

Visualizaciones generativas clásicas de Windows Media Player y su recreación en la web

Windows Media Player (WMP) de los años 2000–2010 incluía varias **visualizaciones clásicas** que generaban animaciones en tiempo real al ritmo de la música. Estas visualizaciones – “*Bars and Waves*”, “*Battery*”, “*Ambience*”, “*Alchemy*”, entre otras – mostraban patrones geométricos y efectos de color sincronizados con el audio ¹. A continuación, se describen cada una de estas visualizaciones, cómo funcionan visualmente, cómo podrían recrearse de forma ligera con HTML, CSS y JavaScript puro (usando principalmente `<canvas>` o SVG), y se sugieren optimizaciones para mantener un buen **rendimiento** (especialmente en dispositivos móviles) sin sacrificar la experiencia visual.

Bars and Waves (Barras y Ondas)

Descripción visual: La visualización “*Bars and Waves*” es una de las más reconocidas y básicas de WMP. Consiste en dos modos principales: uno de **barras** (ecualizador de barras) y otro de **ondas** (línea de forma de onda tipo osciloscopio). En el modo de barras, múltiples barras verticales representan distintas bandas de frecuencia del audio, creciendo y decreciendo en altura según la intensidad de cada banda. En el modo de onda, se traza una línea continua que ondula según la forma de onda del sonido, a menudo centrada horizontalmente. Esta colección incluía variaciones (presets) como *Bars*, *Ocean Mist*, *Firestorm* y *Scope* ¹. Por ejemplo, *Bars* muestra un espectro de barras verticales (generalmente de color verde o amarillo), *Scope* muestra una onda oscilante en el centro, *Firestorm* aplicaba colores cálidos simulando fuego en las ondas, y *Ocean Mist* creaba un efecto de horizonte con ondas de agua y niebla (líneas senoidales suaves con apariencia de olas bajo un cielo). Estas visualizaciones responden al ritmo de la música: las barras vibran con los beats y la onda se agita según el sonido.

Recreación con HTML5 Canvas: Se puede recrear *Bars and Waves* fácilmente usando un elemento `<canvas>` y el API de dibujo 2D. Para el modo **barras**, se usaría el análisis de frecuencias del audio (por ejemplo, con `AnalyserNode` de la Web Audio API) para obtener magnitudes por banda. Cada barra puede ser un rectángulo dibujado con `fillRect(x, y, width, height)` donde `height` depende de la amplitud de esa banda. Ubicamos las barras consecutivamente a lo ancho del canvas. Un pseudocódigo simplificado para dibujar las barras podría ser:

```
const bands = 32; // número de barras de frecuencia
const gap = 2;    // espacio entre barras
const barWidth = (canvas.width - (bands-1)*gap) / bands;
for (let i = 0; i < bands; i++) {
  const amplitude = audioFreqData[i]; // valor de 0 a 255
  const barHeight = (amplitude / 255) * canvas.height;
  ctx.fillRect(i * (barWidth + gap), canvas.height - barHeight, barWidth,
    barHeight);
}
```

En este código, `audioFreqData` sería un array con la magnitud de cada banda de frecuencia en ese instante, y se dibujan barras escaladas a la altura del canvas (invertidas para crecer hacia arriba). Para el

modo **onda**, se toma el waveform (forma de onda temporal) del audio mediante el mismo `AnalyserNode` en modo `getTimeDomainData`. Luego, se recorre ese array de muestras y se usa `moveTo` / `lineTo` para trazar una polilínea. Normalmente se dibuja centrada verticalmente (o con una línea base) y se escala la amplitud de la onda para que ocupe una porción visible del lienzo. Un ejemplo de trazado de onda:

```
ctx.beginPath();
const sliceWidth = canvas.width / waveformData.length;
let x = 0;
for (let sample of waveformData) {
  const y = (sample / 255) * canvas.height; // normalizar muestra 0-255 al
  alto
  if (x === 0) ctx.moveTo(x, y);
  else ctx.lineTo(x, y);
  x += sliceWidth;
}
ctx.stroke();
```

Este trazo dibujaría una línea osciloscopio representando la onda de audio en tiempo real. Se pueden agregar efectos visuales ligeros, como un degradado de color en las barras u ondas, o desenfoque de movimiento simple (dibujando con `globalAlpha` ligeramente menor a 1 para dejar estelas cortas).

Optimización: Para que la visualización *Bars and Waves* sea eficiente y fluida, especialmente en móviles, se pueden aplicar varias optimizaciones: - **Reducir la resolución o número de elementos:** Usar menos barras (p. ej., 32 o 64 en lugar de 128) y muestras de onda con *downsampling* (tomar cada n-ésima muestra) al dibujar la onda reduce la carga de dibujo. - **Canvas persistente para elementos estáticos:** Si hay partes estáticas (ejemplo: un fondo degradado o una cuadrícula de referencia), dibujarlas una vez en un canvas de fondo. Luego, en cada frame, solo redibujar las barras/onda dinámicas en un canvas frontal, evitando repintar todo el fondo cada vez. - **Uso adecuado de `requestAnimationFrame`:** Actualizar la animación con `requestAnimationFrame` sincroniza los dibujos a los refrescos de pantalla, evitando frames innecesarios. También se puede ajustar la frecuencia de muestreo de audio (p. ej., obtener datos cada 50ms en vez de cada frame) si la animación es demasiado exigente. - **Simplificar el dibujo de la onda:** En lugar de dibujar línea por línea todos los puntos, se puede incrementar el valor de x en saltos (p. ej., dibujar un punto cada ~5px) sin perder mucha fidelidad visual, ya que la onda se suavizará visualmente. Esto reduce llamadas de dibujo. - **Aprovechar estilos eficientes:** Dibujar las barras como un solo path complejo en vez de múltiples rectángulos separados (usando subpaths) podría mejorar rendimiento, aunque en canvases modernos el beneficio es marginal. También evitar sombras (`shadowBlur`) o trazos muy gruesos que penalicen la GPU. - **Control de calidad en móviles:** En dispositivos de menor potencia, considerar reducir el tamaño del canvas (por ejemplo, dibujar a la mitad de resolución y escalar con CSS) para disminuir el número de píxeles procesados, a costa de algo de nitidez.

Con estas técnicas, *Bars and Waves* puede recrearse fielmente con un código ligero y mantener un rendimiento fluido.

