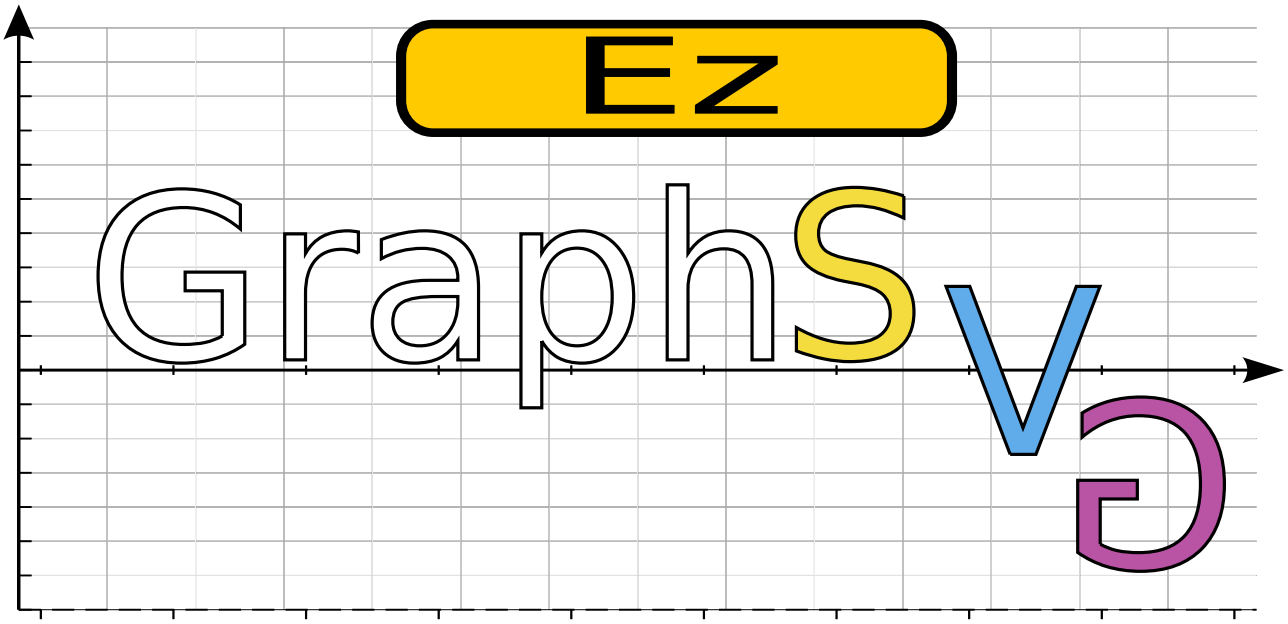
Biblioteca de objetos para la generación dinámica de gráficos de información para la Web

Omar Ochoa Román



http://ezgraphsvg.objectis.net

Agradecimientos

A veces la vida se nos presenta convertida en un jardín de rosas y alegrada por un tibio sol, en sonrisas y gestos de disgusto de enemigos nobles, en un fuego que consume nuestro corazón. Sin embargo, las escenas de esa clase no duran para siempre, sin duda porque los sentimientos cambian y porque todo tiene su límite. Una vez alcanzado el paroxismo sobreviene una dulce calma sin transición, una calma que roza el embrutecimiento, de la misma manera que la excitación roza la locura y después se aleja.

Diríase entonces que nos arrepentimos del frenesí, de las lágrimas, de las palabras que hemos pronunciado, como si el hombre no estuviera hecho para los sentimientos. No obstante, la vida continúa incluso cuando has de enfrentarte a un suplicio que parece insoportable, un suplicio que exprime tu corazón hasta sacarle la última gota de sangre y que deja su trazado zigzagueante. Sí, la vida sigue y el dolor empieza a perder intensidad poco a poco, y luego empieza a desvanecerse.

Pero si la vida pierde su sentido, entonces las metas ya no tienen razón de ser, el final de un día parece nunca ocurrir y tus sueños se desvanecen; por eso quiero agradecer a Ale por estar en mi vida y por hacerme soñar de nuevo.

Gracias papá por tus consejos, por tu ayuda que me has brindado, por creer en mí y por esperar a que llegara este día.

Gracias Jonathan por tus consejos, por estar en el momento preciso cuando era necesario y por esa buena infancia que pasamos juntos.

Gracias Emmanuel por tu ayuda imparcial y sobre todo por ser mi amigo, y también gracias a tu familia por abrirme las puertas de su hogar.

Gracias Ricardo por brindarme todas esas horas de paciencia en explicarme matemáticas.

Gracias María del Carmen por tus sugerencias y contribuciones para mejorar esta tesis.

Gracias a Ernesto por tus sugerencias y por tu valiosa ayuda en Ingeniería de Software.

Gracias y una disculpa para quienes no mencione y que de algún modo u otro influyeron en mi formación académica.

Gracias a todos aquellos que han contribuido al mundo Linux/OpenSource, sin ustedes seguramente esta tesis estaría escrita en Word.

Finalmente, y más importante, quiero agradecer a mi mamá por inculcarme valores, por apoyarme y creer en todo lo que hacía. Gracias por tu ayuda invaluable porque sin tí jamás hubiera llegado hasta aquí y sobre todo gracias por darme la vida.

Índice

Prefacio

Antes de empezar a desarrollar mi tema de tesis, lo único de lo que estaba seguro era de que quería hacer algo que estuviera relacionado con graficación por computadora.

Tomando como experiencia mi servicio social, donde los sistemas que se desarrollaban estaban enfocados para utilizarse vía Web, me vi obligado a aprender HTML, PHP, JavaScript, y, por curiosidad, DOM. Los reportes en línea estaban hechos en tablas que contenían sólo texto y números.

Esto me condujo, primeramente, a querer desarrollar objetos que generaran gráficos de información utilizando PHP y GD (biblioteca de gráficos que permite crear y manipular .jpgs, .pngs, y .wbmps desarrollada por Boutell). Pero seguía habiendo un problema: escoger entre calidad o velocidad de transmisión.

Sin embargo, en una ocasión que estaba haciendo un dibujo en un programa llamado Sketch vi que la opción de exportar traía varios formatos, conocía todos excepto uno: SVG.

Empecé a investigar sobre SVG y al enterarme de sus características, aunado a que ya sabía utilizar JavaScript y el DOM, se me ocurrió que sería buena idea desarrollar una biblioteca de objetos en este formato, no sólo porque seguirían siendo gráficos para la Web superando el problema de la calidad y la transmisión, sino porque me estaría adelantando a lo que se supone que será el futuro de los gráficos Web.

Introducción

Actualmente el desarrollo Web no sólo consiste de diseño HTML, sino también de programación. Ya pasó la época en la que una página Web se componía sólo de algunas imágenes estáticas e información de texto. Lo que ahora distingue a una página Web de otra, es el nivel de interacción con el usuario, su dinamismo y, por supuesto, un atractivo diseño que capte la atención.

Los gráficos en línea han contribuido a popularizar la Web. Además, un recurso muy común en sitios Web que muestran estadísticas y otro tipo de información, es el uso de reportes en línea. Los reportes hechos en texto proporcionan toda la información necesaria, ¿pero no sería mejor si ese reporte tuviera su respectivo gráfico a color?

Por otro lado, la mayoría de los gráficos que hay en la Web hoy en día están en un formato de *mapas de píxeles*. Aunque los gráficos de mapas de píxeles son inmejorables para tratar imágenes escaneadas, como fotografías, y pueden mejorar mucho el diseño de una página Web, ocupan mucho espacio de almacenamiento y pueden volverla pesada; como consecuencia su transmisión en Internet es lenta y el tiempo de descarga puede ser considerable, en especial si se *accesa a Internet vía módem*[1], lo que podría causar que los usuarios se desesperen y abandonen la página, por lo que los gráficos, cuanto menos espacio de almacenamiento ocupen, mejor. También hay que considerar que todo mundo puede ver los gráficos en diferentes plataformas, desde estaciones de trabajo Sun hasta Palm Pilots, pero el color se muestra de forma diferente en estas plataformas. Además es difícil obtener resultados de calidad a la hora de transformar un mapa de píxeles, porque éstos no mantienen ninguna información acerca de la estructura de una imagen, que no sea la información sobre los píxeles.

El problema empeora cuando se requieren *gráficos dinámicos*. Ahora no sólo está presente el problema de la transmisión, sino también el uso de un lenguaje de programación, así como el requerimiento de bibliotecas gráficas para la creación de los mismos.

La presente tesis trata sobre la generación dinámica de gráficos de información para la Web, que sean de alta calidad, eficaces y de un peso ligero. Para ésto, se desarrolló una biblioteca de objetos (EzGraphSVG) con JavaScript y la API DOM para producir gráficos dinámicos en formato SVG.

Organización de la tesis:

- Capítulo 1: Antecedentes de los Gráficos, el DOM y JavaScript. Proporciona información sobre los diferentes tipos de gráficos para la Web, DOM y JavaScript, así como las capacidades de estas dos tecnologías.
- Capítulo 2: Programación Orientada a Objetos en JavaScript. Trata este paradigma de programación orientado a objetos aplicado a JavaScript, así como técnicas de programación ejemplificando los diferentes tópicos (constructores, propiedades, métodos, herencia y polimorfismo).

[1] Los *módems* actuales utilizan tecnología 56K, sin embargo, ni en las mejores condiciones se alcanza esta conexión. La conexión telefónica normal sólo puede ofrecer conexiones entre 42K y 46K, debido al ruido de la línea o interferencias.

- Capítulo 3: Fundamentos de SVG. Describe la sintaxis, estructura, sistema de coordenadas, elementos, figuras básicas y complejas, transformaciones, y el DOM de SVG.
- Capítulo 4: Biblioteca de objetos gráficos. Presenta las características de las gráficas que se programaron (barras, línea, área, círculo), la documentación del análisis y diseño de la biblioteca de objetos EzGraphSVG, así como pruebas de rendimiento de las las gráficas que genera.
- Conclusiones. Muestra los factores de calidad con los que cumple EzGraphSVG.
- Apéndice A: Técnica para convertir coordenadas cartesianas a SVG. Este capítulo ilustra una técnica para convertir coordenadas cartesianas a coordenadas SVG, la cual resulta muy porque en clase de geometría el sistema de coordenadas que se utiliza es el cartesiano.
- Apéndice B: Datos de las pruebas de rendimiento. Contiene las tablas de los datos que se utilizaron para las pruebas de rendimiento.
- Apéndice C: Código de la aplicación. Muestra todo el contenido que tiene CD que acompaña a la tesis así como instrucciones de cómo instalar el software necesario para correr la aplicación.
- Apéndice D: Galería de capturas. Tiene ilustraciones de EzGraphSVG en acción, corriendo en diferentes navegadores Web en Linux y Windows.
- Apéndice E: Programación SVG del lado del servidor. Este capítulo enseña como configurar los tipos MIME para que un servidor entregue archivos SVG. También enseña las instrucciones que necesitan los lenguajes más populares del lado del servidor para crear contenido dinámico en formato SVG. Y por último, tiene una aplicación de encuestas que utiliza PHP, MySQL y EzGraphSVG.
- Apéndice F: Glosario de términos. Contiene una lista de los términos más utilizados en el ámbito de graficación concernientes a esta tesis.
- Bibliografía. Lista toda la bibliografía consultada para la elaboración de esta tesis.

Capítulo 1

Antecedentes de los Gráficos, el DOM y JavaScript

Internet, la red de redes, se originó en 1969[2, p. 199] a partir del proyecto militar ARPANet, el cual fue desarrollado por varios científicos del Departamento de Defensa de los Estados Unidos. ARPANet era una pequeña red de computadoras que permitía transferir información secreta mediante *paquetes conmutados*. Sin embargo, sólo la comunidad científica utilizaba Internet, a pesar de los avances tecnológicos en el área de telecomunicaciones durante la década de los 70's y principios de los 80's.

Con la creación de la *Web* (WWW, World Wide Web) en 1989, Internet empieza a tomar fuerza. Fue desarrollado por un grupo de investigadores bajo la dirección de Tim Berners-Lee en el CERN (Laboratorio Europeo de Física de Partículas). Ellos definieron los conceptos *HTML*, *URL* y *HTTP* que son las herramientas base para construir, localizar y accesar a las *páginas del Web* en cualquier nodo o red conectados a Internet. Antes del Web, los investigadores de las universidades utilizaban Internet con aplicaciones de comunicaciones *(e-mail*, *Telnet*, *Finger*, etc) que se ejecutaban en una *terminal* en modo texto, mediante el protocolo *TCP/IP*. El problema de este tipo de aplicaciones se presentaba cuando las investigaciones contenían archivos de gráficos o vídeo y deseaban consultarlos.

Viola fue el más joven líder en el campo de la tecnología de navegadores Web, ofreciendo los primeros vistazos a gráficos y un sistema de hipertexto basado en ratón, que fue uno de los propósitos originales del Web, pero fue *Mosaic* quien, con su intuitivo y amigable entorno gráfico, popularizó enormemente el acceso al Web. Mosaic fue una de las innovaciones que provocó el desarrollo Web moderno, aunque ésto sólo fue el primer paso hacia la soñada Web multimedia.

1.1. Gráficos

Se dice que una imagen vale más que mil palabras. Aunque esta frase es muy trillada, es cierta. Los gráficos parecen estar sintonizados con la manera como piensa la gente, es decir, los gráficos proporcionan uno de los medios más naturales de comunicación, ya que nuestras habilidades altamente desarrolladas en el reconocimiento de patrones 2D y 3D nos permiten percibir y procesar datos pictóricos de forma rápida y eficiente. Los ingenieros y los científicos siempre han aprovechado el valor de las ilustraciones al expresar los resultados de sus diseños y cálculos en forma gráfica. [3, p. 1]

Probablemente el uso más común de los gráficos es la creación de gráficos 2D y 3D de funciones matemáticas, físicas y económicas; gráficos estadísticos; gráficos de actividades; inventarios y gráficos de producción. Todos estos gráficos son usados para representar de forma significativa y consistente las tendencias y los patrones de datos, así como para clarificar complejos fenómenos y para facilitar la toma de decisiones.

1.1.1. Gráficos por computadora

La graficación por computadora *es la creación, almacenamiento y manipulación de modelos e imágenes de objetos por medio de la computadora*[6, p. 1]. Estos modelos vienen de diversos campos, tales como la física, matemáticas, ingeniería, fenómenos naturales, o simplemente de la imaginación. El campo de los gráficos por computadora es una mezcla de técnicas de arte, programación, geometría y creatividad de invención. Esta disciplina continuamente está cambiando y creciendo gracias a las aportaciones de universidades, estudios de arte, laboratorios de investigación y estudios de animación.

En la actualidad, los gráficos por computadora permiten la interacción con el usuario de tal manera que él controla el contenido, la estructura y la apariencia de los objetos por medio de dispositivos, como el teclado y el ratón, entre otros. Sin embargo, el usuario final desconoce, por lo general, todo el proceso que se encuentra detrás de la creación de las imágenes que observa. Aún la gente que no usa computadoras en su trabajo, encuentra gráficos por computadora en comerciales de televisión y en efectos especiales cinematográficos. Los gráficos por computadora son parte integral de todas las interfaces de usuario de computadora, y es indispensable en la visualización bidimensional (2D) y tridimensional (3D). Áreas tan diversas como la educación, ciencia, ingeniería, arquitectura, diseño asistido por computadora, cartografía, medicina, comercio, milicia, publicidad, entretenimiento, negocios, tecnología, simulación, animación, arte, y, por supuesto, la Web dependen de los gráficos por computadora. [11, p. 1-9][3, p. 1-12]

1.1.2. Gráficos Web

Los gráficos utilizados en la Web se dividen en dos tipos: *mapas de píxeles*[1] y *vectoriales.* Dentro de estas dos clasificaciones, los mapas de píxeles han sido los más utilizados. Los formatos GIF y JPEG son los más populares en el ámbito Web. En gran medida esto se debe a que en los inicios del Web las transmisiones vía módem eran demasiado lentas, y de entre todos los formatos, éstos eran los que ocupaban menos espacio de almacenamiento.

1.1.2.1. Mapas de píxeles

Los *mapas de píxeles* se caracterizan porque están formados por pequeñas celdas. Estas celdas individuales son llamadas *píxeles*. Este tipo de gráficos son almacenados como un arreglo rectangular de valores numéricos. Cada valor numérico representa el valor del píxel almacenado ahí y cada píxel puede ser manipulado por separado[11, pp. 83].

Mapas de píxeles más utilizados en la Web:

- **Graphics Interchange Format (GIF).** El *Formato de Intercambio de Gráficos* es el más idóneo para las imágenes generadas por computadora con áreas de color sólido, debido a su esquema de *compresión sin pérdidas* y admite *color de 8 bits*[2]. Las imágenes GIF admiten *entrelazado*[3]. La ventaja de las imágenes GIF es que son las que más se utilizan y admiten en el Web, además de incluir características avanzadas como *canal alfa*[4] y la animación. El principal inconveniente de este formato es que *es marca registrada de CompuServe, con* ***compresión LZW*** *patentado por Unisys*, lo que implica que es necesario pagar si se desarrollan aplicaciones que utilicen este formato.[15, p. 83][8, p. 82][28, p. 499]

[1] A menudo llamados *mapas de bits,* pero este término, estrictamente hablando, sólo aplica a gráficos que tienen un bit por píxel; para gráficos que tienen múltiples bits por píxel, es más general el término *mapas de píxeles[11, pp. 13].*

[2] Véase *profundidad de color* en la página 102

[3] Véase *entrelazado* en la página 101

[4] Véase *canal alfa* en la página 101

- **Joint Photographic Experts Group (JPEG).** Este *Grupo Asociado de Expertos en Fotografía* es otro formato perfectamente adaptado a la Web. Fue diseñado para comprimir imágenes fotográficas con miles, o millones, de colores o *escala de grises*[5], sin embargo, debido a que es *compresión con pérdidas*, una alta compresión podría reducir severamente la calidad de la imagen. El formato JPEG almacena imágenes de todas las *profundidades de color* ocupando poco espacio de almacenamiento, con lo que se logra un ahorro tanto de espacio como en tiempo de descarga.[15, p. 84][8, p. 81][28, p. 499]

- **Portable Network Graphics (PNG).** El formato *Gráfico de Red Portable* es el candidato a reemplazar al formato GIF y así proporcionar una solución a las cuestiones legales. Imágenes de *color indexado*[6], *escala de grises*, y *color verdadero*[7] son soportados, más un *canal alfa* opcional. PNG soporta *compresión sin pérdida*. [15, p. 84][8, p. 81-82][28, p. 499]

1.1.2.2. Vectoriales

Los *gráficos vectoriales* se describen matemáticamente utilizando elementos geométricos, como líneas, polígonos y elipses. A diferencia de los gráficos de mapas de píxeles, los gráficos vectoriales se pueden manipular (rotar, escalar, trasladar) sin perder calidad y ocupan menos espacio de almacenamiento debido a su naturaleza matemática[19, p. 701].

