Guía Práctica de Clase: Desarrollando con HTML Canvas

El elemento **<canvas>** es una forma popular de dibujar todo tipo de gráficos en pantalla, lo que permite dibujar formas, realizar composiciones de fotos y crear animaciones. En el contexto de los videojuegos 2D, el uso de Canvas es una recomendación, aunque la complejidad del juego puede influir en la elección (siendo Canvas generalmente preferido para juegos 2D).

Módulo 1: Fundamentos y Dibujo de Figuras Básicas

Objetivo: Entender la estructura del Canvas y dibujar rectángulos y triángulos estáticos.

Paso 1.1: Configuración Inicial (HTML y JavaScript)

El elemento `<canvas>` se utiliza para dibujar gráficos mediante secuencias de comandos, típicamente JavaScript.

Estructura HTML: Crea un archivo `index.html`. El elemento `<canvas>` define un espacio en la página. El tamaño predeterminado es 300 píxeles de ancho y 150 píxeles de alto.

```
 <canvas id="miCanvas" width="400" height="300">
    Tu navegador no admite el elemento &amp;lt;canvas&amp;gt;.
</canvas>
<script>
    // Aquí irá tu código JavaScript
</script>
```

1.

Configuración de JavaScript: Para poder dibujar, necesitas acceder al contexto de dibujo 2D del canvas.

```
 function dibujar() {
    const canvas = document.getElementById("miCanvas");
    if (canvas.getContext) {
        // ctx es la "pluma" o contexto de dibujo
        const ctx = canvas.getContext("2d");

        // TODO: Añadir comandos de dibujo
    }
}
// Llamar a la función cuando la página cargue (ejemplo)
// <body onload="dibujar();">
```

2. *El origen de la cuadrícula del canvas se sitúa en la esquina superior izquierda en la coordenada (0,0). Todos los elementos se colocan en relación con este origen (x, y).*

Paso 1.2: Dibujando Rectángulos

El `<canvas>` solo admite dos formas primitivas: rectángulos y trazados (paths).

- **Rectángulo Relleno:** Usa `fillRect(x, y, width, height)`.
- **Rectángulo Contorneado:** Usa `strokeRect(x, y, width, height)`.
- **Borrar:** Usa `clearRect(x, y, width, height)` para hacer un área totalmente transparente.

```
// Dibujar un rectángulo negro relleno (ejemplo: 25, 25, 100, 100)
ctx.fillRect(25, 25, 100, 100);
// Borrar un cuadrado del centro
ctx.clearRect(45, 45, 60, 60);
// Dibujar un contorno
ctx.strokeRect(50, 50, 50, 50);
```

Paso 1.3: Dibujando Trazados (Paths)

Para formas más complejas (líneas, triángulos, círculos), usamos **trazados**.

1. **Iniciar Trazado:** Llama a `beginPath()` para crear un nuevo trazado.
2. **Mover la "Pluma":** Usa `moveTo(x, y)` para establecer el punto de partida sin dibujar.
3. **Dibujar Líneas:** Usa `lineTo(x, y)` para dibujar una línea recta hasta la nueva posición.
4. **Cerrar (Opcional):** Llama a `closePath()` para dibujar una línea recta al inicio del sub-trazado actual. (Si usas `fill()`, se cierra automáticamente, pero no con `stroke()`).
5. **Renderizar:** Usa `fill()` para rellenar la forma o `stroke()` para dibujar el contorno.

Ejemplo: Dibujar un Triángulo Contorneado

```
// Triángulo contorneado
ctx.beginPath();
ctx.moveTo(125, 125); // Punto de partida
ctx.lineTo(125, 45);
ctx.lineTo(45, 125);
ctx.closePath(); // Cierra el triángulo
ctx.stroke();
```

Paso 1.4: Dibujando Círculos (Arcos)

Para arcos o círculos, usamos el método `arc()`.

- `arc(x, y, radius, startAngle, endAngle, counterclockwise)`: Las coordenadas `x` e `y` son el centro del círculo.
- **Importante:** Los ángulos se miden en **radianes**, no en grados. Puedes convertir grados a radianes usando la expresión de JavaScript: `radianes = (Math.PI/180)*grados`. Un círculo completo es `Math.PI * 2`.

Ejemplo: Dibujar un Círculo Relleno (Cara Sonriente)

```
ctx.beginPath();
// Círculo externo (cara completa)
ctx.arc(75, 75, 50, 0, Math.PI * 2, true);

// Ojo izquierdo (Mover la pluma para que no se conecte con la cara)
ctx.moveTo(65, 65);
ctx.arc(60, 65, 5, 0, Math.PI * 2, true);

// Boca (Media luna)
ctx.moveTo(110, 75);
ctx.arc(75, 75, 35, 0, Math.PI, false); // false = en sentido de las agujas del reloj

ctx.stroke();
```

Módulo 2: Animaciones Fluidas

Objetivo: Usar `requestAnimationFrame` para crear movimiento continuo.

Paso 2.1: El Ciclo de Animación

La animación es un componente esencial para los juegos, ya que permite el movimiento de objetos. Para animar en Canvas, usamos el método **`requestAnimationFrame`**. Este método ejecuta una función repetidamente, sincronizando las animaciones con la velocidad de actualización de la pantalla para una experiencia fluida.

1. **Configurar Posición y Velocidad:** Definimos la posición inicial de un objeto (por ejemplo, un cuadrado) y la función principal de dibujo.

2. **Crear la Función `animate`:** Esta función debe hacer tres cosas en cada ciclo: a. Limpiar el canvas. b. Dibujar el objeto en su nueva posición. c. Actualizar la posición del

objeto. d. Llamarse a sí misma usando `requestAnimationFrame`.

