Introduction to WebAssembly

What is WebAssembly?

WebAssembly (WASM) represents a revolutionary step in web development, serving as the fourth language to run natively in web browsers alongside HTML, CSS, and JavaScript. At its core, WebAssembly is a low-level, binary instruction format designed as a portable compilation target for high-level languages like C, C++, C#, Rust, and Go. Unlike JavaScript, which is interpreted or just-in-time compiled, WASM provides near-native performance by executing pre-compiled bytecode in a secure, sandboxed environment.

The technology operates on a stack-based virtual machine that executes instructions at speeds approaching native applications. WASM modules are designed to be small, load quickly, and execute efficiently while maintaining web platform compatibility. The format supports linear memory models and direct memory access patterns, while maintaining browser security through capability-based security models.

WebAssembly's role in my future career

For Computer Science and IT students, WebAssembly opens unprecedented opportunities across multiple domains. In web development, WASM enables CPU-intensive applications like image processing, video editing, gaming engines, and mathematical simulations to run seamlessly in browsers. This technology bridges the gap between desktop application performance and web accessibility.

The gaming industry increasingly relies on WASM to port existing C++ game engines to web platforms, while creative software companies use it to bring desktop-class applications to browsers. Additionally, emerging fields like blockchain development, machine learning inference, and real-time data visualization heavily leverage WASM for performance-critical components. As cloud computing evolves, WASM's sandboxed execution model makes it ideal for serverless and edge computing scenarios.

Integration with my degree

WebAssembly complements various courses throughout my degree. In **Computer Graphics (COS 220)** and **Multimedia Systems**, WASM enables real-time rendering and complex visual effects, directly applying mathematical concepts from **Linear Algebra (WTW 218)** for matrix transformations. **Data Structures and Algorithms (COS 212)** is important because that knowledge becomes crucial when optimizing WASM modules for memory efficiency and computational performance.

Software Engineering (COS 301) principles apply directly to WASM development workflows, particularly in build systems and cross-platform deployment. **Operating Systems (COS 226)** concepts like memory management and process isolation mirror WASM's security model and execution environment. Advanced courses like **Computer Vision** and **Machine Learning** increasingly use WASM for deploying trained models in browsers, while **Database Systems (COS 326)** intersects through in-browser SQL engines and client-side data processing. This technology effectively unifies theoretical computer science concepts with practical web development, making it an essential skill for modern software developers.

Building an Interactive Audio Visualizer with C# and WebAssembly

This tutorial will guide you through creating a real-time audio visualizer that runs in web browsers using C# compiled to WebAssembly. By the end, you'll understand the complete WebAssembly development workflow and have built an interactive multimedia application.

Prerequisites and Setup

Why C# for WebAssembly?

C# offers an excellent entry point into WebAssembly development for several reasons. Unlike lower-level languages like C++ or Rust, C# provides familiar object-oriented syntax while still compiling to high-performance WebAssembly bytecode. Microsoft's .NET runtime includes built-in WebAssembly support, eliminating the need for complex toolchains like Emscripten. This approach allows us to leverage C#'s strong typing system, memory management, and extensive standard library while achieving near-native performance in browsers.

Development Environment Setup

Step 1: Install .NET 8 SDK

Download and install the .NET 8 SDK from Microsoft's official website. This SDK includes the necessary tools for WebAssembly compilation:
https://dotnet.microsoft.com/en-us/download/dotnet/8.0
After you have installed dotnet be sure to restart your CLI for the changes to take effect.

```
# Verify installation
dotnet --version
# Should show 8.0.x or higher
```

Step 2: Install WebAssembly Workload

The WebAssembly workload adds browser-specific compilation targets to .NET:

```
dotnet workload install wasm-tools
```

This workload installs the mono runtime configured for WebAssembly, the necessary build targets, and debugging tools. Without this workload, .NET cannot generate WebAssembly modules from C# code.

Step 3: Verify WebAssembly Support

```
dotnet new list
```

You should see WebAssembly project templates available, confirming proper installation.

Project Initialization and Structure

Creating the WebAssembly Project

Navigate to where you want to store the project then:

```
mkdir AudioVisualizerWasm
cd AudioVisualizerWasm
dotnet new console
```

Configuring for WebAssembly Compilation

Open the `.csproj` file and replace its contents with:

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <OutputType>Exe</OutputType>
    <RuntimeIdentifier>browser-wasm</RuntimeIdentifier>
    <UseAppHost>false</UseAppHost>
    <PublishTrimmed>false</PublishTrimmed>
    <InvariantGlobalization>true</InvariantGlobalization>
    <WasmGenerateRunV8Script>true</WasmGenerateRunV8Script>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
    <AssemblyName>AudioVisualizerWasm</AssemblyName>
    <WasmBuildNative>false</WasmBuildNative>
  </PropertyGroup>
</Project>
```

Key Configuration Explanations:

- `RuntimeIdentifier>browser-wasm`: Tells .NET to compile for WebAssembly instead of native platforms
- `WasmBuildNative>false`: Uses the mono interpreter instead of AOT compilation for faster build times
- `InvariantGlobalization>true`: Reduces bundle size by excluding localization data
- `AllowUnsafeBlocks>true`: Enables pointer operations needed for performance-critical graphics code

Implementing the Core Audio Visualizer Logic

Understanding the Architecture

Our audio visualizer consists of a single main component:

1. **C# WebAssembly Module**: Handles particle physics calculations and audio analysis

This simplified architecture focuses on the core WebAssembly functionality without requiring external browser integration.

Building the Particle System

Create the main `Program.cs` with the particle physics engine:
Remove the default code and add:

```
using System;
using System.Runtime.InteropServices.JavaScript;

namespace AudioVisualizerWasm
{
    public partial class Program
    {
        private static AudioVisualizer visualizer = new AudioVisualizer();

        public static void Main()
        {
            Console.WriteLine("Audio Visualizer WebAssembly module loaded");
        }

        [JSExport]
        public static void UpdateAudioData(string frequencyDataJson)
        {
            // Parse JSON string to float array
            var frequencyData = System.Text.Json.JsonSerializer.Deserialize<float[]>(frequencyDataJson);
            if (frequencyData != null)
                visualizer.UpdateFrequencyData(frequencyData);
        }

        [JSExport]
        public static void UpdateParticles(double deltaTime, double mouseX, double mouseY, bool mousePressed)
        {
            visualizer.UpdateParticles(deltaTime, mouseX, mouseY, mousePressed);
        }

        [JSExport]
        public static int GetParticleCount() => visualizer.GetParticleCount();

        [JSExport]
        public static string GetParticleData()
        {
            return visualizer.GetParticleDataJson();
        }

        [JSExport]
        public static void SetVisualizationMode(int mode)
        {
            visualizer.SetVisualizationMode(mode);
        }

        [JSExport]
        public static void SetSensitivity(double sensitivity)
        {
            visualizer.SetSensitivity((float)sensitivity);
        }

        [JSExport]
        public static double GetTotalEnergy() => visualizer.GetTotalEnergy();
```

