VDA

Librería para Visualización de Datos

García Aguilar Luis Alberto

Contenido

Introducción

Introducción y objetivos

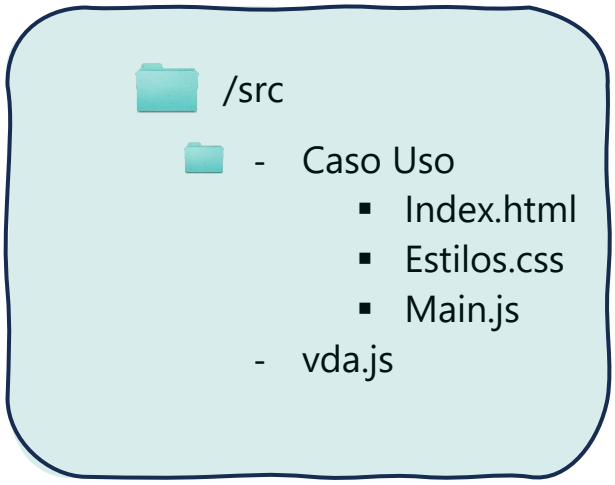
Este documento pretende ser una guía para el uso de nuestra biblioteca de visualización de datos en JavaScript, Vda. Esta guía está diseñada para proporcionar una referencia completa a los métodos disponibles en la biblioteca, facilitando a los desarrolladores la creación de gráficos y visualizaciones interactivas en un entorno web utilizando elementos de canvas.

Se pretende ofrecer una descripción de cada método proporcionado en la biblioteca, incluyendo su propósito, entradas, salidas y casos de uso. La biblioteca tiene como objetivo facilitar al desarrollador el uso de canvas para poder crear una variedad de gráficos y visualizaciones, tales como gráficos de líneas, gráficos de barras, gráficos de burbujas, entre otros.

Configuración inicial

Configuración de la biblioteca

Esta guía considera la siguiente estructura para cada caso de uso:



Para hacer uso de la biblioteca y sus métodos es necesario importarla dentro del proyecto:

Por ejemplo, en el archivo main.js se deberá agregar la siguiente línea de código al inicio del archivo:

```
import {initViewport, drawXAxis} from './lib/vda.js';
```

Entre corchetes se indican los métodos a utilizar y entre comillas se indica la ruta del archivo vda.js

También es posible importar la librería completa de la siguiente manera:

```
import * as vda from './vda.js';
```

Métodos

View Port

Uso

El método initViewport se encarga de inicializar un espacio de trabajo en un elemento canvas. Este método ajusta el tamaño del canvas y configura las coordenadas para trabajar con una matriz inversa, facilitando la representación gráfica de datos en el canvas.

Entradas

- **canvasId** (string): El ID del canvas en el documento HTML.
- **width** (number): Ancho para el canvas.
- **height** (number): Altura para el canvas.

Salidas

context (CanvasRenderingContext2D): Contexto del canvas, configurado para trabajar con el viewport ajustado.

Método

```
initViewport (canvasId, width,
height) {

    ...

    ...

    return context;

 }
```

Texto

Uso

El método drawText permite insertar un texto en el canvas en base a una posición y ángulo indicados.

Entradas

- **ctx** (CanvasRenderingContext2D): Contexto del canvas donde se dibujará.
- **text** (String): Texto a mostrar.
- **x** (number): Posición X en el canvas donde se insertará el texto.
- **y** (number): Posición Y en el canvas donde se insertará el texto.
- **size** (number): Tamaño del texto
- **color** (string): Color.
- **angle** (number): Ángulo del texto.

Salidas

No retorna valor. Dibujará directamente el texto.

Método { }

```
drawText(ctx, text, x, y, size,
color, angle)
{

    ...
 }
```

Eje X

Uso

El método drawXAxis dibuja una línea de eje con marcas (ticks) distribuidas uniformemente a lo largo del eje. Las marcas se generan para un rango de valores especificado por el usuario (start, end) y se espacian según un paso dado (step).

Entradas

- **Context** (CanvasRenderingContext2D): Contexto del canvas donde se dibujará.
- **startX** (number): Valor inicial del rango del eje X.
- **endX** (number): Valor de fin del rango del eje X.
- **xPos** (number): Posición X en el canvas donde se comenzará a dibujar el eje X.
- **yPos** (number): Posición Y en el canvas donde se comenzará a dibujar el eje X.
- **step:** Paso entre cada marca (tick) en el eje X.
- **color** (string): Color.
- **labelSpace** (number): Espacio entre etiquetas y el eje X.

Salidas

No retorna valor. Dibujará directamente el eje X.

Método { }

```
drawXAxis (context, startX, endX, xPos, yPos, step, color, labelSpace)
{

    ...

}
```

Eje Y

Uso

El método drawYAxis dibuja una línea de eje Y con marcas (ticks) distribuidas uniformemente a lo largo del eje. Las marcas se generan para un rango de valores especificado por el usuario (start, end) y se espacian según un paso dado (step).

Entradas

- **Context** (CanvasRenderingContext2D): Contexto del canvas donde se dibujará.
- **startX** (number): Valor inicial del rango del eje Y.
- **endX** (number): Valor de fin del rango del eje Y.
- **xPos** (number): Posición X en el canvas donde se comenzará a dibujar el eje Y.
- **yPos** (number): Posición Y en el canvas donde se comenzará a dibujar el eje Y.
- **step:** Paso entre cada marca (tick) en el eje Y.
- **color** (string): Color.
- **labelSpace** (number): Espacio entre etiquetas y el eje Y.

Salidas

No retorna valor. Dibujará directamente el eje Y.

Método { }

```
DrawYAxis (context, startY, endY,
xPos, yPos, step, color, labelSpace)
{

    ...

 }
```

Eje X a partir de un arreglo de datos

Uso

El método drawXAxisWithIntervals dibuja una línea de eje X en un canvas a partir de valores definidos en un arreglo proporcionado.

Entradas

- **Context** (CanvasRenderingContext2D): Contexto del canvas.
- **xValues** (Array[Object]): Array de datos 1xn que contiene los valores para las marcas del eje.
- **xPos** (number): Posición X inicial para el eje.
- **yPos** (number): Posición Y inicial para el eje.
- **xLabelTextAngle** (number): Ángulo de las etiquetas en el eje.
- **color** (string): Color.
- **labelSpace** (number): Espacio entre etiquetas y el propio eje.
- **canvasPadding** (number): Padding del canvas.
- **Interval** (number): -1: Intervalo calculado en base al rango; >0: intervalo usando el valor indicado; 0: se ocupan todos los valores del arreglo.

Salidas

No retorna valor. Dibujará directamente el eje X.

Método { }

```
drawXAxisWithIntervals(context,
xValues , xPos, yPos, , color,
labelSpace, canvasPadding, 0)

{     ...     }
```

Eje Y a partir de un arreglo de datos

Uso

El método drawYAxisWithIntervals dibuja una línea de eje Y en un canvas a partir de valores definidos en un arreglo proporcionado.

Entradas

- **Context** (CanvasRenderingContext2D): Contexto del canvas.
- **yValues** (Array[Object]): Array de datos 1xn que contiene los valores para las marcas del eje.
- **xPos** (number): Posición X inicial para el eje.
- **yPos** (number): Posición Y inicial para el eje.
- **color** (string): Color.
- **labelSpace** (number): Espacio entre etiquetas y el propio eje.
- **canvasPadding** (number): Padding del canvas.
- **Interval** (number): -1: Intervalo calculado en base al rango; >0: intervalo usando el valor indicado; 0: se ocupan todos los valores del arreglo.

Salidas

No retorna valor. Dibujará directamente el eje Y.

Método { }

```
drawYAxisWithIntervals(context,
yValues , xPos, yPos, color,
labelSpace, canvasPadding, interval)
{
   ...
}
```

Gráfica de puntos

Uso

El método drawDotPlot dibuja una gráfica de puntos basada en un conjunto de datos proporcionados mediante un archivo CSV. Cada punto de datos se representa como un círculo en el gráfico.

Entradas

- **canvas** (Canvas): Canvas en uso.
- **context** (CanvasRenderingContext2D): Contexto.
- **csvFilePath** (String): Ruta al archivo de datos.
- **xColumnName** (String): Nombre para la columna X
- **yColumnName** (String): Nombre para la columna Y.
- **infoColumNames** (Array[String]): Arreglo con el nombre de las columnas que serán presentadas como información al pasar el cursor sobre el gráfico.
- **color** (String): Color de los puntos.
- **filledCircles** (Boolean): True para círculos rellenos.
- **pointRadius** (number): Radio de los puntos.
- **canvasPadding** (number): Padding del canvas.
- **axesProperties** (Object, opt): Objeto que contiene las propiedades definidas para los ejes (Ver caso práctico).

Salidas

No retorna valor.

Método { }

```
drawDotPlot (canvas, context,
csvFilePath, xColumnName,
yColumnName, infoColumnNames, color,
filledCircles, pointRadius,
canvasPadding, axesProperties)
{
        ...
 }
```

Gráfica de líneas

Uso

El método drawLinePlot dibuja una gráfica de línea basada en un conjunto de datos proporcionado mediante un archivo CSV. Cada punto de datos se conecta mediante líneas rectas, creando una representación gráfica de la serie de datos.

Entradas

- **canvas** (Canvas): Canvas.
- **context** (CanvasRenderingContext2D): Contexto del canvas.
- **filePath** (String): Ruta al archivo de datos.
- **xColumnName** (String): Nombre de la columna de datos a usar para el eje X.
- **yColumnName** (String): Nombre de la columna de datos a usar para el eje Y.
- **infoColumNames** (Array[String]): Arreglo con el nombre de las columnas que serán presentadas como información al pasar el cursor sobre el gráfico.
- **color** (String): Color de los puntos.
- **filledCircles** (Boolean): True para círculos rellenos.
- **pointRadius** (number): Radio de los puntos.
- **lineWidth** (number): Grosor de la línea.
- **canvasPadding** (number): Padding del canvas.
- **axesProperties** (Object, opt): Objeto que contiene las propiedades definidas para los ejes (Ver caso práctico).

Salidas

No retorna valor.

Método { }

```
drawLinePlot(canvas, context, filePath, xColumnName, yColumnName, infoColumnNames, color, filledCircles, pointRadius, lineWidth, canvasPadding, axesProperties)

{ ... }
```

Gráfica de barras

Uso

El método drawBarPlot dibuja una gráfica de barras basada en un conjunto de datos proporcionado en un archivo CSV. Cada barra representa una categoría y se dibuja en el canvas con una altura proporcional al conteo de ocurrencias en cada categoría.

Entradas

- **canvas** (Canvas): Canvas.
- **context**: Contexto del canvas.
- **filePath** (String): Ruta al archivo de datos.
- **xColumnName** (String): Nombre de la columna de categorías.
- **canvasPadding** (number): Padding del canvas.

Salidas

No retorna valor.

Método { }

```
drawBarPlot (canvas, context,
filePath, xColumnName,
canvasPadding)
{
      ...
      ...
 }
```

Gráfica de burbujas

Uso

El método drawBubblePlot dibuja una gráfica de burbujas basada en un conjunto de datos en un archivo CSV. Cada burbuja se representa como un círculo en el gráfico, con un radio proporcional a un valor específico indicado.

Entradas

- **canvas** (Canvas): Canvas.
- **context** (CanvasRenderingContext2D): contexto.
- **filePath** (String): Ruta al archivo de datos.
- **xColumnName** (String): Nombre de la columna de datos a usar para el eje X.
- **yColumnName** (String): Nombre de la columna de datos a usar para el eje Y.
- **sizeColumnName** (String): Nombre de la columna que indicará el tamaño de las burbujas.
- **minSixe** (number): Tamaño mínimo de burbuja.
- **maxSixe** (number): Tamaño máximo de burbuja.
- **infoColumNames** (Array[String]): Arreglo con el nombre de las columnas que se desplegarán como información al pasar el cursor sobre el gráfico.
- **color** (String): Color de los puntos.
- **canvasPadding** (number): Padding del canvas.
- **axesProperties** (Object, opt): Objeto que contiene las propiedades definidas para los ejes (Ver caso práctico).

Salidas

No retorna valor.

Método { }

```
drawBubblePlot (canvas, context, filePath,
xColumnName,yColumnName, sizeColumnName,
minSize, maxSize, infoColumnNames,  color,
canvasPadding, axesProperties)

 {    ...   }
```

Mapa de puntos

Uso

El método drawMapDotPlot dibuja un mapa tomando como base un archivo GeoJSON proporcionado, además grafica puntos específicos sobre él, tomando en cuenta el archivo CSV indicado.

Entradas

- **canvas** (Canvas): Canvas.
- **context** (CanvasRenderingContext2D): Contexto.
- **mapDataFile** (String): Ruta al archivo GeoJSON
- **filePath** (String): Ruta al archivo de datos.
- **xColumnName** (String): Nombre de la columna de datos a usar para la longitud.
- **yColumnName** (String): Nombre de la columna de datos a usar para la latitud.
- **infoColumNames** (Array[String]): Arreglo con el nombre de las columnas que se desplegarán como información al pasar el cursor sobre el gráfico.
- **color** (String): Color de los puntos.
- **filledCircles** (Boolean): True para círculos rellenos.
- **pointRadius** (number): Radio de los puntos.
- **canvasPadding** (number): Padding del canvas.

Salidas

No retorna valor.

Método { }

```
drawMapDotPlot(canvas, context,
mapDataFile, filePath, xColumnName,
yColumnName, infoColumNames, color,
filledCircles, pointRadius,
canvasPadding)

{     ...     }
```

Mapa de burbujas

Uso

El método drawMapBubblePlot dibuja un mapa tomando como base un archivo GeoJSON proporcionado, además grafica burbujas sobre él tomando en cuenta un archivo CSV.

Entradas

- **canvas** (Canvas): Canvas.
- **context** (CanvasRenderingContext2D): Contexto.
- **mapDataFile** (String): Ruta al archivo GeoJSON
- **filePath** (String): Ruta al archivo de datos.
- **xColumnName** (String): Nombre de la columna de datos a usar para la longitud.
- **yColumnName** (String): Nombre de la columna de datos a usar para la latitud.
- **sizeColumnName** (String): Nombre de la columna que indica el tamaño de las burbujas.
- **minSizeBubble** (number): Tamaño mínimo de burbuja.
- **maxSizeBubble** (number): Tamaño máximo de burbuja.
- **infoColumNames** (Array[String]): Arreglo con el nombre de las columnas que se desplegarán como información al pasar el cursor sobre el gráfico.
- **color** (String): Color de los puntos.
- **canvasPadding** (number): Padding del canvas.

Salidas

No retorna valor.

Método { }

```
drawMapDotPlot(canvas, context,
mapDataFile, filePath, xColumnName,
yColumnName, sizeColumnName,
minSizeBubble, maxSizeBubble,
infoColumNames, color,
canvasPadding)

{   ...   }
```

Mapa de calor

Uso

El método drawHeatMap dibuja un mapa tomando como base un archivo GeoJSON proporcionado. Posteriormente hace uso de un archivo CSV proporcionado por el usuario para colorea cada región del mapa en base a una variable determinada.

Entradas

- **canvas** (Canvas): Canvas.
- **canvasPadding** (number): Padding del canvas.
- **mapDataFile** (String): Ruta al archivo GeoJSON
- **csvFilePath** (String): Ruta al archivo de datos.
- **variableName** (String): Nombre de la columna de datos a usar para el degradado de color.
- **baseColor** (String): Color base para el degradado.
- **stateNameProperty** (String): Nombre de la propiedad que representa el identificador para cada región del mapa.
- **linkNameProperty** (String): Nombre de la columna en el archivo CSV que identifica cada región del mapa y cuyos valores deben coincidir con los valores en stateNameProperty.
- **infoColumNames** (Array[String]): Arreglo con el nombre de las columnas que se desplegarán como información al pasar el cursor sobre el gráfico.

Salidas

No retorna valor.

Método { }

drawHeatMap (`canvas, canvasPadding, mapDataFile, csvFilePath, variableName, color, stateNameProperty, linkNameProperty, infoColumnNames`)

```
{ ... }
```

Serie de tiempo

Uso

El método drawTimeSeries permite dibujar una serie de tiempo haciendo uso de curvas de bezier. Además, permite agregar eventos y representarlos mediante una banda de tiempo.

Entradas

- **canvas** (Canvas): Canvas.
- **context** (CanvasRenderingContext2D): Contexto.
- **filePath** (String): Ruta al archivo de datos.
- **xColumnName** (String): Nombre de la columna de fecha para usar en el eje de las abscisas.
- **yColumnName** (String): Nombre de la columna de valores referenciados a cada fecha.
- **infoColumNames** (Array[String]): Arreglo con el nombre de las columnas que se desplegarán como información al pasar el cursor sobre el gráfico.
- **color** (String): color de la línea del gráfico.
- **filledCircles** (Boolean): True para círculos rellenos.
- **pointRadius** (number): Radio de los puntos.
- **lineWidth** (number): Grosor de la línea.
- **canvasPadding** (number): Padding del canvas.
- **axesProperties** (Object, opt): Objeto que contiene las propiedades definidas para los ejes (Ver caso práctico).
- **Events (Object,opt) :** Arreglo de eventos (Ver caso práctico)

Salidas

No retorna valor.

Método { }

drawTimeSeries(`canvas, context, filePath, xColumnName,yColumnName, infoColumnNames, color, filledCircles, pointRadius, lineWidth, canvasPadding, axesProperties,events`)

```
{ ... }
```

Mapa de burbujas interactivo (incendios)

Uso

El método drawInteractiveBubbleMapPlot dibuja un mapa a partir de un archivo GeoJSON proporcionado y sobre él, grafica burbujas utilizando los datos de un archivo CSV. Además, permite aplicar filtros dinámicos para controlar la visualización de las burbujas según diferentes criterios, permitiendo una visualización interactiva y personalizada del mapa.

Entradas

- **canvas** (HTMLCanvas): Canvas.
- **context** (CanvasRenderingContext2D): Contexto.
- **bbox** (Oject): coordenadas limitantes del canvas
- **filePath** (String): Ruta al archivo de datos.
- **Data** (Array[Object]): Arreglo de objetos con los datos del archivo CSV.
- **xColumnName** (String): Nombre de la columna de fecha para usar en el eje de las abscisas.
- **yColumnName** (String): Nombre de la columna que contiene los valores de latitud (o eje Y).
- **sizeColumnName** (String): Nombre de la columna que se usa para determinar el tamaño de las burbujas.
- **minSize (number, opt)**: Tamaño mínimo de las burbujas (valor por defecto: 1).
- **maxSize (number, opt)**: Tamaño máximo de las burbujas (valor por defecto: 5).
- **infoColumNames (Array[String])**: Arreglo con los nombres de las columnas que se mostrarán como información al pasar el cursor sobre cada burbuja.
- **colorMapping (Array[Object])**: Mapeo de colores con los cuales se visualizan las burbujas, basado en los valores de los datos.
- **bubbleFillColorColumnName (String)**: Nombre de la columna que define el color de relleno de las burbujas.

- **bubbleRingColorColumnName (String)**: Nombre de la columna que define el color del anillo de las burbujas.
- **canvasPadding (number, opt)**: Padding del canvas para mantener espacio alrededor del área de dibujo (valor por defecto: 2% del ancho del canvas).
- **yearSelected (number)**: Año seleccionado para filtrar los datos a visualizar.
- **impactChecksSelected (Object)**: Objeto que contiene los filtros seleccionados por el usuario para mostrar ciertos tipos de impacto de las burbujas.

Salidas ⇨

No retorna valor.

