfSize * sin(nextAngle));
  }

  // Parte inferior
  for (int i = 0; i < 4; i++) {
    float angle = i * PI/2;
    float nextAngle = (i + 1) * PI/2;
    vertex(0, size, 0);
    vertex(halfSize * cos(angle), 0, halfSize * sin(angle));
    vertex(halfSize * cos(nextAngle), 0, halfSize * sin(nextAngle));
  }
  endShape();
}

// Función para crear un dodecaedro
void dodecahedron(float size) {
  float phi = (1 + sqrt(5))/2; // Proporción áurea
```

```
  float a = size/2;
  float b = size/(2 * phi);
  float c = size * (phi-1)/2;

  // Vertices del dodecaedro
  PVector[] vertices = new PVector[] {
    new PVector(0, a, c),
    new PVector(a, c, 0),
    new PVector(c, 0, a),
    new PVector(b, b, b)
  };

  // Caras del dodecaedro (12 caras pentagonales)
  beginShape(TRIANGLES);
  // Aquí se dibujarían las caras usando los vértices
  // Por simplicidad, dibujamos una aproximación usando triángulos
  for (int i = 0; i < 12; i++) {
    float angle = i * TWO_PI/12;
    float nextAngle = (i + 1) * TWO_PI/12;
    vertex(0, -size/2, 0);
    vertex(size/2 * cos(angle), 0, size/2 * sin(angle));
    vertex(size/2 * cos(nextAngle), 0, size/2 * sin(nextAngle));
  }
  endShape();
}

// Función para crear un toro
void torus(float outerRadius, float innerRadius) {
  int sides = 24;
```

```
  int rings = 16;

  for (int i = 0; i < sides; i++) {
    float phi = map(i, 0, sides, 0, TWO_PI);
    float nextPhi = map(i + 1, 0, sides, 0, TWO_PI);

    beginShape(TRIANGLE_STRIP);
    for (int j = 0; j <= rings; j++) {
      float theta = map(j, 0, rings, 0, TWO_PI);

      float x1 = (outerRadius + innerRadius * cos(theta)) * cos(phi);
      float y1 = innerRadius * sin(theta);
      float z1 = (outerRadius + innerRadius * cos(theta)) * sin(phi);

      float x2 = (outerRadius + innerRadius * cos(theta)) * cos(nextPhi);
      float y2 = innerRadius * sin(theta);
      float z2 = (outerRadius + innerRadius * cos(theta)) * sin(nextPhi);

      vertex(x1, y1, z1);
      vertex(x2, y2, z2);
    }
    endShape();
  }
}

// Función para crear un pentágono
void pentagon(float size) {
  float radius = size/2;
  beginShape();
```

```
  for (int i = 0; i < 5; i++) {
    float angle = TWO_PI * i / 5 - PI/2;
    vertex(radius * cos(angle), radius * sin(angle), 0);
  }
  endShape(CLOSE);

  // Crear efecto 3D extrudiendo el pentágono
  beginShape(TRIANGLE_STRIP);
  for (int i = 0; i <= 5; i++) {
    float angle = TWO_PI * i / 5 - PI/2;
    vertex(radius * cos(angle), radius * sin(angle), -10);
    vertex(radius * cos(angle), radius * sin(angle), 10);
  }
  endShape();
}

// Función para crear un hexágono
void hexagon(float size) {
  float radius = size/2;
  beginShape();
  for (int i = 0; i < 6; i++) {
    float angle = TWO_PI * i / 6;
    vertex(radius * cos(angle), radius * sin(angle), 0);
  }
  endShape(CLOSE);

  // Crear efecto 3D extrudiendo el hexágono
  beginShape(TRIANGLE_STRIP);
  for (int i = 0; i <= 6; i++) {
```

```
  float angle = TWO_PI * i / 6;
  vertex(radius * cos(angle), radius * sin(angle), -10);
  vertex(radius * cos(angle), radius * sin(angle), 10);
 }
 endShape();
}

// Función para crear un rectángulo 2D
void rectangle2D(float width, float height) {
 float halfWidth = width/2;
 float halfHeight = height/2;

 // Dibuja el rectángulo
 beginShape();
 vertex(-halfWidth, -halfHeight, 0);
 vertex(halfWidth, -halfHeight, 0);
 vertex(halfWidth, halfHeight, 0);
 vertex(-halfWidth, halfHeight, 0);
 endShape(CLOSE);