```
        [JSExport]
        public static double GetSpectralCentroid() => visualizer.GetSpectralCentroid();

        [JSExport]
        public static double GetLowFreqEnergy() => visualizer.GetLowFreqEnergy();

        [JSExport]
        public static double GetMidFreqEnergy() => visualizer.GetMidFreqEnergy();

        [JSExport]
        public static double GetHighFreqEnergy() => visualizer.GetHighFreqEnergy();
    }

    public class AudioVisualizer
    {
        private const int MAX_PARTICLES = 200; // Reduced for better performance
        private const int FREQUENCY_BANDS = 128;

        private Particle[] _particles;
        private float[] _frequencyData;
        private Random _random;

        private int _visualizationMode = 0;
        private float _sensitivity = 1.0f;
        private double _time = 0;

        public AudioVisualizer()
        {
            _particles = new Particle[MAX_PARTICLES];
            _frequencyData = new float[FREQUENCY_BANDS];
            _random = new Random();

            // Initialize particles
            for (int i = 0; i < MAX_PARTICLES; i++)
            {
                _particles[i] = new Particle();
                ResetParticle(i);
            }
        }

        public void UpdateFrequencyData(float[] frequencyData)
        {
            int dataLength = Math.Min(frequencyData.Length, FREQUENCY_BANDS);
            Array.Copy(frequencyData, _frequencyData, dataLength);
        }

        public void UpdateParticles(double deltaTime, double mouseX, double mouseY, bool
mousePressed)
        {
            _time += deltaTime;

            // Calculate overall audio energy and spectral characteristics
            float totalEnergy = CalculateEnergyBand(0, FREQUENCY_BANDS);
            float lowFreqEnergy = CalculateEnergyBand(0, 32);
            float midFreqEnergy = CalculateEnergyBand(32, 96);
            float highFreqEnergy = CalculateEnergyBand(96, FREQUENCY_BANDS);
```

```
            // Calculate spectral centroid for dynamic effects
            float spectralCentroid = CalculateSpectralCentroid();

            // Update each particle
            for (int i = 0; i < MAX_PARTICLES; i++)
            {
                UpdateParticle(i, deltaTime, totalEnergy, lowFreqEnergy, midFreqEnergy,
highFreqEnergy, spectralCentroid, mouseX, mouseY, mousePressed);
            }
        }

        private void UpdateParticle(int index, double deltaTime, float totalEnergy, float
lowFreqEnergy, float midFreqEnergy, float highFreqEnergy, float spectralCentroid,
            double mouseX, double mouseY, bool mousePressed)
        {
            ref Particle particle = ref _particles[index];

            if (particle.Life <= 0)
            {
                ResetParticle(index);
                return;
            }

            // Apply different forces based on visualization mode
            switch (_visualizationMode)
            {
                case 0: // Enhanced radial burst with energy layers
                    UpdateEnhancedRadialMode(ref particle, deltaTime, totalEnergy,
lowFreqEnergy, midFreqEnergy, highFreqEnergy, spectralCentroid);
                    break;
                case 1: // Dynamic orbital system
                    UpdateDynamicOrbitalMode(ref particle, deltaTime, totalEnergy,
lowFreqEnergy, midFreqEnergy, highFreqEnergy, spectralCentroid);
                    break;
                case 2: // Spectral wave patterns
                    UpdateSpectralWaveMode(ref particle, deltaTime, totalEnergy,
lowFreqEnergy, midFreqEnergy, highFreqEnergy, spectralCentroid);
                    break;
            }


            // Update position and apply physics
            particle.X += particle.VelocityX * (float)deltaTime;
            particle.Y += particle.VelocityY * (float)deltaTime;
            particle.VelocityX *= 0.98f; // Apply drag
            particle.VelocityY *= 0.98f;
            particle.Life -= (float)deltaTime;

            // Update color based on spectral characteristics
            particle.ColorHue = (particle.ColorHue + totalEnergy * 2.0f + spectralCentroid
* 1.5f) % 360.0f;
        }

        private void UpdateEnhancedRadialMode(ref Particle particle, double deltaTime,
float totalEnergy, float lowFreqEnergy, float midFreqEnergy, float highFreqEnergy, float
```

```
spectralCentroid)
        {
            float centerX = 400; // Canvas center
            float centerY = 300;
            float dx = particle.X - centerX;
            float dy = particle.Y - centerY;
            float distance = (float)Math.Sqrt(dx * dx + dy * dy);

            if (distance > 0)
            {
                // Multi-layered radial forces based on frequency content
                float baseForce = totalEnergy * _sensitivity * 80.0f;
                float lowForce = lowFreqEnergy * _sensitivity * 60.0f;
                float spiralForce = midFreqEnergy * _sensitivity * 40.0f;

                // Main radial explosion
                particle.VelocityX += (dx / distance) * baseForce * (float)deltaTime;
                particle.VelocityY += (dy / distance) * baseForce * (float)deltaTime;

                // Add spiral motion based on spectral centroid
                float spiralAngle = (float)Math.Atan2(dy, dx) + spectralCentroid * 2.0f;
                particle.VelocityX += (float)Math.Cos(spiralAngle) * spiralForce *
(float)deltaTime;
                particle.VelocityY += (float)Math.Sin(spiralAngle) * spiralForce *
(float)deltaTime;

                // High frequency jitter
                float jitter = highFreqEnergy * _sensitivity * 20.0f;
                particle.VelocityX += ((float)_random.NextDouble() - 0.5f) * jitter *
(float)deltaTime;
                particle.VelocityY += ((float)_random.NextDouble() - 0.5f) * jitter *
(float)deltaTime;
            }
        }

        private void UpdateDynamicOrbitalMode(ref Particle particle, double deltaTime,
float totalEnergy, float lowFreqEnergy, float midFreqEnergy, float highFreqEnergy, float
spectralCentroid)
        {
            float centerX = 400;
            float centerY = 300;
            float dx = particle.X - centerX;
            float dy = particle.Y - centerY;
            float distance = (float)Math.Sqrt(dx * dx + dy * dy);

            if (distance > 0)
            {
                // Dynamic orbital speeds based on frequency content
                float orbitalSpeed = (midFreqEnergy + spectralCentroid) * _sensitivity *
40.0f;

                // Orbital force perpendicular to radius
                particle.VelocityX += -dy * orbitalSpeed * (float)deltaTime * 0.01f;
                particle.VelocityY += dx * orbitalSpeed * (float)deltaTime * 0.01f;

                // Radial pulsing based on low frequencies
```

```
                float radialPulse = (lowFreqEnergy - 0.3f) * _sensitivity * 30.0f;
                particle.VelocityX += (dx / distance) * radialPulse * (float)deltaTime;
                particle.VelocityY += (dy / distance) * radialPulse * (float)deltaTime;