Método { }

drawInteractiveBubbleMapPlot (canvas, context, bbox, data, xColumnName, yColumnName, sizeColumnName, minSize=1, maxSize = 5,

infoColumnNames, colorMapping, bubbleFillColorColumnName, bubbleRingColorColumnName, canvasPadding = context.canvas.width * 0.02,yearSelected,impactChecksSelected)

```
{
  ...
}
```

Árbol de adopciones

Uso

El árbol de adopciones es un caso específico que ejemplifica cómo la biblioteca VDA puede ser un recurso para adentrarse en visualizaciones más personalizadas para temas concretos. Esta visualización muestra el número de adopciones registradas entre 2015 y 2023, ofreciendo una representación artística de los datos. Además, incorpora un control interactivo que permite seleccionar el año de visualización y decidir si los datos se presentan de forma acumulada o individual por año. Con este ejemplo, se exhorta al usuario a explorar la creatividad en la visualización de datos.

Entradas

- **data** (Array[Object]): Arreglo de objetos con las adopciones provenientes del archivo CSV.
- **yearColumnName** (String): Nombre de la columna que representa el año en que se realizaron las adopciones.
- **classColumnName** (String): Nombre de la columna que contiene la clasificación binaria de género.
- **selectedYear** (int): Año seleccionado para filtrar los datos.
- **isAccumulative (boolean)**: Indica si los datos se mostraran de manera acumulado o únicamente los datos del año seleccionado.
- **treeCanvasId (String)**: Id del canvas donde se dibujarán las ramas del árbol.
- **flowersCanvasId (String])**: Id del canvas donde se dibujarán las flores del árbol.

Salidas

No retorna valor.

Método { }

createTree (`data, yearColumnName, classColumnName, selectedYear, isAccumulative, treeCanvasId, flowersCanvasId`)

```
{
  ...
}
```

Casos de uso

View Port

El siguiente ejemplo muestra cómo utilizar el método **initViewport** de la biblioteca vda.js.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DataCanvas Ejemplo initView</title>
</head>
<body>
  <! --La siguiente línea define el canvas "miCanvas"-->
  <canvas id="miCanvas"></canvas>
  <script type="module" src="./main.js"></script>
</body>
</html>
```

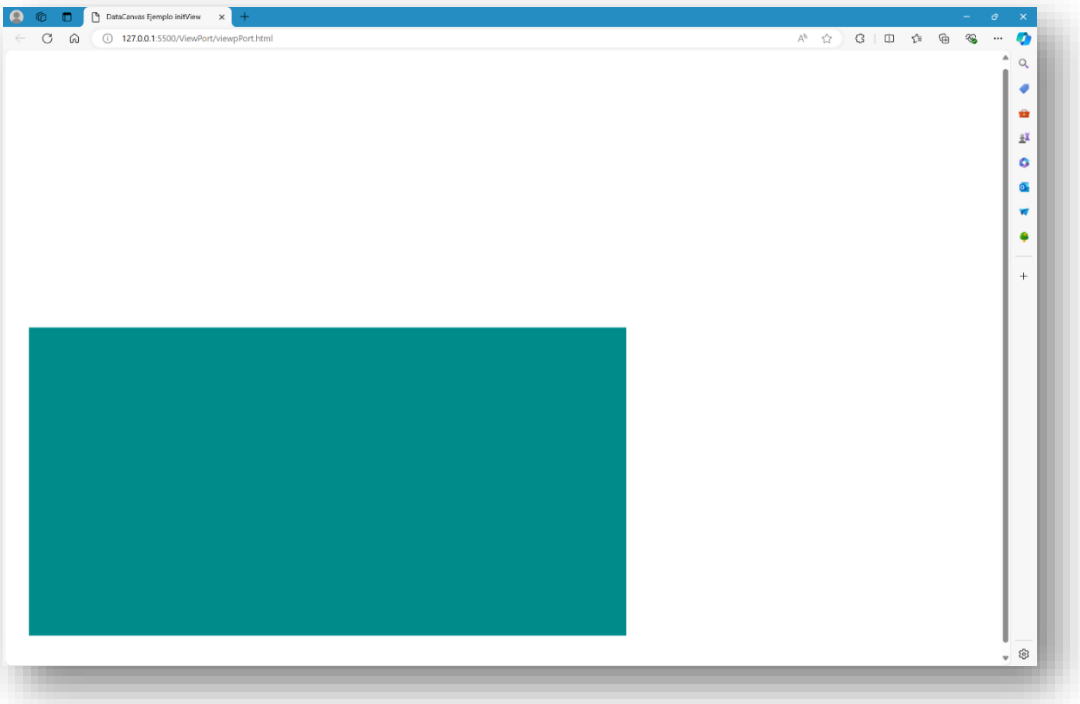
El archivo viewPort.html define un elemento Canvas identificado por un ID con nombre “miCanvas”.

Adicionalmente se tiene el archivo main.js el cual hace uso de la función initViewPort(), de la biblioteca vda.js, la cual recibe el id "miCanvas", y define los parámetros width y height con dimensiones 1000px cada uno. Posteriormente, para verificar el funcionamiento del código, dibujamos un rectángulo comenzando en el punto P(30,50) con ancho 2000 px y altura 500 px.

Main.js

```
import {initViewport} from '../vda.js';

document.addEventListener('DOMContentLoaded', () => {

  // Llamamos a la función initViewPort de la biblioteca vda.js
  const width = 1000;
  const height = 1000;
  const context = initViewport('miCanvas', width, height);

  // Dibujamos rectángulo para verificar que el viewport funciona
  context.fillStyle = 'darkcyan';
  context.fillRect(30, 50, 2000, 500);
});
```

Navegador



Texto

El siguiente ejemplo muestra cómo utilizar el método **drawText** de la biblioteca vda.js.

text.html

```
<!DOCTYPE html>
<html lang="es">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DataCanvas Ejemplo dibujar texto</title>
</head>

<body>
  <!--La siguiente línea define el canvas "miCanvas"-->
  <canvas id="miCanvas"></canvas>
  <script type="module" src="./main.js"></script>
</body>
```

El archivo text.html define un elemento Canvas identificado por un ID con nombre “miCanvas”.

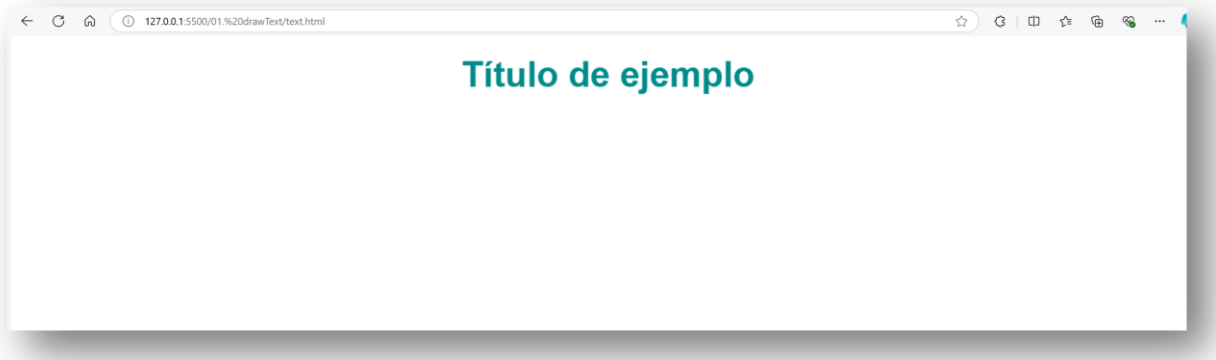
Adicionalmente se tiene el archivo main.js el cual hace uso de la función drawText(), de la biblioteca vda.js, la cual recibe el id "miCanvas", y define los parámetros width y height con dimensiones 1150 px y 1000px, respectivamente.

Posteriormente se definen los valores del texto como el texto mismo, color y tamaño; en este caso, se está posicionando al texto en la parte superior de la pantalla y centrado con respecto al eje horizontal.

Main.js

```
  // Llamamos a la función initViewPort de la biblioteca vda.js
  const width = 1150;
  const height = 10000;
  const context = initViewport('miCanvas', width, height);

  const title = "Título de ejemplo";
  const canvasPadding = 50;
  const angleText = 0;
  const size = 50;
  const color = "darkcyan" ;
  drawText(context,title, width/2 + canvasPadding ,height - canvasPadding,
size ,color,- angleText)

});
```

Navegador



Eje X

El siguiente ejemplo muestra cómo utilizar el método **drawXAxis** de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Canvas Ejemplo Eje X</title>
</head>

<body>
  <canvas id="miCanvas" style="border:1px solid #000000;"></canvas>
  <script type="module" src="./main.js"></script>
</body>

</html>
```

Posteriormente:

- Se importa el método initViewport y drawXAxis desde el archivo main.js.
- Cuando el documento está completamente cargado (DOMContentLoaded), inicializamos el canvas y su contexto mediante el método initViewport().

- Llamamos a drawXAxis con los parámetros necesarios para dibujar el eje X en el canvas.

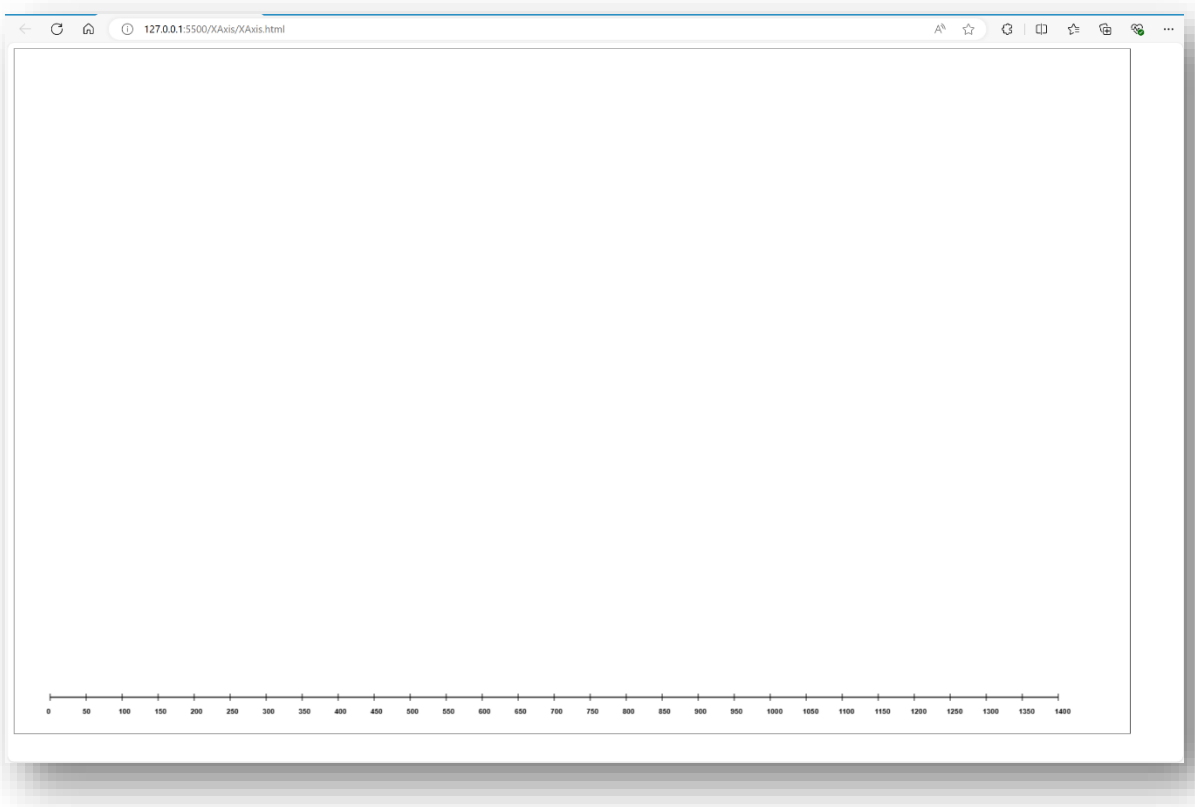
Main.js

```
import { initViewport, drawXAxis } from '../vda.js';

document.addEventListener('DOMContentLoaded', () => {
  const canvasId = 'miCanvas';
  const width = 1550;
  const height = 950;
  const context = initViewport(canvasId, width, height);

  // Configuración para el eje X
  const startX = 0;
  const endX = 1400;
  const step = 50;
  const xPosition = 50;
  const yPosition = 50;
  const color = 'black';
  const labelSpace = 20;

  drawXAxis(context, startX, endX, xPosition,yPosition, step, color,
labelSpace);
});
```

En este caso se está dibujando un eje X que inicia en 0 y finaliza en 1400, la posición del origen se encuentra en (50px, 50px), la separación en cada marca es de 50px y el espacio entre las leyendas (números de referencia en el eje) y el eje es de 20px.

En la siguiente imagen puede observarse el resultado, desplegado mediante un navegador web.

Navegador



Eje Y

El siguiente ejemplo muestra cómo utilizar el método drawYAxis de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Canvas Ejemplo Eje Y</title>
</head>

<body>
  <canvas id="miCanvas" style="border:1px solid #000000;"></canvas>
  <script type="module" src="./main.js"></script>
</body>

</html>
```

Posteriormente:

- Se importa el método initViewport y drawYAxis desde el archivo main.js.
- Cuando el documento está completamente cargado (DOMContentLoaded), inicializamos el canvas y su contexto mediante el método initViewport().

- Llamamos a drawYAxis con los parámetros necesarios para dibujar el eje X en el canvas.

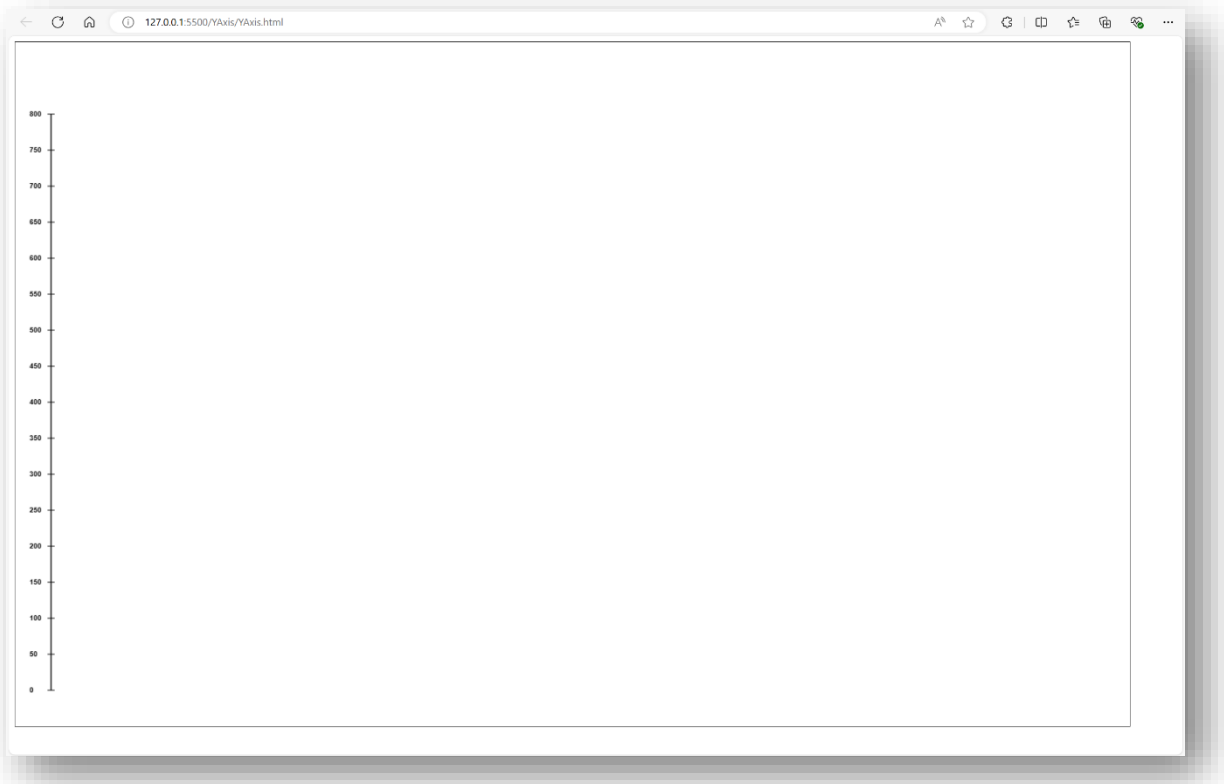
Main.js

```
import { initViewport, drawYAxis } from '../vda.js';

document.addEventListener('DOMContentLoaded', () => {
  const canvasId = 'miCanvas';
  const width = 1550;
  const height = 950;
  const context = initViewport(canvasId, width, height);

  // Configuración para el eje X
  const startY = 0;
  const endY = 800;
  const step = 50;
  const xPosition = 50;
  const yPosition = 50;
  const color = 'black';
  const labelSpace = 30;

  drawYAxis(context, startY, endY, xPosition, yPosition, step, color,
labelSpace);
});
```

En este caso se está dibujando un eje Y que inicia en 0 y finaliza en 800, la posición del origen se encuentra en (50px, 50px), la separación en cada marca es de 50px y el espacio entre las leyendas (números de referencia en el eje) y el eje es de 30px.

En la siguiente imagen puede observarse el resultado, desplegado mediante un navegador web.

Navegador


127.0.0.1:5500/YAxis/YAxis.html
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0

Eje X a partir de un arreglo de datos

El siguiente ejemplo muestra cómo utilizar el método drawXAxisFromArray de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Ejemplo de Eje X a partir de un array</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.
min.js"></script>
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px
solid #000000;"></canvas>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

Posteriormente:

- Se importa el método initViewport, drawXAxisWithCSV y loadCSV desde el archivo main.js.
- Después, inicializamos el canvas y su contexto usando initViewport().

- Cargamos los datos mediante la lectura del archivo CSV haciendo uso de la función loadCSV().
- Finalmente, llamamos a drawXAxisFromArray con los parámetros necesarios para dibujar el eje X en el canvas.

Main.js

```
import { initViewport, drawXAxisWithIntervals, loadCSV } from
'../vda.js';

// Función principal para inicializar el canvas y dibujar el eje X
async function init() {

    //Init viewPort
    const canvasId = 'miCanvas';
    const width = 1550;
    const height = 950;
    const context = initViewport(canvasId, width, height);

    //Carga de datos desde CSV
    const data = await loadCSV("/data/xAxisData.csv");
    const xColumnName = "x"
    let xValues = data.map(row => row[xColumnName]);

    //Parametros para el eje X
    const xPos = 0;
    const yPos = 0;
    const color = "black";
    const labelSpace = 20;
    const canvasPadding = 50
    const xLabelTextAngle = 0;
    const interval = 0;

    // Llamar a la función para dibujar el eje X
    drawXAxisWithIntervals(context, xValues , xPos, yPos,
xLabelTextAngle, color, labelSpace, canvasPadding, interval);
}