Ambience (Ambiente)

Descripción visual: “Ambience” es una visualización clásica de WMP 7/8/9 conocida por sus patrones **suaves y calmados** ². A diferencia de las visualizaciones de barras, Ambience presentaba formas más

orgánicas: líneas fluidas, remolinos, ondas y figuras abstractas que se mueven lentamente y con transiciones de color graduales. Tenía 14 variaciones (presets) con nombres evocativos – por ejemplo: *Swirl* (remolino), *Warp*, *Water*, *Bubble*, *Windmill*, *Down the Drain*, etc ³. Muchas de estas variaciones simulaban elementos de la naturaleza o movimientos suaves: por ejemplo, el preset *Water* mostraba ondas líquidas horizontales ondulando lentamente (fue uno de los más populares de *Ambience* ⁴), *Bubble* podía mostrar burbujas o círculos pulsantes, *Swirl* dibujaba espirales lentas, y *Windmill* mostraba formas rotando como aspas. Los gráficos de *Ambience* solían consistir en **líneas continuas o curvas** que se estiraban y curvaban con la música, a veces duplicadas simétricamente o con efectos de espejado, todo sobre un fondo oscuro. Además, *Ambience* se caracterizaba por **transiciones de color fluidas**: la paleta iba cambiando cíclicamente (de azul claro a rojo, naranja, verde, cyan, magenta, etc.) creando un efecto de degradado animado muy agradable ⁵. Visualmente, la animación reaccionaba a la música de forma sutil: a volúmenes altos ciertas figuras brillaban en blanco o cambiaban dirección repentinamente ⁶, pero mantenía un estilo relajado.

Recreación con Canvas/SVG: Para recrear *Ambience* en la web, podemos usar un `<canvas>` 2D combinando dibujos de líneas curvas, transformaciones y gradientes. Muchos presets de *Ambience* pueden lograrse dibujando **curvas paramétricas o espirales**. Un enfoque general es crear una serie de puntos que siguen funciones matemáticas (seno, coseno, espiral logarítmica, etc.) y dibujar líneas suaves entre ellos. Por ejemplo, para un efecto tipo *Swirl* (espiral), podríamos generar puntos en coordenadas polares aumentando gradualmente el ángulo y el radio, con perturbaciones basadas en el audio. Una pseudo-implementación simplificada para un remolino central podría ser:

```
ctx.beginPath();
const centerX = canvas.width/2, centerY = canvas.height/2;
let radius = 5;
for (let theta = 0; theta < 6*Math.PI; theta += 0.1) {
  // incrementar radio para formar espiral
  radius += 0.5;
  // aplicar ligera variación según audio (ej: usar un valor de bajo
  frecuencia)
  const audioAmp = currentBassLevel * 0.05;
  const r = radius + audioAmp * Math.sin(theta * 4);
  const x = centerX + r * Math.cos(theta);
  const y = centerY + r * Math.sin(theta);
  if (theta === 0) ctx.moveTo(x, y);
  else ctx.lineTo(x, y);
}
ctx.stroke();
```

En este ejemplo, dibujamos una espiral desde el centro, cuyo radio crece y se modula ligeramente con una componente sinusoidal multiplicada por un nivel de audio (*currentBassLevel* podría ser la amplitud de frecuencias bajas). Esto produce una línea que tiembla suavemente al ritmo del bajo. Similarmente, otros efectos: - *Water*: podríamos dibujar varias ondas sinusoidales horizontales apiladas, de baja frecuencia, simulando olas (similar a *Ocean Mist* de Bars and Waves pero con cambios de color lentos). - *Windmill*: dibujar líneas rectas o triángulos rotando suavemente alrededor del centro. - *Bubble*: dibujar círculos semi-translúcidos que crecen y desaparecen. Esto se logra fácilmente con `ctx.arc(x, y, radius, 0, 2*Math.PI)` y ajustando el radio según el audio (p. ej., crear un círculo nuevo cada cierto intervalo cuando la amplitud supera un umbral).

La clave para *Ambience* es también implementar la **transición de colores gradual**. En canvas, se puede lograr cambiando lentamente el color de trazo o relleno frame a frame. Por ejemplo, podemos tener un array de colores (azul, verde, rojo, etc.) y hacer que el color actual interpole hacia el siguiente con un pequeño delta cada frame. Usar `ctx.strokeStyle = 'rgba(r,g,b,alpha)'` con valores cambiantes producirá un cambio suave. Otra técnica es dibujar con muy baja opacidad y sin limpiar completamente el lienzo, de modo que los dibujos anteriores queden como un rastro difuso (efecto de persistencia). Por ejemplo:

```
ctx.fillStyle = "rgba(0, 0, 0, 0.05)";  
ctx.fillRect(0, 0, canvas.width, canvas.height); // semi-borrar fondo
```

antes de dibujar las figuras, para que poco a poco se desvanezcan en lugar de borrarse abruptamente. Esto da un efecto *trailing* que casa bien con la estética de *Ambience*.

Para dibujos complejos, también podría considerarse SVG si las formas son vectoriales relativamente estáticas (por ejemplo, un camino curvo repetido). Sin embargo, la naturaleza continuamente cambiante de *Ambience* (líneas deformándose con audio) hace que canvas sea más apropiado. Con SVG podríamos, por ejemplo, animar un `<path>` que represente una curva Bezier, pero actualizar cientos de vértices en SVG cada frame sería menos eficiente que en canvas.