Gráficos vectoriales para la Web:

- **Flash** es el formato para las animaciones sofisticadas basadas en la Web. La tecnología Flash corresponde a gráficos, texto, animación y aplicaciones para sitios Web. Aunque están compuestas principalmente por gráficos vectoriales, también pueden incluir vídeo, mapas de píxeles y sonidos importados. Las animaciones Flash pueden incorporar interactividad para permitir la introducción de datos de los espectadores y la creación de películas no lineales que pueden interactuar con otras aplicaciones Web. Los diseñadores Web utilizan Flash para crear controles de navegación, logotipos animados, animaciones con sonido sincronizado e incluso sitios Web con capacidad sensorial. Además, y a pesar de las numerosas ventajas que cada versión de Flash incorpora, ésta es una solución externa a lo que es el lenguaje de creación de páginas Web, el HTML.[21, p. 289][19, p.701-702]

- **VML** (Vector Markup Language, Lenguaje de Marcado de Vectores). Es un vocabulario XML, desarrollado principalmente por Microsoft, con participación de AutoDesk, Hewlett-Packard, Macromedia y Visio. VML fue sometido al W3C (World Wide Web Consortium) como especificación en mayo de 1998. Aunque la especificación lleva algún tiempo, VML no ha recibido una aceptación generalizada en la Web. De hecho, Internet Explorer 5 es el primer navegador que ofrece soporte para VML. Sin embargo, esto es lo mejor de VML, ya que, actualmente, ningún otro vocabulario para gráficos de vectores está soportado directamente a nivel de navegador. [21, p. 706][19, p.703-704]

- **SVG** (Scalable Vector Graphics, Gráficos Vectoriales Escalables). Esta basado en XML, de modo que es un lenguaje extensible y el código fuente no sólo queda a la vista, sino que también es editable y, aún mejor, las *etiquetas* están a menudo en inglés; por ejemplo, un círculo está descrito en código como <circle/>. Aunque no haya soporte nativo para SVG en ningún navegador Web a la fecha, existen visores programados en Java, y Adobe ha desarrollado un plug-in gratuito que actualmente se encuentra en su versión 3.01 y que proporciona el mejor soporte para SVG. La tecnología SVG

[5] Véase *escala de grises* en la página 101
[6] Véase *color indexado en la página 102*
[7] Representa más de 16 millones de colores

está planeada para superar muchas de las limitaciones de los gráficos Web tradicionales soportando búsqueda de gráficos, usando hojas de estilo, ocupando mucho menos espacio de almacenamiento, a comparación de un mapa de píxeles de alta calidad, para mayor reducción del tamaño de los archivos SVG soporta compresión *GZIP*, un estándar en la industria para la compresión de archivos; además SVG faculta al diseñador para incrustar mapas de píxeles, superponiendo vectores que incluso puede animar a voluntad o como respuesta a las acciones del usuario. SVG incluye soporte para *pan*[8], *zoom*[9], *anti-aliasing*[10], diseños flexibles, tipografía, transparencias, gradientes suavizados y colores exactos. SVG se integra con las tecnologías Web existentes, como CSS2[11], y es completamente compatible con DOM[12], que abre muchas oportunidades interesantes a la automatización. Una faceta particular de SVG es el soporte que da al contenido dinámico. [19, p.704-705]

1.1.3. Gráficos dinámicos

Los *gráficos dinámicos* son aquellos gráficos que se generan *al vuelo*[13], típicamente combinando datos cuantitativos con imágenes gráficas. Un buen ejemplo de gráfico dinámico es una gráfica de barras que es desplegado en pantalla cuando se carga, basado en datos numéricos que siempre cambian, como las encuestas. Los gráficos dinámicos son diferentes de los gráficos estándar en que no son almacenados en archivos estáticos, tales como los archivos *gif* o *jpeg*. En su lugar, existen como una pareja de contenido gráfico y datos que se unen para formar una imagen final. Esta imagen puede ser cambiada modificando los datos y actualizando la imagen.[16, p. 308]

Estos gráficos son generalmente creados como plantillas, con áreas específicas definidas para completarse de una fuente de datos externa. Observando una plantilla gráfica, se podría tener algún tipo de dificultad visualizando como podría lucir la imagen final, simplemente porque la imagen no está completa. Esto es porque la imagen generalmente tendrá huecos en lugares donde el contenido debería residir en su forma final. Los datos que alimentan a la imagen dinámica podrían lucir insulsos, porque no han sido estilizados aún. Sí los datos para la imagen dinámica residen en una base de datos, puede que no sean visibles en absoluto.[16, p. 308]

Para determinar la necesidad de gráficos dinámicos, hay que considerar que tan a menudo se republican gráficos estáticos similares. La mayoría de los gráficos estadísticos y financieros lucen muy similar, sin embargo son republicados semanalmente (sino es que diario) para continuar graficando el progreso o rechazar cierta información. Como tal, alguien debe recrear el mismo gráfico cada día modificando ciertos puntos o figuras para reflejar algún cambio.[16, p. 308]

Ese trabajo repetitivo, el cual suele ser realizado por un diseñador gráfico, se convierte en tedioso y es un medio ineficiente de publicar información. En su lugar, creando una plantilla maestra y una serie de reglas, estas imágenes pueden ser recreadas con una simple actualización de datos (sea texto, imagen, o algún otro).[16, p. 308]

Crear gráficos por medio de un lenguaje de programación puede resultar, en un principio, más complicado y tedioso que servirse de otras herramientas gráficas mucho más visuales y potentes presentes en la totalidad de las aplicaciones de diseño gráfico. No todo es inconveniente y entre las ventajas del uso de un lenguaje para la creación de gráficos dinámicos destacan:

[8] Véase *pan* en la página 102
[9] Véase *zoom* en la página 102
[10] Véase *anti-aliasing* en la página 102
[11] Véase *CSS* en la página 101
[12] Véase la sección 1.2 en la página 7
[13] Se generan mientras se despliegan en pantalla y no previamente

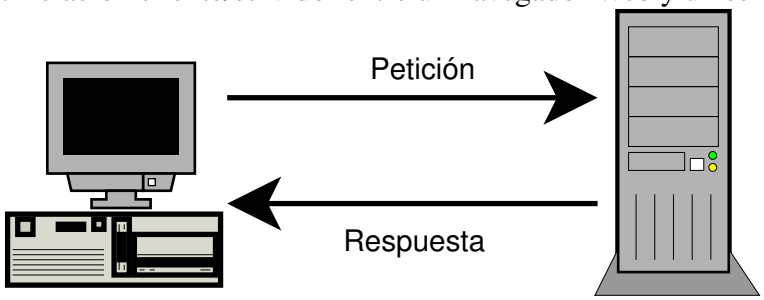
- Creación de botones dinámicos
- Tratamiento automatizado de imágenes recibidas de los usuarios
- Actualización y personalización más flexible
- Ahorro de memoria

1.1.3.1. Gráficos Web dinámicos

El funcionamiento básico de un servidor Web se muestra en la Figura 1.1. Este sistema consiste de dos partes: un *navegador Web* y un *servidor Web*. El contenido del sitio Web está estructurado como una serie de páginas Web predefinidas almacenadas en archivos. El servidor Web aceptaba solicitudes de los navegadores de usuarios (en la forma de mensajes HTTP), localizan la página o páginas solicitadas y las envía al navegador para su visualización, de nuevo utilizando HTTP . Los contenidos de la página Web se expresan, usualmente, en HTML. El código HTML para una página dada contiene texto y gráficos a mostrar en la página, así como los vínculos que albergaban la navegación desde esta página a otras. Esta arquitectura se adapta a un servidor que entrega páginas estáticas.[27, p. 178]

Con esta arquitectura, los gráficos, y toda la página Web en general, son estáticos lo que deja mucho que desear para las aplicaciones Web modernas.

Figura 1.1: Relación cliente/servidor entre un navegador Web y un servidor Web



No obstante, los gráficos Web pueden ser generados dinámicamente al tiempo que una página o gráfico es servido a un cliente y una característica muy interesante de los **gráficos Web dinámicos** es que se pueden enriquecer gracias a la arquitectura de los *sistemas manejadores de datos.*

Sistemas manejadores de datos (Data-driven systems)

Los *sistemas manejadores de datos* son un diseño arquitectónico caracterizado por la separación de datos y código. Consisten en un núcleo de código que se puede comportar de maneras diferentes basado en los datos utilizados para configurarlo. Los sistemas manejadores de datos requieren de una *fuente de datos* de donde recuperarlos, y tienen una arquitectura de ejecución para aplicarlos al sistema mientras los ejecuta. A menudo, también ofrecen un *depósito de datos* que rastrean los datos en el sistema y permiten acceso a él. [23, p. 55]

Las ***fuentes de datos*** que pueden ser utilizados para almacenar los datos son:

- **Archivos de texto.** Los archivos de texto pueden ser utilizados como fuente de datos, pero son un medio primitivo que carece de estructura y herramientas. Los archivos de texto son, sin embargo, muy simples y pueden ser apropiados para pequeños conjuntos de datos. [23, p. 57]

- **XML.** Es un lenguaje estándar para almacenar datos de una forma estructurada y de plataforma independiente. XML es, sin embargo, verboso, y aunque existen herramientas y bibliotecas disponibles para analizarlo, puede tomar mucho tiempo analizar grandes cantidades de datos XML. [23, p. 57]
- **Bases de datos relacionales.** Son las fuentes de datos más importantes en el mundo del almacenamiento. Proporcionan capacidades únicas para almacenar y recuperar grandes cantidades de datos, y tienen motores de ejecución robustos y escalables. Las bases de datos relacionales introducen, sin embargo, una carga elevada de administración e integración. [23, p. 58]
- **Lenguajes de scripts.** Los scripts en lenguajes como PHP, JSP y Python, pueden ser utilizados como fuentes de datos. Los scripts tienen la ventaja de que ya tienen datos estructurados y representan, relativamente, datos complejos, formas compactas y concisas que pueden ser cargados sólo con ejecutarlos. [23, p. 58]

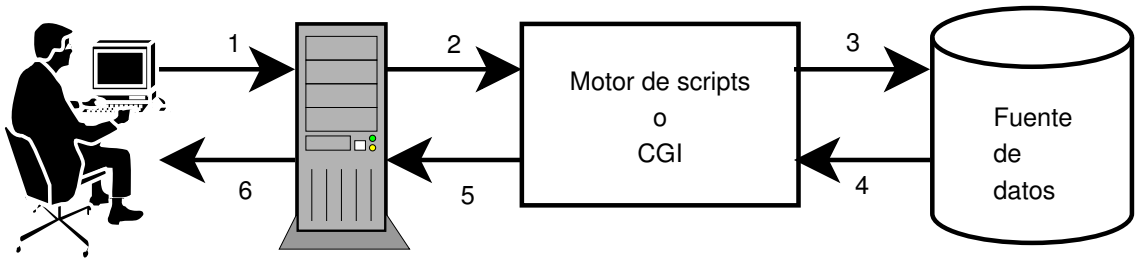
Gráficos manejadores de datos (Data-driven graphics)

Los gráficos cuando son generados dinámicamente mediante fuentes de datos externas, tales como XML o en una RDBMS para proporcionar toda o parte de la información que será desplegado en el gráfico, se llaman ***gráficos manejadores de datos***. El uso de gráficos manejadores de datos proporciona una multiplicidad de posibles aplicaciones, como mostrar el clima o la temperatura local en tiempo real, un reloj analógico que se actualiza a cada segundo, mostrar las encuestas actualizadas de algún evento, etc. así, mientras los datos cambian, también los gráficos, sin trabajo adicional por parte del desarrollador, lo que hacen los gráficos más atractivos para el usuario en general.

Gráficos manejadores de datos basados en Web

Los gráficos manejadores de datos basados en Web consisten en una arquitectura de tres *capas*[14], como se muestra en la Figura 1.2. A continuación se examinan las diferentes etapas:

Figura 1.2: Gráficos manejadores de datos basado en Web



1. El navegador Web de un cliente emite una solicitud HTTP para una página particular Web.
2. El servidor Web recibe la solicitud para la página solicitada, recupera el archivo y lo pasa al motor de scripts o *CGI*[15] para procesarlo.
3. El motor de scripts o CGI empieza a analizar el script. Dentro del código debe haber instrucciones para solicitar los datos de la fuente de datos.

[14] Una *capa* es un concepto abstracto que define un grupo de tecnologías que proporcionan uno o más servicios a sus clientes.

[15] CGI (Common Gateway Interfaz, Puerta de Enlace Común) es un estándar para efectuar la interfaz de programas externos, con un servidor HTTP. Permite usar estos programas para dar formato y procesar salidas a los navegadores Web. [29, p. 342-343]

4. La fuente de datos procesa la petición, recupera los datos y los regresa al motor de scripts o CGI.

5. El motor de scripts o CGI construye el gráfico (usualmente anexado a una página Web) en base a los datos solicitados y regresa el gráfico resultante al servidor Web.

6. El servidor Web regresa al navegador Web la página solicitada (que incluye el gráfico), donde el usuario puede ver finalmente el resultado.

El proceso es básicamente el mismo sin importar el motor de scripts o CGI o fuente de datos que se utilice. A menudo el servidor Web, motor de scripts o CGI, y la fuente de datos corren todos en la misma máquina. Sin embargo, es muy común que la fuente de datos esté en otra máquina por cuestiones de seguridad y desempeño.

1.2. DOM (Document Object Model, Modelo de Objetos de Documento)

Un *modelo de objetos* define la interfaz utilizada por lenguajes de programación para examinar y manipular información estructurada. Un modelo de objetos también define la composición y las características de las partes que lo componen, así como la forma en que se puede actuar sobre ellas.[22, p. 838]

Antes de que el DOM se convirtiera en una especificación oficial, la comunidad Web había empezado a desarrollar modos de crear páginas basadas en programas de *scripts* que pudieran ser usados entre los diferentes navegadores Web. Este primer trabajo condujo al desarrollo de lo que fue llamado *HTML Dinámico* (DHTML), el cual fue utilizado principalmente para cosas como interfaces de usuario más ricas en contenido que lo que HTML podía ofrecer por sí mismo. [18, p. 8]

Las aplicaciones Web datan desde la liberación de Netscape Navigator 2.0 en 1995, el primer navegador Web en tener un *lenguaje de scripts* incrustado en él. Ese lenguaje era *JavaScript*[16] y permitió a los desarrolladores por primera vez manipular sus páginas Web directamente en el navegador del cliente gracias al *modelo de objetos JavaScript*[17]. Desde esos inicios surgió el Modelo de Objetos de Documento.

El Modelo de Objetos de Documento, mejor conocido como DOM, por sus siglas en inglés (Document Object Model), es una especificación, publicada por el W3C, que representa un avance significativo en el manejo de documentos estructurados. La especificación es una API (Application Programming Interface, Interfaz de Programación de Aplicaciones), independiente de plataforma y lenguaje, que describe como accesar y manipular información almacenada con documentos estructurados XML y HTML.[18, p. 4]

De acuerdo con la especificación DOM del W3C, el término *Modelo de Objetos de Documento* fue escogido porque el DOM es un modelo de objetos en el sentido tradicional de la programación orientada a objetos, es decir, los documentos son modelados usando objetos, y el modelo describe la estructura del documento, así como su comportamiento y el comportamiento de sus objetos. [18, p. 4]

1.2.1. Estructura del árbol del Documento

Los documentos DOM son representados por una *estructura de árbol*.[18, p. 18-22]

Un árbol es una estructura que organiza sus elementos, denominados nodos, formando jerarquías y relaciones. Fundamentalmente, la relación clave es la de "*padre-hijo*" entre los nodos del árbol. Si existe una arista (rama) dirigida del nodo *n* al nodo *m*, entonces *n* es el **padre** de *m* y *m* es un **hijo** de *n*. En el árbol de la Figura en la página siguiente, los nodos B y C son hijos del nodo A. Los hijos del mismo padre

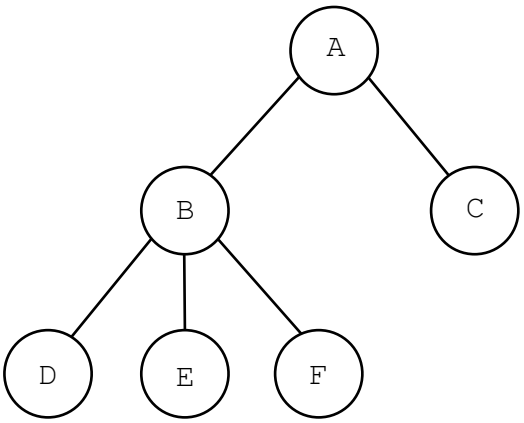
[16] Véase la sección 1.3 en la página 16

[17] Véase la sección 2.2 en la página 21

se llaman **hermanos**, e.g., B y C. Cada nodo de un árbol tiene al menos un padre, y existe un único nodo denominado **raíz** del árbol, que no tiene padre. El nodo A es el raíz del árbol. Un nodo que no tiene hijos se llama **hoja** del árbol. Las hojas del árbol de la Figura 1.3 son C, D, E y F.

La relación padre-hijo entre los nodos se generaliza en las relaciones **ascendiente** (antecesor) y **descendiente**. En la Figura 1.3 A es un antecesor de D, y por consiguiente D es un descendiente de A. Obsérvese que no todos los nodos están relacionados por las relaciones ascendiente/descendiente: B y C, e.g., no están relacionados. Sin embargo, el raíz de cualquier árbol es cualquier nodo del árbol junto con todos sus descendientes. [14, p. 312]

Figura 1.3: Estructura de árbol




```
A       Padre

B,C     hijos de nodo A

D,E,F   hijos de nodo B

B,C     hermanos
```

Considérese el Listado 1 de un documento HTML.

Listado 1 Un documento HTML

```
<html>
   <head>
      <title>Esto es un documento</title>
   </head>
   <body>
      <b>Texto en negritas</b>
      <i>Texto en itálica</i>
   </body>
</html>
```

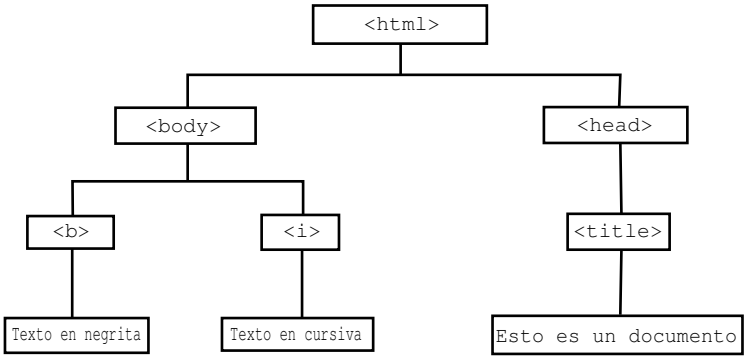
Para el documento del Listado 1, la Figura 1.4 en la página siguiente ilustra la estructura del árbol correspondiente de como el DOM representa las relaciones entre los elementos del documento.