                // High frequency orbital variations
                float orbitVariation = highFreqEnergy * _sensitivity * 15.0f;
                float variationAngle = (float)(_time * 3.0f + particle.Life * 2.0f);
                particle.VelocityX += (float)Math.Cos(variationAngle) * orbitVariation *
(float)deltaTime;
                particle.VelocityY += (float)Math.Sin(variationAngle) * orbitVariation *
(float)deltaTime;
            }
        }

        private void UpdateSpectralWaveMode(ref Particle particle, double deltaTime, float
totalEnergy, float lowFreqEnergy, float midFreqEnergy, float highFreqEnergy, float
spectralCentroid)
        {
            // Multi-frequency wave system
            float lowWaveAmplitude = lowFreqEnergy * _sensitivity * 80.0f;
            float midWaveAmplitude = midFreqEnergy * _sensitivity * 60.0f;
            float highWaveAmplitude = highFreqEnergy * _sensitivity * 40.0f;

            float lowWaveFreq = 0.01f + spectralCentroid * 0.02f;
            float midWaveFreq = 0.02f + spectralCentroid * 0.03f;
            float highWaveFreq = 0.03f + spectralCentroid * 0.04f;

            // Layered wave forces
            float waveForceX = (float)(
                Math.Sin(_time * lowWaveFreq + particle.Y * 0.005f) * lowWaveAmplitude +
                Math.Sin(_time * midWaveFreq + particle.Y * 0.01f) * midWaveAmplitude +
                Math.Sin(_time * highWaveFreq + particle.Y * 0.02f) * highWaveAmplitude
            );

            float waveForceY = (float)(
                Math.Cos(_time * lowWaveFreq + particle.X * 0.005f) * lowWaveAmplitude +
                Math.Cos(_time * midWaveFreq + particle.X * 0.01f) * midWaveAmplitude +
                Math.Cos(_time * highWaveFreq + particle.X * 0.02f) * highWaveAmplitude
            );

            particle.VelocityX += waveForceX * (float)deltaTime;
            particle.VelocityY += waveForceY * (float)deltaTime;

            // Add spiral component based on total energy
            float spiralForce = totalEnergy * _sensitivity * 30.0f;
            float spiralAngle = (float)_time * 2.0f + particle.Life;
            particle.VelocityX += (float)Math.Cos(spiralAngle) * spiralForce *
(float)deltaTime;
            particle.VelocityY += (float)Math.Sin(spiralAngle) * spiralForce *
(float)deltaTime;
        }

        private void ResetParticle(int index)
        {
            ref Particle particle = ref _particles[index];
```

```csharp
            particle.X = 400 + (_random.NextSingle() - 0.5f) * 50;
            particle.Y = 300 + (_random.NextSingle() - 0.5f) * 50;

            float angle = _random.NextSingle() * 2 * (float)Math.PI;
            float speed = _random.NextSingle() * 20 + 10;

            particle.VelocityX = (float)Math.Cos(angle) * speed;
            particle.VelocityY = (float)Math.Sin(angle) * speed;
            particle.Life = _random.NextSingle() * 3 + 2;
            particle.ColorHue = _random.NextSingle() * 360;
        }

        private float CalculateEnergyBand(int startBand, int endBand)
        {
            float energy = 0;
            for (int i = startBand; i < endBand && i < FREQUENCY_BANDS; i++)
            {
                energy += _frequencyData[i];
            }
            return energy / (endBand - startBand);
        }

        private float CalculateSpectralCentroid()
        {
            float weightedSum = 0;
            float totalEnergy = 0;

            for (int i = 0; i < FREQUENCY_BANDS; i++)
            {
                weightedSum += i * _frequencyData[i];
                totalEnergy += _frequencyData[i];
            }

            return totalEnergy > 0 ? (weightedSum / totalEnergy) / FREQUENCY_BANDS : 0.5f;
        }

        public int GetParticleCount() => MAX_PARTICLES;

        public string GetParticleDataJson()
        {
            var particleData = new float[MAX_PARTICLES * 6];
            for (int i = 0; i < MAX_PARTICLES; i++)
            {
                int dataIndex = i * 6;
                particleData[dataIndex] = _particles[i].X;
                particleData[dataIndex + 1] = _particles[i].Y;
                particleData[dataIndex + 2] = _particles[i].VelocityX;
                particleData[dataIndex + 3] = _particles[i].VelocityY;
                particleData[dataIndex + 4] = _particles[i].Life;
                particleData[dataIndex + 5] = _particles[i].ColorHue;
            }
            return System.Text.Json.JsonSerializer.Serialize(particleData);
        }

        public void SetVisualizationMode(int mode) => _visualizationMode = mode;
        public void SetSensitivity(float sensitivity) => _sensitivity = Math.Max(0.1f,
```

```
    Math.Min(3.0f, sensitivity));
            public float GetTotalEnergy() => CalculateEnergyBand(0, FREQUENCY_BANDS);
            public float GetSpectralCentroid() => CalculateSpectralCentroid();
            public float GetLowFreqEnergy() => CalculateEnergyBand(0, 32);
            public float GetMidFreqEnergy() => CalculateEnergyBand(32, 96);
            public float GetHighFreqEnergy() => CalculateEnergyBand(96, FREQUENCY_BANDS);
        }

        public struct Particle
        {
            public float X;
            public float Y;
            public float VelocityX;
            public float VelocityY;
            public float Life;
            public float ColorHue;
        }
    }
```

JSExport Attribute Explanation:

The `[JSExport]` attribute marks methods that JavaScript can call directly. This creates a bridge between the WebAssembly module and browser JavaScript, allowing real-time data exchange.

Performance Optimization Techniques:

- Using `ref` keyword for direct struct modification prevents unnecessary copying
- Pre-allocated arrays avoid garbage collection during real-time updates
- Struct-based particles minimize memory allocation overhead
- Efficient mathematical operations using native .NET math functions

Compilation Process and WebAssembly Generation

Building the WebAssembly Module

```
dotnet publish -c Release
```

If you get missing workload errors, run:

```
dotnet workload restore
```

Then retry the publish command.

This command performs several critical steps:

1. **IL Compilation**: C# source code compiles to Intermediate Language (IL)
2. **IL to WebAssembly Translation**: The mono runtime translates IL to WebAssembly bytecode
3. **Asset Generation**: Creates necessary JavaScript bootstrap files and WebAssembly modules

4. **Optimization**: Applies dead code elimination and other optimizations

Understanding the Generated Files

The build process creates several key files in `bin/Release/net8.0/browser-wasm/AppBundle/`:

- `AudioVisualizerWasm.wasm`: The actual WebAssembly bytecode containing your C# logic
- `dotnet.js`: JavaScript runtime that loads and manages the WebAssembly module
- Various supporting files for the .NET runtime

Creating the Web Interface

To see the visualizer in action, we need to create the web interface that displays the particles and handles audio input.

HTML Interface

Create an `index.html` file in the `AppBundle` directory:
the directory is `bin/Release/net8.0/browser-wasm/AppBundle/`

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Audio Visualizer - WebAssembly</title>
    <style>
      body {
        font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
        background: linear-gradient(
          135deg,
          #1a1a2e 0%,
          #16213e 50%,
          #0f3460 100%
        );
        margin: 0;
        padding: 20px;
        min-height: 100vh;
        /* overflow-x: hidden; */
        overflow-y: auto;
      }

      .container {
        max-width: 1200px;
        margin: 0 auto;
        background: rgba(255, 255, 255, 0.05);
        border-radius: 15px;
        backdrop-filter: blur(10px);
        box-shadow: 0 15px 35px rgba(0, 0, 0, 0.3);
        border: 1px solid rgba(255, 255, 255, 0.1);
      }

      .header {
        background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
        color: white;
        padding: 30px;
        text-align: center;
        position: relative;
      }

      .header h1 {
        margin: 0;
        font-size: 2.8rem;
        text-shadow: 2px 2px 8px rgba(0, 0, 0, 0.5);
        font-weight: 300;
      }

      .header p {
        margin: 10px 0 0 0;
        opacity: 0.9;
        font-size: 1.1rem;
      }

      .controls {
        padding: 25px;
```

```css
  display: flex;
  gap: 15px;
  align-items: center;
  flex-wrap: wrap;
  background: rgba(255, 255, 255, 0.03);
  border-bottom: 1px solid rgba(255, 255, 255, 0.1);
}

.btn {
  padding: 12px 25px;
  border: none;
  border-radius: 25px;
  cursor: pointer;
  font-weight: 600;
  transition: all 0.3s ease;
  font-size: 14px;
  backdrop-filter: blur(10px);
}

.btn:hover {
  transform: translateY(-3px);
  box-shadow: 0 8px 25px rgba(0, 0, 0, 0.2);
}

.btn-primary {
  background: linear-gradient(45deg, #667eea, #764ba2);
  color: white;
  position: relative;
  overflow: hidden;
}
.btn-primary::before {
  content: "";
  position: absolute;
  top: 0;
  left: -100%;
  width: 100%;
  height: 100%;
  background: linear-gradient(
    90deg,
    transparent,
    rgba(255, 255, 255, 0.3),
    transparent
  );
  transition: left 0.5s;
}
.btn-primary:hover::before {
  left: 100%;
}
.btn-primary:hover {
  transform: translateY(-3px) scale(1.05);
  box-shadow: 0 12px 35px rgba(102, 126, 234, 0.4);
}
.btn-success {
  background: linear-gradient(45deg, #56ab2f, #a8e6cf);
  color: white;
}
```

```css
    .btn-warning {
      background: linear-gradient(45deg, #f093fb, #f5576c);
      color: white;
    }
    .btn-info {
      background: linear-gradient(45deg, #4facfe, #00f2fe);
      color: white;
    }

    .control-group {
      display: flex;
      flex-direction: column;
      gap: 8px;
      color: white;
    }

    .control-group label {
      font-size: 12px;
      font-weight: 500;
      opacity: 0.8;
    }

    .slider {
      width: 150px;
      height: 6px;
      background: rgba(255, 255, 255, 0.2);
      border-radius: 3px;
      outline: none;
      -webkit-appearance: none;
    }

    .slider::-webkit-slider-thumb {
      -webkit-appearance: none;
      width: 18px;
      height: 18px;
      background: linear-gradient(45deg, #667eea, #764ba2);
      border-radius: 50%;
      cursor: pointer;
      box-shadow: 0 2px 10px rgba(0, 0, 0, 0.3);
    }

    .canvas-container {
      padding: 0;
      display: flex;
      justify-content: center;
      background: #000;
      position: relative;
    }

    #visualizerCanvas {
      border: none;
      cursor: pointer;
      display: block;
      background: radial-gradient(
        circle at center,
        rgba(26, 26, 46, 0.8) 0%,
```

```css
      rgba(0, 0, 0, 1) 100%
    );
  }

  .audio-controls {
    position: absolute;
    top: 20px;
    left: 20px;
    z-index: 10;
    display: flex;
    flex-direction: column;
    gap: 10px;
  }

  #audioUpload {
    display: none;
  }

  .upload-btn {
    padding: 10px 20px;
    background: rgba(255, 255, 255, 0.1);
    border: 2px dashed rgba(255, 255, 255, 0.3);
    border-radius: 10px;
    color: white;
    cursor: pointer;
    transition: all 0.3s ease;
    font-size: 14px;
    text-align: center;
    backdrop-filter: blur(10px);
  }

  .upload-btn:hover {
    background: rgba(255, 255, 255, 0.2);
    border-color: rgba(255, 255, 255, 0.5);
  }

  .visualizer-info {
    position: absolute;
    bottom: 20px;
    left: 20px;
    background: rgba(0, 0, 0, 0.4);
    border-radius: 15px;
    backdrop-filter: blur(15px);
    padding: 15px 20px;
    color: white;
    font-size: 14px;
    border: 1px solid rgba(255, 255, 255, 0.1);
    box-shadow: 0 10px 30px rgba(0, 0, 0, 0.3);
  }

  .visualizer-info .title {
    font-weight: 600;
    margin-bottom: 5px;
    color: #a8e6cf;
  }
```

```
      .info {
        padding: 25px;
        background: rgba(255, 255, 255, 0.03);
        border-top: 1px solid rgba(255, 255, 255, 0.1);
        color: white;
      }

      .tech-info {
        display: grid;
        grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
        gap: 20px;
        margin-top: 20px;
      }

      .tech-card {
        background: rgba(255, 255, 255, 0.05);
        padding: 20px;
        border-radius: 10px;
        backdrop-filter: blur(10px);
        border: 1px solid rgba(255, 255, 255, 0.1);
        transition: transform 0.3s ease;
      }

      .tech-card:hover {
        transform: translateY(-5px);
      }

      .tech-card h4 {
        margin: 0 0 10px 0;
        color: #a8e6cf;
      }

      .tech-card p {
        margin: 0;
        opacity: 0.8;
        line-height: 1.4;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="header">
        <h1>🎵 Audio Visualizer</h1>
        <p>Real-time particle system powered by C# WebAssembly</p>
      </div>

      <div class="controls">
        <button id="startBtn" class="btn btn-success" disabled>
          ⏳ Loading WebAssembly...
        </button>
        <label
          for="audioUpload"
          class="btn btn-primary"
          style="cursor: pointer; user-select: none;"
          >📁 Choose Audio File</label
        >
```

```html
          <input type="file" id="audioUpload" accept="audio/*" />
          <button id="resetBtn" class="btn btn-warning">🔄 Reset</button>
        </div>

        <div class="canvas-container">
          <div class="audio-controls">
            <div style="color: white; font-size: 12px; opacity: 0.7;">
              Supports: MP3, WAV, OGG, M4A
            </div>
          </div>

          <canvas id="visualizerCanvas" width="800" height="600"></canvas>

          <div class="visualizer-info">
            <div class="title">✨ Spectral Particle System</div>
            <div>128-band frequency analysis with dynamic particle physics</div>
          </div>
        </div>
      </div>

      <script type="module" src="./app.js"></script>
    </body>
  </html>
```