 // Agregar efecto de profundidad con líneas
 beginShape(LINES);
 for (int i = 0; i < 4; i++) {
  float x = (i < 2) ? ((i == 0) ? -halfWidth : halfWidth) : ((i == 2) ? halfWidth : -halfWidth);
  float y = (i < 2) ? -halfHeight : halfHeight;
  vertex(x, y, 0);
  vertex(x, y, -10);
 }
 endShape();
```

```
  // Dibujar la cara posterior
  beginShape();
  vertex(-halfWidth, -halfHeight, -10);
  vertex(halfWidth, -halfHeight, -10);
  vertex(halfWidth, halfHeight, -10);
  vertex(-halfWidth, halfHeight, -10);
  endShape(CLOSE);
}

// Función para crear un rectángulo 3D
void rectangle3D(float width, float height) {
  float halfWidth = width/2;
  float halfHeight = height/2;
  float depth = 10; // Profundidad del rectángulo

  // Cara frontal
  beginShape();
  vertex(-halfWidth, -halfHeight, depth);
  vertex(halfWidth, -halfHeight, depth);
  vertex(halfWidth, halfHeight, depth);
  vertex(-halfWidth, halfHeight, depth);
  endShape(CLOSE);

  // Cara trasera
  beginShape();
  vertex(-halfWidth, -halfHeight, -depth);
  vertex(halfWidth, -halfHeight, -depth);
  vertex(halfWidth, halfHeight, -depth);
```

```
  vertex(-halfWidth, halfHeight, -depth);
  endShape(CLOSE);

  // Conectar caras con TRIANGLE_STRIP
  beginShape(TRIANGLE_STRIP);
  // Lado superior
  vertex(-halfWidth, -halfHeight, depth);
  vertex(-halfWidth, -halfHeight, -depth);
  vertex(halfWidth, -halfHeight, depth);
  vertex(halfWidth, -halfHeight, -depth);
  // Lado derecho
  vertex(halfWidth, halfHeight, depth);
  vertex(halfWidth, halfHeight, -depth);
  // Lado inferior
  vertex(-halfWidth, halfHeight, depth);
  vertex(-halfWidth, halfHeight, -depth);
  // Lado izquierdo
  vertex(-halfWidth, -halfHeight, depth);
  vertex(-halfWidth, -halfHeight, -depth);
  endShape();
}

// Función para crear un prisma triangular
void triangularPrism(float size, float height) {
  float halfSize = size/2;
  float halfHeight = height/2;

  // Calcula los vértices del triángulo base
  float[] xPoints = {-halfSize, halfSize, 0};
```

```
  float[] zPoints = {-halfSize/2, -halfSize/2, halfSize};

  // Base inferior
  beginShape();
  for (int i = 0; i < 3; i++) {
   vertex(xPoints[i], -halfHeight, zPoints[i]);
  }
  endShape(CLOSE);

  // Base superior
  beginShape();
  for (int i = 0; i < 3; i++) {
   vertex(xPoints[i], halfHeight, zPoints[i]);
  }
  endShape(CLOSE);

  // Caras laterales
  beginShape(TRIANGLE_STRIP);
  for (int i = 0; i <= 3; i++) {
   int idx = i % 3;
   vertex(xPoints[idx], -halfHeight, zPoints[idx]);
   vertex(xPoints[idx], halfHeight, zPoints[idx]);
  }
  endShape();
}

// Función para crear un cono
void cone(float radius, float height) {
  int sides = 24;
```

```
  float angleStep = TWO_PI / sides;

  // Base circular
  beginShape(TRIANGLE_FAN);
  vertex(0, height/2, 0); // Centro de la base
  for (int i = 0; i <= sides; i++) {
    float angle = i * angleStep;
    vertex(radius * cos(angle), height/2, radius * sin(angle));
  }
  endShape();

  // Superficie lateral
  beginShape(TRIANGLE_STRIP);
  for (int i = 0; i <= sides; i++) {
    float angle = i * angleStep;
    float x = radius * cos(angle);
    float z = radius * sin(angle);

    vertex(0, -height/2, 0); // Punta del cono
    vertex(x, height/2, z); // Punto en la base
  }
  endShape();
}
```