Cada cuadro en la Figura 1.4 es llamado *nodo* en la terminología DOM. Un nodo es la representación de un objeto de un *elemento*[18] particular en el contenido del documento. Los nodos tienen nombres especiales, dependiendo dónde estén colocados en el árbol del documento y su posición relativa a los otros nodos.[18, p. 21]

Todos los documentos tienen un nodo *raíz*, el cual es el nodo que está colocado en la base de la estructura del árbol. En la Figura 1.4, el nodo raíz está representado por el cuadro que tiene la etiqueta **<html>**. Si un

[18]Los elementos son la unidad fundamental de XML. Los elementos están delimitados por etiquetas (*tags*). Las etiquetas siempre van entre los símbolos "<"y ">". Cuando un elemento tiene contenido se expresa con etiquetas de inicio y cierre, e.g., **<etiqueta>*contenido*</etiqueta>**. Cuando un elemento no tiene contenido, se dice que éste es vacío y se expresa como **<etiqueta/>**. [9, p 16-17]

Figura 1.4: Estructura del árbol del Documento



nodo tienen uno o más nodos bajo él, se les llama nodos *hijo*, y cada nodo es descendiente de su nodo *padre*. El nodo representado en la Figura 1.4 por la etiqueta **<head>** es hijo del nodo **<html>**, y a su vez tiene el nodo hijo **<title>**. Esta relación padre-hijo entre nodos es usado extensamente a través de la API DOM. Los nodos que no tienen hijos se les llama nodos *hoja*, y están localizados en la parte final de árbol. Los nodos *texto* que representan el contenido de las etiquetas **<b>**, **<i>** y **<title>** son nodos hoja.[18, p. 21]

Continuando con la metáfora de la relación padre-hijo, los nodos que comparten el mismo padre se llaman nodos *hermano*. En la Figura 1.4, las etiquetas **<body>** y **<head>** son nodos hermanos. Si dos nodos descienden del mismo nodo en alguna parte en la estructura del árbol, se dice que tienen un nodo común ancestro. En la Figura 1.4 los nodos comunes ancestrales compartidos por los nodos **<b>** y **<i>** son los nodos **<body>** y **<html>**.[18, p. 21]

1.2.2. Niveles del DOM

El DOM está organizado en diferentes niveles, cada uno proporciona sus propios métodos y definiciones. Estos niveles definen ciertas capacidades y servicios que un usuario de ese nivel puede esperar de una aplicación que soporte la API definida en ese nivel. Actualmente, hay dos niveles oficiales del DOM: *Nivel 1* y *Nivel 2*.[18, p. 9-10]

1.2.2.1. Nivel 1

El *nivel 1* del DOM fue adoptado como una Recomendación del W3C en octubre de 1998 (una segunda edición en septiembre del 2000). Está dividido en dos módulos: Núcleo (Core) y HTML. El módulo Núcleo proporciona un conjunto básico de métodos para accesar y manipular objetos del documento en cualquier estructura del documento, acompañado de un conjunto de interfaces extendidas para trabajar con contenido XML. El módulo HTML proporciona mayor nivel de interfaces que son usadas en conjunto con las del Núcleo para trabajar con documentos HTML, aunque las interfaces del núcleo son usualmente suficientes para trabajar con contenido XML y HTML.[18, p. 10-11]

1.2.2.2. Nivel 2

El *nivel 2* del DOM fue adoptado como una Recomendación del W3C en noviembre del 2000. Está dividido en 14 módulos distintos (Núcleo, XML, HTML, Rango, Recorrido, CSS, CSS2, Vistas, Hojas de estilo, Eventos, Eventos de interfaz de usuario, Eventos de ratón, Mutación de eventos, y Eventos HTML) y agrupados en 5 módulos principales (Núcleo, Vistas, Eventos, Estilo, y Recorrido/Rango).[18, p. 11-12]

1.2.3. Tipos de nodos DOM

La especificación DOM define 12 tipos diferentes de nodos que pueden estar contenidos dentro de un documento. Cada tipo de nodo tiene un conjunto relacionado de atributos y capacidades asociados con él. La Tabla 1.1 describe los 12 tipos de nodos.[18, p. 22-23][26, p. 552]

Tabla 1.1: Tipos de nodos

No. nodo	Tipo de nodo	Descripción
1	Element	Representa un elemento entre etiquetas de un documento HTML o XML
2	Attribute	Representa un atributo de un elemento
3	Text	Representa el contenido textual de un elemento
4	CDATASection	Representa la sección *Character Data* en un documento XML
5	EntityReference	Representa una referencia de entidad en el documento
6	Entity	Representa una entidad XML
7	ProcessingInstruction	Representa una instrucción específica para ser usada por el procesador del documento
8	Comment	Representa un comentario
9	Document	Representa el nodo raíz del documento
10	DocumentType	Cada nodo Document tiene un nodo DocumentType que proporciona una lista de entidades definidas en el documento
11	DocumentFragment	Es un fragmento del documento o, dicho de otro modo, un conjunto de nodos
12	Notation	Representa una notación de un documento XML. Las notaciones no tienen nodos padres

1.2.4. El DOM y las CSS

Las CSS (Cascade Style Sheet, Hojas de Estilo en Cascada) son una herramienta eficaz para el diseño de páginas Web, debido a que las CSS separan el estilo de la estructura del documento.[21, p. 925]

1.2.4.1. Anatomía de un *estilo*

Un estilo es una especificación de una propiedad visual y su valor para un elemento. El nombre de la propiedad y el valor separados por dos puntos (:). E.g., si se desea cambiar el color de texto a azul el especificador de estilo sería: *color:blue*.

Para especificar múltiples propiedades en un estilo, se separan los especificadores con un punto y coma (;). El siguiente especificador de estilo asigna el color a rojo, el tamaño a 20 y la alineación a centrado:

```
color:red; font-size:20; text-align: center;
```

1.2.4.2. Agregar estilo a un documento

La información de estilo se puede incluir en un documento de cualquiera de estas formas:

- Utilizar una hoja de estilo externa.
- Incrustar un estilo interno para todo el documento.
- Ofrecer un estilo en línea donde se deba aplicar el documento.

Cada uno de estos métodos de CSS tiene sus ventajas e inconvenientes, como se indica en la Tabla 1.2[21, p. 298].

Tabla 1.2: Comparación entre las distintas formas de aplicar CSS

	Hojas de estilo externas	**Hojas de estilo internas**	**Estilos en línea**
Ventajas	Se puede establecer el estilo para muchos documentos con una única hoja de estilo.	Se puede controlar fácilmente el estilo de los documentos uno a uno. No requiere tiempo adicional de descarga para la información de estilo.	Se puede controlar el estilo hasta una única instancia de carácter. Omite cualquier estilo externo o del documento.
Inconvenientes	Exige un tiempo extra de descarga para la hoja de estilo, lo que podría demorar el procesamiento de la página.	Es necesario volver a aplicar la información de estilo para otros documentos.	Es necesario volver a aplicar la información de estilo a todo el documento y fuera de los documentos Está demasiado vinculado a los elementos.

1.2.5. Modelo de eventos

Un *evento* es una acción destacada que ocurre dentro del navegador, a la que puede responder un script. Ocurren eventos, por ejemplo, cuando el usuario hace clic con el ratón, envía un formulario o incluso cuando mueve el puntero del ratón por encima de un objeto de la página. Un *manejador de eventos* es código asociado a una parte específica del documento y a un evento en particular.[22, p. 320]

1.2.5.1. Modelo de eventos básico

El proceso de control de eventos en el modelo de eventos básico común a todos los navegadores es directo. Los manejadores de eventos se asignan a partes del documento utilizando HTML o JavaScript y cuando ocurre un evento, el navegador revisa la parte pertinente de la jerarquía del documento en busca del manejador apropiado.[22, p. 321]

1.2.5.2. Modelo de eventos moderno

El modelo de eventos moderno tiene un objeto **Event** estático que guarda las constantes y los métodos utilizados por todos los eventos. Cuando ocurre un evento, se clona una instancia transitoria del objeto **Event** y se llena con información sobre ese evento. Una vez que se almacenan en el objeto la posición del ratón, el tipo de evento y otros detalles, se le pasa al manejador pertinente para su procesamiento. Una vez que un evento ha sido tratado, el objeto transitorio se destruye y el navegador espera que ocurra otro evento. Si los eventos ocurren muy próximos en el tiempo, son puestos en la cola y procesados de uno en uno, en el orden en que ocurren.

1.2.5.3. Modelo de eventos del DOM2

El W3C describe eventos del DOM2 como "una interfaz independiente de lenguaje y plataforma que proporciona a los programas y scripts un sistema genérico de eventos". Proporciona una forma normalizada para que interactúen los scripts con documentos estructurados y lo hace con un mínimo de divergencia con modelos de eventos de anteriores navegadores. Como resultado, el modelo de eventos del DOM2 es un híbrido de Netscape 4 y de IE4. Se basa en los aspectos fundamentales de HTML del DOM1 para añadir un modelo de objetos de documento completamente codificable con scripts (que siga la especificación del DOM de W3C) para conseguir su funcionalidad.

Los eventos empiezan su ciclo de vida en la parte superior de la jerarquía y van abriéndose camino descendiendo hasta el objeto de destino. Esto se conoce como la "fase de captura" porque imita el comportamiento de Netscape 4. Durante su descenso, el evento puede ser preprocesado, controlado o redirigido por cualquier objeto participante. Una vez que alcanza su objeto de destino y el manejador (si lo hay) lo haya ejecutado, el objeto comienza su camino de regreso ascendiendo por la jerarquía de nodos. Esto se conoce como la fase de ascenso por la relación obvia con el modelo de IE4. Los eventos ascienden de forma predeterminada, pero se deben capturar explícitamente en su camino descendente de forma semejante a la de Netscape 4.[22, p. 349]

Tipos de eventos

El W3C define diferentes eventos que están divididos en las siguiente categorías:

- **Eventos de interfaz de usuario (UI)**. Estos eventos pueden ser activados por el usuario. Ejemplos de tales eventos son entrada del teclado e interacciones del ratón (mover, clic). Generalmente, cualquier cosa que el usuario pueda hacer con periféricos puede generar un evento UI.[26, p. 555]

- **Eventos lógicos de interfaz de usuario**. Estos eventos son esencialmente los mismos eventos UI; sin embargo, ningún dispositivo externo está incluido. Un ejemplo de este tipo de evento es cuando se adquiere o pierde el foco de un elemento.[26, p. 555]

- **Eventos de ratón**. Estos eventos son básicamente eventos UI; sin embargo, debido a que sólo se pueden activar usando el ratón o cualquier dispositivo semejante (trackball, pluma óptica), están representados en este grupo especial de eventos.[26, p. 555]

- **Eventos de mutación**. La estructura lógica del documento (árbol DOM), puede ser manipulada por diferentes medios. Siempre que esto pase, una mutación de evento ocurre. Estos eventos son siempre activados por la ejecución de scripts, e.g., cuando un nodo en la estructura DOM es eliminado o modificado de alguna manera.[26, p. 555]

Vinculación de eventos a elementos

La forma más fácil de vincular eventos a elementos bajo DOM2 es mediante el modelo tradicional, que consiste en utilizar la técnica de vinculación de atributos HTML. Nada cambia para DOM2 cuando se vinculan eventos utilizando HTML excepto la restricción de que solamente se puede utilizar eventos estándar y que el atributo debe ser soportado por el elemento en HTML 4 [22, p. 353], e.g.:

```
<p onclick="miFuncion()">
  Párrafo de prueba que tiene asignado un evento click
</p>
```

Asignar un manejador de eventos utilizando JavaScript también es muy fácil. Como los eventos del DOM2 se construyen sobre la jerarquía de nodos del DOM1, se accesa al nodo deseado utilizando técnicas estándar DOM en vez de navegar por un modelo propietario de objetos de navegador. Una vez que se ha localizado el nodo, se asigna el manejador de eventos a la propiedad del nodo adecuado, e.g.: [22, p. 335-337, 353-354]

```
function alertUser(evt)
{
  alert("Se capturo un click en: " + evt.screenX);
}
document.getElementById("nodoId").onclick = alertUser;
```

donde:

- **document** es el documento mismo
- **getElementById** es un método para accesar directamente al nodo especificado por **"nodoId"**
- **onclick** es el evento al cual se asigna la función **alertUser**

Propagación de eventos: ascenso y captura

El DOM2 proporciona una nueva forma, más poderosa, de vincular manejadores a elementos utilizando oyentes de eventos. Los *oyentes de eventos* son manejadores de eventos vinculados a un nodo determinado de la jerarquía de objetos que se activa durante una fase específica del ciclo de vida del evento. Los oyentes de eventos se unen a los nodos utilizando el método **addEventListener()**. Se eliminan con **removeEventListener()**. Unir un evento a un nodo utilizando estos métodos difiere de la vinculación de eventos normal en varios aspectos. Primero, estos métodos permiten especificar si el evento se manejará durante la fase de captura (abajo) o durante la fase de ascenso (arriba). Los eventos establecidos con técnicas tradicionales se activan cuando el objeto es el destino durante la fase de ascenso, si la hay para ese evento. En segundo lugar, estos métodos permiten vincular varios manejadores para el mismo evento y para el mismo objeto. Y finalmente, se pueden vincular oyentes a nodos de texto, una capacidad que no estaba disponible antes.[22, p. 355]

La sintaxis de los métodos oyentes es como sigue[22, p. 355]:

```
node.addEventListener(type, eventHandler, direction)
```

donde:

- *node* es el nodo al que se va a vincular el oyente.

- *type* es una cadena que indica el evento que se va a escuchar.
- *eventHandler* es la función que se debe invocar cuando se activa.
- *direction* es un valor lógico que indica si debe escuchar durante la fase de captura (**true**) o durante la fase de ascenso (**false**).

E.g.

```
menu.addEventListener("mouseover", cargaSubMenu, false)
```

Cancelación de eventos

Como con los modelos tradicionales, el DOM2 permite cancelar eventos. Para cancelar eventos el comportamiento predeterminado de un evento, todavía se puede devolver **false** en un manejador. Se puede utilizar también dos nuevos métodos del objeto **Event**: **preventDefault()** y **stopPropagation()**. El método **preventDefault()** tiene el mismo efecto que devolver **false** en el manejador. El método **stopPropagation()** detiene el flujo del evento a través de la jerarquía de objetos después de que termine el manejador activo. Este método se puede invocar durante cualquier parte del ciclo de vida del evento, pero funciona solamente con los eventos que se pueden cancelar; invocarlo sobre un evento que no se pueda cancelar no tiene ningún efecto.[22, p. 356]

La cancelación del comportamiento predeterminado de eventos en el DOM2 es básicamente diferente de la de modelos más antiguos. Una vez que un manejador haya devuelto **false** o invocado el método **preventDefault()** del evento, se cancelará el comportamiento predeterminado del evento, sea cual sea. De este modo, si un manejador devuelve **false** o invoca **preventDefault()** y otro manejador a lo largo del flujo devuelve **true**, el evento todavía se cancela, por lo que hay que tener mucho cuidado al cancelar eventos en el DOM2.[22, p. 357]

La interfaz *Event*

Como se mencionó antes, cuando un *oyente de evento* se vuelve activo y un evento es manejado por una o más funciones, siempre es posible averiguar qué evento y qué elemento activaron la función y cuál era el evento objetivo.

Siempre que una función *manejador de evento* es llamada, esta función automáticamente recibe un parámetro: una referencia al evento que activó la ejecución de la función.

La interfaz **Event** es parte del DOM2 y tiene las propiedades listadas en la Tabla 1.3.[22, p. 930-933]

Tabla 1.3: Propiedades de la interfaz **Event**

Propiedad	Descripción
altKey	Valor lógico que indica si la tecla ALT se pulsó durante el evento.
bubbles	Valor lógico que indica si el evento asciende.
button	Número entero que indica qué botones del ratón se pulsaron durante el evento.
cancelable	Valor lógico que indica si el evento se puede cancelar.
clientX	La coordenada *x* en píxeles de la posición de puntero del ratón relativa al área cliente de la ventana del navegador.
clientY	La coordenada *y* en píxeles de la posición de puntero del ratón relativa al área cliente de la ventana del navegador.
ctrlKey	Valor lógico que indica si la tecla CTRL se pulsó durante el evento.
currentTarget	Hace referencia al elemento cuyo manejador está procesando actualmente el evento.
eventPhase	Valor numérico que indica la fase actual en la que se encuentra el evento (1 para la captura, 2 para su destino, 3 para ascender).
metaKey	Valor lógico que indica si la meta-tecla se pulsó durante el evento.
timeStamp	Hora en que se produjo el evento, en milisegundos desde “1/enero/1970”.
target	Hace referencia al objeto en el que se produjo el evento.
type	Tipo de evento (e.g., click, activate).
screenX	La posición horizontal en píxeles donde se produjo el evento con respecto a la pantalla completa.
screenY	La posición vertical en píxeles donde se produjo el evento con respecto a la pantalla completa.
shiftKey	Valor lógico que indica si la tecla Shift se pulsó durante el evento.

Eventos DOM2

El DOM2 soporta los eventos HTML 4 estándar. La Tabla 1.4 muestra el comportamiento de los eventos en el modelo del DOM2.[22, p. 847]

Tabla 1.4: Eventos DOM2

Evento	¿Asciende?	¿Cancelable?
abort	sí	no
blur	no	no
change	sí	no
click	sí	sí
error	sí	no
focus	no	no
load	no	no
mousedown	sí	sí
mouseup	sí	sí
mouseover	sí	sí
mousemove	sí	no
mouseout	sí	sí
reset	sí	no
resize	sí	no
scroll	sí	no
select	sí	no
submit	sí	sí
unload	no	no

1.3. JavaScript

A principios de 1993 en el *NCSA* (National Center for Supercomputing Applications, Centro Nacional de Aplicaciones de Supercómputo) de la universidad de Illinois, **Marc Andreessen**, junto con el grupo de desarrollo de la universidad trabajó en un proyecto cuyo propósito era leer las *páginas del Web* que estaban en formato *HTML*, pero no en modo texto, sino en forma gráfica, utilizando las capacidades de *hipertexto e hipermedia*. El producto de estas investigaciones fue el *navegador Web* ***Mosaic,*** precursor de los navegadores **Netscape**, **Explorer,** y muchos otros.[2, p. 201]

Tan rápido cundió la necesidad de compartir información en forma de *hipertexto e hipermedia*, que para el mes de octubre de 1993 ya existían 200 servidores Web utilizando el protocolo *HTTP*. En marzo de 1994, *Marc Andreessen* y varios colegas forman la compañía *Mosaic Communications Corp*, que poco después se convertiría en *Netscape*.[2, p. 201]

A mediados de 1995, **Brendan Eich**, ingeniero de Netscape, inventó el lenguaje de programación *LiveScript*. Tenía que servir dos propósitos, uno estar incrustado en el servidor para poder enviar información solicitada por los clientes, y el otro conectar entre si las bases de datos dentro del servidor. Pero no pasó ni un mes después del anuncio de su implementación cuando los ingenieros llegaron a la conclusión de que con un poco de preparación de los navegadores (entonces Netscape con su Navigator 2.0 se encontraba a la cabeza de los pocos que habían) se podían hacer muchas cosas sin salir de la computadora del cliente. En una operación conjunta entre Netscape y Sun Industries se terminó la sintaxis de *LiveScript* con este propósito y se le rebautizó como *JavaScript* (por motivos de marketing)[13, p. 61-62]. El lenguaje se cedió al cuerpo de estándares internacional *ECMA* (European Computer Manufacturers Asociation, Asociación de Fabricantes de Computadoras Europeos) que anunció la aprobación en el verano de 1997 del *ECMA-262*, o ECMAScript, como estándar multiplataforma de Internet para los scripts. Los fabricantes de navegadores aceptan la especificación, pero todavía utilizan el nombre reconocido de JavaScript.