JavaScript Integration

Create a `main.js` file in the `AppBundle` directory:

```js
let dotnetInstance = null;
let canvas = null;
let ctx = null;
let audioContext = null;
let analyser = null;
let dataArray = null;
let source = null;
let isRunning = false;
let animationId = null;
let lastTime = 0;

// Mouse interaction
let mouseX = 0;
let mouseY = 0;
let mousePressed = false;

async function initializeWasm() {
  try {
    console.log("Loading WebAssembly module...");

    // Use the .NET 8 WebAssembly initialization
    const { dotnet } = await import("/_framework/dotnet.js");
    const api = await dotnet.create();

    const assemblyExports = await api.getAssemblyExports("AudioVisualizerWasm");
    dotnetInstance = assemblyExports.AudioVisualizerWasm.Program;

    console.log("Audio Visualizer WebAssembly module loaded successfully");

    initializeCanvas();
    setupEventHandlers();

    // Set default WebAssembly settings
    try {
      dotnetInstance.SetVisualizationMode(0); // Radial Burst mode
      dotnetInstance.SetSensitivity(1.0); // Default sensitivity
    } catch (error) {
      console.warn("Could not set default WebAssembly values:", error);
    }

    startVisualization();

    // Update UI to show module is ready
    const startBtn = document.getElementById("startBtn");
    if (startBtn && startBtn.textContent.includes("Loading")) {
      startBtn.textContent = "🎵 Start Audio";
      startBtn.disabled = false;
    }
  } catch (error) {
    console.error("Failed to initialize WebAssembly:", error);
    const startBtn = document.getElementById("startBtn");
    if (startBtn) {
      startBtn.textContent = "❌ Failed to Load";
      startBtn.disabled = true;
    }
```

```
    }
  }

  function initializeCanvas() {
    canvas = document.getElementById("visualizerCanvas");
    ctx = canvas.getContext("2d");

    // Enable alpha blending for particle trails
    ctx.globalCompositeOperation = "lighter";

    // Set canvas size properly
    canvas.width = 800;
    canvas.height = 600;
  }

  function setupEventHandlers() {
    // Start audio button
    document.getElementById("startBtn").addEventListener("click", startAudio);

    // File upload
    document
      .getElementById("audioUpload")
      .addEventListener("change", handleFileUpload);

    // Reset button
    document
      .getElementById("resetBtn")
      .addEventListener("click", resetVisualizer);

    // Remove mouse interaction - set fixed values
    mouseX = canvas.width / 2;
    mouseY = canvas.height / 2;
    mousePressed = false;
  }

  async function startAudio() {
    try {
      // Request microphone access
      const stream = await navigator.mediaDevices.getUserMedia({ audio: true });
      setupAudioAnalysis(stream);

      document.getElementById("startBtn").textContent = "🎵 Listening...";
      document.getElementById("startBtn").disabled = true;
    } catch (error) {
      console.error("Microphone access denied:", error);
      alert(
        "Please allow microphone access to use the audio visualizer, or load an audio file
instead.",
      );
    }
  }

  let audioElement = null;

  function handleFileUpload(e) {
    const file = e.target.files[0];
```

```
    if (file) {
      if (!dotnetInstance) {
        alert("WebAssembly module is still loading. Please wait and try again.");
        e.target.value = ""; // Clear the file input
        return;
      }

      console.log("Loading audio file:", file.name);
      audioElement = new Audio();
      audioElement.src = URL.createObjectURL(file);
      audioElement.crossOrigin = "anonymous";
      audioElement.controls = true;
      audioElement.loop = true;

      audioElement.addEventListener("loadeddata", () => {
        console.log("Audio file loaded, setting up analysis");
        setupAudioAnalysis(audioElement);
        showAudioControls(file.name);
        audioElement.play();
      });

      audioElement.addEventListener("error", (error) => {
        console.error("Error loading audio file:", error);
        alert("Error loading audio file. Please try a different file.");
        document.getElementById("startBtn").textContent = "🎵 Start Audio";
        document.getElementById("startBtn").disabled = false;
      });

      document.getElementById("startBtn").textContent = "🎵 Playing File";
      document.getElementById("startBtn").disabled = true;
    }
  }

  function setupAudioAnalysis(audioSource) {
    // Create audio context
    audioContext = new (window.AudioContext || window.webkitAudioContext)();
    analyser = audioContext.createAnalyser();

    // Configure analyser
    analyser.fftSize = 256; // 128 frequency bins for better frequency resolution
    analyser.smoothingTimeConstant = 0.3;

    // Create audio source
    if (audioSource instanceof MediaStream) {
      source = audioContext.createMediaStreamSource(audioSource);
    } else {
      source = audioContext.createMediaElementSource(audioSource);
      // Connect to destination so audio plays through speakers
      source.connect(audioContext.destination);
    }

    // Connect audio graph
    source.connect(analyser);

    // Create data array for frequency data
    dataArray = new Uint8Array(analyser.frequencyBinCount);
```

```
  console.log("Audio analysis setup complete");
}

function startVisualization() {
  console.log("Starting visualization...", { isRunning, animationId });

  // Stop any existing animation first
  if (animationId) {
    cancelAnimationFrame(animationId);
    animationId = null;
  }

  isRunning = true;
  lastTime = performance.now();
  animate(lastTime);
}

function animate(currentTime) {
  if (!isRunning) {
    console.log("Animation stopped");
    return;
  }

  // Ensure we always continue the animation loop
  animationId = requestAnimationFrame(animate);

  // Initialize lastTime if needed
  if (lastTime === 0) {
    lastTime = currentTime;
  }

  const deltaTime = Math.min((currentTime - lastTime) / 1000.0, 0.1); // Cap delta time to
prevent jumps
  lastTime = currentTime;

  // Clear canvas with proper fade for trail effect
  ctx.fillStyle = "rgba(0, 0, 0, 0.15)";
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  // Get audio frequency data
  let frequencyData = new Array(128).fill(0);
  let hasAudioData = false;

  if (analyser && dataArray) {
    analyser.getByteFrequencyData(dataArray);
    hasAudioData = true;

    // Normalize frequency data to 0-1 range
    for (let i = 0; i < Math.min(dataArray.length, 128); i++) {
      frequencyData[i] = dataArray[i] / 255.0;
    }
  } else {
    // Generate animated demo data when no audio
    const time = currentTime * 0.001;
    for (let i = 0; i < 128; i++) {
```

```
      const baseFreq = (Math.sin(time * 1.5 + i * 0.2) + 1) * 0.25;
      const variation = Math.sin(time * 3 + i * 0.1) * 0.15;
      const noise = Math.random() * 0.1;
      frequencyData[i] = Math.max(0, Math.min(1, baseFreq + variation + noise));
    }
  }

  // Always render visualization - either WebAssembly or fallback
  let rendered = false;

  if (dotnetInstance) {
    try {
      // Convert frequencyData array to JSON string for WebAssembly
      dotnetInstance.UpdateAudioData(JSON.stringify(frequencyData));
      dotnetInstance.UpdateParticles(deltaTime, mouseX, mouseY, mousePressed);

      renderParticles();
      rendered = true;
    } catch (error) {
      console.warn("WebAssembly rendering failed, using fallback:", error);
    }
  }

  // Use fallback visualization if WebAssembly failed or isn't available
  if (!rendered) {
    renderDemoVisualization(frequencyData);
  }
}

function renderParticles() {
  if (!dotnetInstance) return;

  const particleCount = dotnetInstance.GetParticleCount();

  // Note: For simplicity, we'll get particle data one by one
  // In a real implementation, you'd want to optimize this
  for (let i = 0; i < particleCount; i++) {
    // Since we can't easily access the particle data directly,