Optimización: Al recrear *Ambience*, es importante conservar la fluidez sin saturar la CPU/GPU: - **Simplificar la geometría:** Limitar la cantidad de puntos usados para dibujar curvas. Por ejemplo, si una espiral se ve bien con 100 puntos, no usar 1000. Podemos ajustar la resolución de las curvas dinámicamente: menos puntos en dispositivos lentos. - **Uso moderado de efectos alpha:** Aunque los trazos semitransparentes y persistencia de dibujado dan un look suave, abusar de altos valores de alpha puede requerir dibujar muchas capas sobrepuestas. Conviene usar un alpha bajo (ej. 0.1) y no sobre-dibujar demasiadas figuras simultáneamente. - **Actualizar a menor framerate si es complejo:** Si ciertos presets son muy costosos (ej. dibujar 10 espirales a la vez), se puede optar por dibujarlos a, digamos, 30 FPS en lugar de 60 FPS, usando `setTimeout` o alternando frames (ej. dibujar solo en frames pares para aligerar carga). - **Reutilizar cálculos trigonométricos:** Las figuras como las espirales y ondas usan muchas operaciones `sin/cos`. Calcular estas funciones es costoso, así que se puede precalcular tablas de seno/coseno para los rangos necesarios al iniciar, o calcular una vez por frame ciertos valores base y solo ajustar parámetros. Asimismo, usar variables temporales para valores repetidos (evitar recalcular `Math.sin(theta)` varias veces con el mismo `theta`). - **Capar la complejidad de audio análisis:** El análisis de audio en tiempo real también consume recursos. Podemos reducir el tamaño de la FFT o utilizar solo unas pocas bandas relevantes para influir la animación (por ejemplo, solo volumen general y quizás nivel de graves para controlar ciertos movimientos), en lugar de tratar de reaccionar a 128 bandas simultáneamente. - **Canvas Off-screen (opcional):** Para efectos complejos de mezcla de colores, podría usarse un canvas fuera de pantalla donde se dibuja la forma compleja, y luego copiarlo al canvas principal escalado o con algún filtro. Esto aísla el cómputo pesado. No obstante, en general *Ambience* puede implementarse suficientemente ligero con un solo canvas si se controlan los puntos anteriores.

Con estas medidas, es posible lograr los remolinos y ondulaciones características de *Ambience* con un rendimiento suave incluso en móviles, manteniendo la estética relajante del original.

Battery (Batería)

Descripción visual: “Battery” es una visualización introducida en WMP 8 que expandió la idea de Ambience con **más energía y variedad de formas** ⁷. De hecho, Battery es considerada la “hermana mayor” de Ambience, con 26 presets (el doble que Ambience) cubriendo un espectro amplio de efectos ⁸. Visualmente, Battery abarca desde ondas y líneas similares a Ambience hasta figuras geométricas, efectos caleidoscópicos y fondos dinámicos. Por ejemplo, algunos de sus presets tenían nombres descriptivos como *BrightSphere* (esfera brillante), *Dance of the Freaky Circles* (círculos bailando), *Kaleidovision* (probablemente un caleidoscopio), *SepiaSwirl* (remolino en tonos sepia), *ChemicalNova*, etc. ⁹ ¹⁰. Un preset destacado llamado *Randomization* alternaba aleatoriamente entre patrones. En Battery podíamos ver, dependiendo del preset, elementos como: - **Formas circulares y esféricas:** círculos concéntricos que laten al ritmo de la música o esferas luminosas flotando (ej. *BrightSphere*). - **Efectos kaleidoscopio:** divisiones simétricas de la pantalla con patrones que se repiten girando, generando una imagen caleidoscópica en movimiento (*Kaleidovision*). - **Matrices de partículas:** puntos o estrellas que explotan y forman matrices animadas, similar en concepto a Particle. - **Fondos de color e imágenes:** Battery a veces incluía fondos con gradientes o texturas (por ejemplo, *sepiaswirl* insinuaba tonos sepia, *gemstonematrix* podía mostrar un fondo tipo “malla de gemas”). También podía haber destellos o flashes en colores brillantes según la intensidad del audio.

En resumen, Battery ofrece “formas de onda, gamas de colores y fondos muy diversos” ⁷, proporcionando desde ambientes tranquilos hasta explosiones visuales más vívidas, todo sincronizado con la música.

Recreación con Canvas: Dada la variedad de Battery, recrearla implica implementar **múltiples modos de visualización** y quizás permitir cambiar entre ellos (similar a presets). Afortunadamente, muchos de estos modos pueden construirse combinando las técnicas ya mencionadas para Bars & Waves y Ambience, junto con algunas adicionales: - *Barras y ondas mejoradas:* Battery incluye formas de onda, por lo que se puede reutilizar el código de barras/onda pero incorporando efectos de color cambiantes o duplicando la onda en formas novedosas (por ejemplo, dibujar dos ondas espejadas verticalmente para formar una figura de “ojo”). - *Círculos y partículas:* Para los círculos danzantes, se puede generar un conjunto de círculos cuyo radio u posición oscilan con la música. Por ejemplo, dibujar N círculos concéntricos en el centro, cuyo radio aumenta/disminuye según el volumen bass o beats. O dispersar pequeños círculos por la pantalla moviéndose aleatoriamente (tipo partículas) y cambiar su tamaño/opacidad con el audio. - *Caleidoscopio:* Este efecto se puede lograr dividiendo el canvas en secciones angulares y dibujando un patrón repetido con rotación. Un método sencillo: dibujar formas en solo 1/6 del canvas (por ejemplo, en un triángulo sector de 60°) y luego copiar/rotar esa porción 6 veces alrededor del centro. En canvas 2D, se puede hacer usando `ctx.save(); ctx.rotate(angle); ... ctx.restore();` repetidamente. Por ejemplo, para un caleidoscopio hexagonal:

```
for(let i=0; i<6; i++){
  ctx.save();
  ctx.translate(cx, cy);
  ctx.rotate((Math.PI/3) * i); // rotar i * 60 grados
  ctx.translate(-cx, -cy);
  drawPatternSlice(); // función que dibuja la figura en un sector
  (predefinido)
  ctx.restore();
}
```

Donde `drawPatternSlice()` dibujaría quizás una línea ondulada o unos puntos en posiciones relative al sector. El resultado es que seis copias de esa forma aparecen rotadas, creando simetría radial.

- **Fondo y colores:** Podemos cambiar el fondo del canvas para ciertos presets. Con `ctx.fillStyle` podemos llenar el fondo de un color sólido o un gradiente. Por ejemplo, en un preset "sepia" usar un color de fondo marrón oscuro. También se pueden superponer rectángulos translúcidos de colores cambiantes para destellos. Por ejemplo, si el audio excede cierto umbral, dibujar rápidamente un flash: `ctx.fillStyle = "rgba(255,255,255,0.5)"; ctx.fillRect(0,0,width,height);` para un destello blanco semitransparente.

La implementación en código podría organizarse con un sistema de presets definidos por configuraciones. Por ejemplo, un objeto `presetConfig` con propiedades que indiquen qué elementos dibujar: `{ bars: true, kaleidoscope: false, circleCount: 10, background: "#000", palette: ["#ff0", "#f00", ...] }`. El motor principal lee esta config y ejecuta las rutinas de dibujo correspondientes. Esto mantiene el código mantenible pese a la variedad.