1.3.1. Características de lenguaje

JavaScript es un lenguaje dinámico de scripts orientado a objetos e independiente de la plataforma. El núcleo del lenguaje está incrustado, principalmente, en los navegadores Web para interpretar los scripts. Estos scripts pueden hacer que exista interactividad dinámica entre el usuario y la página, controlar el navegador o crear páginas completas sin tener que recurrir al servidor nuevamente. Su sintaxis se parece mucho a la de otros lenguajes como Perl, Java o C++, debido principalmente a que todos ellos son lenguajes que trabajan con objetos, pero aunque su nombre comience con "Java", probablemente tenga más cosas en común con Perl, sobre todo en el tratamiento de eventos y de las *expresiones regulares*[19].

JavaScript puede funcionar tanto como lenguaje estructural (jerárquico) como lenguaje orientado a objetos. En JavaScript los objetos son creados programáticamente anexando métodos y propiedades a otros objetos (vacíos) en tiempo de ejecución, opuesto a la definición de clases sintácticas comunes en lenguajes compilados como C++ y Java. Una vez que un objeto ha sido construido puede ser usado como prototipo para crear objetos similares. Las capacidades dinámicas de JavaScript incluyen construcción de objetos en tiempo de ejecución, lista variable de parámetros, funciones como variables, creación de scripts dinámicos (vía *eval*), introspección de objetos (vía *for...in*), y recuperación de código fuente.

Gracias a su facilidad de uso en la Web, JavaScript ha eclipsado a otros populares lenguajes de Internet, como Java, ActiveX e incluso CGI. Al poder incrustar directamente código en páginas HTML, JavaScript se encuentra en más páginas Web que cualquier otro lenguaje de scripts o compilado.

Su uso está muy extendido en tareas que van desde validación de los datos de formularios a la creación de complejas interfaces de usuario. Aunque en JavaScript se pueden escribir auténticas aplicaciones (en toda la extensión de la palabra), la mayor parte de los programadores sólo lo usan para decorar las páginas Web.

[19] Una expresión regular es un patrón (especificación de un grupo de caracteres) que se desea buscar en una cadena, e.g., buscar los caracteres "SRT" en ese orden específico. [29, p. 244]

Capítulo 2

Programación Orientada a Objetos en JavaScript

Escribir y mantener grandes programas de software es un verdadero reto. Se lleva años investigando diferentes técnicas en la ingeniería de software con el objetivo de intentar determinar las metodologías más efectivas para crear productos de alta calidad. En los primeros tiempos de la programación no había mucha sofisticación en la organización del código o en su planificación. La mayor parte de los programas eran cosa directa de *lenguaje ensamblador*, escritos sin la ayuda de nada que pudiera parecerse a la ingeniería de software de hoy en día.

Según empezaba a evolucionar los lenguajes de alto nivel, iba quedando claro que la modularización de código y la programación estructurada eran aspectos importantes de cualquier proyecto de cierta magnitud. Los requisitos del software continuaban aumentando en cuanto a complejidad, impulsando a los programadores a investigar nuevos métodos para el proceso de creación de software. La *Programación Orientada a Objetos* surgió de la idea de que se puede pensar en software como un conjunto de objetos abstractos que interactúan entre sí. Aunque el enfoque orientado a objetos no es la solución para todos los problemas encontrados durante el desarrollo de software, se presta a la creación de software altamente modular y susceptible de mantenimiento.

Los lenguajes de programación modernos soportan, o por lo menos tienen una aproximación, al desarrollo orientado a objetos.

Actualmente la *Programación Orientada a Objetos (OOP*, Object-Oriented Programming) está generalmente considerada como la mejor forma de estructurar programas, pero raramente se utiliza un estilo totalmente OOP en JavaScript, a pesar de que el lenguaje mismo se basa en los principios de la Programación Orientada a Objetos: *encapsulación, herencia, polimorfismo.*[22, p. 168-170]

JavaScript es un lenguaje de scripts orientado a objetos, basado en prototipos en lugar de clases. Debido a estas bases diferentes, puede ser menos aparente cómo JavaScript permite la creación de jerarquías de objetos y cómo heredan sus métodos y propiedades.

Los objetos en JavaScript se clasifican en cuatro grupos:

- Los objetos ***definidos por el usuario*** son objetos personalizables creados por el programador para dar estructura y coherencia a una tarea de programación en particular. Este capítulo se concentrará en la creación y utilización de estos objetos.

- Los objetos ***incorporados*** son proporcionados por el propio lenguaje JavaScript. Éstos incluyen los objetos asociados a tipos de datos primarios (**String**, **Number** y **Boolean**), objetos que permiten la creación de objetos definidos por el usuario y tipos compuestos (**Object** y **Form**) y objetos que simplifican tareas frecuentes, como **Date**, **Math** y **RegExp**.

- Los objetos ***de navegador*** son objetos no especificados como parte del lenguaje de JavaScript, pero que la mayor parte de los navegadores comúnmente soportan. Ejemplos de objetos de navegador son **Window,** el objeto que permite el control de las ventanas del navegador y la interacción con el usuario, y **Navigator**, el objeto que proporciona la información acerca de la configuración del cliente.
- Los objetos ***de documento*** son parte del Modelo de objetos de documento (DOM).

2.1. Lenguajes basados en clases vs basados en prototipos

Los lenguajes orientados a objetos basados en clases, como Java y C++, están fundados en el concepto de dos diferentes entidades: *clases* e *instancias*.

- Una *clase* es una plantilla que define todas las propiedades (variables) y métodos (funciones) que caracterizan un cierto conjunto de objetos. Por ejemplo, la clase **Empleado** podría representar un conjunto de todos los empleados.[20, Chapter 8 - Details of the Object Model]

  E.g.

```
class Empleado{
  private String nombre;
  private int salario;
  private String puesto;
  public Empleado(String nombre, int salario, String puesto){
    this.nombre = nombre;
    this.salario = salario;
    this.puesto = puesto;
  }
}
```

- Una *instancia*, por otro lado, es un miembro individual de una clase y es fundamentalmente diferente de una clase. Por ejemplo, **victoria** podría representar una instancia de una clase **Empleado**, representando un individuo particular como un empleado. Una instancia tiene exactamente las propiedades y métodos de su clase base.[20, Chapter 8 - Details of the Object Model]

  E.g.

```
Empleado victoria = new Empleado(“Victoria”)
```

Un lenguaje orientado a objetos basado en prototipos, como JavaScript, no reconoce esta distinción; simplemente tiene objetos. Sin embargo, hablando de JavaScript, *instancia* puede ser utilizado informalmente para referirse a objetos creados por una función constructor particular.

Un lenguaje basado en prototipos tiene la noción de un *objeto prototipo*, un objeto usado como una plantilla de donde obtener las propiedades y métodos iniciales para un nuevo objeto, permitiendo que el segundo objeto comparta las propiedades y métodos del primer objeto.

2.1.1. Definición de una clase

En lenguajes basados en clases, una clase se define por separado en una *definición de clase*. Un método constructor puede especificar valores iniciales para las propiedades de las instancias y llamar a otros métodos de ser necesario en tiempo de creación. Se suele usar el operador *new* en asociación con el método constructor para crear instancias de clase.[20, Chapter 8 - Defining a Class]

2.1.2. Subclases y herencia

En un lenguaje basado en clases, se crea una jerarquía de clases a través de las definiciones de clase. En una definición de clase, se puede especificar que la nueva clase es una *subclase* de una clase ya existente. La subclase hereda todas las propiedades y métodos de la clase base, y adicionalmente puede añadir nuevas propiedades y métodos, o modificar las heredadas. Por ejemplo, supóngase que la clase **Empleado** incluye sólo las propiedades **nombre** y **depto**, y **Gerente** es una subclase de **Empleado** que añade **reportes**. En este caso, una instancia de la clase **Gerente** tendría las tres propiedades: **nombre**, **depto** y **reportes**.[20, Chapter 8 - Subclasses and Inheritance]

2.1.3. Añadir y remover propiedades

En los lenguajes basados en clases, típicamente se crea una clase en tiempo de compilación y después se crean las instancias de la clase en tiempo de compilación o en tiempo de ejecución. No se puede cambiar el número o tipo de propiedades de una clase una vez que se ha definido. Sin embargo en JavaScript se pueden añadir o remover propiedades de cualquier objeto en tiempo de ejecución. Si se añade una propiedad a un objeto que es usado como el prototipo de un conjunto de objetos, los objetos para los cuales es prototipo también tendrán la nueva propiedad.

2.1.4. Resumen de diferencias

La Tabla 2.1[20, Chapter 8 - Summary of Differences] proporciona un breve resumen de las diferencias mencionadas.

Tabla 2.1: Comparación de modelos de objetos basados en clases y prototipos

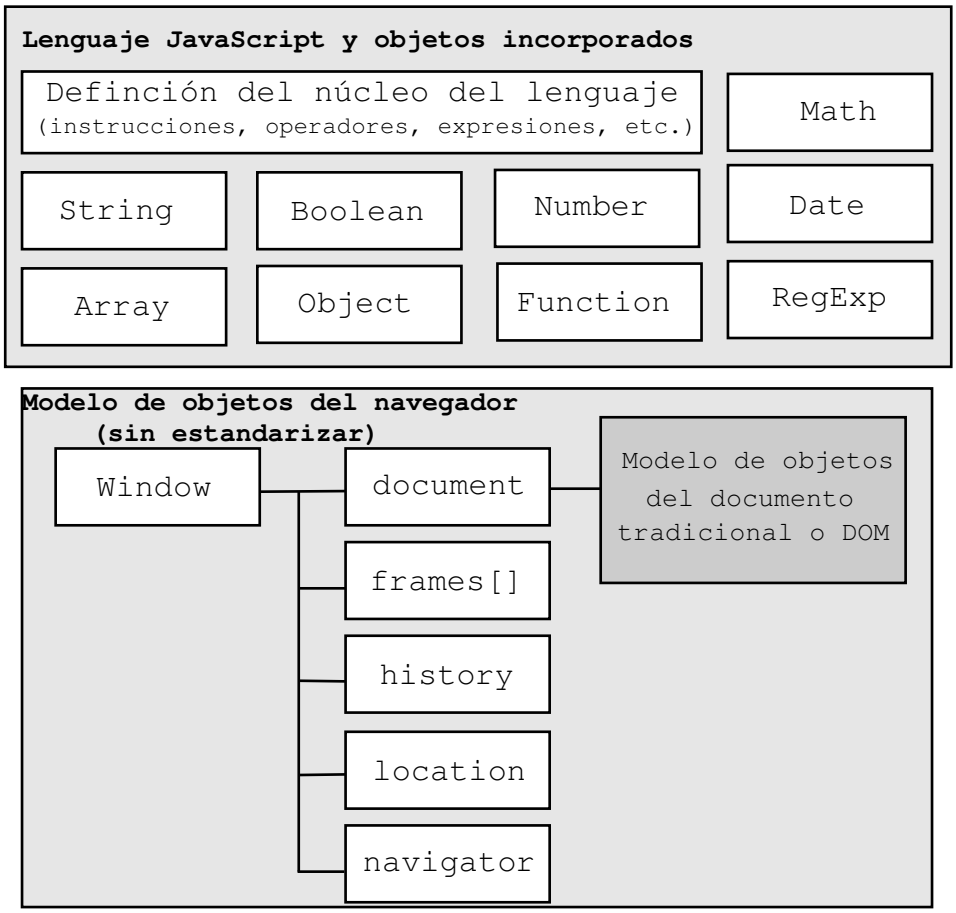
Basado en clases (Java)	**Basado en prototipos (JavaScript)**
Clases e instancias son entidades diferentes	Todos los objetos son instancias
Se define una clase como una *definición de clase*; se instancia una clase con métodos constructores	Se define y se crea un conjunto de objetos con funciones constructor
Se crea un objeto con el operador *new*	Se crea un objeto con el operador *new*
Se construye una jerarquía de objetos usando las definiciones de clase para definir subclases de clases existentes	Se construye una jerarquía de objetos asignando a un objeto como el prototipo asociado con la función constructor
Se heredan las propiedades siguiendo la cadena de clases	Se heredan propiedades siguiendo la cadena de prototipos
La definición de clase especifica todas las propiedades de todas las instancias de una clase. No se pueden añadir propiedades dinámicamente en tiempo de ejecución	La función constructor o prototipo especifica un conjunto inicial de propiedades. Se pueden añadir o remover propiedades dinámicamente a objetos individuales o a conjuntos enteros de objetos
Se ocultan atributos y/o métodos mediante la palabra clave *private*	Todos los atributos y métodos son públicos

2.2. Modelo de objetos JavaScript

En JavaScript se emplean dos modelos de objetos fundamentales: un Modelo de Objetos de Navegador (BOM, acrónimo del inglés Browser Object Model) y un Modelo de Objetos de Documento (DOM). El BOM proporciona acceso a las diversas características de un navegador, como la propia ventana del navegador, las características de la pantalla, el historial del navegador, etc. El DOM, por su parte, proporciona acceso al contenido de la ventana del navegador (concretamente al documento). [22, p. 237]

La Figura 2.1 presenta de forma general los distintos aspectos de JavaScript, incluidos sus modelos de objetos. Hay cuatro piezas esenciales[22, p. 238-239]:

Figura 2.1: Modelo de objetos JavaScript



- Los elementos fundamentales de JavaScript (tipos de datos, operadores, instrucciones, etc.)
- Los objetos fundamentales relacionados esencialmente con los tipos de datos (**Date, String, Math,** etc.)
- Los objetos del navegador (**Window, Navigator, Location,** etc.)
- Los objetos del documento (**Document, Form, Image,** etc.)

2.3. Objetos definidos por el usuario

2.3.1. Creación de objetos

Los objetos se crean con el operador ***new***. Este operador hace que el constructor en cuestión cree un nuevo objeto. La naturaleza del objeto creado está determinada por el constructor específico que se invoca. Un constructor y su prototipo definen un tipo de objeto y todos los objetos creados con ese constructor son instancias de ese tipo.[22, p. 153]

Un objeto tiene *propiedades*, que son las variables de JavaScript. Cada objeto puede tener varios *métodos* asociados, que son las funciones de JavaScript. En JavaScript, las propiedades o métodos de un objeto se acceden mediante la notación [20, capítulo 7][17, p. 180]:

```
Objeto.Propiedad
Objeto.Método()
```

El siguiente ejemplo muestra cómo definir un constructor definiendo una función:

```
function Empleado(){
}
```

Esta función por sí misma no hace absolutamente nada. Sin embargo, se puede invocar como un método constructor:

```
var emp = new Empleado();
```

Con lo que se crea una instancia del objeto *Empleado*. Obviamente, este objeto no es útil.

Cuando se invoca un constructor, el intérprete asigna espacio para el nuevo objeto e implícitamente pasa el objeto a la función. El constructor puede acceder a él utilizando *this*, una palabra clave especial que contiene una referencia al nuevo objeto. Por ejemplo:

```
function Empleado(){
  this.nombre = "Vlad";
}
```

Se añadió una nueva propiedad de instancia *nombre* al objeto que se está creando. Observe que todos los objetos *Empleado* tendrán esta propiedad porque todos están creados con el constructor *Empleado()*. Después de crear un objeto con el constructor, se puede acceder a la propiedad *nombre* de la siguiente manera:

```
var emp = new Empleado();
var nom = emp.nombre;
```

2.3.2. Destrucción de objetos y reciclaje de basura

Cuando se crean objetos, JavaScript asigna automáticamente la memoria para que se puedan utilizar. Lo que realmente sucede es que el intérprete asigna la memoria y pasa una referencia de este nuevo objeto (vacío) al constructor que ha sido invocado. La cuestión es que el intérprete no sólo maneja automáticamente la asignación de la memoria, sino que también "limpia" lo que va dejando. Esta característica del lenguaje se llama *reciclaje de basura* (*Garbage Collection* en inglés).[22, p. 158]

Los lenguajes con reciclaje de basura ejercen una continua vigilancia sobre sus datos. Cuando una serie de datos no son ya accesibles al programa, el espacio que ocupan es recuperado por el intérprete y devuelto al bloque de memoria disponible.[22, p. 158]

2.3.3. Definición de métodos

Un método es una función asociada con un objeto. Una forma de definir un método es utilizando una *literal de función*. Las literales de función utilizan la palabra clave *function*, pero sin un nombre de función explícito[22, p. 138]. Por ejemplo:

```
function Empleado(){
  this.toString = function(){ return "objeto Empleado";}
}
```

2.3.4. Propiedades prototipo

Todos los objetos tienen una propiedad *prototype* que le da su estructura. El prototipo define el código y los datos que todos los objetos de ese tipo tienen en común.[22, p. 165]

Otra forma de añadir propiedades o métodos es usando el prototipo del objeto:

```
function Empleado(){
  this.nombre = "";
}
Empleado.prototype.toString = function(){ return "objeto Empleado"; }
```

Un prototipo es *compartido*, es decir, hay solamente una copia de éste que es usado por todos los objetos creados con el mismo constructor y, por lo tanto, un cambio en el prototipo será visible para todos los objetos que lo comparten.[22, p. 166]

2.3.5. Propiedades estáticas

Además de propiedades de instancia y propiedades de prototipos, JavaScript permite definir *propiedades estáticas* (también conocidas como *propiedades de clase*). Como los constructores son funciones y las funciones son instancias del objeto *Function*, podemos añadir propiedades a los constructores. Tales propiedades son propiedades estáticas. Por ejemplo:

```
Empleado.esTemporal = true;
```

define una propiedad estática del objeto *Empleado* añadiendo una variable de instancia al constructor. Es importante mencionar que las propiedades estáticas solamente existen en un único lugar, como miembros de constructores. Por tanto, se accede a ellas a través del constructor y no de una instancia del objeto.[22, p 167]

2.3.6. Encadenamiento de prototipos (Prototype Chain)

La herencia en JavaScript se logra a través de los prototipos. Está claro que las instancias de un objeto en particular "heredan" el código y datos presentes en el prototipo del constructor. Es también posible obtener un nuevo tipo de objeto de un tipo ya existente. Las instancias de tales objetos lo heredan todo, las propiedades de su tipo "base" y también cualquier propiedad añadida por el nuevo tipo [22, p. 167]. Por ejemplo, defináse un nuevo tipo de objeto que hereda todas las capacidades del objeto *Empleado* por encadenamiento de prototipos:

```
EmpleadoDerivado.prototype = new Empleado();
function EmpleadoDerivado(sexo){
  this.sexo = sexo;
}
```

Los objetos *EmpleadoDerivado* tendrán todas las propiedades del objeto *Empleado* y las de *EmpleadoDerivado*.