// Llama a la función principal para inicializar todo
init();
```

En este caso, el arreglo de datos fue creado a partir del archivo xAxisData.csv, el cual contiene la una columna de datos con el encabezado “x”. El archivo puede consultarse en la carpeta “data” del paquete “VDA”.

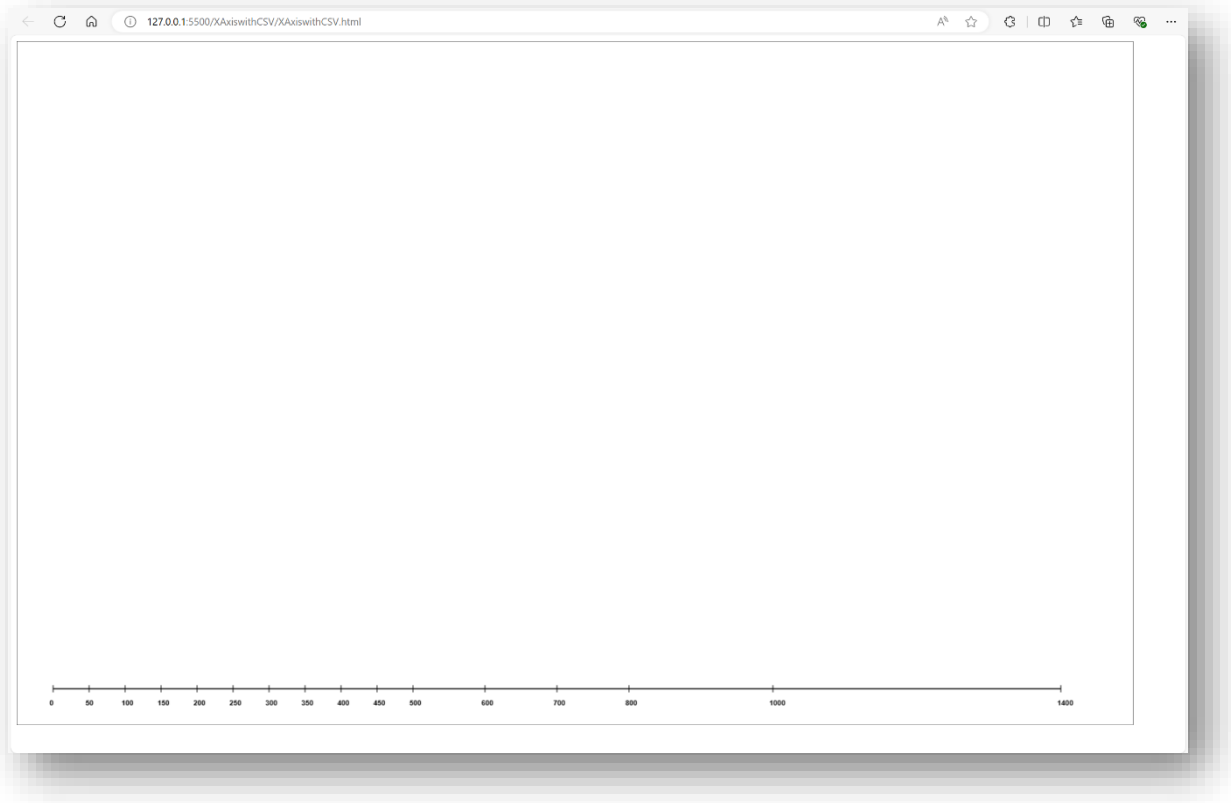
Nótese que el parámetro Interval está en 0, esto indica que los valores para los intervalos en el eje se tomaran directamente del arreglo de datos. Los valores aceptados para este parámetro son:

1. > 0: Cuando se coloca un valor mayor a cero, ese valor será el número de intervalos a definir.
2. = 0: Si el valor es igual a cero, se tomarán los valores del arreglo otorgado.
3. -1: Este valor indica que se calcule un intervalo de acuerdo a los datos otorgados.

Una vez desplegado, observamos un eje X que inicia en 0 y finaliza en 1400, y cuyos valores son tomados de la columna con nombre “x” del archivo CSV cuya ruta es `'../data/xAxisData.csv'`.

La posición del origen se encuentra en (50 px, 50px), el color de la línea se indica negro y el espacio entre las leyendas (números de referencia en el eje) y el eje es de 20px.

Navegador


127.0.0.1:5500/XAxiswithCSV/XAxiswithCSV.html
0
50
100
150
200
250
300
350
400
450
500
600
700
800
1000
1400

Eje Y a partir de un arreglo de datos

El siguiente ejemplo muestra cómo utilizar el método drawYAxisFromArray de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Ejemplo de Eje Y a partir de un array</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.
min.js"></script>
</head>
<body>
    <canvas id="miCanvas" style="border:1px solid #000000;"></canvas>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

Posteriormente:

- Se importa el método initViewport, drawYAxisWithCSV y loadCSV desde el archivo main.js.
- Después, inicializamos el canvas y su contexto usando initViewport().
- Cargamos los datos mediante la lectura del archivo CSV haciendo uso de la función loadCSV().

- Finalmente, llamamos a drawYAxisFromArray con los parámetros necesarios para dibujar el eje X en el canvas.

Main.js

```
import { initViewport, drawYAxisWithIntervals, loadCSV } from
'../vda.js';

// Función principal para inicializar el canvas y dibujar el eje X
async function init() {

    const canvasId = 'miCanvas';
    const width = 1550;
    const height = 950;
    const context = initViewport(canvasId, width, height);

    //Carga de datos desde CSV
    const data = await loadCSV("/data/yAxisData.csv");
    const yColumnName = "y";
    let yValues = data.map(row => row[yColumnName]);

    //Parametros para dibujar eje Y
    const yPosition = 0;
    const XPosition = 0;
    const color = "black";
    const labelSpace = -10;
    const canvasPadding = 50;
    const interval = 0;

    // Llamar a la función para dibujar el eje Y
    drawYAxisWithIntervals(context, yValues , XPosition, yPosition,
color, labelSpace, canvasPadding, interval);

}

// Llama a la función principal para inicializar todo
init();
```

En este caso, el arreglo de datos fue creado a partir del archivo yAxisData.csv, el cual contiene la una columna de datos con el encabezado “y”. El archivo puede consultarse en la carpeta “data” del paquete “VDA”.

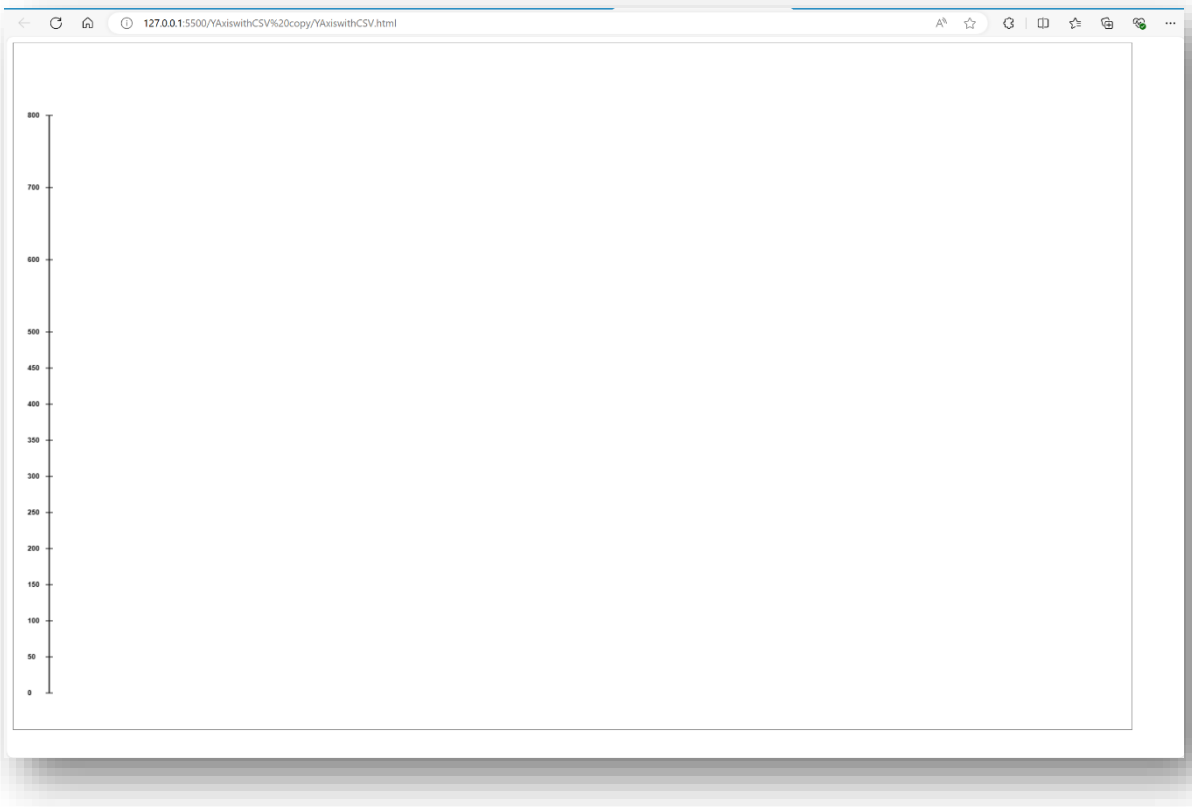
Nótese que el parámetro Interval está en 0, esto indica que los valores para los intervalos en el eje se tomaran directamente del arreglo de datos. Los valores aceptados para este parámetro son:

1. > 0: Cuando se coloca un valor mayor a cero, ese valor será el número de intervalos a definir.
2. = 0: Si el valor es igual a cero, se tomarán los valores del arreglo otorgado.
3. -1: Este valor indica que se calcule un intervalo de acuerdo a los datos otorgados.

Una vez desplegado, observamos un eje Y que inicia en 0 y finaliza en 800, y cuyos valores son tomados de la columna con nombre “y” del archivo CSV cuya ruta es `'../data/yAxisData.csv'`.

La posición del origen se encuentra en (50px, 50px), el color de la línea se indica negro y el espacio entre las leyendas (números de referencia en el eje) y el eje es de 30px.

Navegador


127.0.0.1:5500/YAxiswithCSV%20copy/YAxiswithCSV.html
800
700
600
500
450
400
350
300
250
200
150
100
50
0

Gráfica de puntos

El siguiente ejemplo muestra cómo utilizar el método drawDotPlot de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Ejemplo Gráfica de puntos</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.
min.js"></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px
solid #000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

- Se importa los métodos initViewport , drawDotPlot y loadCSV desde el archivo main.js, y opcionalmente, drawXAxisFromArray, drawYAxisFromArray.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Llamamos a drawDotPlot con los parámetros necesarios para el gráfico.

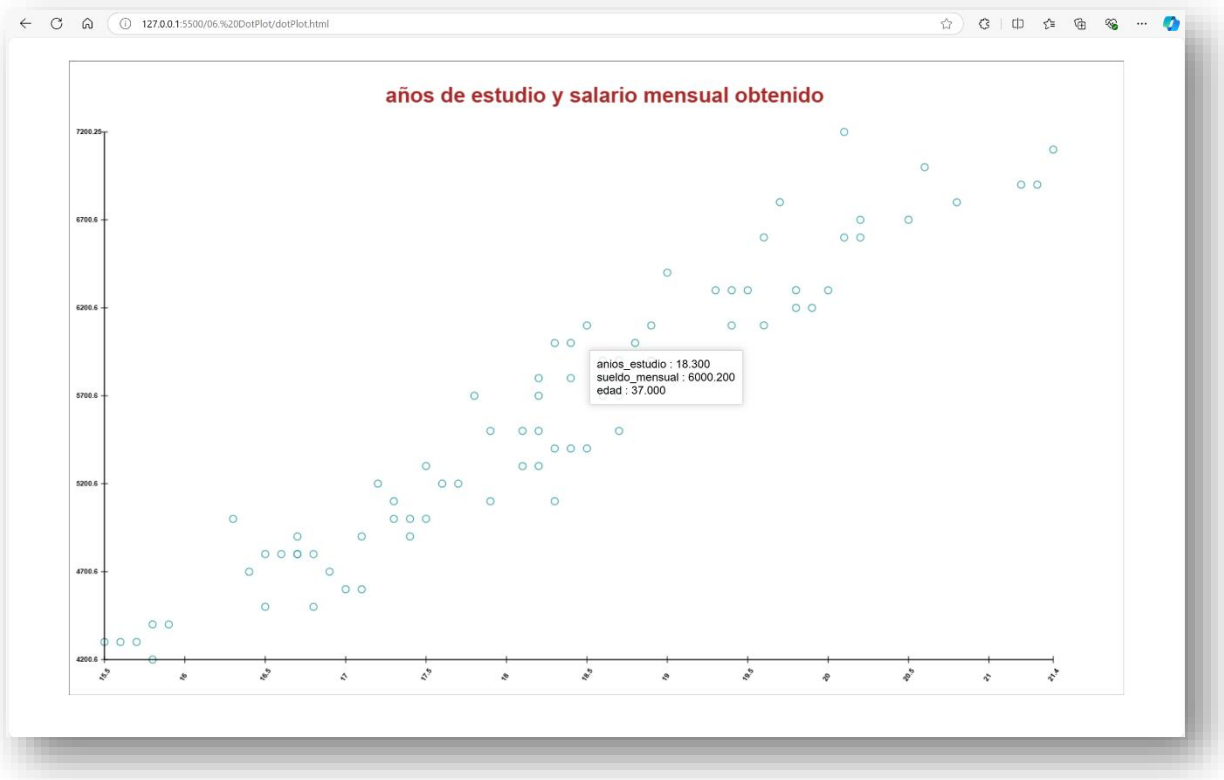
Main.js

```
Import {initViewport,drawDotPlot,drawText,loadCSV} from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500;
  const height = 900;
  const context = initViewport(canvasId, width, height);

  //Carga de datos desde CSV
  const filePath  = "/data/salarios.csv";
  const data = await loadCSV(filePath);

  //Parametros para la gráfica
  const canvas = document.getElementById("miCanvas");
  const xColumnName = "anios_estudio";
  const yColumnName = "sueldo_mensual";
  let infoColumnNames = [xColumnName, yColumnName, "edad"]
  const color = "darkcyan"
  const filledCircles = false;
  const pointRadius = 5;
  const canvasPadding = 50;

  //Parametros para los ejes
  const axesProperties = {
    xPos: 0,
    yPos: 0,
    xLabelSpace: 20,
    yLabelSpace: -10,
    xLabelTextAngle: 55,
    color: "black",
    xAxeType: 0.5,
    yAxeType: 500
  };

  drawDotPlot(canvas, context, filePath, xColumnName,yColumnName, infoColumnNames,
  color, filledCircles, pointRadius, canvasPadding,axesProperties);

  const xPos =  width/2 - 6*canvasPadding ;
  const yPos = height -canvasPadding;
  const titleSize = 30;
  const titleColor = "brown"
  drawText(context,"años de estudio y salario mensual obtenido", xPos, yPos,
titleSize ,titleColor,- 0)
});
```

Nótese que se hace uso del objeto axesPorperties, el cual contiene los valores necesarios para dibujar los ejes X y del gráfico.

En la siguiente imagen podemos apreciar el gráfico de puntos dibujado en el navegador web.

Navegador



Gráfica de líneas

El siguiente ejemplo muestra cómo utilizar el método drawLinePlot de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Ejemplo Gráfica de línea</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.
min.js"></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px
solid #000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

- Se importa los métodos initViewport , drawLinePlot y loadCSV desde el archivo main.js, y opcionalmente, drawXAxisFromArray, drawYAxisFromArray.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Llamamos a drawDotPlot con los parámetros necesarios para el gráfico.

Main.js

```
import { initViewport, drawLinePlot, drawText, loadCSV } from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500;
  const height = 900;
  const context = initViewport(canvasId, width, height);

  //Carga de datos desde CSV
  const filePath  = "/data/pesoEstatura.csv";
  const data = await loadCSV(filePath);

  //Parametros para la gráfica
  const canvas = document.getElementById("miCanvas");
  const xColumnName = "estatura";
  const yColumnName = "peso";
  let infoColumnNames = [xColumnName, yColumnName, "edad"];
  const color = "darkcyan";
  const filledCircles = true;
  const pointRadius = 10;
  const lineWidth = 1;
  const canvasPadding = 50;

  //Parametros para los ejes
  const axesProperties = {
    xPos: 0,
    yPos: 0,
    xLabelSpace: 20,
    yLabelSpace: -10,
    xLabelTextAngle: 20,
    color: "black",
    xAxeType: 0,
    yAxeType: 0
  };

  drawLinePlot(canvas, context, filePath, xColumnName,yColumnName, infoColumnNames,
color, filledCircles, pointRadius, lineWidth, canvasPadding, axesProperties);


  const xPos =  width/2 - 2*canvasPadding ;
  const yPos = height -canvasPadding;
  const titleSize = 30;
  const titleColor = "brown"
  drawText(context,"estatura vs peso", xPos, yPos, titleSize ,titleColor,- 0)

});
```

Nótese que se hace uso del objeto axesPorperties, el cual contiene los valores necesarios para dibujar los ejes X y Y del gráfico.

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador

Gráfica de barras

El siguiente ejemplo muestra cómo utilizar el método drawBarPlot de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo Gráfica de barras</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js"
></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px solid
#000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

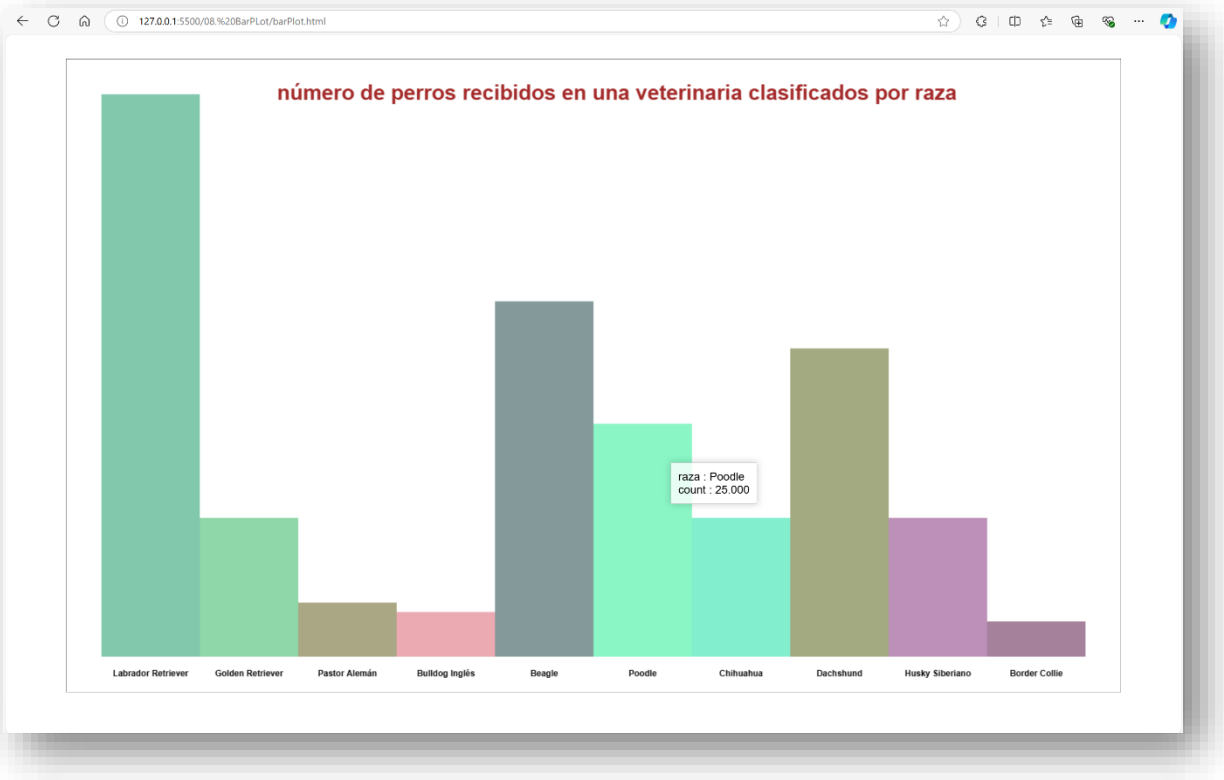
- Se importa los métodos initViewport , drawBarPlot y loadCSV desde el archivo main.js, y opcionalmente, drawXAxisFromArray, drawYAxisFromArray.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Llamamos a drawBarPlot con los parámetros necesarios para el gráfico.

Main.js

```
import { initViewport, drawBarPlot, drawText, loadCSV } from
'../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500;
  const height = 900;
  const context = initViewport(canvasId, width, height);

  //Carga de datos desde CSV
  const filePath  = "/data/perros.csv";
  const data = await loadCSV(filePath);

  //Parametros para la gráfica
  const canvas = document.getElementById("miCanvas");
  const xColumnName = "raza"
  const canvasPadding = 50;

  drawBarPlot(canvas, context, filePath, xColumnName,
canvasPadding);

  const xPosTitle =  width/2 - 9*canvasPadding ;
  const yPosTitle = height - canvasPadding;
  const titleSize = 30;
  const titleColor = "brown"

  drawText(context,"número de perros recibidos en una veterinaria
clasificados por raza", xPosTitle, yPosTitle, titleSize
,titleColor,- 0)

});
```

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador



Gráfica de burbujas

El siguiente ejemplo muestra cómo utilizar el método drawBubblePlot de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo Gráfica de burbujas</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js"
></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px solid
#000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

- Se importa los métodos initViewport , drawBubblePlot y loadCSV desde el archivo main.js, y opcionalmente, drawXAxisFromArray, drawYAxisFromArray.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Llamamos a drawBubblePlot con los parámetros necesarios para el gráfico.

Nótese que se hace uso del objeto axesPorperties, el cual contiene los valores necesarios para dibujar los ejes X y Y del gráfico.