    // we'll create a simple particle rendering system
    renderParticleEffect(i);
  }
}

function renderParticleEffect(particleIndex) {
  // Generate enhanced particle effects based on overall frequency spectrum
  if (!analyser || !dataArray) return;

  const time = Date.now() * 0.001;
  const centerX = canvas.width / 2;
  const centerY = canvas.height / 2;

  // Get overall energy from frequency data
  const totalEnergy = dataArray
    ? Array.from(dataArray).reduce((a, b) => a + b, 0) /
      (dataArray.length * 255)
    : 0.5;
```

```
    const lowEnergy = dataArray
      ? Array.from(dataArray.slice(0, 20)).reduce((a, b) => a + b, 0) / (20 * 255)
      : 0.5;
    const midEnergy = dataArray
      ? Array.from(dataArray.slice(20, 80)).reduce((a, b) => a + b, 0) /
        (60 * 255)
      : 0.5;
    const highEnergy = dataArray
      ? Array.from(dataArray.slice(80, 128)).reduce((a, b) => a + b, 0) /
        (48 * 255)
      : 0.5;

    // Create multiple particle systems
    const systems = [
      {
        energy: lowEnergy,
        baseAngle: particleIndex * 0.1,
        radius: 80,
        speed: 0.3,
        color: 240,
        size: 3,
      },
      {
        energy: midEnergy,
        baseAngle: particleIndex * 0.15,
        radius: 120,
        speed: 0.5,
        color: 120,
        size: 2,
      },
      {
        energy: highEnergy,
        baseAngle: particleIndex * 0.2,
        radius: 160,
        speed: 0.8,
        color: 0,
        size: 1.5,
      },
    ];

    systems.forEach((system, sysIndex) => {
      if (system.energy > 0.1) {
        const angle = system.baseAngle + time * system.speed + sysIndex;
        const radiusVariation = Math.sin(time * 2 + particleIndex * 0.1) * 30;
        const finalRadius = system.radius + system.energy * 100 + radiusVariation;

        const x = centerX + Math.cos(angle) * finalRadius;
        const y = centerY + Math.sin(angle) * finalRadius;

        // Main particle with glow
        const hue = (system.color + time * 30 + particleIndex * 5) % 360;
        const size = system.size + system.energy * 8;

        // Outer glow
        const gradient = ctx.createRadialGradient(x, y, 0, x, y, size * 3);
        gradient.addColorStop(
```

```
          0,
          `hsla(${hue}, 90%, 70%, ${system.energy * 0.8})`,
        );
        gradient.addColorStop(
          0.5,
          `hsla(${hue}, 80%, 50%, ${system.energy * 0.4})`,
        );
        gradient.addColorStop(1, `hsla(${hue}, 70%, 30%, 0)`);

        ctx.fillStyle = gradient;
        ctx.beginPath();
        ctx.arc(x, y, size * 3, 0, Math.PI * 2);
        ctx.fill();

        // Core particle
        ctx.fillStyle = `hsla(${hue}, 95%, 85%, ${system.energy * 0.9})`;
        ctx.beginPath();
        ctx.arc(x, y, size, 0, Math.PI * 2);
        ctx.fill();

        // Energy burst effects
        if (system.energy > 0.7) {
          for (let burst = 0; burst < 3; burst++) {
            const burstAngle = angle + burst * Math.PI * 0.67;
            const burstDistance = size * 2 + Math.sin(time * 8 + burst) * 10;
            const burstX = x + Math.cos(burstAngle) * burstDistance;
            const burstY = y + Math.sin(burstAngle) * burstDistance;

            ctx.fillStyle = `hsla(${hue + 60}, 100%, 90%, ${system.energy * 0.6})`;
            ctx.beginPath();
            ctx.arc(burstX, burstY, size * 0.3, 0, Math.PI * 2);
            ctx.fill();
          }
        }

        // Connection lines to center when energy is high
        if (totalEnergy > 0.5 && Math.random() < system.energy * 0.3) {
          ctx.strokeStyle = `hsla(${hue}, 80%, 60%, ${system.energy * 0.5})`;
          ctx.lineWidth = 1 + system.energy * 2;
          ctx.lineCap = "round";
          ctx.beginPath();
          ctx.moveTo(centerX, centerY);
          ctx.lineTo(x, y);
          ctx.stroke();
        }
      }
    });
  }

  // HSV to RGB color conversion utility
  function hsvToRgb(h, s, v) {
    let r, g, b;
    const i = Math.floor(h * 6);
    const f = h * 6 - i;
    const p = v * (1 - s);
    const q = v * (1 - f * s);
```

```
  const t = v * (1 - (1 - f) * s);

  switch (i % 6) {
    case 0:
      ((r = v), (g = t), (b = p));
      break;
    case 1:
      ((r = q), (g = v), (b = p));
      break;
    case 2:
      ((r = p), (g = v), (b = t));
      break;
    case 3:
      ((r = p), (g = q), (b = v));
      break;
    case 4:
      ((r = t), (g = p), (b = v));
      break;
    case 5:
      ((r = v), (g = p), (b = q));
      break;
  }

  return [Math.round(r * 255), Math.round(g * 255), Math.round(b * 255)];
}

function showAudioControls(fileName) {
  const controlsContainer = document.querySelector(".audio-controls");

  // Remove existing controls if any
  const existingControls = document.getElementById("audioPlayerControls");
  if (existingControls) {
    existingControls.remove();
  }

  // Create audio controls container
  const audioControls = document.createElement("div");
  audioControls.id = "audioPlayerControls";
  audioControls.style.cssText = `
        background: rgba(0,0,0,0.7);
        border-radius: 10px;
        padding: 15px;
        margin-top: 10px;
        color: white;
        backdrop-filter: blur(10px);
    `;

  // File name display
  const fileNameDiv = document.createElement("div");
  fileNameDiv.textContent = `🎵 ${fileName}`;
  fileNameDiv.style.cssText =
    "margin-bottom: 10px; font-size: 14px; font-weight: bold;";

  // Insert audio element for controls
  audioElement.style.cssText = "width: 100%; height: 40px;";
```

```
    audioControls.appendChild(fileNameDiv);
    audioControls.appendChild(audioElement);

    controlsContainer.appendChild(audioControls);
  }

  function resetVisualizer() {
    console.log("Resetting visualizer...");

    // Stop animation
    isRunning = false;
    if (animationId) {
      cancelAnimationFrame(animationId);
      animationId = null;
    }

    // Stop and clear audio
    if (audioElement) {
      audioElement.pause();
      audioElement.currentTime = 0;
      audioElement = null;
    }

    // Close audio context
    if (audioContext && audioContext.state !== "closed") {
      audioContext.close();
      audioContext = null;
    }

    // Clear audio references
    analyser = null;
    dataArray = null;
    source = null;

    // Clear canvas completely
    if (ctx && canvas) {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.fillStyle = "black";
      ctx.fillRect(0, 0, canvas.width, canvas.height);
    }

    // Remove audio controls
    const audioControls = document.getElementById("audioPlayerControls");
    if (audioControls) {
      audioControls.remove();
    }

    // Reset file input
    document.getElementById("audioUpload").value = "";

    // Reset buttons
    const startBtn = document.getElementById("startBtn");
    startBtn.textContent = "🎵 Start Audio";
    startBtn.disabled = false;

    // Reset to default values
```

```
    // Reset WebAssembly if available
    if (dotnetInstance) {
      try {
        dotnetInstance.SetSensitivity(1.0);
        dotnetInstance.SetVisualizationMode(0);
      } catch (error) {
        console.warn("Error resetting WebAssembly state:", error);
      }
    }

    updateVisualizationDisplay();

    console.log("Visualizer reset complete");

    // Force restart visualization after a brief pause
    setTimeout(() => {
      console.log("Restarting visualization after reset...");
      startVisualization();
    }, 200);
  }

  function renderDemoVisualization(frequencyData) {
    if (!ctx || !canvas) return;

    const centerX = canvas.width / 2;
    const centerY = canvas.height / 2;
    const time = Date.now() * 0.001;

    // Calculate overall energy levels for dramatic effects
    const totalEnergy =
      frequencyData.reduce((a, b) => a + b, 0) / frequencyData.length;
    const lowFreqs = frequencyData.slice(0, 32).reduce((a, b) => a + b, 0) / 32;