2.3.7. Propiedades comunes de los objetos

La abstracción y la herencia son dos de los principios esenciales de la programación orientada a objetos. En JavaScript todos los objetos derivan del objeto ***Objec***t, éste es el tipo de objeto más genérico. Todos los demás objetos se obtienen de él, según las reglas de herencia de JavaScript. Como "superobjeto", define varias propiedades que son comunes a todos sus descendientes. La Tabla 2.2 muestra estas propiedades.[22, p. 162]

Tabla 2.2: Propiedades del objeto *Object*

Propiedad	Descripción
constructor	Una referencia de sólo lectura a la función que sirve de objeto del constructor.
prototype	El prototipo para el objeto. Este objeto define las propiedades y los métodos comunes a todos los objetos de este tipo.
toSource()	Devuelve una cadena que contiene una literal de JavaScript que describe el objeto.
toString()	Devuelve al objeto una cadena, de forma predeterminada "[objeto Object]". Muy a menudo sobreescrito para proporcionar funcionalidad específica.
unwatch(property)	Deshabilita la vigilancia de la propiedad del objeto indicada por la cadena *property.*
valueOf()	Devuelve el valor primitivo asociado con el objeto, de forma predeterminada a la cadena "[objeto Object]". A menudo sobreescrito para proporcionar funcionalidad específica.
watch(property, handler)	Establece una vigilancia sobre la propiedad del objeto indicada en la cadena *property*. Cuando el valor de la propiedad cambia, se invoca al manejador de función *handler* con tres argumentos: el nombre de la propiedad, el valor antiguo y el nuevo valor al que se está estableciendo. El manejador *handler* puede sobrescribir la configuración del nuevo valor devolviendo un valor, que se establece en su lugar.

2.3.8. Mejoramiento de métodos existentes de objetos

Las clases de objetos integrados de JavaScript (Object, Math, Date, RegExp, etc.), incluyen varios métodos predefinidos útiles en la mayoría de los casos. Habrá ocasiones, sin embargo, en las cuales se necesite más funcionalidad de la que pueden proporcionar.

Se pueden mejorar los métodos existentes de dos maneras, dependiendo del tipo de objeto con el que se esté trabajando. Hay algunos objetos de los que no se puede crear instancias, los cuales son únicos y

universales. No se puede, por ejemplo, crear una instancia del objeto *Math*. Esto contrasta con las clases de objeto, de los cuales se pueden crear instancias (String, Array, Date, etc.). [7, p. 211]

Cuando no se puede crear una instancia de un objeto se puede mejorar un método del mismo usando la técnica de Listado 2. El script mejora el método **max()** (empleado para encontrar el mayor de dos números) del objeto Math, ahora encuentra el mayor de tres números. El script funciona de la siguiente manera:

- En la primera línea el nuevo objeto **MathPlus** está asignado al método **Math** del prototipo de la clase Object (es decir, el objeto MathPlus se convierte en el nuevo objeto Math).
- En la segunda línea se define el objeto MathPlus**.**
- En la cuarta línea se emplea la propiedad **__proto__** para hacer que el objeto MathPlus herede toda la funcionalidad del objeto Math, debido a que no se puede crear una instancia del objeto Math. En este punto, MathPlus puede hacer todo lo que Math hace. Así, **Math.round(34.7)**; y **MathPlus.round(34.7)**; hacen los mismo. La razón de ello es que no se puede manipular directamente los métodos del objeto Math (o cualquier otro objeto del que no se puedan crear instancias). Así para evitar este obstáculo, se crea un objeto sustituto que herede la funcionalidad de Math (MathPlus), se cambia su funcionalidad y, en un extraño giro, se indica a JavaScript pensar en el objeto sustituto MathPlus como si fuera el nuevo objeto Math. [7, p. 212]
- En las siguientes dos líneas de código, se definen dos nuevos métodos para el objeto MathPlus: **viejoMax**, al cual se le da la funcionalidad del método **Math.max()** original, y **max**, que se configura como el nuevo y mejorado método **max()**. Se debe crear **viejoMax** porque el nuevo método **max()** aún requiere de la funcionalidad del método **max()** original. Sin embargo, era necesario darle un nuevo nombre porque el método mejorado también se llama **max**. La función que define el nuevo método **max()** está configurada para aceptar tres valores **(x, y, z)**. La definición del nuevo método emplea esos valores de parámetros en varias formas para encontrar el mayor de tres números. [7, p. 212]

Listado 2 MathPlus

```
Object.prototype.Math = MathPlus;
function MathPlus()
{
  this.__proto__ = Math;
  this.viejoMax = Math.max;
  this.max = function(x,y,z)
  {
    var primeraComp = this.viejoMax(x,y);
    var segundaComp = this.viejoMax(primeraComp, z);
    return(segundaComp);
  }
}
```

2.3.9. Definición de métodos o propiedades personalizados para objetos incorporados

Cuando se necesita un método o propiedad inédito se debe crear uno propio.

Para crear un nuevo método para un objeto integrado, por ejemplo el objeto Math, se usa la siguiente sintaxis:

```
Math.prototype.nuevoMetodo = function(arg1, arg2, ..., argN)
{
  ...
  ...
}
```

Aquí se puede ver que para agregar un método al objeto Math, simplemente se le debe anexar la función a su prototipo. Este método estará disponible para usarse desde cualquier punto.

Para crear una nueva propiedad para un objeto integrado, al igual que con los métodos, basta con anexarla a su prototipo.

```
Math.prototype.nuevaPropiedad = null;
```

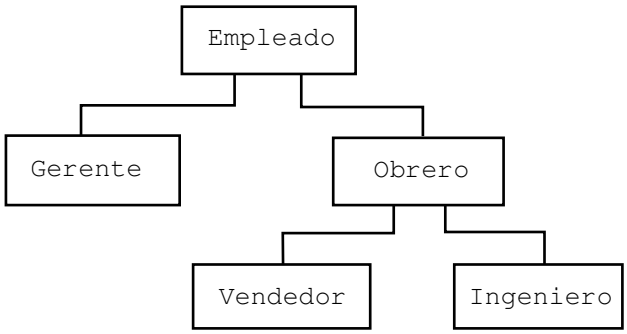
2.3.10. Buena práctica en la creación de objetos

El encadenamiento de prototipos explica por qué es a veces aconsejable proporcionar propiedades y métodos específicos a objetos definidos por el usuario que sobrescriben el comportamiento del objeto base. Si un programador espera que el comportamiento de una propiedad o método pueda cambiar de objeto base a objeto derivado, el comportamiento se debería sobrescribir en el hijo. E.g., es aconsejable proporcionar un método *toString()* específico a los objetos definidos por el usuario para sobrescribir la funcionalidad del método *toString()* de *Object* para saber el nombre del objeto. [22, p. 168]

2.4. Ejemplo de Programación Orientada a Objetos

El resto de este capítulo utiliza la jerarquía **Empleado** que se muestra en la Figura 2.2, el cual fue extraído del capítulo 8 (“Details of the Object Model”) de “Core JavaScript Guide 1.5 [20], para mostrar las técnicas de OOP en JavaScript.

Figura 2.2: Una simple jerarquía de objetos



Este ejemplo utiliza los siguientes objetos:

- **Empleado** tiene la propiedad *nombre* (cuyo valor predeterminado es una cadena vacía) y *depto* (cuyo valor predeterminado es “general”).
- **Gerente** está basado en **Empleado**. Añade la propiedad *reportes* (cuyo valor predeterminado es un arreglo vacío, destinado a tener un arreglo de objetos **Empleado** como su valor).

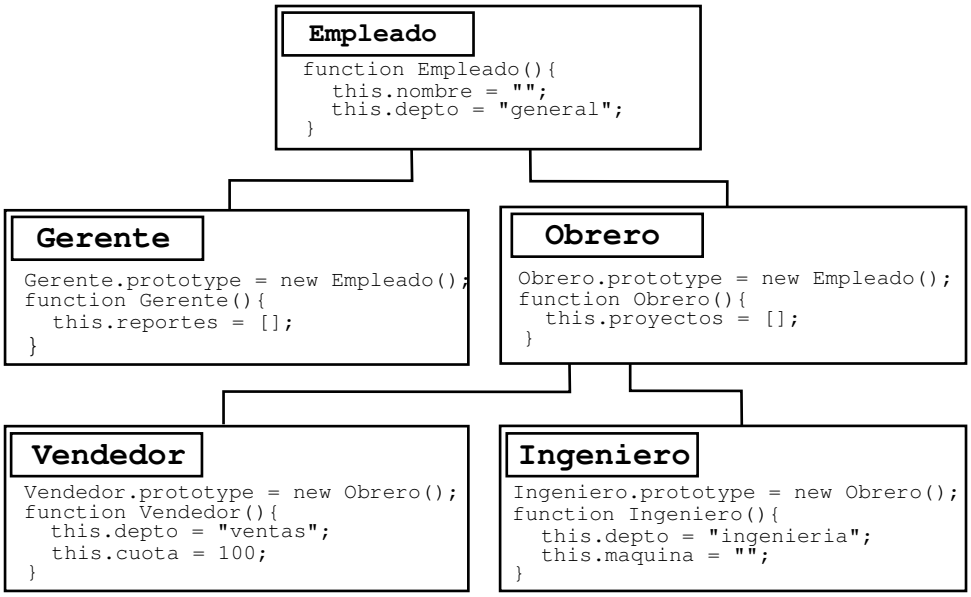
- **Obrero** también está basado en **Empleado**. Añade la propiedad *proyectos* (cuyo valor es una arreglo vacío, destinado a tener un arreglo de cadenas como su valor).
- **Vendedor** está basado en **Obrero**. Añade la propiedad *cuota* (cuyo valor predeterminado es 100). También sobrescribe la propiedad *depto* con el valor "ventas", indicando que todos los vendedores están en el mismo departamento.
- **Ingeniero** también está basado en **Obrero**. Añade la propiedad *maquina* (cuyo valor predeterminado es una cadena vacía) y también sobreescribe la propiedad *depto* con el valor "ingeniería".

2.4.1. Creación de la jerarquía

Esta sección muestra cómo definir funciones constructor simples e inflexibles para implementar la jerarquía Empleado.

En estas definiciones, no se puede especificar ningún valor para las propiedad cuando se crea el objeto. El recién objeto creado simplemente tiene los valores predeterminados, que pueden ser cambiados más tarde. La Figura 2.3 ilustra la jerarquía con estas simples definiciones.

Figura 2.3: Definiciones de la jerarquía



Usando estas definiciones, se puede crear instancias de estos objetos que tienen valores predeterminados para sus propiedades. La Figura 2.4 ilustra como crear nuevos objetos y muestra los valores de las propiedades de dichos objetos.

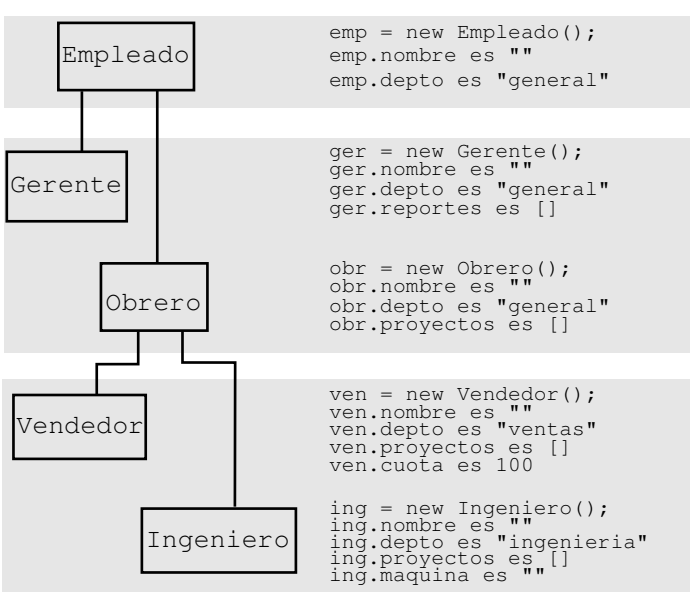
2.4.2. Propiedades de objetos

Esta sección trata sobre cómo los objetos heredan propiedades de otros objetos mediante encadenamiento de prototipos y qué pasa cuando se añade una propiedad en tiempo de ejecución.

2.4.2.1. Herencia de propiedades

Supóngase que se crea el objeto *obr* como un *Obrero* como se muestra en la Figura 2.4 con la siguiente sentencia:

Figura 2.4: Creación de objetos con definiciones simples




```
obr = new Obrero();
```

Cuando JavaScript encuentra el operador *new*, crea un nuevo objeto genérico y pasa este nuevo objeto como el valor de la palabra clave *this* a la función constructor *Obrero*. La función constructor explícitamente asigna el valor de la propiedad *proyectos*. También asigna el valor interno de la propiedad *__proto__* al valor de *Obrero.prototype*. La propiedad *__proto__* determina la cadena prototipo usada para regresar los valores de las propiedades. Una vez que estas propiedades son asignadas, JavaScript regresa el nuevo objeto y las sentencias de asignación colocan la variable *obr* a ese objeto.

Este proceso no pone los valores explícitamente en el objeto *obr* (valores locales) para las propiedades de *obr* heredadas mediante encadenamiento de prototipos. Cuando se pide el valor de una propiedad, JavaScript primero revisa sí el valor existe en ese objeto. Si existe, el valor es regresado. Si el valor no es local, JavaScript revisa la cadena prototipo (usando la propiedad *__proto__*). Si un objeto en la cadena prototipo tiene un valor para la propiedad, ese valor es regresado. Si la propiedad no se encuentra, JavaScript informa que el objeto no tiene esa propiedad. De esta forma, el objeto *obr* tiene las siguientes propiedades y valores:

```
obr.nombre == “”;
obr.depto == “general”;
obr.proyectos == [];
```

El objeto *obr* hereda valores para las propiedades *nombre* y *depto* del objeto prototipo en *obr.__proto__*. Se asigna un valor local para la propiedad *proyectos* por el constructor *Obrero*. Esto proporciona la herencia de propiedades y de sus valores en JavaScript.

Debido a que estos constructores no permiten proporcionar valores específicos de instancia, esta información es genérica. Los valores de las propiedades son los predeterminados compartidos por todos los nuevos objetos creados a partir de *Obrero*. Se puede, por supuesto, cambiar los valores de cualquiera de estas propiedades. Por ejemplo, se podría dar información específica para *obr* como la siguiente:

```
obr.nombre = “Alán O. R.”
obr.depto = “admin”;
obr.proyectos = [”mozilla”];
```

2.4.2.2. Adición de propiedades

En JavaScript se pueden añadir propiedades a cualquier objeto en tiempo de ejecución. No hay restricciones para usar sólo las propiedades proporcionadas por la función constructor. Para añadir una propiedad que es específica a un objeto singular, se asigna el valor al objeto de la siguiente manera:

```
obr.bonus = 3000;
```

Ahora el objeto *obr* tiene una propiedad *bonus*, pero ningún otro *Obrero* tiene esta propiedad. Si se desea eliminar esa propiedad se debe utilizar el operador *delete*, por ejemplo:

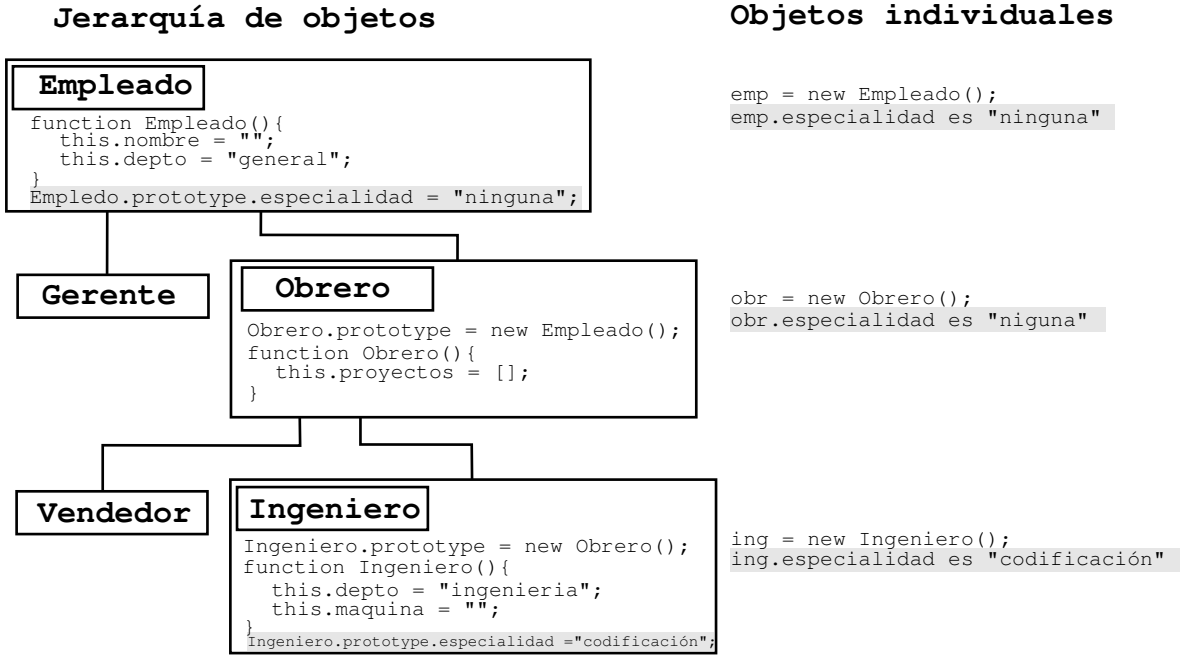
```
delete obr.bonus;
```

Si se añade una nueva *propiedad prototipo* a un objeto que está siendo usado como el prototipo para una función constructor, se añade esa propiedad a todos los objetos que heredan propiedades del prototipo. Por ejemplo, se puede añadir una propiedad *especialidad* a todos los empleados con la siguiente sentencia:

```
Empleado.prototype.especialidad = “ninguna”;
```

Tan pronto como JavaScript ejecuta este código, el objeto *obr* también tendrá la propiedad *especialidad* con el valor de *“ninguna”*. La Figura 2.5 muestra el efecto de añadir esta propiedad al prototipo *Empleado* y sobreescribiéndolo para el prototipo *Ingeniero*.