```
//Parametros para los ejes
const axesProperties = {
    xPos: 0,
    yPos: 0,
    xLabelSpace: 20,
    yLabelSpace: -10,
    xLabelTextAngle: 0,
    color: "black",
    xAxeType: -1,
    yAxeType: -1
};
```

Adicionalmente, hacemos uso de la función drawText para colocar un título y un subtítulo a la gráfica.

Título:

drawText(context,"Libros: precio vs número de páginas", xPosTitle, yPosTitle, titleSize ,titleColor,- 0)

Subtítulo:

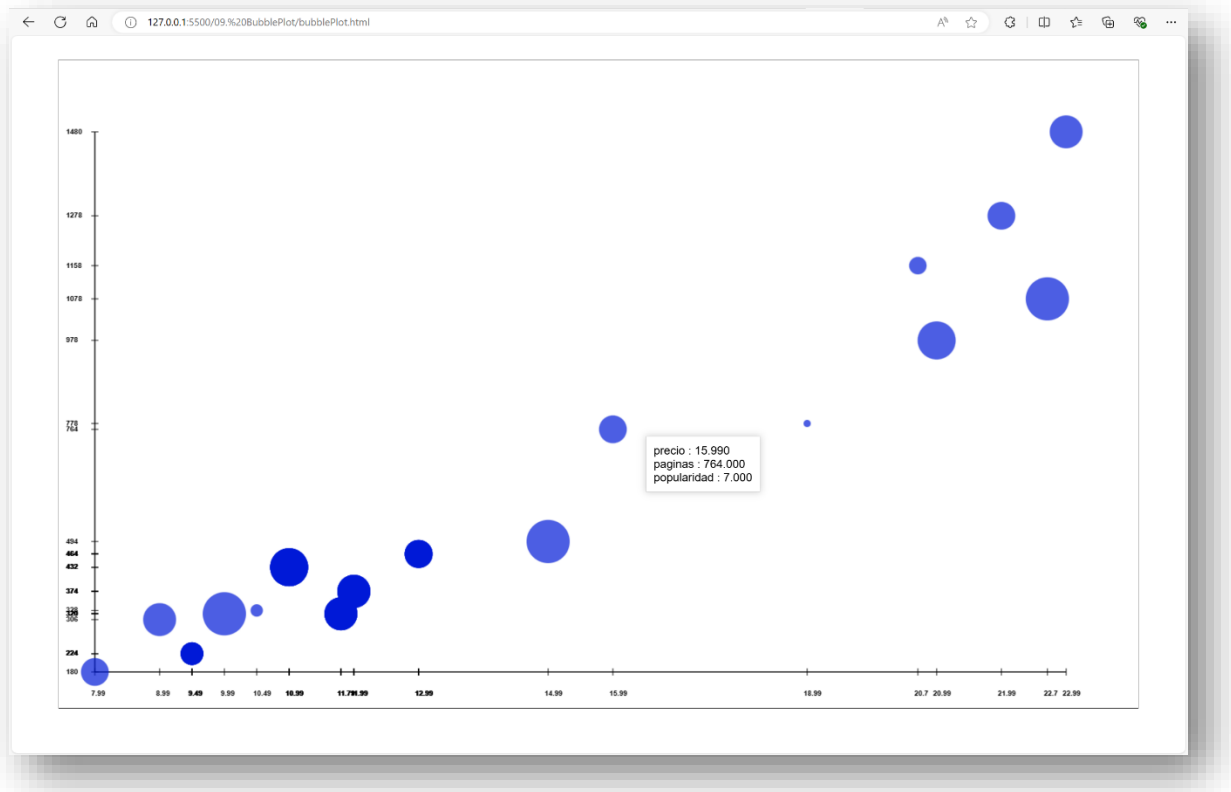
drawText(context,"(El tamaño de burbuja equivale a la popularidad del libro)", xPosSubtitle, yPosSubtitle, subtitleSize ,titleColor,- 0)

Main.js

```
import {initViewport,drawBubblePlot,drawText,loadCSV} from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500;
  const height = 900;
  const context = initViewport(canvasId, width, height);

  //Carga de datos desde CSV
  const filePath  = "/data/libros.csv";
  const data = await loadCSV(filePath);

  //Parametros para la gráfica
  const canvas = document.getElementById("miCanvas");
  const xColumnName = "precio";
  const yColumnName = "paginas";
  const zColumnName = "popularidad";
  let infoColumnNames = [xColumnName, yColumnName, zColumnName];
  const minSize = 5;
  const maxSize = 30;
  const transparence = 0.5;
  const color = 'rgba(0, 25, 215, 0.7)';
  const canvasPadding = 50;

    //Parametros para los ejes
    const axesProperties = {
      xPos: 0,
      yPos: 0,
      xLabelSpace: 20,
      yLabelSpace: -10,
      xLabelTextAngle: 0,
      color: "black",
      xAxeType: -1,
      yAxeType: -1
    };

  drawBubblePlot(canvas, context, filePath, xColumnName,yColumnName, zColumnName,
minSize, maxSize, infoColumnNames,  color, canvasPadding,axesProperties);


  const xPosTitle =  width/2 - 5*canvasPadding ;
  const yPosTitle = height - canvasPadding;
  const titleSize = 30;
  const titleColor = "brown"
  const xPosSubtitle = width/2 - 4*canvasPadding ;
  const yPosSubtitle = height - 2*canvasPadding;
  const subtitleSize = 15;

  drawText(context,"Libros: precio vs número de páginas", xPosTitle, yPosTitle,
titleSize ,titleColor,- 0)
  drawText(context,"(El tamaño de burbuja equivale a la popularidad del libro)",
xPosSubtitle, yPosSubtitle, subtitleSize ,titleColor,- 0)

});
```

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador



Mapa de puntos

El siguiente ejemplo muestra cómo utilizar el método drawMapDotPlot de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo: Mapa de puntos</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js"
></script>
    <script src="https://d3js.org/d3.v6.min.js"></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px solid
#000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

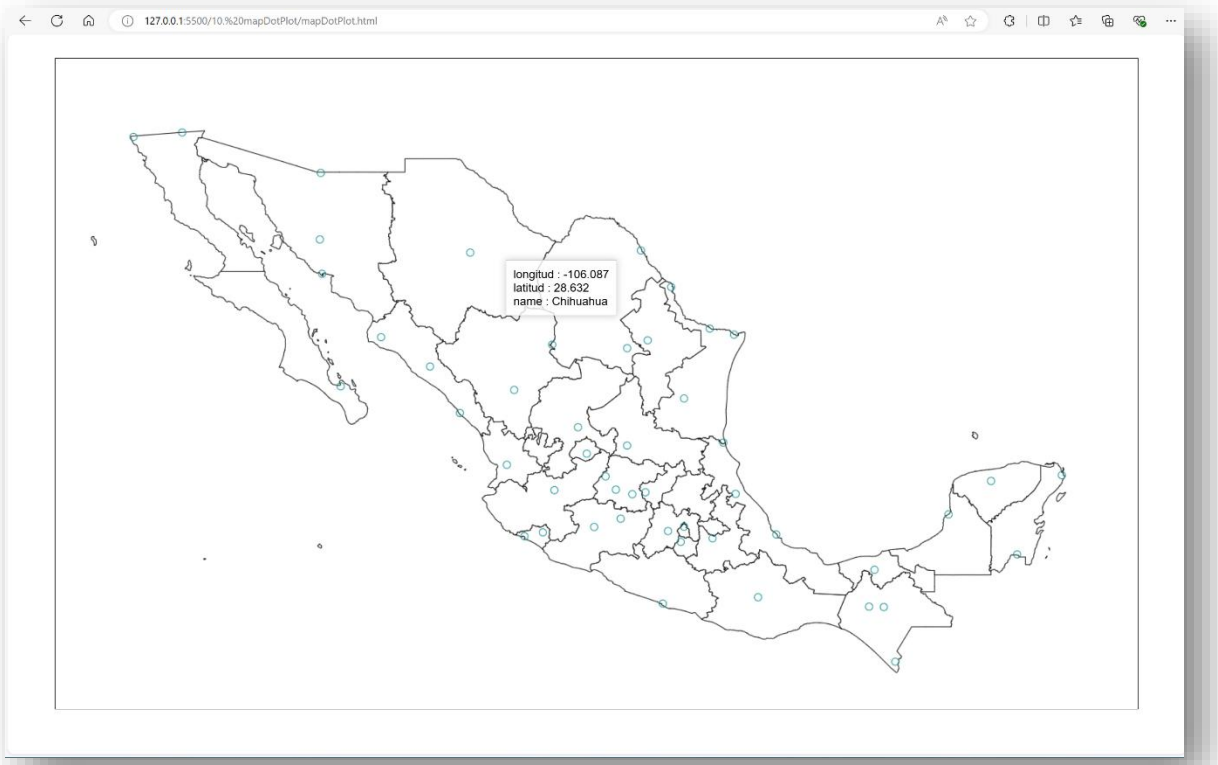
- Se importan los métodos initViewport y drawMapDotPlot desde el archivo main.js.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Llamamos a drawMapDotPlot con los parámetros necesarios para el gráfico.

Main.js

```
import { initViewport, drawText, drawMapDotPlot } from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500;
  const height = 900;
  const context = initViewport(canvasId, width, height);

  const canvas = document.getElementById(canvasId);
  const canvasPadding = 50;
  const mapDataFile = "/data/states.geojson"

  //Carga de datos desde CSV
  const filePath  = "/data/puntos_mapa.csv";

  //Parametros para la gráfica
  const longitudeColumnName = "longitud"
  const latitudeColumnName = "latitud";
  let infoColumnNames=[longitudeColumnName,latitudeColumnName,"name"]
  const color = "darkcyan"
  const filledCircles = true;
  const pointRadius = 7;

  drawMapDotPlot(canvas, context, mapDataFile, filePath,
longitudeColumnName, latitudeColumnName, infoColumnNames, color,
filledCircles, pointRadius, canvasPadding)

  const xPos =  width/2 - 8*canvasPadding ;
  const yPos = height -canvasPadding;
  const titleSize = 30;
  const titleColor = "brown"
  drawText(context,"Algunas ciudades de México y sus localización
geográfica", xPos, yPos, titleSize ,titleColor,- 0)

});
```

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador



Mapa de burbujas

El siguiente ejemplo muestra cómo utilizar el método drawMapBubblePlot de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo: Mapa de burbujas</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js"
></script>
    <script src="https://d3js.org/d3.v6.min.js"></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px solid
#000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

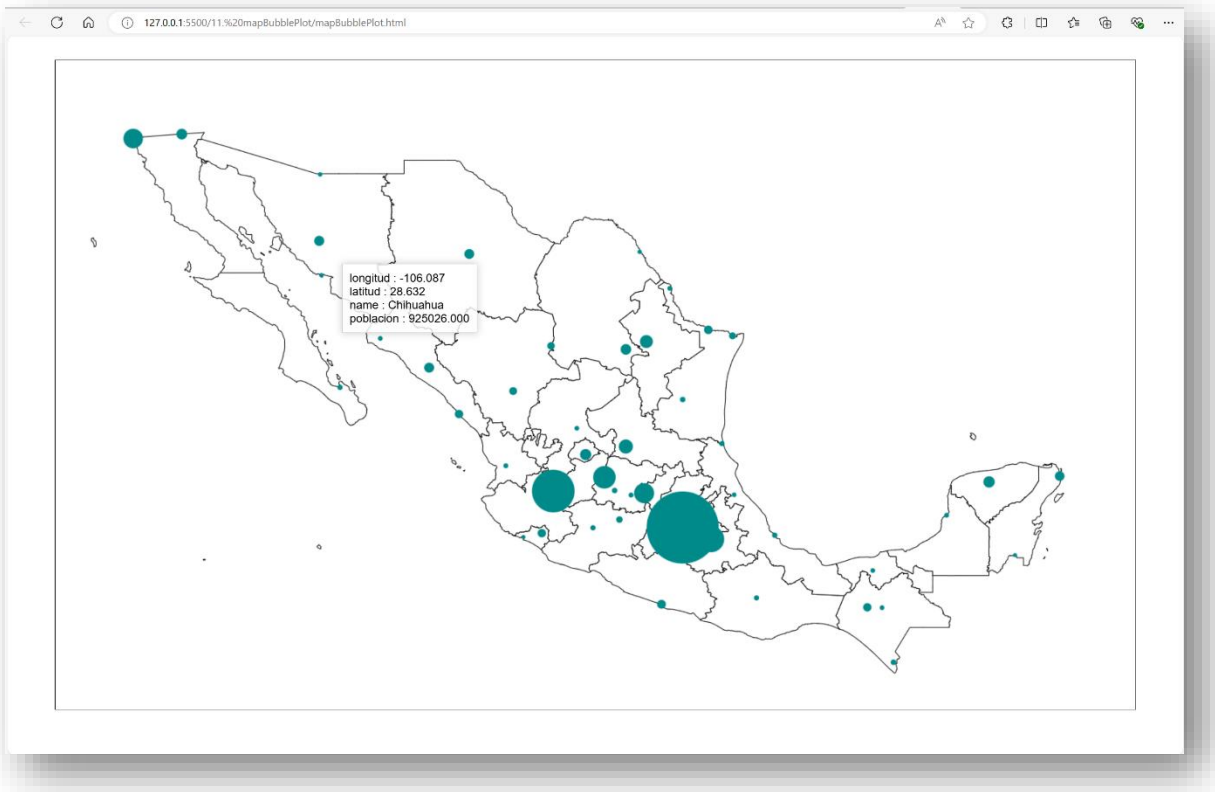
- Se importan los métodos initViewport y drawMapBubblePlot desde el archivo main.js.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Llamamos a drawMapBubblePlot con los parámetros necesarios para el gráfico.

Main.js

```
import { initViewport, drawText, drawMapBubblePlot } from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500;
  const height = 900;
  const context = initViewport(canvasId, width, height);

  const canvas = document.getElementById(canvasId);
  const canvasPadding = 50;
  const mapDataFile = "/data/states.geojson";

  //Carga de datos desde CSV
  const csvFilePath  = "/data/bubbles_mapa.csv";

  //Parametros para la gráfica
  const longitudeColumnName = "longitud";
  const latitudeColumnName = "latitud";
  const sizeColumnName = "poblacion";
  const minSizeBubble = 2;
  const maxSizeBubble = 50;
  let infoColumnNames = [longitudeColumnName, latitudeColumnName, "name" ,
"poblacion"];
  const color = "darkcyan";

  drawMapBubblePlot(canvas, context, mapDataFile, csvFilePath, longitudeColumnName,
latitudeColumnName, sizeColumnName,minSizeBubble, maxSizeBubble, infoColumnNames,
color, canvasPadding)

  const xPos =  width/2 - 8*canvasPadding ;
  const yPos = height -canvasPadding;
  const titleSize = 30;
  const titleColor = "brown"
  const xPosSubtitle = width/2 - 3.5*canvasPadding ;
  const yPosSubtitle = height - 2*canvasPadding;
  const subtitleSize = 15;

  drawText(context,"Algunas ciudades de méxico y su localización geográfica", xPos,
yPos, titleSize ,titleColor,- 0)
  drawText(context,"(El tamaño de burbuja equivale al número de habitantes)",
xPosSubtitle, yPosSubtitle, subtitleSize ,titleColor,- 0)

});
```

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador



Mapa de color (República Mexicana)

El siguiente ejemplo muestra cómo utilizar el método drawHeatMap de la biblioteca vda.js.

Definimos un elemento canvas (miCanvas) donde se dibujarán los elementos gráficos.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo: Mapa de color México</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js"
></script>
    <script src="https://d3js.org/d3.v6.min.js"></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px solid
#000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

- Se importan los métodos initViewport y drawHeatMap desde main.js.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Proporcionamos los archivos GeoJson y CSV para la geometría del mapa y los datos a representar respectivamente.
- Definimos los valores `linkNameProperty`, `stateNameProperty`, que son los identificadores de las columnas que comparten ambos sets de datos y que sirven como llaves únicas para enlazar ambos.
- Llamamos a drawHeatMap con los parámetros necesarios para el gráfico.

Main.js

```
import { initViewport, drawText, drawHeatMap } from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500;
  const height = 900;
  const context = initViewport(canvasId, width, height);

  //Obtención de polígonos para dibujar
  const canvas = document.getElementById(canvasId);
  const canvasPadding = 50;
  const mapDataFile = "/data/states.geojson";

  // Parámetros para el heatMap
  const csvFilePath = '/data/colorMap_Mexico.csv';
  const variableName = 'urbanizacion'; // Nombre de la columna en el CSV
  //const variableName = 'poblacion'; // Nombre de la columna en el CSV
  //const variableName = 'temperatura'; // Nombre de la columna en el CSV
  const baseColor = "purple"
  const stateNameProperty="state_name";//nombre de la propiedad en archivo geojson
  const linkNameProperty = "nombre"; //nombre de la columna con la que se comparará
el stateNameProperty
  let infoColumnNames = ["nombre", "temperatura", "poblacion", "urbanizacion"];

  drawHeatMap(canvas, canvasPadding, mapDataFile, csvFilePath, variableName,
baseColor, stateNameProperty, linkNameProperty, infoColumnNames);

  const xPos =  width/2 - 4*canvasPadding ;
  const yPos = height -canvasPadding;
  const titleSize = 30;const titleColor = "brown";
  const xPosSubtitle = width/2 - 4.2*canvasPadding ;
  const yPosSubtitle = height - 1.8*canvasPadding;
  const subtitleSize = 15;

  drawText(context,"Estados de la República Mexicana",xPos,yPos,titleSize
,titleColor,- 0);
  drawText(context,"(El degradado de color indica el grado de urbanización en cada
estado)", xPosSubtitle, yPosSubtitle, subtitleSize ,titleColor,- 0);

});
```

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador

Mapa de color (USA)

Este ejemplo sirve para enfatizar que es posible indicar un archivo GeoJSON propio y no únicamente los dos establecidos en estos casos de uso; esto permite dibujar cualquier geometría siempre y cuando se respete la estructura de este tipo de archivos.

Definimos un elemento canvas (miCanvas) donde se dibujará el mapa de calor.

viewPort.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo: Mapa de color USA</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js"
></script>
    <script src="https://d3js.org/d3.v6.min.js"></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px solid
#000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

En este caso, también nos es posible agregar una etiqueta (de forma opcional)

```
<div id="infoOverlay"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el gráfico.