Optimización: Para que Battery corra bien en la web sin sacrificar sus vistosos efectos:

- **Seleccionar elementos clave por preset:** No intentes dibujarlo *todo a la vez*. Cada preset de Battery destaca un efecto, así que en la implementación asegurarse de **activar solo las rutinas necesarias**. Por ejemplo, si el preset actual es caleidoscopio, no dibujar también partículas y barras simultáneamente (a menos que se desee, pero eso incrementa carga).
- **Reusar lógica entre presets:** Muchos presets comparten componentes (ondas, círculos, partículas). Usar funciones comunes para dibujarlos y solo variar parámetros. Esto no solo reduce código, también mejora rendimiento debido a que funciones optimizadas se reutilizan (mejor caché de CPU, etc.).
- **Caleidoscopio eficiente:** Al implementar el efecto caleidoscopio, dibujar en un área pequeña y replicar por transformaciones es eficiente porque aprovecha la simetría. Asegurarse de no recalcular la misma figura 6 veces de manera distinta. Lo ideal es dibujar la figura base una vez en una pequeña área (por ejemplo, usar un canvas off-screen del tamaño de un sector), luego usar `drawImage` para copiarlo rotado. Sin embargo, `drawImage` con rotación no es directa en 2D API, así que la alternativa es la técnica con `ctx.rotate` dentro de un bucle como se mostró, que igualmente evita recalcular la geometría de la figura para cada sector.
- **Limiter partículas/círculos:** Si un preset tiene 100 partículas, considerar reducir ese número en dispositivos lentos. Las partículas pueden moverse con física simple (velocidad, aceleración) actualizada en JS; conviene usar estructuras de datos eficientes (arrays simples) y evitar cálculos innecesarios. Por ejemplo, si las partículas tienen todas el mismo comportamiento salvo posición, quizás usar un senoide común para su movimiento vertical en lugar de calcular 100 senos independientes.
- **Control de frames adaptativo:** Monitorizar el rendimiento (medir el tiempo de dibujo de un frame). Si un preset complejo está tardando demasiado, se puede implementar un **auto-throttle**: reducir temporalmente la tasa de frames (ej. saltarse dibujar ciertos frames) o simplificar efectos (menos segmentos en las ondas, etc.). Esto podría hacerse dinámicamente comprobando `performance.now()` entre frames.
- **Minimizar cambios de estado del canvas:** Dibujar muchos elementos con distintos estilos puede incurrir en cambios frecuentes de `strokeStyle`, `fillStyle`, etc. Agrupar dibujos por estilo puede mejorar rendimiento. Por ejemplo, dibujar todos los círculos de un mismo color en un mismo bucle antes de cambiar de color para otros elementos.
- **Evitar filtrados costosos:** Efectos como desenfocado (canvas `shadowBlur`) o composiciones complejas (`globalCompositeOperation` inusual) deben usarse con moderación. Si un preset requiere un **blur** o brillo, valora pre-generar una imagen difuminada y luego dibujarla, en lugar de aplicar blur en cada frame a gran cantidad de píxeles.

Siguiendo estas recomendaciones, incluso las animaciones más complejas de Battery pueden ser recreadas en el navegador manteniendo una buena tasa de refresco y sin sobrecargar en exceso el CPU o GPU.

Alchemy (Alquimia)

Descripción visual: “Alchemy” es la visualización que debutó con WMP 9, caracterizada por su naturaleza **impredecible y generativa**. A diferencia de otras, Alchemy no tenía múltiples presets nombrados, sino un único modo *Random* que continuamente mutaba y generaba nuevos patrones ¹¹ ¹². Pertenece a la familia de Ambience ¹³, por lo que visualmente guarda parentesco con Ambience y Battery: presenta formas fluidas, colores cambiantes y movimientos sincronizados con la música. Sin embargo, Alchemy lleva esto al extremo de la espontaneidad: en una misma sesión, la visualización puede mostrar algo parecido a un remolino de humo, que luego se transforma en una red de líneas eléctricas, luego en un torbellino de partículas, etc., todo enlazado por transiciones suaves. El nombre *Alchemy* sugiere esa mezcla y cambio constante, casi “mágico”, de las formas y colores. Por ejemplo, podíamos ver en Alchemy momentos de “**líneas aleatorias**” (como garabatos que se dibujan solos), formas geométricas que aparecen y se disuelven, y destellos al ritmo de beats fuertes. En general, los colores son vivos y van rotando, similares a la paleta de Ambience (muchos tonos intensos de azul, verde, magenta, etc.). Es como si Alchemy tomara todos los trucos de las otras visualizaciones y los mezclara en una sola, aplicando distintos al azar.

Recreación con Canvas (enfoque generativo): Para recrear *Alchemy*, necesitamos implementar un sistema generativo que produzca variaciones visuales automáticamente. Un enfoque es definir un conjunto de posibles **efectos base** (por ejemplo: *línea sinuosa central*, *red de líneas aleatorias*, *explosión de partículas*, *ondulaciones circulares*, *patrón caleidoscópico*, etc.) y que el programa elija y mezcle estos efectos durante la reproducción. Podemos estructurarlo así: 1. **Estados o escenas aleatorias:** Definir una lista de “escenas” o configuraciones de dibujo. Cada una podría corresponder a un mini-preset. Por ejemplo, escenaA dibuja una espiral de color X, escenaB dibuja un patrón de partículas radiales, escenaC dibuja líneas aleatorias que se van conectando, etc. 2. **Transición entre escenas:** Cada cierta cantidad de tiempo o compases de música, Alchemy debería transicionar a una nueva escena aleatoria. La transición puede ser suave (fundidos) para imitar el comportamiento original. Podemos lograrlo haciendo que los nuevos elementos aparezcan gradualmente mientras los antiguos se desvanecen. Por ejemplo, introduciendo un factor de interpolación alfa de 0 a 1 para la nueva escena mientras disminuimos el alfa de la anterior. 3. **Parámetros aleatorios:** Incluso dentro de una misma escena, introducir aleatoriedad. Por ejemplo, si dibujamos “líneas de garabato”, que el ángulo o la densidad de las líneas sea aleatorio cada vez que se activa ese efecto. Usar `Math.random()` con una semilla controlada (para que cambios no sean demasiado bruscos frame a frame) es útil.