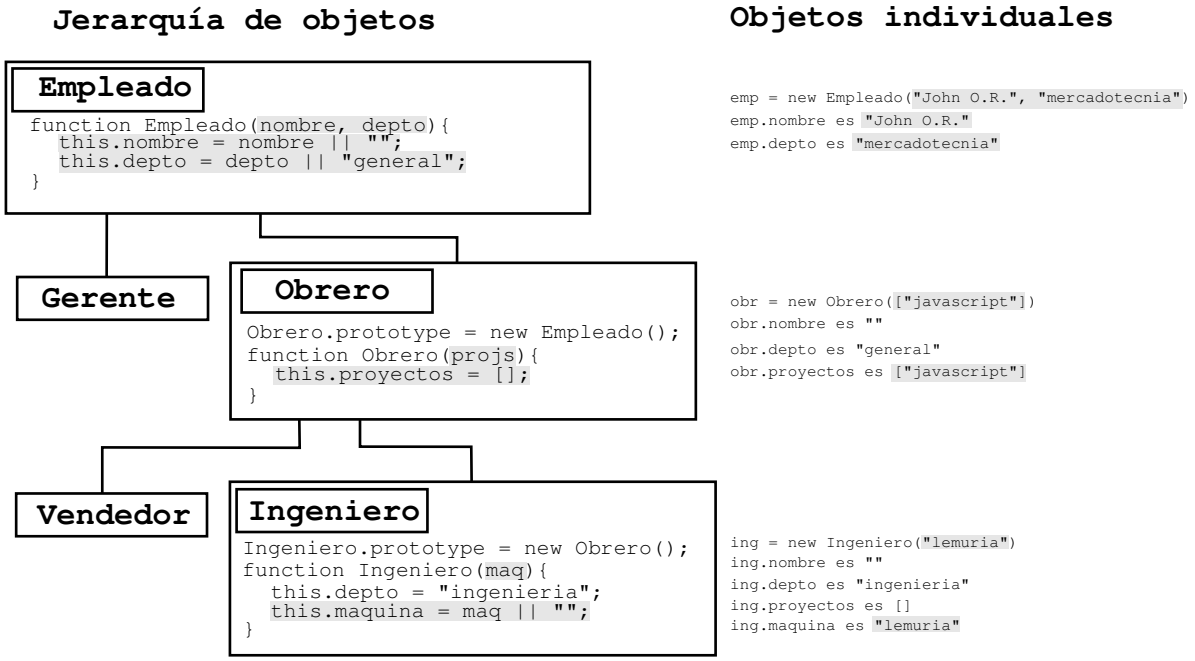
Figura 2.5: Añadiendo propiedades



2.4.3. Constructores más flexibles

Las funciones constructor mostradas hasta ahora no permiten especificar valores para las propiedades cuando se crea una instancia. Se pueden proporcionar argumentos a los constructores para inicializar valores de propiedad en las instancias. La Figura 2.6 ilustra como hacer esto.

Figura 2.6: Especificando propiedades en un constructor, forma 1



El operador lógico OR (||) evalúa su primer argumento. Si ese argumento es verdadero, el operador lo regresa. De lo contrario, el operador regresa el valor del segundo argumento. Por lo tanto, esta línea de código prueba si *nombre* tiene un valor para la propiedad *nombre*. Si lo tiene, asigna ese valor a *this.nombre*. De lo contrario, le asigna una cadena vacía.

Con estas definiciones, cuando se crea una instancia de un objeto, se pueden especificar valores para las propiedades definidas localmente. Como se muestra en la Figura 2.6, se utiliza la siguiente sentencia para crear un nuevo *Ingeniero*:

```
ing = new Ingeniero(“vlad”);
```

Las propiedades de *ing* ahora son:

```
ing.nombre == “”;
ing.depto == “ingeniería”;
ing.proyectos == [];
ing.maquina == “vlad”;
```

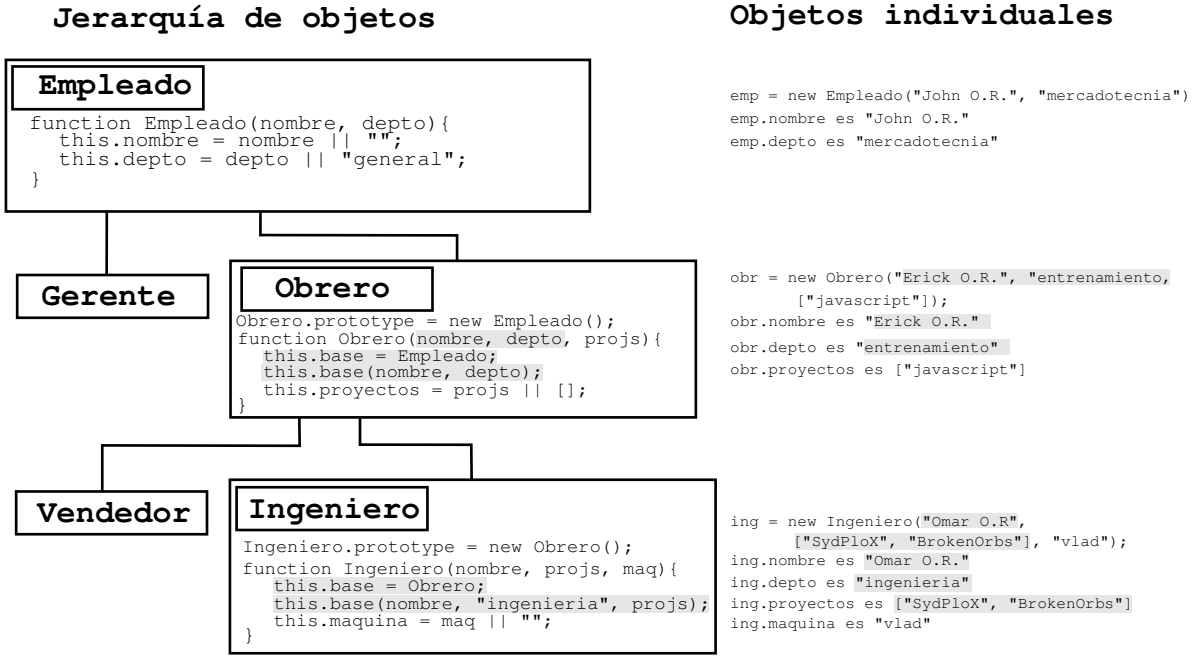
Con estas definiciones no se puede especificar un valor inicial para una propiedad heredada, como el *nombre*. Si se desea especificar un valor inicial para las propiedades heredadas en JavaScript, se necesita añadir más código a la función constructor.

Hasta ahora, la función constructor ha creado un objeto genérico y después especificado propiedades y valores locales para el nuevo objeto. Se puede hacer que el constructor añada más propiedades llamando directamente a la función constructor para un objeto de nivel superior mediante el encadenamiento de prototipos. La Figura 2.7 muestra estas nuevas definiciones.

Obsérvese algunas de estas definiciones en detalle. Aquí está la nueva definición para el constructor *Ingeniero*:

```
function Ingeniero(nombre, projs, maq){
```

Figura 2.7: Especificando propiedades en un constructor, forma 2




```
  this.base = Obrero;
  this.base(nombre, “ingeniería”, projs);
  this.maq = maq || “”;
}
```

Supóngase que se crea un nuevo objeto *Ingeniero* con el siguiente código:

```
ing = new Ingeniero(“Omar O.R.”, [”SydPloX”, “BrokenOrbs”], “vlad”);
```

JavaScript realiza estos pasos:

1. El operador *new* crea un objeto genérico y asigna su propiedad *__proto__* a *Ingeniero.prototype*.

2. El operador *new* pasa el nuevo objeto al constructor *Ingeniero* como el valor de la palabra clave *this*.

3. El constructor crea una nueva propiedad llamada *base* para ese objeto y le asigna el valor del constructor *Obrero*. Esto hace al constructor *Obrero* un método del objeto *Ingeniero*.
   El nombre de la propiedad *base* no es especial. Se puede usar cualquier nombre permitido por JavaScript.

4. El constructor llama al método *base*, pasando dos argumentos al constructor (“*Omar O.R.*” y *[”SydPloX”, “BrokenOrbs”]*) y también la cadena *“ingeniería”*. Usando explícitamente *“ingeniería”* en el constructor, indica que todos los objetos tienen el mismo valor para la propiedad heredada *depto*, y este valor sobrescribe el valor heredado de *Empleado*.

5. Debido a que *base* es un método de *Ingeniero*, desde el que se llama a *base*, JavaScript anexa la palabra clave *this* al objeto creado en el **paso 1**. Así, la función Obrero pasa en turno los argumentos “*Omar O.R.*” y *[”SydPloX”, “BrokenOrbs”]* a la función constructor *Empleado*. Hacia el regreso de la función constructor *Empleado*, la función *Obrero* utiliza el argumento restante para asignarlo a la propiedad *proyectos*.

6. Hacia el regreso del método *base*, el constructor *Ingeniero* inicializa la propiedad *maquina* del objeto a*"vlad"*.

7. Hacia el regreso del constructor, JavaScript asigna el nuevo objeto a la variable *ing*.

Se podría pensar que habiendo llamado al constructor *Obrero* desde adentro del constructor *Ingeniero*, se configuraría la herencia apropiadamente para los objetos *Ingeniero*. Este no es el caso. Llamando al constructor *Obrero* asegura que un objeto *Ingeniero* empezará con las propiedades especificadas en todas las funciones constructor que son llamadas. Sin embargo, si después se agregan propiedades a los prototipos *Empleado* u *Obrero*, esos prototipos no serán heredados por el objeto *Ingeniero*. Por ejemplo, asúmase que se tiene el siguiente código:

```
function Ingeniero(nombre, projs, maq){
  this.base = Obrero;
  this.base(nombre, "ingeniería", projs);
  this.maquina = maq || "";
}
ing = new Ingeniero("Omar O.R.", ["SydPloX", "BrokenOrbs"], "vlad");
Empleado.prototype.especialidad = "python";
```

El objeto *ing* no hereda la propiedad *especialidad*. Aún se necesita configurar explícitamente el prototipo para asegurar la herencia dinámica. Asúmase que se tiene el siguiente código:

```
Ingeniero.prototype = new Obrero();
function Ingeniero(nombre, projs, maq){
  this.base = Obrero;
  this.base(nombre, "ingeniería", projs);
  this.maquina = maq || "";
}
ing = new Ingeniero("Omar O.R.", ["SydPloX", "BrokenOrbs"], "vlad");
Empleado.prototype.especialidad = "python";
```

Ahora el valor de la propiedad *especialidad* del objeto *ing* es *"python"*.

2.4.4. El valor *undefined*

Retomando el siguiente código de la sección anterior:

```
function Empleado(nombre, depto){  ... }
function Obrero(nombre, depto, projs){  ... }
function Ingeniero(nombre, projs, maq){
  this.base = Obrero;
  this.base(nombre, "ingenieria", projs);
  this.maquina = maq || "";
}
```

se puede hacer que el constructor llame al constructor del objeto que hereda y se puede construir un objeto Ingeniero de la siguiente forma:

```
ing = new Ingeniero("Román", ["SydPloX", "BrokenOrbs", "ODE"], "vlad");
```

Sin embargo, si no se desea asignar nada a la propiedad *depto* y mantener el valor predeterminado ("general"), entonces se tiene que asignar explícitamente el valor ***undefined***, y se tiene la siguiente definición:

```
function Ingeniero(nombre, projs, maq){
  this.base = Obrero;
  this.base(nombre, undefined, projs);
  this.maquina = maq || "";
}
```

Al crear un nuevo objeto *Ingeniero*

```
ing = new Ingeniero("Omar O.R.", ["SydPloX", "BrokenOrbs", "ODE"], "vlad")
```

se tiene lo siguiente:

```
ing.name == "Omar O.R."
ing.depto == "general"
ing.proyectos == ["SydPloX", "BrokenOrbs", "ODE"]
ing.maq = "vlad"
```

Al pasar el parámetro *undefined*, el constructor de nivel superior asigna el valor "general" a la propiedad *depto.* Del mismo modo, si al construir un objeto Ingeniero no se desea asignar nada a la propiedad *maq*, basta con asignarle el valor *undefined* o no pasar el último parámetro, por ejemplo:

```
ing = new Ingeniero("Omar O.R.", ["SydPloX", "BrokenOrbs", "ODE"], undefined);
ó
ing = new Ingeniero("Omar O.R.", ["SydPloX", "BrokenOrbs", "ODE"]);
```

de este modo, se tiene lo siguiente:

```
ing.name == "Omar O.R."
ing.depto == "general"
ing.proyectos == ["SydPloX", "BrokenOrbs", "ODE"]
ing.maq = ""
```

2.5. Herencia de propiedades

Las secciones anteriores describen cómo los constructores y prototipos en JavaScript proporcionan jerarquías y herencia. Esta sección trata algunas sutilezas que no fueron aparentes en explicaciones previas.

2.5.1. Valores locales vs valores heredados

Cuando se accesa a una propiedad de un objeto, JavaScript primero revisa las *propiedades de instancia* del objeto en busca del nombre deseado. Si no se encuentra, se comprueban las *propiedades prototipo* del objeto. Este proceso se repite en forma recursiva hasta llegar a la cadena de herencia del objeto de nivel superior. Si no se encuentra la propiedad, entonces el objeto no posee tal propiedad.

El resultado de estos pasos depende de cómo se definen las cosas en el camino. En las definiciones originales mostradas en la Figura en la página 27 se tienen las siguientes:

```
function Empleado(){
  this.nombre = "";
  this.depto = "general";
}
function Obrero(){
  this.proyectos = [];
}
Obrero.prototype = new Empleado();
```

Con estas definiciones, supóngase que se crea un objeto *obr* como una instancia de *Obrero* con la siguiente sentencia:

```
obr = new Obrero();
```

El objeto *obr* tiene una propiedad local, *proyectos*. Los valores para las propiedades *nombre* y *depto* no son locales para *obr* y de este modo se obtienen de la propiedad *__proto__* del objeto *obr*. Así, *obr* tiene estos valores de propiedades:

```
obr.nombre == "";
obr.depto == "general";
obr.proyectos == [];
```

Ahora supóngase que se cambia el valor de la propiedad *nombre* en el prototipo asociado con *Empleado*:

```
Empleado.prototype.nombre = "Desconocido"
```

A primera vista, podría esperarse que los nuevos valores se propaguen a todas las instancias de *Empleado*. Sin embargo, no ocurre así.

Cuando se crea cualquier instancia del objeto *Empleado*, esa instancia tiene un valor para la propiedad *nombre* (una cadena vacía). Esto significa que cuando se asigna el prototipo *Obrero* creando un nuevo objeto *Empleado*, *Obrero.prototype* tiene un valor local para la propiedad *nombre*. Por lo tanto, cuando JavaScript busca la propiedad *nombre* del objeto *obr* (una instancia de *Obrero*), JavaScript encuentra el valor local para esa propiedad en *Obrero.prototype*. Ya no busca más arriba en la cadena a *Empleado.prototype*.

Si se desea cambiar el valor de la propiedad de un objeto en tiempo de ejecución y que el nuevo valor sea heredado por todos los descendientes del objeto, no se puede definir la propiedad en la función constructor del objeto. En su lugar, se le añade al prototipo asociado del constructor. Por ejemplo:

```
function Empleado(){
  this.depto = "general";
}
Empleado.prototype.nombre = "";
Obrero.prototype = new Empleado();
function Obrero(){
  this.proyectos = [];
}
obr = new Obrero();
Empleado.prototype.nombre = "Desconocido";
```

En este caso, la propiedad *nombre* de *obr* se convierte en "Desconocido".

Como estos ejemplos muestran, si se desea tener valores predeterminados para las propiedades del objeto y si se quiere cambiar los valores predeterminados en tiempo de ejecución, se debe asignar las propiedades en el prototipo del constructor, no en la función constructor misma.

2.5.2. Determinando relaciones de instancia

Es posible que se quiera conocer qué objetos están en la cadena prototipo de un objeto, para que se pueda determinar de qué objetos este objeto hereda sus propiedades.

A partir de JavaScript 1.4, el interprete proporciona un operador *instanceof* para probar la cadena prototipo. Este operador funciona exactamente como la función *instanceOf* que se describe en esta página.

Como se discutió en la sección 2.4.2.1 en la página 27, cuando se usa el operador *new* con una función constructor para crear un nuevo objeto, JavaScript asigna la propiedad *__proto__* del nuevo objeto al valor de la propiedad de la función constructor. Se puede usar esto para probar el encadenamiento de prototipos.

Por ejemplo, supóngase que se tiene el mismo conjunto de definiciones mostradas en la Figura 2.7 en la página 31. Se crea un objeto como sigue:

```
ing = new Ingeniero("Omar O.R.", ["psx"], "vlad");
```

Con este objeto, todas las siguientes sentencias son verdaderas:

```
ing.__proto__ == Ingeniero.prototype;
ing.__proto__.__proto__ == Obrero.prototype;
ing.__proto__.__proto__.__proto__ == Empleado.prototype;
ing.__proto__.__proto__.__proto__.__proto__ == Object.prototype;
ing.__proto__.__proto__.__proto__.__proto__.__proto__ == null;
```

Sabiendo esto, se puede escribir una función *instanceOf* como la siguiente:

```
function instanceOf(object, constructor)
{
  while (object != null)
  {
    if (object == constructor.prototype)
      return true;
    object = object.__proto__;
  }
  return false;
}
```

Con esta definición, todas las siguientes expresiones son ciertas:

```
instanceOf (ing, Ingeniero)
instanceOf (ing, Obrero)
instanceOf (ing, Empleado)
instanceOf (ing, Object)
```

Pero la siguiente expresión es falsa:

```
instanceOf (ing, Vendedor)
```

2.5.3. Información global en los constructores

Cuando se crean constructores, hay que ser muy cuidadosos si se asigna información global en el constructor. Por ejemplo, supóngase que se desea un Id único que sea automáticamente asignado a cada nuevo empleado. Se podría usar la siguiente definición para *Empleado*:

```
var idContador = 1;
function Empleado (nombre, depto){
  this.nombre = nombre || “”;
  this.depto = depto || “general”;
  this.id = idContador++;
}
```

Con esta definición, cuando se crea un nuevo *Empleado*, el constructor asigna el siguiente Id en secuencia y después incrementa el contador global Id. De este modo, en las siguientes sentencias *emp1.id* es 1 y *emp2.id* es 2:

```
emp1 = new Empleado(“Lisa S.”, “investigación”);
emp2 = new Empleado(“Bart S.”, “limpieza”);
```

A primera vista parece bien. Sin embargo, *idContador* se incrementa cada vez que un objeto *Empleado* es creado, para cualquier propósito. Si se crea la jerarquía completa de *Empleado* mostrada en este capítulo, el constructor de *Empleado* es llamado cada vez que se asigna un prototipo. Supóngase que se tiene el listado 3.

Listado 3 Información. global en los constructores

```
var idContador = 1;
function Empleado (nombre, depto){
  this.nombre = nombre || “”;
  this.depto = depto || “general”;
  this.id = idContador++;
}
Empleado.prototype = new Empleado();
function Gerente(nombre, depto, reportes) {...}
Obrero.prototype = new Empleado();
function Obrero(nombre, depto, projs) {...}
Ingeniero.prototype = new Obrero();
function Ingeniero(nombre, projs, maq) {...}
Vendedor.prototype = new Obrero();
function Vendedor(nombre, projs, cuota) {...}
ing = new Ingeniero(“Omar O. R.”);
```

Además supóngase que las definiciones omitidas aquí tienen la propiedad *base* y llaman al constructor de nivel superior mediante el encadenamiento de prototipos. En este caso, para cuando el objeto *ing* sea creado, *ing.id* será 3.

Dependiendo de la aplicación, puede o no importar que el contador haya sido incrementado de más. Si es importante el valor exacto de este contador, una posible solución es el siguiente constructor:

```
function Empleado(nombre, depto){
  this.nombre = nombre || “”;
  this.depto = depto || “general”;
  if(nombre)
    this.id = idCounter++;
}
```

Cuando se crea una instancia de *Empleado* para usarse como prototipo, no se pasan argumentos al constructor. Usando esta definición del constructor, cuando no se pasan argumentos, el constructor no asigna un valor al id y no actualiza el contador. Por lo tanto, para que un *Empleado* tenga asignado un id, se debe especificar un nombre para el empleado. En este ejemplo, *ing.id* sería 1.