    const midFreqs = frequencyData.slice(32, 96).reduce((a, b) => a + b, 0) / 64;
    const highFreqs =
      frequencyData.slice(96, 128).reduce((a, b) => a + b, 0) / 32;

    // Create multiple visual layers for depth

    // Layer 1: Spiraling frequency rings
    for (let ring = 0; ring < 5; ring++) {
      const ringRadius = 60 + ring * 40;
      const ringSpeed = 0.2 + ring * 0.1;

      for (let i = 0; i < frequencyData.length; i += 2) {
        const angle =
          (i / frequencyData.length) * Math.PI * 4 + time * ringSpeed + ring;
        const intensity = frequencyData[i];

        if (intensity > 0.1) {
          const radius = ringRadius + intensity * 80;
          const x = centerX + Math.cos(angle) * radius;
          const y = centerY + Math.sin(angle) * radius;

          const hue = (i * 2.8125 + ring * 72 + time * 30) % 360;
          const alpha = Math.max(0.1, intensity * 0.8);
```

```
      ctx.fillStyle = `hsla(${hue}, 90%, ${60 + ring * 8}%, ${alpha})`;
      ctx.beginPath();
      ctx.arc(x, y, 2 + intensity * 8, 0, Math.PI * 2);
      ctx.fill();

      // Add trailing particles
      const trailX = centerX + Math.cos(angle - 0.3) * (radius * 0.8);
      const trailY = centerY + Math.sin(angle - 0.3) * (radius * 0.8);
      ctx.fillStyle = `hsla(${hue}, 80%, 50%, ${alpha * 0.4})`;
      ctx.beginPath();
      ctx.arc(trailX, trailY, 1 + intensity * 3, 0, Math.PI * 2);
      ctx.fill();
    }
  }
}

// Layer 2: Central energy burst
const burstRadius = 30 + totalEnergy * 150 + Math.sin(time * 6) * 20;
const burstParticles = Math.floor(20 + totalEnergy * 80);

for (let i = 0; i < burstParticles; i++) {
  const angle = (i / burstParticles) * Math.PI * 2 + time * 2;
  const distance = Math.random() * burstRadius;
  const x = centerX + Math.cos(angle) * distance;
  const y = centerY + Math.sin(angle) * distance;

  const hue = (time * 120 + i * 10) % 360;
  const alpha = (1 - distance / burstRadius) * totalEnergy;

  ctx.fillStyle = `hsla(${hue}, 95%, 70%, ${alpha})`;
  ctx.beginPath();
  ctx.arc(x, y, 1 + totalEnergy * 4, 0, Math.PI * 2);
  ctx.fill();
}

// Layer 3: Frequency bands as expanding arcs
const numArcs = 8;
for (let arc = 0; arc < numArcs; arc++) {
  const startAngle = (arc / numArcs) * Math.PI * 2 + time * 0.5;
  const endAngle = startAngle + Math.PI / 4;
  const baseRadius = 100 + arc * 25;

  const freqBandSize = Math.floor(frequencyData.length / numArcs);
  const bandStart = arc * freqBandSize;
  const bandEnd = Math.min(bandStart + freqBandSize, frequencyData.length);
  const bandEnergy =
    frequencyData.slice(bandStart, bandEnd).reduce((a, b) => a + b, 0) /
    freqBandSize;

  if (bandEnergy > 0.1) {
    const arcRadius = baseRadius + bandEnergy * 100;
    const hue = (arc * 45 + time * 40) % 360;

    ctx.strokeStyle = `hsla(${hue}, 85%, 60%, ${bandEnergy * 0.8})`;
    ctx.lineWidth = 3 + bandEnergy * 8;
```

```
      ctx.lineCap = "round";
      ctx.beginPath();
      ctx.arc(centerX, centerY, arcRadius, startAngle, endAngle);
      ctx.stroke();

      // Add inner glow
      ctx.strokeStyle = `hsla(${hue}, 100%, 80%, ${bandEnergy * 0.4})`;
      ctx.lineWidth = 1 + bandEnergy * 3;
      ctx.beginPath();
      ctx.arc(centerX, centerY, arcRadius - 5, startAngle, endAngle);
      ctx.stroke();
    }
  }

  // Layer 4: Floating orbs responding to different frequency ranges
  const orbTypes = [
    { freq: lowFreqs, count: 6, baseRadius: 200, color: 240, speed: 0.3 },
    { freq: midFreqs, count: 8, baseRadius: 160, color: 120, speed: 0.5 },
    { freq: highFreqs, count: 12, baseRadius: 120, color: 0, speed: 0.8 },
  ];

  orbTypes.forEach((orbType, typeIndex) => {
    for (let i = 0; i < orbType.count; i++) {
      const angle =
        (i / orbType.count) * Math.PI * 2 + time * orbType.speed + typeIndex;
      const radius = orbType.baseRadius + Math.sin(time * 2 + i) * 30;
      const x = centerX + Math.cos(angle) * radius;
      const y = centerY + Math.sin(angle) * radius;

      const orbSize = 3 + orbType.freq * 15;
      const hue = (orbType.color + time * 20 + i * 30) % 360;

      // Outer glow
      const gradient = ctx.createRadialGradient(x, y, 0, x, y, orbSize * 2);
      gradient.addColorStop(0, `hsla(${hue}, 90%, 70%, ${orbType.freq * 0.6})`);
      gradient.addColorStop(
        0.7,
        `hsla(${hue}, 80%, 50%, ${orbType.freq * 0.3})`,
      );
      gradient.addColorStop(1, `hsla(${hue}, 70%, 30%, 0)`);

      ctx.fillStyle = gradient;
      ctx.beginPath();
      ctx.arc(x, y, orbSize * 2, 0, Math.PI * 2);
      ctx.fill();

      // Core orb
      ctx.fillStyle = `hsla(${hue}, 95%, 80%, ${orbType.freq * 0.9})`;
      ctx.beginPath();
      ctx.arc(x, y, orbSize, 0, Math.PI * 2);
      ctx.fill();
    }
  });

  // Layer 5: Energy lightning effects
  if (totalEnergy > 0.6) {
```

```
    for (let bolt = 0; bolt < 5; bolt++) {
      const startAngle = Math.random() * Math.PI * 2;
      const startRadius = 20 + Math.random() * 40;
      const endRadius = 150 + Math.random() * 200;

      let currentRadius = startRadius;
      let currentAngle = startAngle;

      ctx.strokeStyle = `hsla(${Math.random() * 60 + 180}, 100%, 90%, ${totalEnergy *
0.7})`;
      ctx.lineWidth = 1 + totalEnergy * 3;
      ctx.lineCap = "round";
      ctx.beginPath();

      let x = centerX + Math.cos(currentAngle) * currentRadius;
      let y = centerY + Math.sin(currentAngle) * currentRadius;
      ctx.moveTo(x, y);

      while (currentRadius < endRadius) {
        currentRadius += 10 + Math.random() * 20;
        currentAngle += (Math.random() - 0.5) * 0.5;

        x = centerX + Math.cos(currentAngle) * currentRadius;
        y = centerY + Math.sin(currentAngle) * currentRadius;
        ctx.lineTo(x, y);
      }

      ctx.stroke();
    }
  }

  // Central core with multiple layers
  const coreHue = (time * 60) % 360;
  const coreRadius = 15 + totalEnergy * 40;

  // Outer core glow
  const coreGradient = ctx.createRadialGradient(
    centerX,
    centerY,
    0,
    centerX,
    centerY,
    coreRadius * 1.5,
  );
  coreGradient.addColorStop(
    0,
    `hsla(${coreHue}, 100%, 90%, ${totalEnergy * 0.8})`,
  );
  coreGradient.addColorStop(
    0.6,
    `hsla(${coreHue}, 90%, 70%, ${totalEnergy * 0.4})`,
  );
  coreGradient.addColorStop(1, `hsla(${coreHue}, 80%, 50%, 0)`);

  ctx.fillStyle = coreGradient;
  ctx.beginPath();
```

```
  ctx.arc(centerX, centerY, coreRadius * 1.5, 0, Math.PI * 2);
  ctx.fill();

  // Inner core
  ctx.fillStyle = `hsla(${coreHue + 180}, 100%, 95%, ${0.6 + totalEnergy * 0.4})`;
  ctx.beginPath();
  ctx.arc(centerX, centerY, coreRadius, 0, Math.PI * 2);
  ctx.fill();

  // Core pulse ring
  const pulseRadius = coreRadius + Math.sin(time * 8) * 10;
  ctx.strokeStyle = `hsla(${coreHue + 60}, 100%, 80%, ${totalEnergy * 0.6})`;
  ctx.lineWidth = 2;
  ctx.beginPath();
  ctx.arc(centerX, centerY, pulseRadius, 0, Math.PI * 2);
  ctx.stroke();
}

function updateVisualizationDisplay() {
  // Remove any existing mode display
  const existingDisplay = document.getElementById("currentMode");
  if (existingDisplay) {
    existingDisplay.remove();
  }
}

// Initialize when DOM is ready
document.addEventListener("DOMContentLoaded", () => {
  initializeWasm();
  updateVisualizationDisplay(); // Show initial mode
});

// Handle page visibility changes to restart animation
document.addEventListener("visibilitychange", () => {
  if (!document.hidden && isRunning) {
    console.log("Page became visible, restarting animation...");
    // Give a moment for the page to fully load
    setTimeout(() => {
      startVisualization();
    }, 100);
  }
});

// Handle window focus to ensure animation continues
window.addEventListener("focus", () => {
  if (isRunning) {
    console.log("Window focused, ensuring animation is running...");
    setTimeout(() => {
      startVisualization();
    }, 50);
  }
});
```