En código, podríamos tener algo como:

```
let currentScene = null;
let nextScene = null;
let transitionProgress = 0;

function pickRandomScene() {
  const scenes = [sceneSpiral, sceneParticles, sceneMesh, sceneCircles];
  const choice = scenes[Math.floor(Math.random()*scenes.length)];
  return choice;
}

function renderFrame() {
  if (!currentScene) currentScene = pickRandomScene();
  if (nextScene) {
    // Mezclar escenas durante transición
  }
}
```

```

currentScene.draw(1 - transitionProgress);
nextScene.draw(transitionProgress);
transitionProgress += 0.01;
if (transitionProgress >= 1) {
    // finalizar transición
    currentScene = nextScene;
    nextScene = null;
    transitionProgress = 0;
}
} else {
    // Dibujar escena actual normalmente
    currentScene.draw(1);
    // aleatoriamente, iniciar cambio de escena
    if (shouldChangeScene()) {
        nextScene = pickRandomScene();
        transitionProgress = 0;
    }
}
requestAnimationFrame(renderFrame);
}

```

En este pseudocódigo, cada escena tendría un método `draw(alpha)` que dibuja sus elementos con cierta opacidad o intensidad dada por `alpha`. Así, durante la transición, la escena vieja y la nueva se superponen inversamente proporcional. La condición `shouldChangeScene()` podría basarse en tiempo (por ejemplo cada 10 segundos) y/o en eventos de la música (por ejemplo un pico de volumen podría disparar un cambio para mayor dinamismo).

Dibujo de los elementos: Las técnicas de dibujo dentro de cada escena serían similares a las ya discutidas: usar canvas para dibujar líneas, arcos, partículas, etc., moduladas por audio. La diferencia es que ahora cada efecto se activa solo en su escena correspondiente. Por ejemplo: - *sceneSpiral*: dibuja una espiral de la forma descrita en Ambiente, quizás con parámetros aleatorios (número de vueltas, grosor) elegidos al iniciar la escena. - *sceneParticles*: dibuja partículas moviéndose (similar a Particle/Particle preset), con colores aleatorios. - *sceneCircles*: dibuja círculos concéntricos pulsantes (similar a algunos presets de Battery). - *sceneMesh*: dibuja una “malla” de líneas conectando puntos aleatorios que vibran con el audio, dando aspecto de red eléctrica.

Con este sistema, la visualización resultante cambiará de forma impredecible como en Alchemy.

Optimización: Dado que Alchemy potencialmente combina varios efectos, hay que ser cuidadoso para no sobrecargar: - **Una transición a la vez:** Evitar que múltiples transiciones se solapen. Siempre terminar una transición antes de iniciar la siguiente, así no estamos dibujando tres escenas a la vez (eso duplicaría/triplicaría el coste). - **Limitar complejidad por escena:** Como regla, cada escena por sí sola debe ser ligera. Si definimos una escena muy pesada (por ejemplo, dibujar 500 partículas + 100 líneas cada frame), tal escena podría por sí sola bajar el rendimiento. Testear cada módulo de escena y simplificarlo si consume demasiado. Es preferible tener más escenas variadas pero sencillas, que pocas muy complejas. - **Reciclar canvas u objetos:** Al cambiar de escena, en lugar de descartar todo y crear estructuras nuevas, podemos reutilizar. Por ejemplo, un array de partículas puede reciclarse entre escenas de partículas, solo re-inicializando posiciones. También mantener un único canvas y simplemente cambiar lo que se dibuja; no crear/destroy canvases. - **Disminuir la frecuencia de cambios aleatorios:** Dentro de una escena, aplicar variaciones suaves. Si cada frame recalculamos

aleatoriamente posiciones totalmente nuevas, el dibujo puede parpadear y además no aprovecha cálculo previo. Mejor usar propiedades que varían lentamente (por ejemplo, una rotación que va incrementando cada frame en vez de asignar un ángulo random cada vez). Esto no solo es visualmente más agradable sino que también permite coherencia temporal y menos picos de cálculo. - **Adaptación a fps:** Similar a Battery, si el sistema detecta que la animación de Alchemy está yendo lenta (por demasiadas cosas a la vez), se puede implementar una *degradación graciosa*: por ejemplo, reducir el número de elementos en la escena actual dinámicamente, o prolongar más la duración de la escena actual (para no iniciar otra transición pronto). Así se evita agravar una situación de carga alta con aún más cambios. - **Evitar memory leaks:** Como se generan muchas variaciones, asegurarse de limpiar referencias de objetos de escenas viejas si ya no se usan (por ejemplo, si se crearon buffers off-screen, liberarlos o dejarlos que el GC los recicle). Mantener el bucle principal limpio.

Con estas precauciones, *Alchemy* puede recrearse de forma bastante fiel: la clave es lograr esa aleatoriedad controlada. El resultado será una visualización muy dinámica pero estable en rendimiento, capaz de correr incluso en navegadores móviles modernos sin problemas mayores.

Particle (Dotplane)

Descripción visual: “Particle” (originalmente llamado *Dotplane* en WMP7) presenta un efecto distinto a los anteriores: una **cuadrícula plana de puntos** que reaccionan a la música ¹⁴. Imagina un plano bidimensional lleno de pequeños puntos luminosos organizados en filas y columnas, generalmente sobre fondo negro. En WMP, los puntos eran de colores como rojo, púrpura, azul y cyan ¹⁴ formando un patrón de color repetitivo o degradado en la malla. Al sonar la música, esta matriz de puntos podía comportarse de dos maneras según el preset: - En el preset estático *Particle*, los puntos vibran o cambian de intensidad ligeramente con el ritmo, a veces generando ondas que recorren la cuadrícula (por ejemplo, un pulso que se propaga a través de los puntos al compás del beat). - En el preset *Rotating Particle*, la entera malla de puntos **rota en el espacio** lentamente ¹⁵, dando la impresión de un plano que gira en 3D. Durante la rotación, los puntos cambian de perspectiva (más juntos o separados según el ángulo) y posiblemente de tamaño o brillo según su distancia. Este preset podía resetear la rotación tras un tiempo para evitar girar indefinidamente debido a limitaciones técnicas ¹⁶.

El efecto general es como ver un “**campo de estrellas**” estilizado o un plano digital que se mueve con la música. Es relativamente sencillo visualmente (no hay líneas ni formas complejas, solo puntos), pero produce un ambiente hipnótico, sobre todo cuando rota.