2.5.4. No existe la herencia múltiple

Algunos lenguajes orientados a objetos como C++ y Python permiten la herencia múltiple. Es decir, un objeto puede heredar las propiedades y valores de objetos base sin relación alguna. JavaScript no permite herencia múltiple.

La herencia de valores de propiedad ocurre en tiempo de ejecución. JavaScript busca la cadena prototipo de un objeto para encontrar el valor. *Debido a que un objeto tiene un único prototipo asociado, JavaScript no puede heredar dinámicamente de más de una cadena prototipo.*

En JavaScript, se puede tener una función constructor que llame a más de una función constructor desde él. Esto da la ilusión de herencia múltiple. Por ejemplo, considere el código del listado 4.

Listado 4 No existe herencia múltiple

```
function Pasatiempo(hobby){
  this.hobby = hobby || “dibujo”;
}
Ingeniero.prototype = new Obrero;
function Ingeniero(nombre, projs, maq, hobby){
  this.base1 = Obrero;
  this.base1(nombre, “ingeniería”, projs);
  this.base2 = Pasatiempo;
  this.base2(hobby);
  this.maquina = maq || “”;
}
ing = new Ingeniero(“Omar O. R.”, [”SydPloX”], “lemuria”];
```

Además supóngase que la definición de Obrero no ha cambiado. En este caso, el objeto *ing* tiene las siguientes propiedades:

```
ing.nombre == “Omar O. R.”
ing.depto == “ingeniería”
ing.proyectos == [”SydPloX”]
ing.maquina == “lemuria”
ing.hobby == “dibujo”
```

De este modo, *ing* obtiene la propiedad *hobby* del constructor *Pasatiempo*. Sin embargo, supóngase que después se añade una propiedad al prototipo del constructor *Pasatiempo*:

```
Pasatiempo.prototype.deportes = [”basket ball”, “box”, “fútbol”]
```

El objeto *ing* no heredará esta nueva propiedad.

Capítulo 3

Fundamentos de SVG

SVG es un nuevo formato de archivos gráficos y un lenguaje de desarrollo Web basado en XML. SVG habilita a los desarrolladores y diseñadores Web a crear dinámicamente gráficos de alta calidad con datos en tiempo real con estructura precisa y control visual.

Con esta innovadora tecnología, los desarrolladores SVG pueden crear una nueva generación de aplicaciones Web basado en gráficos manejadores de datos, interactivos y personalizados.

SVG ofrece las siguientes ventajas y posibilidades:

1. Tiene todas las ventajas asociadas a un formato vectorial: es escalable, compacto, con formas simples editables a través de *curvas Bézier*[1], con entornos suavizados, transparencias, La calidad de colores es excelente, y capaz de incluir, si es preciso, mapas de píxeles.

2. El tamaño de los SVG es muy compacto.

3. El texto que incluyen es editable: admite las *fuentes escalables*[2]. Esto es una diferencia enorme con los actuales GIF o JPG: el texto que contiene se puede editar, seleccionar, ser indexado por los buscadores, etc.

4. El archivo SVG no es binario: se trata de un fichero de texto plano. Esto significa que se puede editar en un simple editor de texto, y sus contenidos se pueden indexar, buscar, utilizar *sistemas de control de versión*[3], etc.

5. Es compatible con los estándares actuales de la Web (e.g. CSS, *XSL*[4] o DOM), y lo más importante, con los futuros.

6. SVG trabaja efectivamente con hojas de estilo para controlar los elementos y atributos de presentación separando la estética del gráfico del contenido. CSS puede ser usado para características de fuente (tamaño, familia, color, etc.), y también para propiedades de elementos gráficos. Por ejemplo, se puede controlar y cambiar el color del trazado, el color de relleno, y la opacidad de un rectángulo desde una hoja de estilo externa.

---

[1] Una forma de dibujar líneas curvas en términos de un punto inicial y un punto final, y un número de puntos de control. SVG soporta curvas Bézier cuadráticas (dos puntos de control) y curvas cúbicas Bézier (tres puntos de control). [26, p. 1060]

[2] Las fuentes escalables más comunes son *TrueType* y *Type 1* (también llamada *Postscript type 1*). Ambos tipos de fuentes contienen la información para los caracteres en forma de trazados vectoriales.

[3] El control de versión es un proceso automatizado para administrar los cambios hechos en los archivos de código fuente. [25, p. 120]

[4] e**X**tensible **S**ytle **L**anguage es un mecanismo para aplicar estilos de formato a los documentos XML. [19, p. 130]

7. SVG tiene soporte completo *Unicode*[5] para mostrar texto en muchos lenguajes, verticalmente, horizontalmente y bidireccional, logrando una riqueza tipográfica sin precedentes. Un desarrollador de contenido puede fácilmente incrustar una fuente inusual, habilitando que el texto sea mostrado como debe, sin suposiciones sobre las fuentes disponibles del usuario.

8. Se puede crear gráficos interactivos mediante la inclusión de código (scripts) que modifican el gráfico dinámicamente.

9. Al ser un dialecto XML, es un formato extensible: los fabricantes podrán adoptarlo como formato nativo de sus aplicaciones, añadiendo las características específicas que deseen, pero siempre mantendrá la compatibilidad básica y universal con toda aplicación que reconozca el formato.

10. Los efectos visuales, como *rollover*[6] y comportamientos, se manipulan fácilmente vía script controlando color, forma, tamaño, texto u opacidad.

11. Puede generarse dinámicamente en un servidor Web por medio de lenguajes del lado del servidor como JSP, PHP o Perl. Por ejemplo, pueden crearse al momento gráficos de excelente calidad con las cotizaciones de bolsa en tiempo real; un reloj analógico, con minutos y segundos.

12. SVG permite que el flujo de trabajo, contenido (datos), presentación (gráficos), y lógica (programación), sean desarrollados en paralelo, reduciendo tiempo de desarrollo y distribuyendo el trabajo más eficientemente. Separando los elementos de este flujo de trabajo, SVG habilita a los desarrolladores y a los diseñadores a enfocarse en su área de trabajo

3.1. Estructura de un documento SVG

Para manipular un documento SVG, es necesario conocer su estructura.

3.1.1. Sintaxis

En la mayoría de los lenguajes de marcado, las etiquetas se relacionan con etiquetas específicas; por ejemplo la etiqueta *table* instruye específicamente al navegador que dibuje su contenido dentro de una tabla. En XML, las etiquetas sólo indican la manera en que se almacenan los datos y no como se visualizan.

Las tres reglas básicas de la sintaxis XML y SVG (para asegurar documentos bien formados) son las siguientes[16, p. 23]:

1. Todas las etiquetas son caso sensitivo, es decir, si se designa <aa> como un elemento, <aA> se referirá a un elemento diferente.

2. Todas las etiquetas deben estar cerradas, es decir, las etiquetas deben seguir una de las siguientes dos convenciones: <etiqueta>...</etiqueta> o <etiqueta />.

3. Todos los valores de atributo deben estar entre comillas simples o dobles, e.g., <etiqueta atributo=”valor” atributo=’valor’/>.

[5] Los caracteres Unicode son representados por valores de 16 bits (0-65535) y, por lo tanto, pueden manejar casi cualquier tipo de caracter imaginable.

[6] Efectos que se ocurren cuando el usuario activa un evento de ratón sobre el objeto. Requieren de por lo menos dos imágenes, una para cuando se pasa el ratón sobre el objeto y otra para cuando se quita el ratón.

Aparte de tener un documento XML o SVG bien formado, estos documentos deben ser validos. Una forma de hacer que un documento XML sea valido es agregándole un DTD[7].

Existen dos términos muy importantes en la sintaxis XML[16, p. 23-24]:

- Elemento. El *nombre del tipo de elemento* puede considerarse como el nombre de la etiqueta. Determina cómo funcionará la etiqueta y sí consiste de una etiqueta vacía (o cerrada), o de dos etiquetas (una de apertura y una de cierre).
- Atributo. Un atributo describe un elemento proporcionándole información adicional. Las especificaciones de atributo consisten de un nombre de atributo y un valor de atributo.

Por ejemplo:

```
<circle r="50" />
```

circle es el nombre del tipo de elemento

r es el nombre del atributo

50 es el valor del atributo.

3.1.1.1. El atributo *id*

El atributo ***id*** nombra al elemento que lo describe; en muchos casos, actúa como el "rótulo" de un elemento. La sintaxis es muy simple:

```
id="Identificador"
```

donde *Identificador* es cualquier texto. El valor del atributo *id* se ajusta a la convención de nombramiento de XML. Cualquier valor *id* puede contener sólo caracteres alfanuméricos, guiones bajos, puntos y guiones; los espacios o los caracteres especiales no están permitidos. Sin embargo, los valores del atributo *id* no pueden empezar con un punto, guión, o número.

La importancia del atributo *id* es que se puede nombrar elementos de un modo descriptivo, a fin de facilitar la búsqueda de elementos en un documento. También, muchas funciones interactivas de SVG necesitan de un valor *id* para saber a quien se aplica la función. Sin el atributo *id* y su valor asociado, los documentos SVG serían muy estáticos.[16, p. 24]

3.1.2. Estructura

La estructura de un documento SVG es una serie de elementos, algunos de los cuales encierran otros elementos.

En cada documento SVG, el contenido está encerrado entre las etiquetas del elemento *svg*: <svg>...</svg>. Esto es referido, algunas veces, como el elemento documento o raíz. Todo el contenido SVG debe aparecer dentro de estas dos etiquetas. Los únicos datos que debería aparecer fuera de estas dos etiquetas serán las definición de datos de un documento XML.

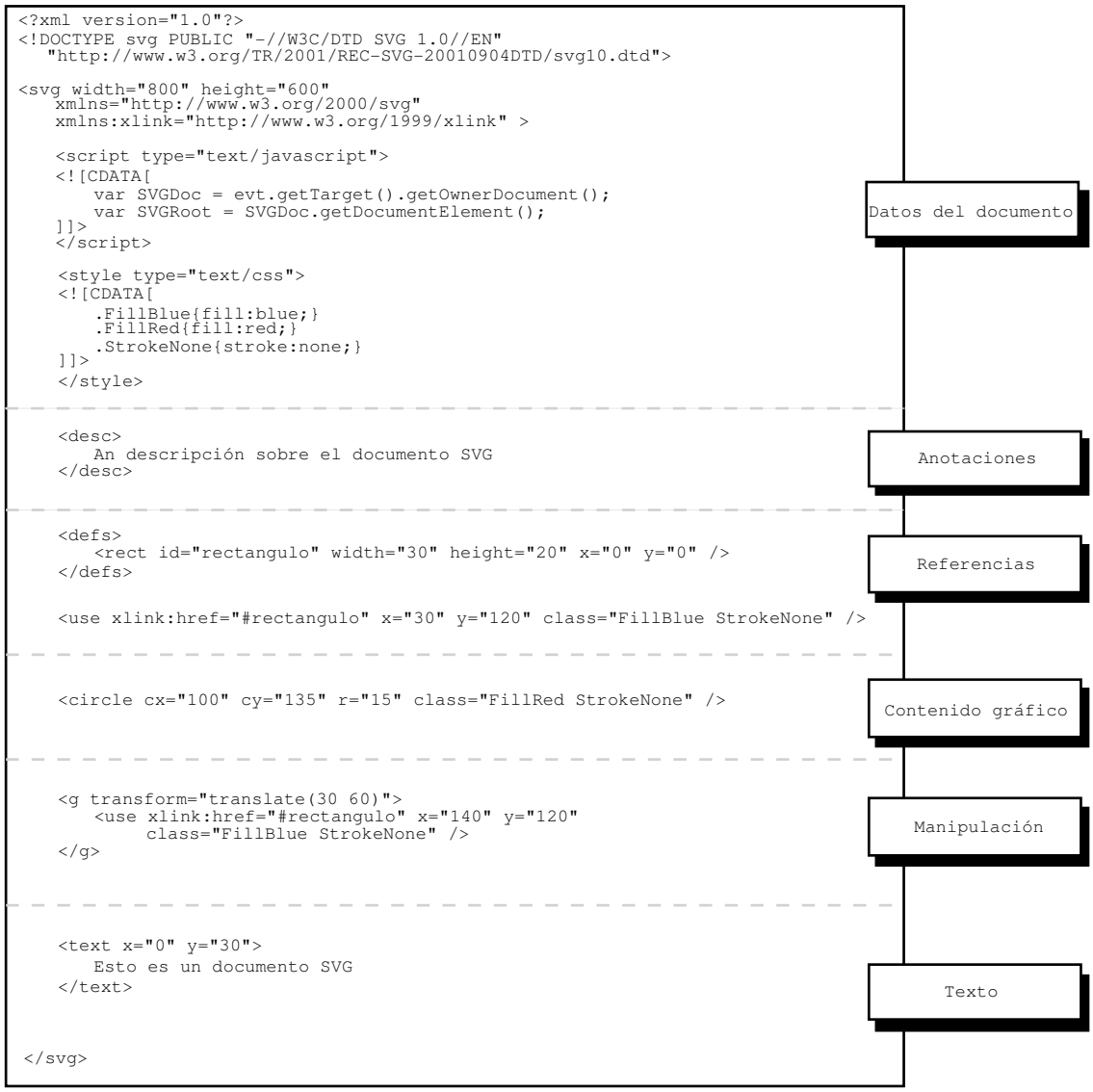
Aunque no existe una clasificación formal, el contenido SVG generalmente consiste de elementos pertenecientes a unos de las siguientes seis categorías[16, p. 25]:

[7] DTD (Document Type Definition, Definición de Tipo de Documento). Indica la estructura y la gramática de un documento.

- Datos del documento (definiciones de nombre de espacio, información de encabezado XML, hojas de estilo, scripting).
- Anotaciones (descripciones, comentarios).
- Material de referencia (definiciones, símbolos).
- Contenido gráfico (figuras básicas, curvas, mapas de píxeles).
- Manipulación de datos (animación, transformaciones, enmascaramiento).
- Texto

La Figura 3.1[16, p. 26] ilustra los seis tipos diferentes de información utilizados en un documento SVG.

Figura 3.1: Estructura de un documento SVG



En la Figura 3.1 las dos primeras líneas en la categoría “datos del documento” son comunes para algunos documentos XML, definiendo el código subsecuente como perteneciente a un lenguaje específico XML y referenciando un DTD.

Los elementos **<?xml version...>** y **<!DOCTYPE...>** al principio del archivo son usados para determinar si el siguiente código SVG es válido. La primera línea explica que el documento es una pieza de datos XML, mientras que la segunda línea proporciona una liga al **DTD** de SVG del W3C.

Un DTD permite al código XML describir su contenido para verificarse siempre que el documento sea analizado, por lo tanto, SVG requiere su propio DTD. El DTD de SVG es usado para validar el documento SVG revisando ítems como la estructura, los elementos, y sus atributos (y correspondientes valores).

Debido a que el número de aplicaciones basadas en XML se ha incrementado, es necesario indicar de algún modo a qué aplicaciones pertenecen cada elemento. Para distinguir elementos nombrados del mismo modo unos de otros, los documentos XML utilizan el *nombre de espacio* (*name space*). Un nombre de espacio es, en esencia, un nombre de URI (Uniform Resource Identifier), es un atributo (***xmlns***) y se coloca en la etiqueta de inicio de un elemento raíz. En el caso de un documento SVG está dado por:

```
<svg xmlns="http://www.w3.org/2000/svg">
```

Todas las ligas en SVG dependen de **XLink** (XML Linking Language). Todas la ligas internas y externas, incluyendo aquellas que utilizan fragmentos de identificadores, hacen uso de los atributos de XLink. La declaración explícita del nombre de espacio de XLink no es necesaria porque la declaración está incluida en el DTD de SVG, pero si el visor SVG no tiene acceso al DTD, será necesario incluirlo dentro del elemento *svg*, i.e.,

```
<svg xmlns:xlink="http://www.w3.org/1999/xlink">
```

El elemento **<script>** sirve para incluir código de programación. El código puede estar contenido entre las etiquetas <script> y </script>, o ligado a un archivo externo (mediante el atributo xlink:href). El elemento script de SVG tiene el atributo *type* para indicar el lenguaje que se va a utilizar, e.g., "text/javascript" o "text/ecmascript". Todo el código dentro de las etiquetas <script> y </script> debe estar contenido en la sección CDATA, por razones de sintaxis.

El elemento **<style>** sirve para definir una CSS. Al igual que el código de scripts, las definiciones CSS deben estar en la sección CDATA.

La sección **CDATA** comienza con **"<**![CDATA[" y termina con **"]]>**". Todo lo que esté contenido dentro de estos dos delimitadores no será analizado como código XML. El contenido de una sección CDATA es código JavaScript o definiciones CSS, cuya sintaxis es muy diferente a la de XML, y es procesado por ellos mismos.

El elemento **<desc>** sirve para proporcionar descripciones de un documento SVG, un elemento individual y su propósito, comentarios, etc.; esta información no se despliega.

El elemento **<defs>** contiene definiciones. Tales definiciones típicamente serán referenciadas, y posiblemente reutilizadas muchas veces, por elementos que aparezcan más adelante en un documento SVG. Los elementos anidados dentro de un elemento <defs>no serán desplegados, pero estarán disponibles para ser referenciados por otros elementos.

La mayoría de los elementos pueden ser reutilizados mediante el elemento **<use>**. El elemento <use> puede hacer referencia a diferentes elementos como figuras básicas o complejas, agrupaciones, mapas de píxeles y símbolos utilizando el atributo *xlink:href* con valor *#id*. El elemento referenciado es desplegado en la posición y tamaño definido por los atributos (x, y, width, height) del elemento <use>.

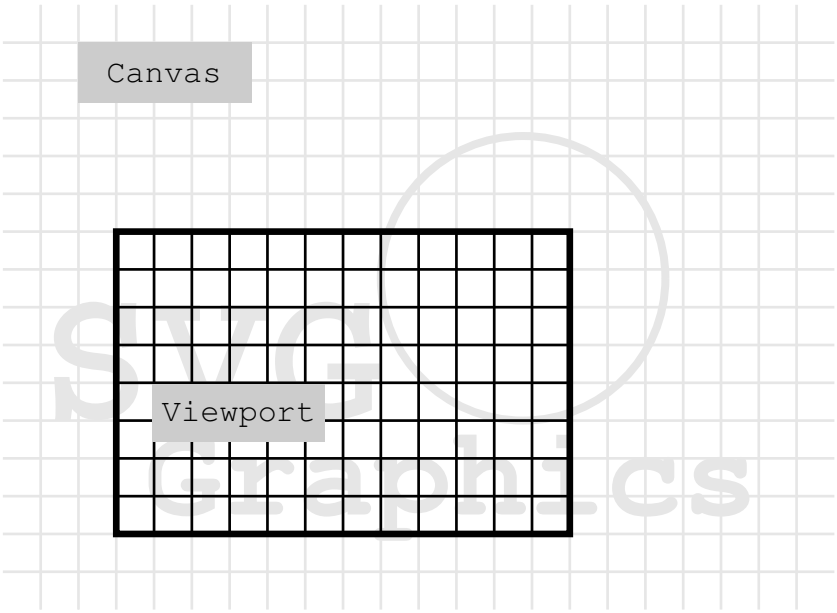
El elemento **<g>** tiene como propósito agrupar elementos SVG, de tal modo que todo lo que contenga se considera como un solo elemento. Un elemento <g> puede contener elementos gráficos (figuras básicas o complejas y texto), también puede contener elementos <desc> para proporcionar información.