Posteriormente:

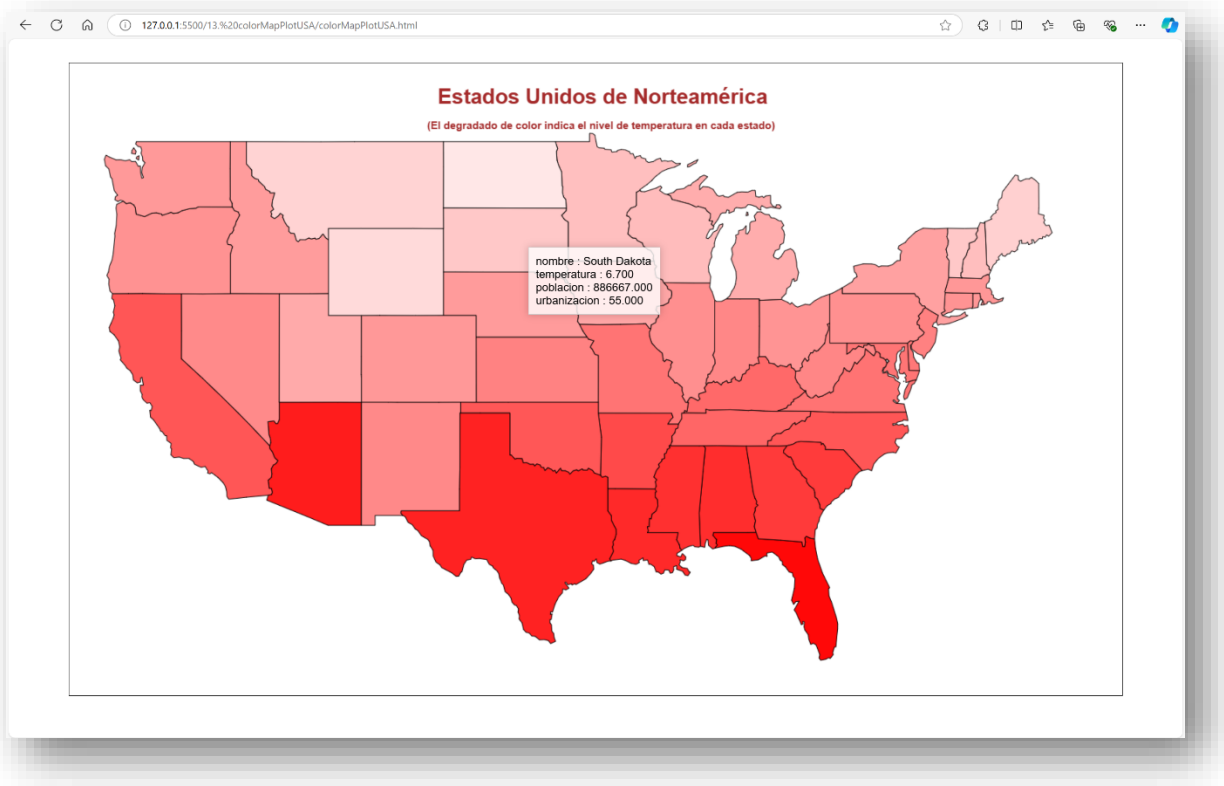
- Se importan los métodos initViewport y drawHeatMap desde main.js.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Proporcionamos los archivos GeoJson y CSV para la geometría del mapa y los datos a representar respectivamente.
- Definimos los valores `linkNameProperty`, `stateNameProperty`, que son los identificadores de las columnas que comparten ambos sets de datos y que sirven como llaves únicas para enlazar ambos.
- Llamamos a drawHeatMap con los parámetros necesarios para el gráfico.

Main.js

```
import { initViewport, drawText, drawHeatMap } from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500; const height = 900;
  const context = initViewport(canvasId, width, height);

  //Obtención de polígonos para dibujar
  const canvasPadding = 50;
  const mapDataFile = "/data/usaStates_simple.geojson"

  // Parámetros para el heatMap
  const canvas = document.getElementById(canvasId);
  const csvFilePath = '/data/colorMap_USA.csv';
  //const variableName = 'urbanizacion'; // Nombre de la columna en el CSV
  //const variableName = 'poblacion'; // Nombre de la columna en el CSV
  const variableName = 'temperatura'; // Nombre de la columna en el CSV
  const stateNameProperty ="name"//nombre de la propiedad en archivo geojson
  const linkNameProperty = "nombre" //nombre de la columna con la que se
comparará el stateNameProperty
  let infoColumnNames=["nombre","temperatura","poblacion","urbanizacion"];

  drawHeatMap(canvas, canvasPadding, mapDataFile, csvFilePath, variableName,
"red", stateNameProperty, linkNameProperty, infoColumnNames);

  const xPos =  width/2 - 4.5*canvasPadding ;
  const yPos = height -canvasPadding;
  const titleSize = 30; const titleColor = "brown";
  const xPosSubtitle = width/2 - 4.8*canvasPadding ;
  const yPosSubtitle = height - 1.8*canvasPadding; const subtitleSize = 15;

  drawText(context,"Estados Unidos de Norteamérica", xPos, yPos, titleSize
,titleColor,- 0);
  drawText(context,"(El degradado de color indica el nivel de temperatura en
cada estado)", xPosSubtitle, yPosSubtitle, subtitleSize ,titleColor,- 0);
});
```

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador



Serie de tiempo

En este ejemplo se hace uso de una serie de eventos cronológicos. Se dibuja una colección de puntos con base en el conjunto de fechas dadas en el formato “yyyy-mm-dd” y un valor asociado a cada una de ellas. El valor en las abscisas será definido por la fecha, mientras que el valor en las ordenadas estará definido por el valor asociado a cada fecha; el archivo con estos datos puede obtenerse de la carpeta data time_series_100.csv del repositorio de GitHub.

Mediante el método drawTimeSeries podremos dibujar las líneas que conectan los puntos en la serie de tiempo, haciendo uso de curvas de Bezier. Las curvas de Bezier son un tipo de curva paramétrica ideal para modelar formas suaves, se definen mediante un conjunto de puntos de control y son extremadamente flexibles.

Existen curvas de bezier lineales, cuadráticas y cúbicas; la librería VDA hace uso de curvas de bezier cuadráticas para dibujar las líneas entre los puntos, estas curvas están definidas por cuatro puntos: un punto inicial

Como se puede ver en el código del archivo mai.js, utilizaremos el método bezierCurveTo, el cual es parte de la API de dibujo en canvas de HTML5, el cual se define de la siguiente manera:

bezierCurveTo(cp1x,cp1y,cp2x,cp2y,x,y)

donde:

1. cp1x, cp1y: Coordenadas del primer punto de control
2. cp2x, cp2y: Coordenadas del segundo punto de control
3. x, y: Coordenadas del punto final

Notar que el punto inicial es el último punto que se dibujó.

timeSeriesPlot.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo Gráfica línea de tiempo</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js"
></script>
    <link rel="stylesheet" href="./styles.css">
</head>
<body>
    <canvas id="miCanvas" width="1200" height="900" style="border:1px solid
#000000;"></canvas>
    <div id="infoOverlay" class="info-overlay"></div>
    <div id="eventToolTip" style="position: absolute; display: none;
background: rgba(0, 0, 0, 0.7); color: white; padding: 5px; border-radius:
5px;"></div>
    <script type="module" src="./main.js"></script>
</body>
</html>
```

Adicionalmente, el método permite agregar una serie de bandas de tiempo para enfatizar eventos dentro de la serie de tiempo. Estos eventos son enviados como un objeto de array al método, tal como puede observarse en el código del archivo main.js

Para visualizar los eventos, será necesario incluir la siguiente etiqueta en el archivo timeSeriesPlot.html

```
<div id="eventTootlTip"></div>
```

la cual nos permitirá agregar una capa de información desplegable cada vez que el mouse sea deslizado sobre el título de cada evento.

Finalmente, en el archivo main.js:

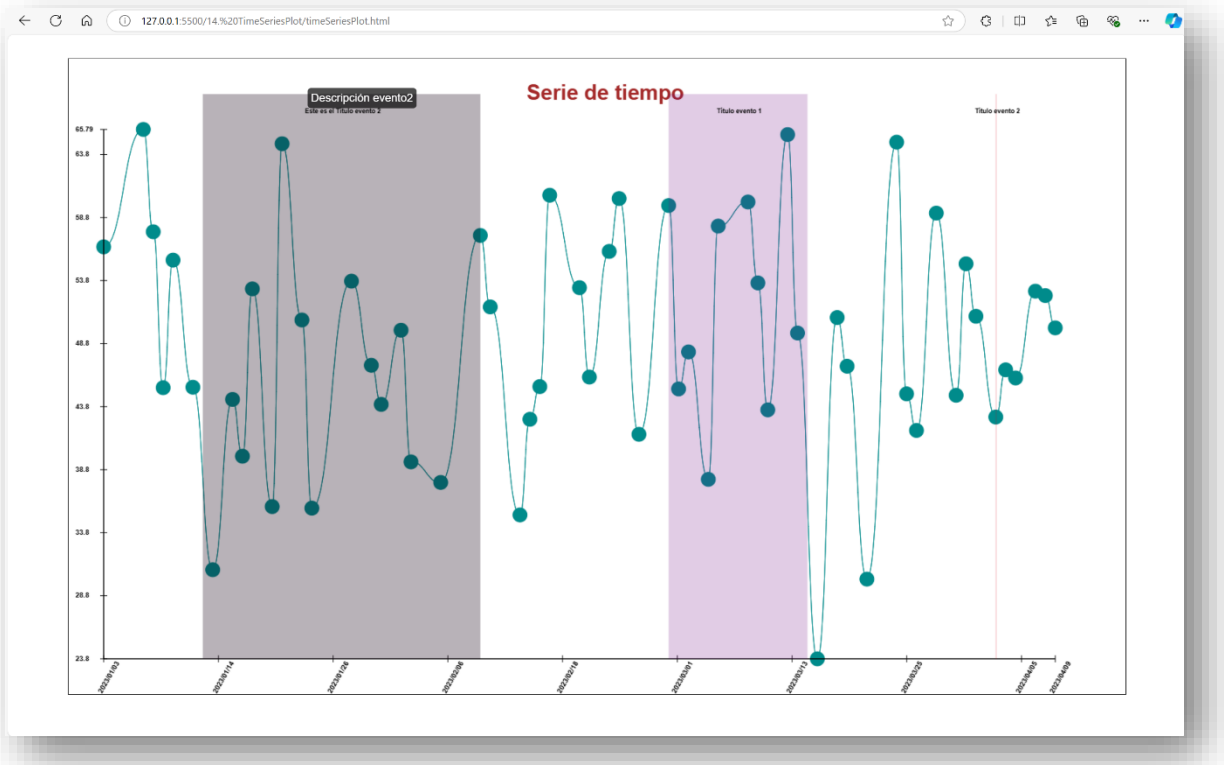
- Se importan los métodos initViewport y drawTimeSeries desde main.js.
- Inicializamos el canvas y su contexto mediante el método initViewport().
- Proporcionamos el archivo CSV que contiene la serie de tiempo.
- Definimos parámetros para la gráfica, ejes en el plano y array de eventos.
- Llamamos a drawTimeSeries para dibujar el gráfico.

Main.js

```
import { initViewport, drawTimeSeries, drawText, loadCSV } from '../vda.js';

document.addEventListener('DOMContentLoaded', async () => {

  //Init ViewPort
  const canvasId = 'miCanvas';
  const width = 1500; const height = 900;
  const context = initViewport(canvasId, width, height);

  //Carga de datos desde CSV
  const filePath  = "/data/time_series_100.csv";
  const data = await loadCSV(filePath);

  //Parametros para la gráfica
  const canvas = document.getElementById("miCanvas");
  const xColumnName = "date";
  const yColumnName = "value";
  let infoColumnNames = [xColumnName, yColumnName];
  const color = "darkcyan";
  const filledCircles = true; const pointRadius = 10;
  const lineWidth = 1;
  const canvasPadding = 50;

  //Parametros para los ejes
  const axesProperties = {
    xPos: 0,
    yPos: 0,
    xLabelSpace: 2,
    yLabelSpace: -10,
    xLabelTextAngle: 60,
    color: "black",
    xAxeType: -1,
    yAxeType: -1
  };

  //Eventos definidos para la serie de tiempo
  const events = [
    { startDate: '2023-01-13', endDate: '2023-02-10', title: 'Título evento 2', desc:
'Descripción 2', color: 'rgba(20, 0, 20, 0.3)' },
    { startDate: '2023-03-01', endDate: '2023-03-15', title: 'Título evento 1', desc:
'Descripción 1', color: 'rgba(105, 0, 123, 0.2)' },
    { startDate: '2023-04-03', endDate: '2023-04-03', title: 'Título evento 2', desc:
'Descripción 2', color: 'rgba(220, 0, 20, 0.3)' }
  ];

  // Lamada a la función para dibujar la serie de tiempo
  drawTimeSeries(canvas, context, filePath, xColumnName,yColumnName, infoColumnNames, color,
filledCircles, pointRadius, lineWidth, canvasPadding, axesProperties,events);

  const xPos =  width/2 - 2*canvasPadding ;
  const yPos = height -canvasPadding;
  const titleSize = 30;
  const titleColor = "brown"
  drawText(context,"Serie de tiempo", xPos, yPos, titleSize ,titleColor,- 0)
});
```

En la siguiente imagen podemos apreciar el gráfico de línea dibujado en el navegador web.

Navegador



Mapa de burbujas interactivo (incendios)

Este ejemplo tiene como objetivo crear un **mapa interactivo** que visualice los incendios ocurridos entre 2015 y 2021. El mapa utiliza burbujas para representar los incendios, donde el tamaño de cada burbuja es proporcional al total de hectáreas afectadas por el incendio. Además, el impacto del incendio y el tipo de incendio están representados mediante colores específicos, lo que permite a los usuarios comprender visualmente la magnitud y características de cada evento.

El mapa cuenta con los siguientes controles interactivos:

1. **Control de Año (Deslizador):**
    - Es posible seleccionar un año entre 2015 y 2021 utilizando un control deslizante. Se actualizará el mapa para mostrar los incendios correspondientes al año seleccionado.
2. **Control de Impacto (Checkboxes):**
    - Se ofrecen tres opciones de impacto (mínimo, moderado, severo) mediante casillas de verificación. Es posible seleccionar uno o más tipos de impacto, y el mapa se actualizará para mostrar solo los incendios que coincidan con los tipos seleccionados.

Datos utilizados en el mapa: Los datos que alimentan el mapa provienen de un archivo CSV que contiene las siguientes columnas:

- **Anio**: Año del incendio.
- **Latitud y Longitud**: Ubicación geográfica del incendio (para posicionar la burbuja en el mapa).

- **Tipo_incendio**: Tipo de incendio (de copa, superficial, subterráneo, etc.). Este valor se usa para definir el color del contorno de la burbuja.
- **Tipo_impacto**: Tipo de impacto (mínimo, moderado, severo). Este valor se usa para definir el color de relleno de la burbuja.
- **Total_hectareas**: Tamaño del incendio en hectáreas. Este valor determina el tamaño de la burbuja en el mapa.

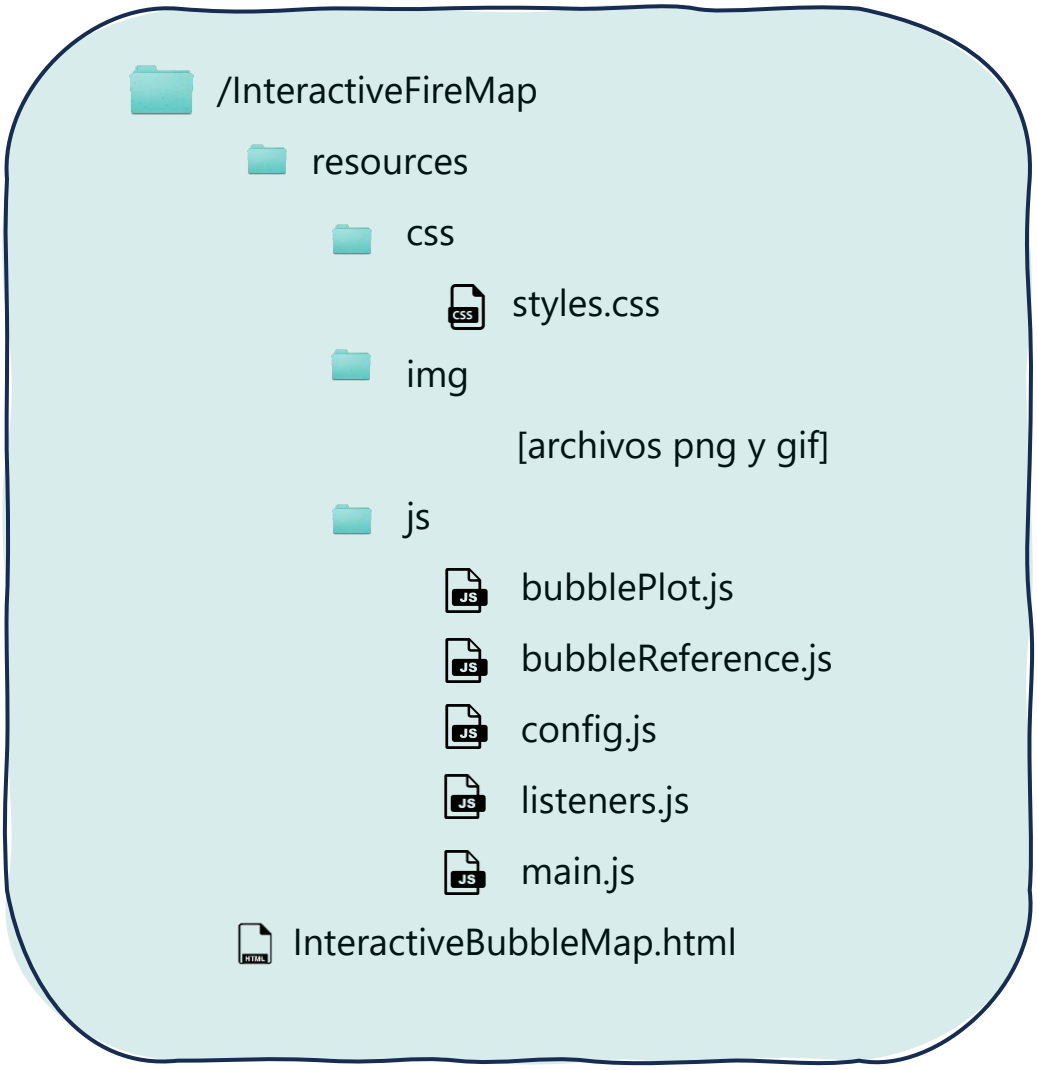
Referencias:

1. **Referencia Dinámica de Tamaño**:
    - Cada burbuja en el mapa tendrá una referencia dinámica que permite comparar el tamaño del incendio con dimensiones conocidas, como el tamaño de Ciudad Universitaria, el Auditorio Nacional, o el Estadio Azteca. Esto ayudará a los usuarios a visualizar la magnitud del incendio en términos más familiares.
2. **Referencia Estática de Colores**:
    - Se incluirá una leyenda estática en el mapa que explique el significado de los colores de las burbujas (relacionados con el impacto) y el color del contorno de las burbujas (relacionados con el tipo de incendio). Esto facilitará la interpretación visual de los datos y mejorará la experiencia del usuario.

Interacción del Mapa: El usuario podrá interactuar con el mapa mediante los controles de impacto y de año. Al realizar una selección en cualquiera de estos controles, el mapa se actualizará dinámicamente para reflejar solo los incendios que coincidan con los filtros seleccionados.

Anteriormente, se mostró un mapa de burbujas, por lo que usaremos este ejemplo y código como base para este ejemplo, añadiendo los controles interactivos para los filtros. El código fue dividio en varios archivos .js con el fin de hacerlo más entendible y modular.

La estructura del proyecto es la siguiente:



Se explicará primeramente el archivo html y posteriormente las dependencias a los archivos java script.

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <!-- Meta Información -->
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Incendios Forestales en México</title>

    <!-- Estilos -->
    <link rel="stylesheet" href="resources/css/styles.css">

    <!-- Scripts Externos -->
    <script src =
"https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js">
    </script>
    <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
    <div class="container">
        <!-- Título y Descripción del Mapa -->
        <h1 id="mapTitle">Incendios Forestales en México</h1>
        <p id="mapDesc">
            Datos recabados por la CONAFOR a partir de 2015 a 2021.
            El tamaño de las burbujas representa el total de hectáreas
            afectadas; se tiene un rango de afectación general de
            <span id="minRange">0.005</span> a
            <span id="maxRange">23,809</span> hectáreas.
        </p>

        <!-- Canvases para el Mapa y las Burbujas -->
        <canvas id="mapCanvas"></canvas>
        <canvas id="bubbleCanvas"></canvas>

        <!-- Superposición de Información -->
        <div id="infoOverlay" class="info-overlay"></div>

        <!-- Referencia de Colores para las Burbujas -->
        <canvas id="bubbleColorReferenceCanvas"></canvas>
    </div>
```

index.html

```
<!-- Control Deslizante para Seleccionar el Año -->
  <input type="range" id="yearSlider" min="2015" max="2021" value="2015"
  step="1" list="years_">
  <datalist id="years_">
    <option value="2015">2015</option> <option value="2016">2016</option>
    <option value="2017">2017</option> <option value="2018">2018</option>
    <option value="2019">2019</option> <option value="2020">2020</option>
    <option value="2021">2021</option>
</datalist>

<!-- Filtros de Impacto -->
<div id="impactChecks">
    <span>Impacto:</span> <input type="checkbox" class="Impactcheck"
  id="minimo" name="categoria" value="minimo">
    <label for="minimo">Mínimo</label><br> <input type="checkbox"
  class="Impactcheck" id="moderado" name="categoria" value="moderado">
    <label for="moderado">Moderado</label><br> <input type="checkbox"
  class="Impactcheck" id="severo" name="categoria" value="severo">
    <label for="severo">Severo</label><br>
</div>

<!-- Referencia del Tamaño de las Burbujas -->
<div class="bubbleSizeReference">
    <div class="bubbleSizeReference-header">
        <span>Referencia para el tamaño de las burbujas</span>
    </div>
    <div class="bubbleSizeReference-content">
        <p><img src="resources/img/cu.png" alt="Ciudad Universitaria">
        750 hect. aprox. <br> C. Universitaria</p>
        <p><img src="resources/img/fundidora.png" alt="Parque Fundidora">
        100 hect. aprox. <br> Parque Fundidora</p>
        <p><img src="resources/img/azteca.png" alt="Estadio Azteca"> 50
        hect. aprox. <br> Estadio Azteca</p>
        <p><img src="resources/img/auditorio.png" alt="Auditorio
        Nacional"> 10 hect. aprox. <br> Auditorio Nacional</p>
        <p><img src="resources/img/zocalo.png" alt="Zócalo CDMX"> 5 hect.
        aprox. <br> Zócalo, CDMX</p>
        <p><img src="resources/img/park.png" alt="Parque España"> 1
        hectárea aprox.<br> Parque España</p>
        <!-- Canvas para Referencia del Tamaño -->
        <canvas id="bubbleSizeReferenceCanvas"></canvas>
    </div>
</div>
```

index.html

```
    <!-- Cargando -->
    <div id="loading">
        <img src="resources/img/loading.gif" alt="Cargando...">
    </div>

    <!-- Scripts del Proyecto -->
    <script type="module" src="resources/js/main.js"></script>
</body>
</html>
```