Recreación con Canvas/SVG: Lo esencial para recrear Particle es **dibujar una grilla de puntos** y manipularla. Podemos hacerlo con canvas dibujando múltiples pequeños círculos o cuadrados. Supongamos una matriz de N x M puntos. Podemos almacenar sus coordenadas en un array bidimensional para fácil acceso. Una implementación básica:

```
const rows = 16, cols = 16;
let points = [];
for(let i=0; i<rows; i++){
  points[i] = [];
  for(let j=0; j<cols; j++){
    points[i][j] = { x: j * spacing, y: i * spacing };
  }
}
```

Aquí `spacing` sería la separación entre puntos (e.g., 20 px). Con esto tenemos la cuadrícula base. Para dibujarla:

```
ctx.fillStyle = "#0ff"; // color cian por ejemplo
for(let i=0; i<rows; i++){
  for(let j=0; j<cols; j++){
    const p = points[i][j];
    ctx.fillRect(p.x, p.y, 3, 3); // dibujar punto como cuadradito 3x3
  }
}
```

Esto dibujaría una malla regular de puntos cian. Para hacerla más interesante y “WMP-like”, debemos agregar:

- **Color por punto:** En WMP, los puntos tenían diferentes colores. Podemos asignar un color dependiendo de la posición, por ejemplo alternando columnas rojas/azules, o usando un degradado. Un truco: usar el índice para variar color: si `(i+j)` es par -> rojo, si es impar -> azul, etc., o algo más sofisticado como calcular color por coordenada normalizada.
- **Reacción al audio:** Para lograr respuesta al sonido, podríamos mapear la intensidad de ciertas frecuencias a transformaciones de la cuadrícula. Por ejemplo, tomar el nivel de graves para hacer *temblar* la malla (ligeros desplazamientos aleatorios a los puntos cuando hay bajos fuertes), o usar frecuencia media para crear ondas: seleccionar una fila o columna y desplazar sus puntos un poco en X o Y según la amplitud (creando una onda transversal en la malla). Otra opción: variar el **tamaño** o brillo de los puntos con el volumen global (puntos más grandes/brillantes en partes más fuertes).
- **Rotación 3D:** La parte más compleja es simular la rotación en perspectiva (como *Rotating Particle*). Podemos implementar una simple proyección 3D manualmente: asumir la cuadrícula está en un plano 3D y aplicar una rotación, luego proyectar a 2D. Por ejemplo, asignemos a cada punto también una coordenada z (que inicializamos 0 para un plano frontal). Para rotar sobre el eje Y (izq-der):

```
const angle = t * 0.01; // ángulo incremental con el tiempo
for (let i=0; i<rows; i++){
  for (let j=0; j<cols; j++){
    let px = j * spacing - centerX; // trasladar origen al centro del plano
    let py = i * spacing - centerY;
    let pz = 0;
    // Rotación alrededor del eje Y:
    let cosA = Math.cos(angle), sinA = Math.sin(angle);
    let x_rot = px * cosA - pz * sinA;
    let z_rot = px * sinA + pz * cosA;
    // Proyección perspectiva (simple):
    let depth = 500; // distancia del ojo
    let scale = depth / (depth + z_rot);
    let screenX = centerX + x_rot * scale;
    let screenY = centerY + py * scale;
    // guardar screenX, screenY en p.x, p.y
  }
}
```

Este código rotaría los puntos en torno al eje vertical que pasa por el centro de la cuadrícula. La proyección simple hace que puntos con `z_rot` hacia atrás se junten (`scale < 1`) y hacia delante se expandan (`scale > 1`), dando la ilusión 3D. Luego se dibujan en esas posiciones proyectadas. También se

puede aprovechar `z_rot` para alterar el tamaño del punto (`ctx.fillRect` más grande si el punto está “cerca”).

Para rotar continuamente, incrementamos `angle` cada frame. Podemos también rotar en torno a X para inclinar hacia arriba/abajo de forma similar.

Optimización: Particle es relativamente ligero, pero la rotación 3D manual puede costar si la malla es muy densa: - **Tamaño de malla moderado:** No hace falta 100x100 puntos; con una grilla de ~16x16 (256 puntos) o 20x20 (400 puntos) es suficiente para apreciar el efecto. En móviles incluso 10x10 podría bastar. Ajustar el `spacing` (separación) para llenar la pantalla adecuadamente con menos puntos. - **Cálculo matemático vectorizado:** Al proyectar puntos, son muchas multiplicaciones y sumas. Podemos intentar usar estructuras vectoriales o TypedArrays, pero en JS puro probablemente lo mejor es escribir bucles simples (como arriba) y confiar en la optimización JIT. Sin embargo, evitar *branching* dentro de los bucles (por ejemplo, no usar `if` dentro de cada punto si se puede calcular de antemano cosas) ayuda. En el pseudocódigo de rotación, calculamos `cosA` y `sinA` una vez por frame, fuera de los loops, para no recalcular por punto. - **Double buffering estático:** Si no implementamos rotación, la cuadrícula sería estática salvo vibración. En tal caso, podríamos dibujar todos los puntos en un canvas fijo y luego simplemente cambiar sus propiedades (como brillo) con CSS filters – aunque esto es avanzado y quizá innecesario. Pero, si rotamos, todo cambia cada frame, así que hay que redibujar. - **Usar `requestAnimationFrame` correctamente:** Como siempre, para sincronizar la rotación suave. Si se quiere limitar la velocidad de rotación en sistemas lentos, se podría rotar el plano sólo en cada 2º frame para la mitad de carga (aunque esto reduciría la suavidad del giro). - **Redibujar sólo si hay cambio:** Si la música está en una sección silenciosa y la visualización Particle no rota (ej. solo parpadea con audio), podemos pausar la repintura continua y solo dibujar cuando haya eventos de audio significativos. Esto ahorra CPU. Por ejemplo, si en un preset solo queremos que reaccione a audio, podríamos no llamar a `requestAnimationFrame` constantemente, sino desencadenar un redraw cuando el analyser detecte un pico. Sin embargo, para la mayoría de implementaciones es más sencillo seguir con animación continua pero tener en cuenta posibles pausas (por ejemplo, detener animación si el audio está pausado). - **Canvas sobre SVG:** Dibujar 200-300 círculos en SVG cada frame (actualizando sus atributos) sería mucho menos eficiente que hacerlo en canvas 2D, por lo que definitivamente usar canvas para este caso de muchos elementos pequeños. - **Evitar subpíxel en exceso:** Dibujar puntos exactamente en coordenadas enteras (o con .5 offset si se quiere alinearlos con píxeles) puede evitar difuminado por anti-aliasing. Dado que son puntos, si se dibujan en posiciones fraccionales, el navegador los renderizará con suavizado, lo cual en 300 puntos puede costar un poco. Bloquearlos en píxeles ayuda a la GPU a dibujar puntos claros. En la proyección 3D, esto es difícil porque habrá decimales, pero se podría considerar redondear la posición final del punto (sacrificando un pelín de precisión visual por rendimiento).