Running the Complete Application

1. Navigate to the output directory:
   `cd bin\Release\net8.0\browser-wasm\AppBundle`

2. Install a simple HTTP server (if you don't have one):
   `dotnet tool install --global dotnet-serve`

3. Start the server:
   `dotnet serve -p 8080`

4. Open your browser and go to:
   http://localhost:8080

What You'll Experience:

- **Enhanced Radial Mode**: Particles emanate from center with spiral patterns based on frequency analysis
- **Dynamic Orbital Mode**: Particles orbit the center with pulsing based on low frequencies
- **Spectral Wave Mode**: Complex wave patterns driven by different frequency bands
- **Real-time Audio Response**: Particles react to music, voice, or any audio input
    - for voice input click 'Start Audio'
- **Smooth 60 FPS Performance**: WebAssembly delivers consistent frame rates with complex calculations

Understanding WebAssembly in Practice

What We've Accomplished

This tutorial demonstrates WebAssembly's core value proposition: bringing high-performance, compiled language capabilities to web browsers. Our audio visualizer performs complex mathematical calculations in real-time, showcasing the computational power available through WebAssembly.

Real-World Applications

The techniques demonstrated here apply to numerous scenarios:

- **Game Development**: Unity and Unreal Engine use similar WebAssembly compilation for browser games
- **Scientific Computing**: Complex simulations and data analysis tools
- **Image/Video Processing**: Real-time filters and effects
- **Cryptocurrency**: Blockchain operations and wallet management
- **CAD Applications**: 3D modeling and engineering tools

Performance Benefits

By implementing particle physics calculations in WebAssembly, we achieve:

- **Predictable Performance**: Compiled code eliminates JavaScript JIT compilation variability
- **Memory Efficiency**: Structured data layout reduces memory fragmentation

- **Mathematical Precision**: Native floating-point operations for accurate physics simulation
- **Scalability**: Handles 200+ particles with complex physics calculations

This tutorial illustrates how WebAssembly enables developers to combine web accessibility with high-performance computational capabilities, opening new possibilities for sophisticated browser-based applications.

To access the repository please visit this link, https://github.com/nolzftw/IMY320-IA1