Meta Información (head)

1. Título: Se establece el título de la página como "Incendios Forestales en México".
2. Estilos: Se enlaza una hoja de estilos CSS externa (resources/css/styles.css) para el diseño visual de la página.
3. Scripts Externos:
    - PapaParse: Se añaden bibliotecas de JavaScript para parsear archivos CSV y podr leer los datos de incendios).
    - D3.js: Como el el resto de ejemplos, se añade la biblioteca para el manejo de gráficos y visualizaciones.

Cuerpo (body)

1. Contenedor Principal (.container): Este elemento div envuelve todos los elementos del contenido visual.
2. Título y Descripción del Mapa:
    - Se muestra un título y una breve descripción que explica el propósito del mapa: mostrar incendios forestales entre 2015 y 2021, con información dinámica adicional sobre el mínimo y máximo de hectáreas afectadas.
3. Canvas para el Mapa y las Burbujas:
    - #mapCanvas: Este canvas es el lienzo para dibuja el mapa.
    - #bubbleCanvas: Aquí se dibujan las burbujas que representan los incendios, con el tamaño y color dependiendo de los datos.
4. Superposición de Información (#infoOverlay):
    - Este div se usa para mostrar información adicional de los incendios cuando se desliza el cursor sobre las burbujas en el mapa.
5. Referencia de Colores para las Burbujas:
    - Este canvas permite mostrar una eferencia con los colores de las burbujas, correspondientes al tipo de impacto o tipo de incendio.

Controles Interactivos

1. Control Deslizante para Año (#yearSlider):
    - Permite seleccionar un año entre 2015 y 2021.
2. Filtros de Impacto (#impactChecks):
    - Permiten seleccionar el tipo de impacto de los incendios: mínimo, moderado o severo.
3. Div para la Referencia del Tamaño:
    - Se muestra una serie de ejemplos visuales que permiten a los usuarios comparar el tamaño de las burbujas con áreas conocidas como la Ciudad Universitaria, el Estadio Azteca, etc.
    - Cada área de comparación tiene una imagen representativa y una descripción con el tamaño en hectáreas.
    - El canvas con el identificador #bubbleSizeReferenceCanvas se usa para graficar estas comparaciones visualmente.

Carga de Datos

1. Elemento de Carga (#loading):
    - Muestra una imagen animada de carga mientras los datos del mapa están siendo procesados o cargados.

Script Principal

1. La llamada a main.js controla la lógica de todo el proceso de lectura, procesamiento y visualización de datos.

Ahora, se explicaran individualmente los archivos .js para entender todo el proceso a nivel técnico.

main.js

```
import { initViewport, drawMap, loadCSV } from '../../../vda.js';
import { drawBubbleColorReference } from './bubbleReference.js';
import { setupListeners } from './listeners.js';
import { drawBubblesWithLoading } from './bubblePlot.js';
import { config } from './config.js';

document.addEventListener('DOMContentLoaded', async () => {
  const contexts = {};
  for (const key in config.canvas) {
    if (key !== 'width' && key !== 'height') {
      contexts[key] = initViewport(config.canvas[key], config.canvas.width,
config.canvas.height);
    }
  }

  const bbox = await drawMap(
    document.getElementById(config.canvas.map),
    config.canvasPadding,
    config.mapDataFile,
    'white',
    'black'
  );

  const data = await loadCSV(config.csvFilePath);

  drawBubblesWithLoading(data, config.bubbleParams.defaultYear, contexts,
bbox, config);
  drawBubbleColorReference(contexts.reference, 'white', config.colorMapping);
  setupListeners(data, config, contexts, bbox);
});
```

Primeramente, se importan las funciones necesarias de los modulos js mediante la instrucción import {}. Como se puede ver, la ejecución del código está encapsulada dentro de un evento que se dispara cuando el contenido HTML de la página se ha cargado completamente (función DOMContentLoaded), lo que asegura que todo el contenido del

documento esté disponible antes de intentar manipularlo o mostrar datos.

Dentro de este evento, creamos un objeto llamado contexts, que servirá para almacenar las configuraciones de los diferentes "canvas" donde se representarán el mapa, las burbujas y las referencias. El uso de varios camvas se debe a la necesidad de limpiar el espacio cada vez que se escoja un filtro ya que no se requiere limpiar todo el contenido, sino únicamente las burbujas, la superposición de los canvas esta controlada mediante estilos css, especificamnete la propiedad z-index la cual admite un valor numérico y da prioridad a aque elemento con mayor valor, por ejemplo el elemento con valor 5 será posicionado delante del elemento con valor 1 y el elemento con valor 10 será posicionado por encima de los dos anteriores.

Luego, se utiliza la función initViewport para inicializar cada una de estas áreas de visualización.

A continuación, se llama a la función drawMap para dibujar el mapa enel canvas destinado al mapa. La función devuelve un valor llamado bbox, que es un área de coordenadas que define los límites del mapa.

Después, se carga un archivo CSV con datos relacionados con los incendios forestales utilizando la función loadCSV.

Con los datos cargados, se llama a la función drawBubblesWithLoading, que dibuja las burbujas en el mapa representando los incendios.

Después de dibujar las burbujas, se agrega una referencia estática de colores que explica qué representa cada color de las burbujas. Esto se logra mediante la función drawBubbleColorReference, que dibuja la leyenda utilizando la configuración de colores definida en la variable config.colorMapping.

Por último, la función setupListeners configura los eventos que estarán escuchando cada vez que un filtro es modificado. Estos “listeners”

permiten que el usuario interactúe con el mapa, como cambiar el año de los datos mostrados.

config.js

```
export const config = {
    canvas: {
      map: 'mapCanvas',
      bubble: 'bubbleCanvas',
      reference: 'bubbleColorReferenceCanvas',
      width: 1500,
      height: 900,
    },
    mapDataFile: '../../../data/states.geojson',
    csvFilePath: '../../../data/incendios.csv',
    canvasPadding: 50,
    bubbleParams: {
      longitude: 'Longitud',
      latitude: 'Latitud',
      size: 'Total_hectareas',
      minSize: 3,
      maxSize: 60,
      defaultYear: 2015,
    },
    infoColumns: ['Longitud', 'Latitud', 'Total_hectareas', 'Deteccion',
'Duracion', 'Tipo_impacto', 'Tipo_incendio'],
    colorMapping: [
      { color: 'rgba(237, 209, 48, 0.5)', texto: 'Impacto mínimo', tipo: 'fill'
},
      { color: 'rgba(249, 152, 7, 0.65)', texto: 'Impacto moderado', tipo: 'fill'
},
      { color: 'rgba(255, 3, 58, 0.5)', texto: 'Impacto severo', tipo: 'fill' },
      { color: 'rgb(255, 255, 255)', texto: 'Otro', tipo: 'stroke' },
      { color: 'rgba(0, 255, 210, 1)', texto: 'Subterráneo', tipo: 'stroke' },
      { color: 'rgb(133, 0, 184)', texto: 'De copa', tipo: 'stroke' },
      { color: 'rgba(237, 225, 48, 1)', texto: 'Superficial', tipo: 'stroke' },
      { color: 'rgba(199, 0, 57, 1)', texto: 'Mixto', tipo: 'stroke' },
    ],
    colorColumns: {
      fill: 'Tipo_impacto',
      ring: 'Tipo_incendio',
    },
  };
```

Este archivo de configuración centraliza todos los parámetros necesarios para dibujar el mapa, las burbujas y la leyenda de colores.

Primeramente, define un objeto de configuración llamado config, con todos los parámetros necesarios para el mapa interactivo y la visualización de los incendios forestales:

canvas: Este objeto define las configuraciones de los elementos canvas que se utilizan para dibujar el mapa, las burbujas y la referencia de color y sus atributos son.

- map: Especifica el ID del canvas donde se dibuja el mapa (en este caso, 'mapCanvas').
- bubble: Especifica el ID del canvas donde se dibuja las burbujas que representan los incendios ('bubbleCanvas').
- reference: Define el ID del canvas donde se dibuja la referencia de color para las burbujas ('bubbleColorReferenceCanvas').
- width y height: Establecen las dimensiones de los canvas para el mapa y las burbujas, 1500 píxeles de ancho y 900 píxeles de alto, en este caso.

mapDataFile: Especifica la ruta al archivo GeoJSON que contiene los datos geoespaciales para dibujar el mapa de México.

csvFilePath: Es la ruta al archivo CSV que contiene los datos de los incendios.

canvasPadding: Establece el margen (en píxeles) que se debe dejar alrededor del mapa y las burbujas, para evitar que se dibujen demasiado cerca del borde del lienzo.

bubbleParams: Este objeto contiene los parámetros relacionados con las burbujas que se dibujan en el mapa:

- longitude y latitude: Especifican los nombres de las columnas en el archivo CSV que contienen las coordenadas geográficas de los incendios.
- size: Define el nombre de la columna que contiene el tamaño del incendio (en hectáreas). Este valor se usa para determinar el tamaño de las burbujas.
- minSize y maxSize: Establecen el tamaño mínimo y máximo para las burbujas, en píxeles. Esto es útil para asegurarse de que las burbujas tengan un tamaño adecuado y visible.
- defaultYear: Especifica el año por defecto que se debe mostrar en el mapa (2015), el cual se usa como filtro inicial para mostrar los incendios de ese año.

infoColumns: Este arreglo define las columnas del archivo CSV que contienen información relevante sobre los incendios. Estas columnas incluyen la longitud, latitud, tamaño del incendio (en hectáreas), detección, duración, tipo de impacto y tipo de incendio. Estos datos se usan para mostrar información adicional sobre cada incendio al desplazar el mouse sobre una burbuja.

colorMapping: Este arreglo define cómo se deben colorear las burbujas en el mapa, dependiendo del impacto y tipo de incendio. Cada objeto dentro de colorMapping tiene:

- color: El color con el que se rellena la burbuja o el borde (dependiendo del tipo).

- texto: El texto descriptivo que se muestra para cada tipo de impacto o tipo de incendio.
- tipo: Indica si el color se aplica al relleno de la burbuja ('fill') o al borde de la burbuja ('stroke').

Por ejemplo, el tipo de impacto "mínimo" tiene un color amarillo con opacidad de 0.5 y se aplica al relleno de la burbuja. El tipo de incendio "subterráneo" tiene un color verde y se aplica al borde de la burbuja.

colorColumns: Este objeto define qué columnas del archivo CSV se deben usar para determinar el color del relleno y el borde de las burbujas:

- fill: Especifica la columna que se utilizará para determinar el color del relleno de la burbuja (en este caso, Tipo_impacto).
- ring: Define la columna que se utilizará para determinar el color del borde de la burbuja (en este caso, Tipo_incendio).

bubblePlot.js

```
import { drawInteractiveBubbleMapPlot } from '../../../vda.js';

export function drawBubblesWithLoading(data, selectedYear, contexts, bbox,
config, checkboxState = {}) {
    document.getElementById('loading').style.display = 'block';
    setTimeout(() => {
      drawInteractiveBubbleMapPlot(
        document.getElementById(config.canvas.bubble),
        contexts.bubble,
        bbox,
        data,
        config.bubbleParams.longitude,
        config.bubbleParams.latitude,
        config.bubbleParams.size,
        config.bubbleParams.minSize,
        config.bubbleParams.maxSize,
        config.infoColumns,
        config.colorMapping,
        config.colorColumns.fill,
        config.colorColumns.ring,
        config.canvasPadding,
        selectedYear,
        checkboxState
      );
      document.getElementById('loading').style.display = 'none';
    }, 0);
  }
```

Este código define una función llamada drawBubblesWithLoading, que:

1. Muestra una animación de carga (loading) que se muestra mientras se procesa y dibuja el mapa.
2. Llama a la función que dibuja el mapa y las burbujas con los datos proporcionados (esta función será explicada más adelante).
3. Cuando se termina de dibujar el mapa y las burbujas, se oculta la anaimación de carga.

Función drawInteractiveBubbleMapPlot

```
export async function drawInteractiveBubbleMapPlot(canvas, context, bbox, data,
xColumnName, yColumnName, sizeColumnName, minSize=1, maxSize = 5, infoColumNames,
colorMapping,bubbleFillColorColumnName, bubbleRingColorColumnName, canvasPadding
= context.canvas.width * 0.02,yearSelected,impactChecksSelected) {

  //Filtramos los puntos de acuerdo a los controles seleccionados
  if(yearSelected!=null){
    data= data.filter(d => d.Anio == yearSelected);
  }

  // Filtramos los puntos por los impactos seleccionados
  const selectedImpacts = [];
  if (impactChecksSelected) {
    if (impactChecksSelected.minimo) selectedImpacts.push("Impacto mínimo");
    if (impactChecksSelected.moderado) selectedImpacts.push("Impacto moderado");
    if (impactChecksSelected.severo) selectedImpacts.push("Impacto severo");
  }

  if (selectedImpacts.length > 0) {
      data = data.filter(d => selectedImpacts.includes(d.Tipo_impacto));
  }

  //Ordenamos los puntos
  const sortedData = sortData(data, sizeColumnName);

  //Agregamos a sortedData dos nuevos registros con los valores minimos y maximos
en latitud y longitud del mapa para tener el marco completo de referencia
  const newRecords = [
    { [xColumnName]: bbox[0][0], [yColumnName]: bbox[0][1],
[sizeColumnName]:1000, isDummy: true },
    { [xColumnName]: bbox[1][0], [yColumnName]: bbox[1][1],
[sizeColumnName]:1000,isDummy: true }
  ];

  sortedData.push(...newRecords);

  // Mapear datos a valores de canvas
  let dataWithCanvasValues = sortedData.map(row => ({
    ...row,
    xCanvas: mapValue(row[xColumnName], canvasPadding, context.canvas.width - 2 *
canvasPadding, sortedData.map(d => d[xColumnName])),
    yCanvas: mapValue(row[yColumnName], canvasPadding, context.canvas.height - 2
* canvasPadding, sortedData.map(d => d[yColumnName])),
    sizeCanvas: mapValue(row[sizeColumnName], minSize, maxSize, sortedData.map(d
=> d[sizeColumnName]))
  }));
```

Función drawInteractiveBubbleMapPlot

```
//Una vez que se mapearon los puntos con el marco completo de referencia,
eliminamos los puntos dummy para que no sean dibujados
  dataWithCanvasValues = dataWithCanvasValues.filter(record => !record.isDummy);

  //Limpiamos el canvas destinado a las burbujas
  clearCanvas(context);

  //Dibujamos los puntos
  drawBubblesWithFillAndRingColor(context,dataWithCanvasValues,colorMapping,bubbl
eFillColorColumnName, bubbleRingColorColumnName);

  //Dibujamos referencia dinámica de tamaños para las burbujas
  const radiosRef = [1, 5, 10, 50, 100, 750]; // Valores de referencia
  const radiosRefCanvasValues = radiosRef.map(radio =>
    mapValue(radio, minSize, maxSize, sortedData.map(d => d[sizeColumnName]))
  );
  const xPos = 170;
  const yPos = 70;
  const yStep = 73;
  drawInteractiveBubbleSizeReferences(context, sortedData, sizeColumnName,
radiosRefCanvasValues, xPos, yPos, yStep);

  // Evento de hover sobre el canvas
  canvas.addEventListener('mousemove', function(event) {
    handleHover(event, canvas, dataWithCanvasValues, infoColumNames, "dot",
canvasPadding);
  });
}
```

El código define la función drawInteractiveBubbleMapPlot, que es la responsable de dibujar las burbujas en el mapa, para esto, realiza los siguientes pasos:

1. **Filtrar datos por año**: En caso de que se haya seleccionado un año, los datos se filtran para incluir solo aquellos cuya columna Anio coincide con el año seleccionado.

2. **Filtrar datos por impacto**: En caso de que se hayan seleccionado una o más casillas de verificación, se filtran los datos para incluir solo los que coincidan con los tipos seleccionados (Impacto mínimo, Impacto moderado o severo).
3. **Ordenar datos**: Se ordenan los datos según el tamaño de las burbujas (columna sizeColumnName) para asegurarse de que las burbujas más grandes se dibujen primero y no cubran a las más pequeñas.
4. **Mapeo de valores al canvas**: Cada fila de datos se transforma para incluir:
    - xCanvas y yCanvas: Coordenadas mapeadas al sistema de coordenadas del canvas.
    - sizeCanvas: Tamaño mapeado entre los valores mínimo y máximo definidos.
5. **Limpiar el canvas**: Antes de dibujar las nuevas burbujas, se limpia el canvas usando la función clearCanvas.
6. **Dibujar burbujas**: Se hace realiza un llamado a la función drawBubblesWithFillAndRingColor (la cual se explicara más adelante), que dibuja las burbujas en el canvas según su posición, tamaño y colores de relleno y anillo.
7. **Dibujar referencias de tamaño**: Se dibujan una referencia de tamaños dinámica que se ajusta a los filtros seleccionados, esto ayuda a dimensionar facilmente los datos visualizados.
8. **Agregar interactividad**: Se configura un evento para que al deslizar el mouse sobre una burbuja se muestre la información de dicho incendio.