Siguiendo estas optimizaciones, la visualización *Particle* se puede reproducir con éxito: un entramado de puntos oscilantes o rotativos que, pese a involucrar cálculos 3D básicos, debería funcionar a buen ritmo incluso en hardware modesto, dado el limitado número de primitivas que maneja.

Spikes (Púas/Astros)

Descripción visual: “*Spikes*” es otra visualización clásica de WMP (presente en versiones 7 a 10) con un estilo más geométrico y minimalista. Consiste principalmente en **formas circulares que se expanden y contraen** con la música ¹⁷. El nombre *Spikes* (espinas/púas) proviene de la apariencia que toma la forma en ciertos momentos: aunque básicamente es un círculo, su contorno puede volverse irregular, como un círculo con púas o protuberancias que cambian dinámicamente (*preset Amoeba*, por ejemplo, parecía una ameba cambiando de forma). Los presets incluidos eran *Spike* y *Amoeba* ¹⁸: - En el preset

Spike, la visualización mostraba un anillo u orbe en el centro de la pantalla, normalmente de color amarillo (en modo predeterminado) ¹⁹. Este anillo pulsaba al ritmo del audio: con sonidos fuertes, el círculo se agrandaba rápidamente y quizás lanzaba “ondas” o destellos desde su perímetro, para luego volver a su tamaño normal. - En el preset *Amoeba*, el círculo tomaba una forma más orgánica: en lugar de un borde perfectamente redondo, tenía ondulaciones y picos que variaban constantemente, dando una impresión de criatura unicelular cambiando de forma. Con la música, estas ondulaciones se acentuaban o giraban alrededor del centro. A volumen alto, la figura podía vibrar intensamente y sus “púas” crecían más largas ²⁰.

El fondo de Spikes era simple (negro), y la figura podía cambiar de color dependiendo del modo (por ejemplo, en algunos modos de WMP aparecía verde pálido) ¹⁹. No había múltiples objetos: siempre es un solo ente circular (o dos en alguna variación secundaria), lo que le da un enfoque más limpio comparado con visualizaciones de muchas partes.

Recreación con Canvas/SVG: *Spikes* se puede recrear eficientemente dibujando formas de círculo y modificando su radio y contorno según la música: - **Círculo pulsante básico:** Para el efecto principal, dibujar un círculo centrado (`ctx.arc(centerX, centerY, radius, 0, 2*Math.PI)`). El `radius` se actualizará en cada frame en función del audio. Por ejemplo, podríamos tomar el nivel de volumen global o de bajos, normalizarlo y usarlo para incrementar el radio base. Pseudocódigo:

```
const baseRadius = 50;
const maxRadiusIncrease = 30;
// volumeLevel entre 0 y 1
const currentRadius = baseRadius + volumeLevel * maxRadiusIncrease;
ctx.beginPath();
ctx.arc(cx, cy, currentRadius, 0, 2*Math.PI);
ctx.stroke();
```

Esto hará que el círculo “respire” con la música (más grande con sonido fuerte, más pequeño en silencios). - **Amoeba (contorno irregular):** Para lograr las púas o irregularidades, podemos usar una representación polar del círculo y alterar el radio en función del ángulo. Por ejemplo:

```
ctx.beginPath();
const spikes = 20; // número de irregularidades
const spikeLength = volumeLevel * 15; // longitud extra de púa según volumen
for(let i=0; i<=spikes; i++){
  const theta = (i / spikes) * 2 * Math.PI;
  // calcular radio modulado con una senoide
  const irregularity = Math.sin(theta * spikes * 0.5 + phase) * 0.5 + 0.5;
  const r = currentRadius + irregularity * spikeLength;
  const x = cx + r * Math.cos(theta);
  const y = cy + r * Math.sin(theta);
  if(i===0) ctx.moveTo(x, y); else ctx.lineTo(x, y);
}
ctx.closePath();
ctx.fill();
```

En este fragmento, subdividimos el círculo en `spikes` segmentos angulares. Calculamos un factor `irregularity` usando una onda sinusoidal que varía con el ángulo (y se podría animar en el tiempo

mediante `phase`). Esto produce un radio que oscila creando baches. El valor `spikeLength` depende del volumen actual, así las púas se alargan con el sonido. Finalmente dibujamos con `fill` o `stroke`. Usar `fill` con un color sólido daría una masa tipo ameba rellena; `stroke` solo el contorno (más parecido a WMP que era línea).

Para animar esta ameba, actualizaríamos `phase` lentamente (para rotar las irregularidades alrededor) y recalculáramos `volumeLevel` cada frame.

- **Destellos con la música:** Para darle dramatismo, en picos de audio podríamos dibujar un breve destello. Ej: si `volumeLevel` supera cierto umbral (beat detectado), dibujar rápidamente un círculo extra o un efecto radial. Una idea sencilla: dibujar el círculo con un color más brillante o un radio ligeramente mayor *solo* en ese frame, creando un “flash”. Esto se logra incorporando la lógica en el cálculo de radius (p.ej., si beat detectado, añadir +X al radius sólo en ese instante).
- **SVG alternativa:** Como Spikes es esencialmente un solo path (posiblemente complejo en el caso de ameba), podría dibujarse con SVG `<path>` o `<circle>` y actualizar sus atributos via JS. Por ejemplo, `<circle cx=cx cy=cy r=...>` actualizando `r` para el pulso. Para la forma ameba, habría que recalcular la `d` de un path en cada frame, lo cual es similar a calcular puntos para canvas. Dado que es un solo elemento, SVG podría manejarlo bien. Sin embargo, canvas es perfectamente apto y probablemente más sencillo de controlar en cuanto a animación continua.