Ejemplo Básico de Movimiento Lineal (Eje X):

```
let x = 0; // Posición inicial en X
const speed = 2; // Velocidad de movimiento

function animate() {
    const ctx = canvas.getContext("2d");

    // a. Limpiar el canvas en cada ciclo
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // b. Dibujar el cuadrado (usando la posición actualizada x)
    ctx.fillStyle = "green";
    ctx.fillRect(x, 50, 20, 20);

    // c. Actualizar la posición (incrementar x)
    x = x + speed; // El cuadrado se mueve progresivamente hacia la derecha

    // d. Repetir el ciclo
    requestAnimationFrame(animate);
}

// Iniciar la animación
requestAnimationFrame(animate);
```

Para controlar la velocidad, simplemente se modifica el incremento de la variable de posición, por ejemplo, aumentando el valor de `speed`.

Paso 2.2: Movimiento Bidimensional y Rebote

Para movimientos en dos direcciones (diagonal), necesitamos variables para el eje Y.

Ejemplo de Rebote Diagonal:

```
let x = 50;
let y = 50;
let speedX = 3; // Velocidad horizontal
let speedY = 3; // Velocidad vertical

function animateDiagonal() {
    const ctx = canvas.getContext("2d");
```

```
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // Dibujar el objeto (ejemplo: cuadrado rojo)
    ctx.fillStyle = "red";
    ctx.fillRect(x, y, 20, 20);

    // Actualizar posición
    x += speedX;
    y += speedY;

    // Lógica de rebote (inversión de dirección al tocar los límites)
    if (x + 20 > canvas.width || x < 0) {
        speedX = -speedX;
    }
    if (y + 20 > canvas.height || y < 0) {
        speedY = -speedY;
    }

    requestAnimationFrame(animateDiagonal);
}

requestAnimationFrame(animateDiagonal);
```

Si el objeto alcanza los límites del canvas, la dirección del movimiento se invierte, creando un efecto de rebote.

Módulo 3: Proyecto de Juego 2D - Detección de Colisiones

Objetivo: Implementar lógica de juego, específicamente detección de colisiones y destrucción de objetos (como en un juego de romper ladrillos).

Paso 3.1: Configuración de Objetos de Juego (Ladrillos)

En un juego 2D, necesitamos una manera de rastrear los objetos. Usaremos un array para almacenar los "ladrillos".

1. **Definir Propiedades del Ladrillo:** Para saber si un ladrillo debe dibujarse o no, le agregamos una propiedad `status`.
2. **Inicializar Ladrillos:** Usamos bucles para crear una cuadrícula de ladrillos, asignando su posición y estableciendo `status: 1` (activo).

```
var bricks = [];
```

```
// Configura columnas y filas (ejemplo: 5 columnas, 3 filas)
const brickColumnCount = 5;
const brickRowCount = 3;

for (c = 0; c < brickColumnCount; c++) {
    bricks[c] = [];
    for (r = 0; r < brickRowCount; r++) {
        bricks[c][r] = { x: 0, y: 0, status: 1 }; // Status 1 = debe dibujarse
    }
}
```

3. **Función `drawBricks()`:** Actualiza la función de dibujo para pintar solo los ladrillos con `status == 1`.

Paso 3.2: Detección de Colisiones

Ya que el `<canvas>` no tiene funciones incorporadas para saber si una bola toca un rectángulo, debemos calcularlo mediante lógica matemática. Una manera sencilla es verificar si el centro de la bola está dentro de las coordenadas de un ladrillo.

1. **La Lógica de Colisión (Cuatro Condiciones):** El centro de la bola está dentro del ladrillo si se cumplen estas cuatro condiciones simultáneamente:
    - La posición `x` de la bola es mayor que la posición `x` del ladrillo.
    - La posición `x` de la bola es menor que la posición del ladrillo más su ancho.
    - La posición `y` de la bola es mayor que la posición `y` del ladrillo.
    - La posición `y` de la bola es menor que la posición del ladrillo más su altura.
2. **Implementar `collisionDetection()`:** Recorre todos los ladrillos activos (`b.status == 1`) y verifica la colisión.

```
function collisionDetection() {
    for (c = 0; c < brickColumnCount; c++) {
        for (r = 0; r < brickRowCount; r++) {
            var b = bricks[c][r]; // Asignamos el objeto ladrillo a 'b' para mayor legibilidad
            if (b.status == 1) { // Solo verificamos si el ladrillo está activo
                // Condición de colisión traducida a JavaScript
                if (x > b.x && x < b.x + brickWidth && y > b.y && y < b.y + brickHeight) {
                    // 1. Invertir la dirección de la bola
                    dy = -dy;

                    // 2. Desactivar el ladrillo para que desaparezca
                    b.status = 0; // El ladrillo ya no se dibujará
```

```
            }
        }
    }
  }
}
```

En el ejemplo anterior, `x` y `y` son las coordenadas del centro de la bola, y `dy` controla su movimiento vertical.

Paso 3.3: Integración en el Ciclo de Juego

Asegúrate de que la función `collisionDetection()` se ejecute en cada fotograma del juego.

```
function draw() {
  // ... código de limpieza y dibujo de la bola/paleta ...

  collisionDetection(); // Llamada a la función de colisión

  // ... código de actualización de movimiento ...

  // requestAnimationFrame(draw);
}
```

Al llamar a `collisionDetection()` dentro de la función principal de dibujo, se comprueba si la bola colisiona con cada ladrillo en cada fotograma, permitiendo que los ladrillos se rompan.