Función drawBubblesWithFillAndRingColor

```
export function drawBubblesWithFillAndRingColor(context, dataWithCanvasValues,
colorMapping, colorFillColumnName, colorRingColumnName ){

  dataWithCanvasValues.forEach(point => {
    // Buscar el color de relleno en el colorMapping
    const fillColor = colorMapping.find(
      ref => ref.texto === point[colorFillColumnName] && ref.tipo === "fill"
    ).color;

    // Buscar el color de contorno en el colorMapping
    const ringColor = colorMapping.find(
      ref => ref.texto === point[colorRingColumnName] && ref.tipo === "stroke"
    ).color;

    // Dibujar la burbuja con relleno
    context.fillStyle = fillColor;
    context.beginPath();
    context.arc(point.xCanvas, point.yCanvas, point.sizeCanvas, 0, 2 * Math.PI);
    context.fill();

    // Dibujar el contorno de la burbuja
    context.strokeStyle = ringColor;
    context.lineWidth = 2; // Ajustar grosor si es necesario
    context.stroke();
  });
}
```

La función drawBubblesWithFillAndRingColor es muy similar a la función que ya se había presentado para dibujar burbujas, sin embargo, en esta ocasión se añade una característica para incluir color al borde de la burbuja y no solo al relleno.

bubbleReference.html

```
import { drawText } from '../../../vda.js';

export function drawBubbleColorReference(context, textColor, colorMapping) {
  context.save();
  let [x, y] = [1030, 870];
  colorMapping.forEach(({ color, texto, tipo }) => {
    context.beginPath();
    context.arc(x, y, 10, 0, 2 * Math.PI);
    if (tipo === 'fill') {
      context.fillStyle = color;
      context.fill();
    } else if (tipo === 'stroke') {
      context.lineWidth = 1.5;
      context.strokeStyle = color;
      context.stroke();
    }
    drawText(context, texto, x + 30, y, 10, textColor, 0);
    y -= 33;
  });
  context.restore();
}
```

Este código define una función llamada drawBubbleColorReference que dibuja una referencia visual, la cual muestra círculos de diferentes colores con etiquetas que describen el significado de estos.

Ilustración 1. Referencia de colores para las burbujas

Al inicio, context.save() guarda el estado actual del canvas para que cualquier cambio en los estilos (colores, líneas, etc.) sea reversible. Esto asegura que no se afecte el resto del dibujo del canvas.

Posteriormente, se recorre el arreglo colorMapping utilizando forEach.

Para cada elemento, se dibuja un círculo en las coordenadas (x, y). Si el estilo es fill, se rellena el círculo con el color especificado; si el estilo es stroke, se dibuja solo el contorno del círculo con el color indicado.

Al final, se llama a context.restore() para restaurar el estado original del canvas. Esto asegura que los cambios en estilos no afecten otros dibujos fuera de la función.

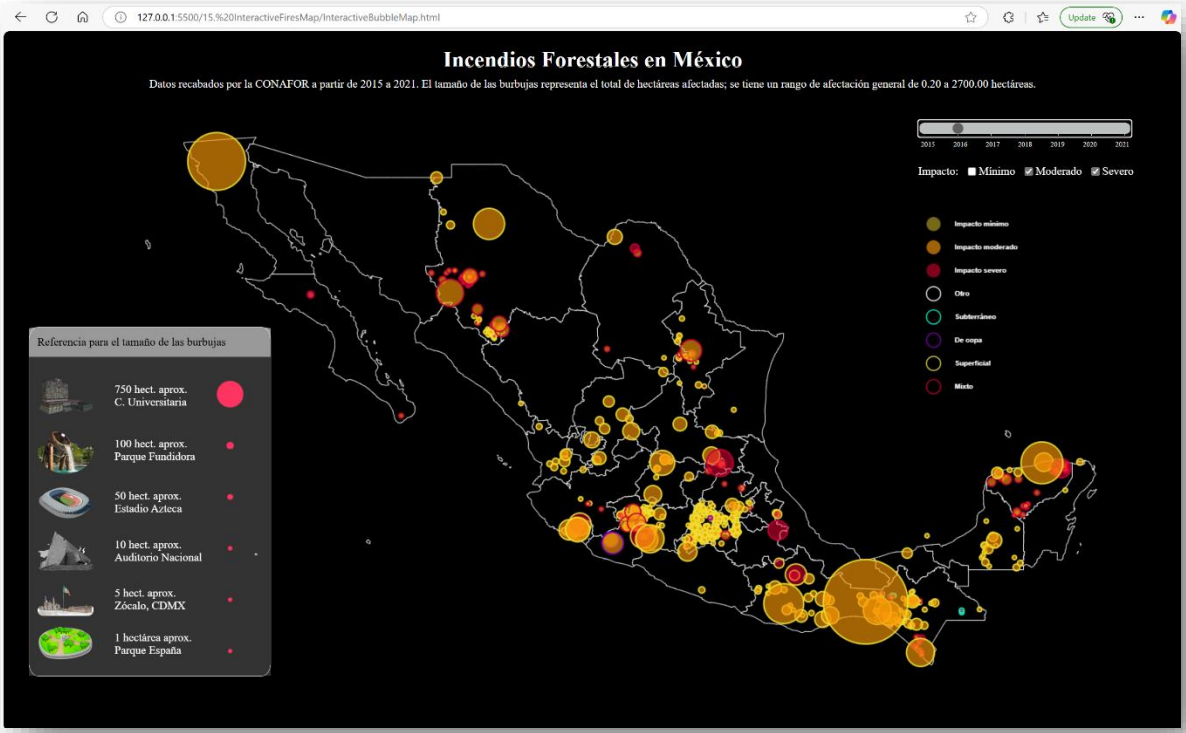
listeners.html

```
import { drawBubblesWithLoading } from './bubblePlot.js';

export function setupListeners(data, config, contexts, bbox) {
  const yearSlider = document.getElementById('yearSlider');
  const checkboxes = document.querySelectorAll('.Impactcheck');

  let selectedYear = config.bubbleParams.defaultYear;
  const checkboxState = { minimo: false, moderado: false, severo: false };

  yearSlider.addEventListener('change', () => {
    selectedYear = yearSlider.value;
    sendFilters();
  });

  checkboxes.forEach((checkbox) => {
    checkbox.addEventListener('change', () => {
      checkboxState[checkbox.id] = checkbox.checked;
      sendFilters();
    });
  });

  function sendFilters() {
    drawBubblesWithLoading(data, selectedYear, contexts, bbox, config,
checkboxState);
  }
}
```

Este código define la función setupListeners, que tiene como objetivo agregar funcionalidades interactivas a los controles de la interfaz gráfica, permitiendo actualizar el gráfico de burbujas basado en un control deslizante (para cambiar el año) y una serie de casillas de verificación (para filtrar por tipo de impacto).

En la siguiente imagen podemos apreciar el gráfico dibujado en el navegador web.

Navegador



Árbol de adopciones

El árbol de adopciones es un caso específico que ejemplifica cómo la biblioteca VDA permite explorar el factor artístico en la representación de datos, incentivando la creatividad y la libertad visual. Con este ejemplo, se busca inspirar a los usuarios a descubrir nuevas formas de representar información, trascendiendo los enfoques tradicionales y explorando visualizaciones más expresivas y personalizadas.

Esta visualización muestra el número de adopciones registradas entre 2015 y 2023, e incorpora un control interactivo que permite seleccionar el año de visualización y elegir entre una vista acumulada o individual.

La inspiración detrás de este gráfico proviene de la obra Almendro en flor de Vincent Van Gogh. El árbol simboliza las conexiones familiares y el crecimiento a través de la adopción. Su estructura crece de forma exponencial, con ramas que se dividen sucesivamente en dos. Cada flor representa a un niño que se reintegra a una familia mediante la adopción. Para su diseño, se utilizaron pentágonos de Sierpinski, un fractal generado a partir de subdivisiones sucesivas de un pentágono, lo que da como resultado un patrón armónico y dinámico.

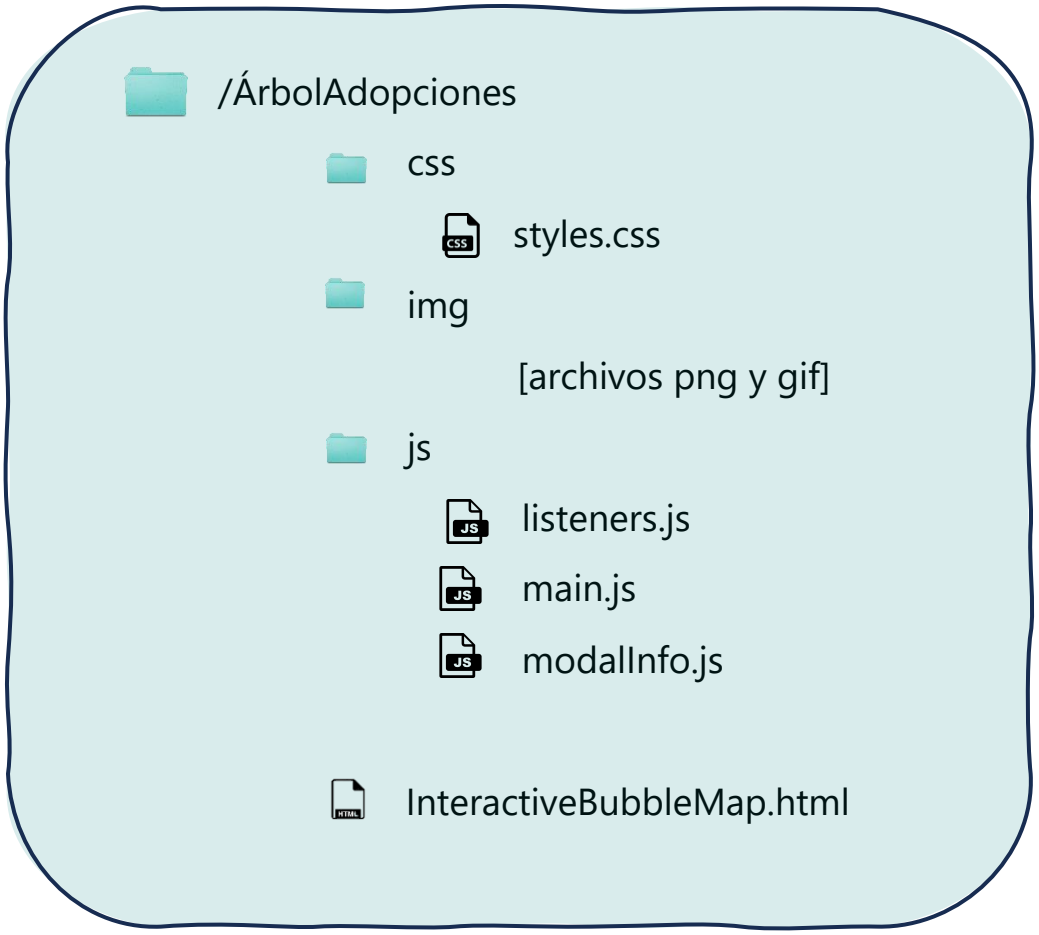
Además, el color juega un papel fundamental para distinguir visualmente a los niños y niñas adoptados, facilitando la interpretación de los datos de forma intuitiva.

La interacción con el árbol se realiza mediante diferentes controles:

1. Un control deslizante en la parte superior izquierda para seleccionar el año de visualización.
2. Un checkbox que permite alternar entre la vista acumulada de adopciones y la del año seleccionado.
3. Dos marcadores en la parte derecha que muestran el conteo total de adopciones.

Este proyecto resalta cómo la combinación de disciplinas puede generar experiencias visuales significativas mientras que el factor artístico mejora la narrativa. Los elementos artísticos como el árbol y las flores generan una conexión emocional con el espectador, de esta manera, el gráfico no solo comunica estadísticas, sino que también resalta las historias humanas detrás de los números.

La estructura del caso de uso es la siguiente:



Se explicará primeramente el archivo html y posteriormente las dependencias a los archivos java script.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Adopciones de niños en México</title>

    <!-- Enlace al archivo CSS para estilos -->
    <link rel="stylesheet" href="css/styles.css">

    <!-- Scripts Externos -->
      <script src =
      "https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.j
      s">  </script>
    <script src="https://d3js.org/d3.v7.min.js"></script>
</head>

<body>
    <!-- Contenedor del modal de descripción -->
    <div class="overlay" id="overlay">
        <!-- Botón para cerrar el modal -->
        <span class="closeModal" id="close">&times;</span>

        <!-- Descripción del propósito del gráfico -->
        <p>
            <span style="font-weight: bold;">Árbol familiar</span>
        </p>
        <p>El gráfico que verás a continuación representa un árbol familiar
      inspirado en la pintura de Vincent Van Gogh "Almendro en flor".</p>
        <p>La información presentada está relacionada con adopciones de niños
      y niñas realizadas en toda la República Mexicana.</p>
        <p>Puedes elegir el año con el control deslizante, además de decidir
      si deseas observar datos específicos de un año o datos acumulados hasta
      cierto año.</p>
```

index.html

```
  <p>En la esquina superior derecha se mostrará el número total de
adopciones de niños (color azul) y niñas (color rosa).</p>
          <p>Cada flor en el árbol simboliza la reintegración de un
niño o niña al árbol familiar: una flor rosa representa una niña
adoptada, mientras que una flor azul representa a un niño adoptado.</p>

<!-- Botón para cerrar el modal e ir al gráfico -->
    <p>
       <button class="closeModal" id="closeIntButton">
            Ver el árbol
       </button>
    </p>
</div>

<!-- Botón para abrir el modal de descripción -->
<div>
    <button id="openModal">Ver descripción</button>
</div>

<!-- Canvas para el árbol y las flores -->
    <canvas id="arbol"></canvas>
    <canvas id="flores" style="z-index: 10;"></canvas>

<!-- Contenedor del control deslizante -->
    <div class="slider-container">
    <!-- Control deslizante para seleccionar el año -->
        <input type="range" min="2014" max="2023" value="0"
        class="slider" id="year-slider" list="tickmarks">

    <!-- Etiquetas de los años disponibles -->
        <datalist id="tickmarks">
          <option value="0">---</option>
          <option value="2015">2015</option>
          <option value="2016">2016</option>
          <option value="2017">2017</option>
          <option value="2018">2018</option>
          <option value="2019">2019</option>
          <option value="2020">2020</option>
          <option value="2021">2021</option>
          <option value="2022">2022</option>
          <option value="2023">2023</option>
        </datalist>
    </div>
```

index.html

```
            <!-- Indicador del número de adopciones -->
            <div id="marcador">
                <!-- Número total de niños adoptados -->
                <div id="marca_ninios" class="circle">0</div>
                <!-- Número total de niñas adoptadas -->
                <div id="marca_ninias" class="circle">0</div>
            </div>

            <!-- Contenedor del checkbox para datos acumulados -->
            <div id="acummulativeCheckControl">
                <input type="checkbox" id="acum_check">
                  <label id="acum_check_text"
            for="acum_check">Acumulado</label>
            </div>

            <!-- Archivo JavaScript con la lógica del árbol y las
            interacciones -->
            <script type="module" src="js/main.js"></script>
            <script type="module" src="js/modalInfo.js"></script>
      </body>
</html>
```

Explicaremos a continuación cada sección.

1. **Enlaces a Archivos Externos:**

   Se incluye un archivo de estilo styles.css para aplicar estilos a la página.

   <link rel="stylesheet" href="css/styles.css">

2. **Bibliotecas Externas:**

   Se incluye PapaParse para analizar archivos CSV y D3.js: para crear gráficos interactivos, en este caso, el árbol de adopciones.

```
<script src=
"https://cdnjs.cloudflare.com/ajax/libs/PapaParse/5.3.0/papaparse.min.js">
</script>
<script src="https://d3js.org/d3.v7.min.js"></script>
```

3. **Modal (Ventana Emergente) de Descripción:**

   Se utiliza para mostrar información explicativa sobre el gráfico antes de que el usuario interactué con él. Este modal puede abrirse haciendo clic en el botón "Ver descripción". El botón "Ver el árbol" cierra el modal y permite al usuario ver el gráfico.

```
<div class="overlay" id="overlay">
  <span class="closeModal" id="close">&times;</span>
  <p>...</p>
  <button class="closeModal" id="closeIntButton">Ver el árbol</button>
</div>
```

4. **Canvas para el Árbol y las Flores:**

   - El Id #arbol se utiliza para dibujar el gráfico del árbol familiar, utilizando D3.js.
   - El Id #flores se utiliza para dibujar las flores, que representan las adopciones de niños y niñas. Se posiciona encima del primer canvas con un índice z (z-index: 10) para asegurarse de que este canvas se encuentre por encima del canvas para dibujar las ramas del árbol.

5. **Control Deslizante (Slider):**

   Permite al usuario seleccionar un año entre 2015 y 2023 para ver los datos de adopción correspondientes.

```
<input type="range" min="2014" max="2023" value="0" class="slider" id="year-slider" list="tickmarks">
```

6. **Marcadores para el Número de Adopciones:**

   Dos divs con clase circle se usan para mostrar el número total de niños y niñas adoptados. Estos números se actualizarán dinámicamente mediante JavaScript según la selección del año y la opción acumulada.

```
<div id="marcador">
   <div id="marca_ninios" class="circle">0</div>
   <div id="marca_ninias" class="circle">0</div>
</div>
```

7. **Checkbox para Datos Acumulados:**

   Permite al usuario optar por ver los datos acumulados (hasta el año seleccionado) o solo los datos de adopciones del año seleccionado. Este checkbox es controlado por JavaScript para actualizar la visualización.

```
<div id="acummulativeCheckControl">
   <input type="checkbox" id="acum_check">
   <label id="acum_check_text" for="acum_check">Acumulado</label>
</div>
```

8. **Scripts de Lógica:**

- Se incluyen dos archivos JavaScript:
    - **main.js:** Contiene la lógica para manejar la visualización del gráfico, la creación del árbol y las flores, y las interacciones del usuario con los controles.
    - **modalInfo.js:** Maneja las interacciones con el modal de instrucciones.

```
<script type="module" src="js/main.js"></script>
<script type="module" src="js/modalInfo.js"></script>
```

Ahora, se explicaran individualmente los archivos .js para entender todo el proceso a nivel técnico.

main.js

```
import { loadCSV} from '../../vda.js';
import { setupListeners } from './listeners.js';

// Hacer la variable global anexándola a window
window.animationIDs = [];

document.addEventListener('DOMContentLoaded', async () => {

  //Init treeParams

  //Carga de datos desde CSV
  const filePath  = "../../data/adopciones.csv";
  const data = await loadCSV(filePath);

  //Nombres de las columnas para el año y el género
  const yearColumnName = "year";
  const classColumnName = "gender";

  //Canvas para el árbol y canvas para las flores
  const treeCanvasId = "arbol";
  const flowersCanvasId = "flores";

  //Enviamos los filtros para crear el árbol
  setupListeners(data, yearColumnName, classColumnName, treeCanvasId,
flowersCanvasId);

});
```

1. Importación de Módulos

import { loadCSV } from '../../vda.js';

import { setupListeners } from './listeners.js';

- **loadCSV** se importa desde ../../vda.js, para la lectura y carda de datos.
- **setupListeners** se importa desde ./listeners.js para gestionar los eventos e interacciones del usuario con el gráfico.

2. animation Ids

Se define una variable global **animationIDs** dentro del objeto window para almacenar los identificadores de las animaciones y poder cancelarlas en caso necesario.

3. Carga de Datos desde CSV

- Se define la ruta del archivo CSV adopciones.csv, ubicado en ../../data/.

- loadCSV(filePath) se ejecuta de manera asíncrona (await), devolviendo una promesa con los datos cargados desde el CSV.

4. Asignación de canvas

- treeCanvasId = "arbol": canvas donde se dibujarán las ramas.
- flowersCanvasId = "flores": canvas donde se dibujarán las flores.

5. Envío de filtros

A través de listeners, se configuran los controles de usuario (slider y checkbox) para filtrar los datos en función del año y género.

listeners.js

```
import { createTree } from '../../vda.js';

export function setupListeners(data, yearColumnName, classColumnName,
treeCanvasId, flowersCanvasId) {
    const yearSlider = document.getElementById("year-slider");
    const accumulativeCheck = document.getElementById("acum_check");

    //Cada vez que cambien los filtros se enviarán para construir un nuevo
árbol
    yearSlider.addEventListener('input', sendFilters);
    accumulativeCheck.addEventListener('change', sendFilters);

    function sendFilters() {
      // Obtener el valor del año seleccionado
      const value = (yearSlider.value - yearSlider.min) / (yearSlider.max -
yearSlider.min) * 100;

      // Verificar si el checkbox "acumulativo" está marcado
      if (accumulativeCheck.checked) {
        // Si está marcado, poner el relleno con un gradiente basado en el
valor del slider
        yearSlider.style.background = `linear-gradient(to right, transparent
0%, cyan ${value}%, transparent ${value}%, transparent 100%)`;
      } else {
        // Si no está marcado, quitar el relleno (dejar fondo simple)
        yearSlider.style.background = 'none';
      }

      // Obtener el valor del año seleccionado y si es acumulativo
      const selectedYear = parseInt(yearSlider.value, 10);
      const isAccumulative = accumulativeCheck.checked;

      createTree(data, yearColumnName, classColumnName, selectedYear,
isAccumulative, treeCanvasId, flowersCanvasId);
    }
}
```

1. Función SetupListeners

SetupListeners es una función exportada que configura los eventos de interacción en la visualización y recibe los siguientes parámetros:

- data: Datos del archivo CSV.
- yearColumnName: Nombre de la columna que contiene los años en el archivo CSV.
- classColumnName: Nombre de la columna que contiene el género de los niños en el archivo CSV.
- treeCanvasId: ID del canvas donde se dibujarán las ramas.
- flowersCanvasId: ID del canvas donde se dibujarán las flores.

2. Obtención de filtros

Se obtienen los parámetros provenientes de los filtros seleccionados por el usuario.