Optimización: Como Spikes implica dibujar muy pocos elementos (1 forma principal), ya es muy liviano. Aún así, algunas consideraciones: - **Disminuir segmentos en la ameba:** En el ejemplo pusimos `spikes = 20` para dividir el contorno. Este número determina la suavidad de la forma. Se podría usar incluso menos (8, 12) para formas más angulares y simplificar cálculo. O, si se quiere más suavidad, se puede usar curva Bézier entre puntos en vez de muchos segmentos lineales. Por ejemplo, en canvas podríamos usar `quadraticCurveTo` para suavizar la transición entre púas sin multiplicar excesivamente el número de vértices. - **Calcular solo lo necesario:** Si el preset activo es *Spike* (círculo perfecto), no calcular irregularidades en absoluto. Es trivial, pero asegurarse de separar esos modos. Similarmente, si *Amoeba* está activo, podríamos fijar un número de púas moderado. - **Uso de `requestAnimationFrame` e inercia:** No es necesario actualizar el círculo a 60 FPS si la música no cambia tan rápido. Sin embargo, dado que la música sí puede tener ritmo rápido, probablemente conviene 60 FPS para que los pulsos grandes coincidan con los beats. Podemos, sin embargo, aplicar una ligera inercia al radio para evitar cambios bruscos frame a frame (lo que también sirve de micro-optimización). Por ejemplo, en lugar de asignar directamente `currentRadius = baseRadius + vol * maxIncrease`, hacer `currentRadius = currentRadius * 0.8 + targetRadius * 0.2`. Esto suaviza la animación y evita “saltos” que podrían ocurrir con lecturas de volumen variables. Al mismo tiempo, suavizar reduce la alta frecuencia de cambio, permitiendo que quizás no necesitemos recalcular completamente la forma en cada frame si el cambio fue mínimo. En otras palabras, si `currentRadius` apenas cambió, podríamos omitir redibujar el contorno con tantos puntos (aunque redibujar un path no es costoso, cualquier ahorro cuenta). - **Evitar rellenos complejos:** Si usamos `fill`, tener en cuenta que rellenar un polígono complejo con transparencias o gradientes podría ser más costoso que un simple trazo. Un contorno con `stroke` probablemente sea suficiente y más cercano a la estética original. Si se quiere un relleno, usar colores sólidos en lugar de patrones o gradientes elaborados es mejor para la GPU. - **One canvas, no extra layers:** Spikes no requiere capas adicionales. Dibujar el círculo directamente sobre el fondo cada frame es suficiente. Podemos simplemente limpiar con `ctx.clearRect` todo el canvas y dibujar de nuevo la forma. Dado lo mínimo del dibujo, limpiar no supone problema. En caso de querer dejar estelas (pero la visualización original no lo hacía tanto), habría que considerar no limpiar por completo, pero eso es extra.

En síntesis, *Spikes* es sencillo de mantener eficiente. Con un solo path dibujado por frame, prácticamente cualquier dispositivo moderno puede manejarlo a 60 FPS. Aplicando los detalles anteriores garantizamos que la experiencia visual siga siendo fluida: un círculo que late y se deforma orgánicamente al son de la música, sin que la CPU ni la GPU rompan a sudar.

Conclusión y consideraciones finales

Hemos repasado las visualizaciones generativas clásicas de Windows Media Player – Bars and Waves, Ambience, Battery, Alchemy, Particle, Spikes – describiendo sus patrones visuales característicos y proponiendo cómo recrearlos con tecnologías web ligeras. Empleando principalmente la etiqueta `<canvas>` de HTML5 para dibujar gráficas 2D (y eventualmente técnicas básicas de proyección para simular 3D), es posible **interpretar fielmente** la esencia de estas visualizaciones. Cada una presenta desafíos diferentes (desde manejar muchas primitivas en Bars and Waves/Particle hasta generar formas fluidas en Ambience/Alchemy), pero en todos los casos se puede lograr un resultado visualmente atractivo optimizando el código y los cálculos:

- **Canvas 2D** proporciona suficientes herramientas (líneas, arcos, rellenos, transformaciones) para la mayoría de efectos. No se requirieron frameworks ni librerías externas, lo cual mantiene el código manejable y ligero.
- **SVG** es viable en casos con pocos elementos vectoriales (por ejemplo, Spikes), pero en general canvas ofrece un control de pixel más directo, útil para las animaciones continuas basadas en audio.
- **Rendimiento:** aplicando técnicas como reducción de complejidad geométrica, control de frame rate, reutilización de cálculos y separación de capas, aseguramos que incluso dispositivos móviles puedan renderizar estas visualizaciones en tiempo real sin caídas notorias de frames. Recordemos que muchos de estos efectos se originaron en épocas de hardware modesto, por lo que sus recreaciones bien optimizadas hoy ocupan una fracción del poder de cómputo disponible.

En definitiva, revivir las clásicas animaciones de Windows Media Player en la plataforma web es absolutamente factible. Con HTML, CSS y JavaScript puro hemos delineado cómo construir cada visualización emblemática. Esto no solo permite disfrutar de la nostalgia visual, sino que también sirve como ejercicio de creatividad técnica: combinar matemáticas, audio y gráficos de manera eficiente para producir arte generativo interactivo. Siguiendo las pautas anteriores, un desarrollador puede implementar un “visualizador web” multi-modo que rinda homenaje a aquellas experiencias de los 2000, con código moderno, ligero y mantenible, listo para funcionar en navegadores actuales. 1 7

1 Set up the Visualizations | Windows Media Player

<https://www.windows-media-player.com/set-up-the-visualizations/>

2 3 4 5 6 Ambience | Windows Media Player Visualization Wiki | Fandom

<https://wmpvis.fandom.com/wiki/Ambience>

7 8 9 10 Battery | Windows Media Player Visualization Wiki | Fandom

<https://wmpvis.fandom.com/wiki/Battery>

11 12 13 Alchemy | Windows Media Player Visualization Wiki | Fandom

<https://wmpvis.fandom.com/wiki/Alchemy>

14 15 16 Particle | Windows Media Player Visualization Wiki | Fandom

<https://wmpvis.fandom.com/wiki/Particle>

17 18 19 20 Spikes | Windows Media Player Visualization Wiki | Fandom

<https://wmpvis.fandom.com/wiki/Spikes>