```
const yearSlider = document.getElementById("year-slider");
const accumulativeCheck = document.getElementById("acum_check");
```

Cada vez que el usuario modifica los filtros, se hace una nueva llamada para pintar el árbol.

3. Llamada a createTree

Se llama a la función enviando los parámetros iniciales, e incluyendo también, los filtros obtenidos de la interaccipón del usuario.

Función createTree()

```
import { loadCSV} from '../../vda.js';
import { setupListeners } from './listeners.js';

// Hacer la variable global anexándola a window
window.animationIDs = [];

document.addEventListener('DOMContentLoaded', async () => {

  //Init treeParams

  //Carga de datos desde CSV
  const filePath  = "../../data/adopciones.csv";
  const data = await loadCSV(filePath);

  //Nombres de las columnas para el año y el género
  const yearColumnName = "year";
  const classColumnName = "gender";

  //Canvas para el árbol y canvas para las flores
  const treeCanvasId = "arbol";
  const flowersCanvasId = "flores";

  //Enviamos los filtros para crear el árbol
  setupListeners(data, yearColumnName, classColumnName, treeCanvasId,
flowersCanvasId);

});
```

1. Cancelación de animaciones previas

Con el fin de evitar superposiciones de animaciones, se detienen las existentes antes de dibujar un nuevo árbol.

```
stopAllAnimations(window.animationIDs);
```

2. Inicializa parámetros del árbol

InitTreeParams prepara los parámetros para dibujar el árbol como tamaños del canvas, profundidad del árbol, velocidad de animación, entre otros.

```
let treeParams = initTreeParams(treeCanvasId, flowersCanvasId);
```

3. Cálculo de ramas

Con el objetivo de simular el crecimiento natural de un árbol se generan dos ramas independientes, que posteriormente serán iterativamente bifurcadas, con ángulos aleatorios.

```
let arbol1 = calculateBranches(-Math.PI / 2 + Math.random() * Math.PI / 4, treeParams);

let arbol2 = calculateBranches(-Math.PI / 2 - Math.random() * Math.PI / 4, treeParams);
```

3. Filtrado de datos y cálculo de flores

Se filtran los datos de acuerdo a los parámetros elegidos por el usuario.

```
let filteredData;
if (isAccumulative) {
   filteredData = data.filter(d => d[yearColumnName] <=
parseInt(selectedYear));
} else {
   filteredData = data.filter(d => d[yearColumnName] ===
parseInt(selectedYear));
}
```

Posteriormente se calcula el número total de adopciones, dados los datos del archivo CSV, así como el total de niños y niñas adoptados.

3. Generación del árbol y actualización de marcadores

Se llama a la función drawTree() para renderizar el árbol visualmente utilizando las estructuras generadas. Los parámetros que recibe esta función son el cálculo de las posiciones de las ramas y los parámetros inicializados del árbol, respectivamente.

Más adelante se explicará esta función con mayor detalle.

```
drawTree(arbol1, treeParams);
drawTree(arbol2, treeParams);
```

Para optimizar el rendimiento y lograr que el árbol pareciera más frondoso, se optó por dibujar dos árboles de manera simultánea.

Finalmente, se actualiza el total de niños y niñas adoptados en los marcadores correspondientes.

```
actualizarMarcador(treeParams.numNinios, treeParams.numNinias);
```

Función drawTree()

```
export function drawTree(branches,treeParams) {

    // Nivel de profundidad actual que se está dibujando.
    var curDepth = 1;

    /**
     * Función para dibujar las ramas de la profundidad actual.
     * Al finalizar, incrementa la profundidad y continúa con la siguiente.
     */
    function drawNextDepth() {
        if (curDepth <= treeParams.maxDepth) {

            // Tiempo inicial de la animación para calcular el progreso.
            var startTime = performance.now();

            /**
             * Función que dibuja las ramas de la profundidad actual con una
animación progresiva.
             *
             * @param {number} timestamp - Tiempo actual proporcionado por
`requestAnimationFrame`.
             */
            function drawStep(timestamp) {

                // Calcular el progreso de la animación (entre 0 y 1).
                var progress = Math.min((timestamp - startTime) /
treeParams.animationSpeed, 1);

                // Estilo de las ramas (color marrón oscuro).
                treeParams.treeContext.strokeStyle = "#1F1916";

                // Dibujar cada rama de la profundidad actual de forma
progresiva.
                branches.forEach(branch => {
                    if (branch.depth === curDepth) {

                        // Calcular las coordenadas actuales del punto final
de la rama.
                        var currentEndX = branch.startX + progress *
(branch.endX - branch.startX);
                        var currentEndY = branch.startY + progress *
(branch.endY - branch.startY);
```

Función drawTree()

```
     // Dibujar la rama actual.
               treeParams.treeContext.beginPath();
               treeParams.treeContext.lineWidth = branch.width;
               treeParams.treeContext.moveTo(branch.startX, branch.startY);
               treeParams.treeContext.lineTo(currentEndX, currentEndY);
               treeParams.treeContext.stroke();
               }
              });

     // Si la animación no está completa, continuar con el siguiente cuadro.
                if (progress < 1) {
                    var id = requestAnimationFrame(drawStep);
                    window.animationIDs.push(id);
                } else {
                    // Si la animación de la profundidad actual terminó:
                    branches.forEach(branch => {
              // Dibujar flores en ramas de profundidad actual si se permite.
                         if (branch.depth === curDepth) {
                               if(curDepth >=2 && treeParams.flowerNumber <
                               treeParams.maxNumFlowers)
                               {
                               treeParams.treeContext.globalCompositeOperation
                               = "destination-under";

                              // Llamar a la drawFlower para dibujar la flor.
                              drawFlower(branch.endX,branch.endY,treeParams);
                             }
                         }
                    });
                    curDepth++; // Pasamos a la siguiente profundidad

                    // Llamar recursivamente para la siguiente profundidad.
                    drawNextDepth();
                }
            }

            // Animación de profundidad actual
            var id = requestAnimationFrame(timestamp => drawStep(timestamp));
            window.animationIDs.push(id);// Guardar el ID de la animación.
        }
    }
    // Iniciar la animación comenzando por la primera profundidad
    drawNextDepth();
}
```

La función drawTree se encarga de dibujar el árbol a partir de un conjunto de ramas previamente calculadas. La animación se realiza de manera progresiva, dibujando cada nivel de profundidad de las ramas antes de pasar al siguiente.

1. Inicialización de la animación

- Se define la variable curDepth para conocer el nivel actual de profundidad que se está dibujando.
- Se inicia la animación llamando a la función drawNextDepth(), que se encargará de dibujar cada nivel del árbol secuencialmente. Notesé que esta función es recursiva.

2. Dibujo progresivo de las ramas

Para cada nivel de profundidad (curDepth), se inicia una animación con requestAnimationFrame, que permite actualizar el lienzo de manera fluida, es decir que, cada rama se dibuja desde su punto de inicio y avanza gradualmente hasta el punto final.

3. Adición de flores

A partir de la segunda profundidad (curDepth >= 2), se dibujan flores en las ramas, siempre y cuando no se haya alcanzado el número máximo de flores (adopciones).

4. Control de animaciones

Cada llamada a requestAnimationFrame devuelve un ID de animación, que se almacena en window.animationIDs. Esto permite detener las animaciones en curso si es necesario.

5. Recursividad

Una vez que todas las ramas de un nivel han sido dibujadas, la función avanza al siguiente nivel (curDepth++) y vuelve a llamar a drawNextDepth(), repitiendo el proceso hasta alcanzar la máxima profundidad (treeParams.maxDepth).

Función drawFlower()

```
import { loadCSV} from '../../vda.js';
import { setupListeners } from './listeners.js';

// Hacer la variable global anexándola a window
window.animationIDs = [];

document.addEventListener('DOMContentLoaded', async () => {

  //Init treeParams

  //Carga de datos desde CSV
  const filePath  = "../../data/adopciones.csv";
  const data = await loadCSV(filePath);

  //Nombres de las columnas para el año y el género
  const yearColumnName = "year";
  const classColumnName = "gender";

  //Canvas para el árbol y canvas para las flores
  const treeCanvasId = "arbol";
  const flowersCanvasId = "flores";

  //Enviamos los filtros para crear el árbol
  setupListeners(data, yearColumnName, classColumnName, treeCanvasId,
flowersCanvasId);

});
```

La función drawFlower se encarga de dibujar una flor en una posición específica del árbol. Para lograr esto, genera un tamaño aleatorio y asigna un color basado en la proporción de "niños" y "niñas" representados en los datos. Finalmente, utiliza la función drawSierpinskiPentagon para dibujar la flor con una estructura fractal.

1. Generación del tamaño de la flor

Se define un tamaño aleatorio entre 10 y 60 unidades para dar variabilidad a las flores. Adicionalmente, se incrementa el contador de flores treeParams.flowerNumber para llevar un registro del número total de flores dibujadas.

2. Asignación del color de la flor

Se define un color por defecto (1, representando "niño") y se verifica si la flor es par o impar en la secuencia de dibujo, si es par, se intenta asignar el color "niña" (0), siempre que no se haya alcanzado el número máximo de niñas representadas; si es impar, se intenta asignar el color "niño" (1), siguiendo el mismo criterio.

3. Dibujo de la flor

- Se llama a la función drawSierpinskiPentagon para representar la flor como un pentágono fractal, con los siguientes parámetros:
    - X, Y: Coordenadas de la flor en el lienzo.
    - flowerSize: Tamaño de la flor.
    - 3: Nivel de profundidad para el pentágono.
    - flowerColor: Color asignado a la flor.
    - treeParams: Parámetros generales del árbol.

Función drawSierpinskiPentagon()

```
function drawSierpinskiPentagon(x, y, size, depth,color,treeParams) {

    // Caso base: si la profundidad es 0, no se dibuja nada y se detiene la
recursión.
    if (depth === 0) {
        return;
    }

    // Calcula los puntos (vértices) del pentágono basados en los valores de
seno y coseno.
    var points = [];
    for (var i = 0; i < 5; i++) {
        points.push({
            x: x + size * treeParams.cosValues[i],
            y: y + size * treeParams.sinValues[i]
        });
    }

    // Centro del pentágono (en este caso coincide con `x, y`).
    var centerX = x;
    var centerY = y;

    // Crea el gradiente de color azul desde el centro hacia los bordes
    var gradient = treeParams.flowersContext.createRadialGradient(centerX,
centerY, 0, centerX, centerY, size);

    if(color===1){
        gradient.addColorStop(0, 'white'); // Blanco
        gradient.addColorStop(0.9, '#164C8F');// Azul oscuro.
        gradient.addColorStop(0.9, '#164C8F'); // Azul oscuro.
        gradient.addColorStop(1, 'white'); //Blanco
    }
    else{
        gradient.addColorStop(0, 'white'); // Blanco.
        gradient.addColorStop(0.9, '#DE78A1'); // Rosa oscuro
        gradient.addColorStop(0.7, '#DE78A1'); //Rosa oscuro
        gradient.addColorStop(1, 'white'); // Blanco.
    }
```

Función drawSierpinskiPentagon()

```
 // Animación para dibujar los pentágonos de forma progresiva si la
profundidad es menor a 2.
    if (depth < 2) {
        // Progreso inicial de la animación.
        var progress = 0;

         //Función para animar el crecimiento progresivo del pentágono.
        function drawZoomStep() {
            // Incrementar el progreso.
            progress +=  0.026;

            // Calcular el tamaño actual basado en el progreso.
            var currentSize = size * progress;

            // Calcular nuevos puntos del pentágono basados en tamaño actual.
            var newPoints = [];
            for (var i = 0; i < 5; i++) {
                newPoints.push({
                    x: x + currentSize * treeParams.cosValues[i],
                    y: y + currentSize * treeParams.sinValues[i]
                });
            }

            // Dibujar el pentágono con el gradiente calculado.
            treeParams.flowersContext.beginPath();
            treeParams.flowersContext.fillStyle = gradient; // Establece
color de relleno
            treeParams.flowersContext.moveTo(points[0].x, points[0].y);
            for (var i = 0; i < 5; i++) {
                treeParams.flowersContext.lineTo(newPoints[i].x,
newPoints[i].y);
            }
            treeParams.flowersContext.closePath();
            treeParams.flowersContext.fill();

            // Continuar la animación si el progreso no ha alcanzado el 100%.
            if (progress < 1) {
                var id = requestAnimationFrame(drawZoomStep);
                window.animationIDs.push(id);
            }
        }
```

Función drawSierpinskiPentagon()

```
     // Iniciar la animación de zoom del pentágono.
        var id = requestAnimationFrame(drawZoomStep);
        window.animationIDs.push(id);

    }

    // Calcular los puntos centrales para los pentágonos internos (más
pequeños).
    var innerPoints = [];
    for (var i = 0; i < 5; i++) {
        innerPoints.push({
            x: x + (size / 2) * treeParams.cosValues[i],
            y: y + (size / 2) * treeParams.sinValues[i]
        });
    }

    // Llamada recursiva para dibujar pentágonos internos con tamaño
reducido.
    for (var i = 0; i < 5; i++) {
        drawSierpinskiPentagon(innerPoints[i].x, innerPoints[i].y, size / 3,
depth - 1, color, treeParams);
    }

}
```

La función drawSierpinskiPentagon es una implementación recursiva que dibuja un fractal en forma de pentágonos. Está inspirada en el patrón fractal de Sierpinski, pero adaptada para trabajar con pentágonos en lugar de triángulos, creando un efecto visual de crecimiento progresivo y el cual es utilizado para representar las flores del árbol.

1. Caso base de la recursión:

Cuando la profundidad llega a 0, la función deja de dibujar y se detiene, evitando más llamadas recursivas.

2. Cálculo de los vértices del pentágono:

La función calcula los puntos que forman un pentágono regular en función de su centro (x, y) y su tamaño (size). Los puntos se calculan utilizando las funciones trigonométricas seno y coseno basadas en los ángulos de un pentágono regular.

3. Creación de gradiente de color:

Se creó un gradiente radial que va del centro del pentágono hasta sus bordes con el fin de darle profundidad y movilidad a las flores.

4. Animación de crecimiento:

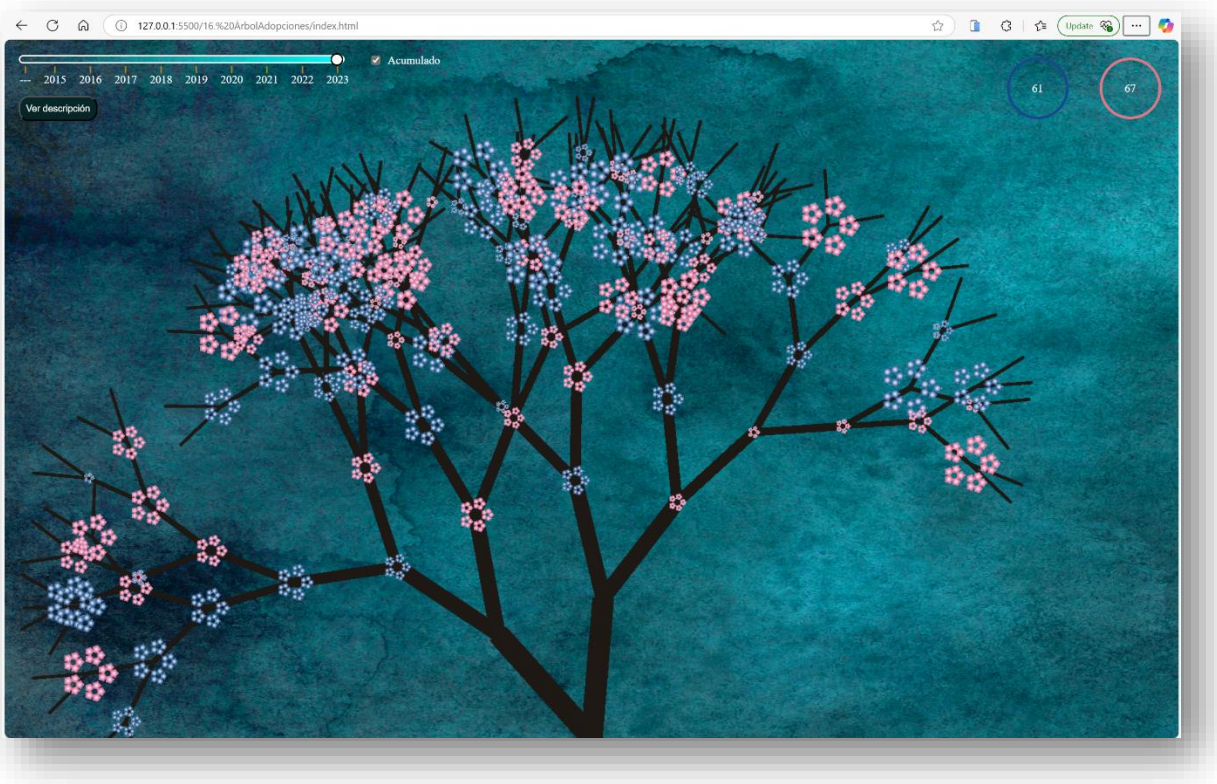
Si la profundidad es menor a 2, se inicia una animación en la que el pentágono crece progresivamente desde el centro hacia afuera. Esto se logra mediante requestAnimationFrame para crear un efecto visual fluido del crecimiento. El tamaño del pentágono aumenta a medida que avanza la animación, hasta alcanzar su tamaño final.

5. Recursividad para los pentágonos internos:

Posteriormente, la función calcula los puntos de los pentágonos internos (siguiente nivel) ubicados dentro de los pentágonos actuales (nivel actual). Luego, dd manera recursiva se dibujan más pentágonos dentro de estos nuevos puntos, reduciendo el tamaño de cada uno de ellos en cada llamada a la función recursiva.

En la siguiente imagen podemos apreciar el árbol dibujado en el navegador web.

Navegador



Referencias

Repositorio GitHub. El código para implementar los casos de uso, así como los archivos de datos csv y geoJson utilizados en esta guía, pueden obtenerse directamente del siguiente repositorio.

VDA

Librería para Visualización de Datos

García Aguilar Luis Alberto