3.2. Coordenadas

3.2.1. Canvas y Viewport

Los gráficos SVG están contenidos en un "hipotético espacio bidimensional infinito" (*canvas*). Antes de que se defina el contenido de un archivo SVG, es necesario definir el tamaño del documento SVG. El área para graficar que el documento intenta usar se llama *viewport*. Se establece el tamaño de este *viewport* con los atributos *width* y *height* del elemento *svg*. La Figura 3.2[1, p. 53] ilustra esto.

Figura 3.2: Colocación de un *viewport* en un *canvas*



Los atributos *width* y *height* pueden especificarse en unidades numéricas, y los documentos SVG pueden usar un número de unidades de medición para definir la posición y resolución. La unidad predeterminada de medida en SVG es el "*valor de espacio de usuario"* (*user space value*), pero SVG también soporta una variedad alternativa de unidades.

Un ***valor de espacio de usuario*** es una unidad abstracta en la que la recomendación describe como el entorno del documento antes de que sea analizado para concordar con los requerimientos del dispositivo de visualización. Las unidades son intencionalmente abstractas para que un documento SVG pueda ser portado fácilmente a diferentes "espacios", como monitores de computadora, pantallas de teléfonos celulares, o una hoja de papel. En monitores de computadora, esta unidad se convierte en *píxel*, como la unidad predeterminada. La Tabla 3.1 muestra cada una de las unidades disponibles, su nomenclatura en SVG, y sus valores en píxeles[16, p. 40-41].

También se pueden usar porcentajes para definir el ancho y alto de un documento. Cuando la resolución de la ventana del navegador cambia, las dimensiones del documento SVG cambiarán en proporción a los nuevos valores.

3.2.2. Sistema de coordenadas SVG

El sistema de coordenadas SVG es diferente al sistema de coordenadas cartesiano, porque la intersección de los ejes y la dirección en que se incrementan las unidades difiere. Como SVG es un lenguaje para gráficos bidimensionales, todas sus unidades de medición se reflejan en dos ejes. Igual que en geometría, el *eje **X***

Tabla 3.1: Unidades admitidas en SVG

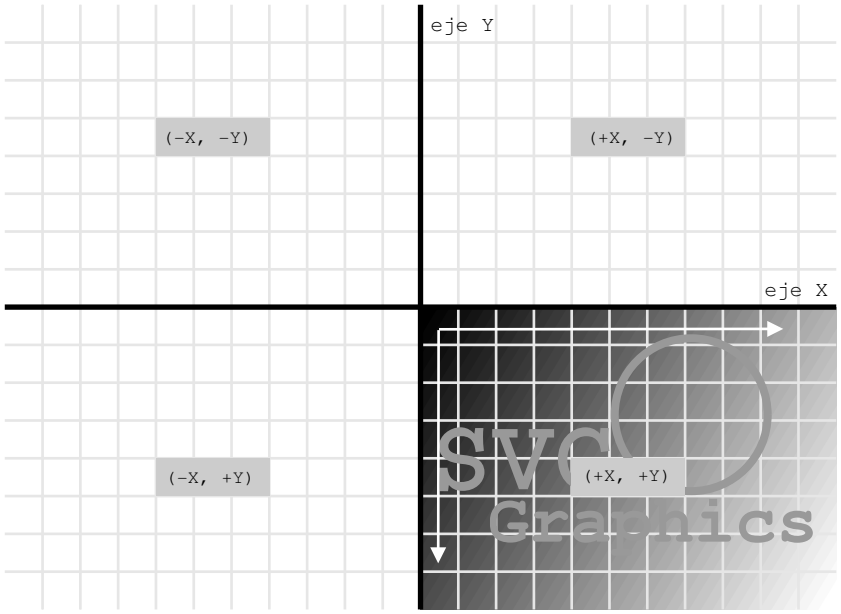
Nombre de la Unidad	Nomenclatura SVG	Valor en píxeles
Píxel	px	1
Punto	pt	1.25
Pica	pc	15
Milímetros	mm	3.543307
Centímetros	cm	35.43307
Pulgadas	in	72

representa el eje horizontal, y el *eje **Y*** representa el eje vertical. Para representar las coordenadas se utiliza la notación (X,Y).

El W3C definió la intersección de los dos ejes en el punto (0,0), y este punto está la esquina superior izquierda del *viewport*. Las unidades en ***X*** se incrementan positivamente hacia la derecha y viceversa. Similarmente, las unidades en ***Y*** se incrementan positivamente hacia abajo y viceversa.

La Figura 3.3 ilustra cómo funciona el sistema de coordenadas en SVG. El cuadrante inferior derecho que está en degradado refleja la orientación de la esquina superior izquierda de cualquier documento SVG, mientras que las flechas blancas muestran los valores positivos de ***X*** y ***Y***.[16, p. 42]

Figura 3.3: Sistema de coordenadas SVG



3.3. Figuras, texto, estilos y máscaras

3.3.1. Figuras básicas

SVG proporciona diversos elementos que describen figuras gráficas usadas comúnmente.

La Tabla 3.2 resume las figuras básicas disponibles en SVG.

En todos, excepto los dos últimos elementos de la Tabla 3.2, se pueden especificar los atributos como simples números, en cuyo caso las unidades de medición serán *valor de espacio de usuario*, o se puede

Tabla 3.2: Elementos de figuras básicas

Figura	Descripción	Elemento
Línea	Dibuja una línea desde un punto (x1, y1) hasta el punto (x2, y2), los cuales son atributos	<line x1=”inicio” y1=”inicio” x2=”fin” y2=”fin” />
Rectángulo	Dibuja un rectángulo cuya esquina superior izquierda es (x, y) y su tamaño es *width* y *height*. Las esquinas redondeadas se logran mediante los atributos *rx* y *ry*	<rect x=”izq” y=”arriba” width=”ancho” height=”alto” />
Círculo	Dibuja un círculo con un radio *r* dado y centrado en el punto (cx, cy)	<circle cx=”centro-x” cy=”centro-y” r=”radio” />
Elipse	Dibuja una elipse con dos radios dados *rx* y *ry*, y centrado en el punto (cx, cy)	<ellipse cx=”centro-x” cy=”centro-y” rx=”radio-x” ry=”radio-y” />
Polilínea	Dibuja un conjunto arbitrario de líneas conectadas descritas por un conjunto de puntos. Los puntos están especificados como pares ordenados (x y) y separados por comas	<polyline points=”x y, x y, ..., x y” />
Polígono	Dibuja un polígono formado por líneas conectadas entre sí descritas por un conjunto de puntos. Los puntos están especificados como pares ordenados (x y) y separados por comas	<polygon points=”x y, x y, ..., x y” />

añadir unidades específicas como cm, pt, in, etc. Por ejemplo:

```
<line x1=”1cm” y1=”30” width=”50mm” height=”10pt”/>
```

3.3.2. Figuras complejas

Las figuras básicas están limitadas a formar únicamente la figura que describen. El elemento *path* puede crear cualquier tipo de figura bidimensional.

Un *path* es una figura arbitraria que puede ser rellenada y trazada. Un *path* puede consistir de líneas rectas, líneas curvas o combinación de ambas. Puede ser una figura abierta o cerrada. Debido a que el elemento *path* puede soportar potencialmente la creación de un gran número de figuras, la sintaxis del elemento *path* es algo compleja.

La Tabla 3.3[4, p. 93-94] resume la sintaxis del elemento *path*.

Tabla 3.3: Comandos del elemento *path*

Comando	Argumentos	Efecto
M m	x y	Mueve a la coordenada indicada
L l	x y	Dibuja una línea en las coordenadas indicadas
H h	x	Dibuja una línea horizontal indicada por la coordenada x
V v	y	Dibuja una línea vertical indicada por la coordenada y
A a	rx ry rotación-eje-x longitud-arco dirección x y	Dibuja un arco elíptico desde el punto actual al punto (x, y). Los puntos están en una elipse con radio-x (*rx*) y radio-y (*ry*). La elipse está rotada (rotación-eje-x) grados. Sí el arco es menor que 180 grados, la *longitud-arco* es uno. Sí el arco se dibuja en dirección positiva, *dirección* es uno, de lo contrario es cero.
Q q	x1 y1 x y	Dibuja una curva Bézier cuadrática desde el punto actual al punto (x, y) usando un punto de control (x1, y1).
T t	x y	Dibuja una curva Bézier cuadrática desde el punto actual al punto (x, y). El punto de control será la reflexión del punto previo de control Q. Si no hay una curva previa, el punto actual será utilizado como el punto de control.
C c	x1 y1 x2 y2 x y	Dibuja una curva Bézier cúbica desde el punto actual al punto(x, y) usando el punto de control (x1, y1) para el inicio de la curva y (x2, y2) como el punto de control para el final de la curva.
S s	x2 y2 x y	Dibuja una curva Bézier cúbica desde el punto actual al punto (x, y) usando (x2, y2) como el punto de control para este nuevo punto terminal. El primer punto de control será la reflexión del previo punto de control C. Si no hay curva previa, el punto actual será usado como el primer punto de control.

3.3.3. Texto

SVG proporciona cuatro elementos para la distribución de texto: *text*, *tspan*, *tref*, y *textPath.* En esta tesis sólo se explicarán los dos primeros debido a que son los más utilizados.

Elemento *text*

El elemento *text* es el más esencial para el trazado de texto en SVG, todo el texto que esté entre las etiquetas <text>...</text> será desplegado. Todas las distribuciones de texto utilizan el elemento *text*, ya sea solo o en combinación con otros elementos de texto relacionados.

El despliegue de texto en pantalla requiere que se defina la colocación del texto mediante los atributos ***x*** y ***y*** del elemento texto. El comportamiento predeterminado es que el valor de ***x*** define el lado izquierdo del texto, y el valor de ***y*** define la *linea base* de los caracteres. [26, p. 310]

Elemento *tspan*

El elemento *tspan* proporciona una alternativa más accesible de contener texto. Este elemento siempre está anidado dentro del elemento *text*; no puede ser usado solo. La posición de un elemento *tspan* está dada por los atributos ***x*** y ***y*** (para definir una posición absoluta en la página) o por los atributos ***dx*** y ***dy*** (para definir una posición relativa a su elemento texto padre o el precedente elemento *tspan*).

3.3.4. Estilos

SVG proporciona varias técnicas para alterar la apariencia visual de cualquier figura gráfica o texto. Por ejemplo, hacer una línea sólida o punteada, delgada o gruesa, un contorno o relleno de cualquier color opaco o transparente. Se puede especificar el tamaño, color, tipo y estilo de una fuente. Todos estos aspectos de presentación de un objeto SVG pueden ser definidas usando una o más de las siguientes formas de manipular la aplicación de estilos:

- Estilos en línea
- Estilos internos
- Estilos externos
- Presentación de atributos individuales

Los estilos en SVG están basados en la especificación CSS (Cascade Style Sheet, Hojas de Estilo en Cascada), la cual también es producida por el W3C.

3.3.4.1. Estilos en línea

En los estilos en línea, que se aplican individualmente a los elementos, se asigna como valor al atributo *style* una serie de propiedades visuales junto con sus valores de la siguiente manera:

```
<elemento style="propiedad1:valor; propiedad2:valor;...;propiedadN:valor" />
```

3.3.4.2. Estilos internos

Si se desea el mismo estilo para una serie de elementos, es más conveniente crear una definición de una *hoja de estilo* (*stylesheet*) internamente. Además se pueden utilizar selectores de clase (mediante el atributo *class*), selectores individuales (mediante el atributo *id*), aplicar diferentes clases a un mismo elemento y sobrescribir una propiedad en línea. Una hoja de estilo interna se define entre las etiquetas <style>...</style>, el Listado 5 muestra un ejemplo.

Listado 5 Hoja de estilo interna

```
<style type="type/css">
<![CDATA[
  .advertencia { fill:yellow; stroke:#FF0000;}
  .transparencia {fill-opacity:0.25;}
  circle {stroke-width:4;}
  #texto1 {font-style:italic; font-size:20px;
     stroke-opacity:0.5;}
]]>
</style>
```

3.3.4.3. Estilos externos

Si se desea aplicar un conjunto de estilos a múltiples documentos SVG, los más conveniente es escribir las definiciones de estilo en un archivo externo con extensión *.css* (sin las etiquetas <style type="text/css"> <![CDATA[...]> </style>. Para llamar una hoja de estilo externa, se agrega una línea en el encabezado de la información del documento[16, p. 93]:

```
<?xml version="1.0" ?>
<?xml-stylesheet href="ubicación_archivo.css" type="text/css" ?>
```

3.3.4.4. Presentación de atributos individuales

A pesar de las ventajas que ofrecen los estilos, en cualquiera de sus modalidades, para la presentación de información, SVG permite especificar la información en forma de atributos de presentación, i.e, cada propiedad se puede escribir como un atributo. E.g.,

```
<elemento fill:"red" stroke="black" stroke-width="2" />
```

3.3.4.5. Especificación de colores

Es posible especificar colores de relleno o de línea exterior de una figura en cualquiera de las siguientes formas[4, p. 42]:

- ***none***, indica que ninguna línea exterior será dibujada, o que no tendrá relleno.
- **Un *nombre de color***, el cual puede ser: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, o yellow.
- ***Seis dígitos hexadecimales* #RRGGBB**, cada par describe los valores rojo, verde y azul.
- ***rgb(r, g, b)***, cada valor en el rango 0-255 o 0 % a 100 %.

3.3.4.6. Características de trazados y rellenos

Para poder ver una línea o la línea externa de una figura, se debe especificar características de trazado (stroke). Una línea exterior de una figura se dibuja después de que su interior ha sido rellenado. Todas estas características, mostradas en la Tabla 3.4[4, p. 43], son propiedades de estilizado, y van en el atributo *style*.

Tabla 3.4: Características de trazado

Atributo	Valores
stroke	El color de trazado como se describe en la sección 3.3.4.5.
stroke-width	Grosor del trazado; puede estar dado en coordenadas de usuario o con una unidad específica.
stroke-opacity	Un número entre el rango 0.0-1.0; 0.0 es completamente transparente, 1.0 es completamente opaco.
stroke-dasharray	Una serie de números que indican la longitud de los guiones y espacios, con los cuales una línea es dibujada. Estos números están en coordenadas de usuario.
stroke-linecap	Figura del final de una línea; tiene uno de los siguientes valores: *butt* (el predeterminado), *round*, o *square*.
stroke-linejoin	La figura de las esquinas de un polígono o un polilínea; tiene uno de los siguientes valores: *miter* (punta, el predeterminado), *round* (redondo), o *bevel* (plano).
stroke-miterlimit	Máxima proporción de longitud del punto de intersección al grosor de las líneas que se dibujan; el valor predeterminado es 4.

Se puede controlar la forma en que el interior de una figura se rellena usando de los atributos de relleno que se muestran en la Tabla 3.5. Una figura se rellena antes de que su línea exterior sea dibujada.

Tabla 3.5: Características de relleno

Atributo	Valores
fill	El color de relleno, como se describe en la sección 3.3.4.5.
fill-opacity	Un número en el rango 0.0-1.0; 0.0 es completamente transparente, 1.0 es completamente opaco.

3.3.5. Máscaras

Una máscara en SVG es exactamente lo contrario de una máscara que se usa en una fiesta de disfraces. Con una máscara de fiesta de disfraces, las partes que son opacas ocultan el rostro; las partes que son translúcidas permiten ver el rostro borrosamente, y los hoyos (los cuales son transparentes) permiten ver el

rostro claramente. Una máscara SVG, por otra parte, transfiere su transparencia al objeto que enmascara. Donde la máscara es opaca, los *píxeles* del objeto enmascarado son opacos. Donde la máscara es translúcida, también es el objeto, y las partes transparentes de la máscara hacen las partes correspondientes del objeto enmascarado invisibles. [4, p. 141]

Se utiliza el elemento <mask> para crear una máscara. Se pueden especificar la resolución de la máscara con los atributos *x*, *y*, *width*, y *height*. Este elemento debe estar en la sección *defs* y debe tener el atributo *id* para que pueda ser utilizado más tarde.

Entre la etiquetas inicial <mask> y final </mask> están cualquier forma básica, texto o curvas que se deseen utilizar como máscaras.

La transparencia está determinada por el atributo *fill* y/o *opacity* de los elementos que conforman la máscara. Aquellos valores más cercanos al negro, son transparentes; los valores más cercanos al blanco, son opacos.

El Listado 6 muestra un ejemplo de máscara. Un círculo esta enmascarado por un cuadrado y sólo se ve una cuarta parte del círculo.

Listado 6 Máscara SVG

```
<defs>
  <mask id=”rectMask”>
    <rect x=”0” y=”0” width=”10” height=”10” style=”fill: #FFFFFF” />
  </mask>
</defs>
<g style=”mask: url(#rectMask)>
  <circle cx=”0” cy=”0” r=”10” />
</g>
```

3.4. Transformaciones

Las transformaciones son la característica que más maximiza las posibilidades de un simple objeto. Mientras el elemento *use* permite reutilizar un objeto múltiples veces, las transformaciones permiten modificar el sistema de coordenadas de un objeto. Aplicando translación, escalamiento, torsión, rotación, una transformación puede cambiar dramáticamente la apariencia original de un objeto.

La Tabla 3.6 resume las transformaciones disponibles en SVG.

Para aplicar transformaciones, se utiliza el atributo *transform* en los elementos gráficos que se desea modificar. El valor del atributo consiste de una combinación de propiedades de transformación y su valor(es). E.g.,

```
transform="tipoTransf(valor)"
```

donde *tipoTransf* es una de las transformaciones listadas en la Tabla 3.6, y *valor* es el parámetro(s) para modificar el elemento.

Se pueden aplicar múltiples transformaciones a un objeto asignando como valor al atributo *transform* varios tipos de transformaciones separados por un espacio en blanco. E.g.,

```
transform="translate(10, 10) rotate(15) scale(0.5)"
```

El orden de las transformaciones es de derecha a izquierda.

Tabla 3.6: Transformaciones en SVG

Transformación	Descripción
translate(x, y)	Mueve el sistema de coordenadas especificado por los parámetros (x, y). Si no se especifica un valor (y), se asume que es cero.
scale(xFactor, yFactor)	Multiplica el sistema de coordenadas especificado por los parámetros (xFactor, yFactor). xFactor multiplica al eje X y yFactor multiplica el eje Y. Los factores pueden ser fracciones o negativos.
scale(factor)	Multiplica el sistema de coordenadas de manera equitativa.
rotate(ángulo)	Rota el sistema de coordenadas especificado por el parámetro (ángulo). El centro de rotación es el origen (0, 0). La medida de los ángulos es en el sentido de las manecillas del reloj.
rotate(ángulo, cX, cY)	Rota el sistema de coordenadas especificado por los parámetros (ángulo, cX, cY). El centro de rotación es el punto (cX, cY).
skewX(ángulo)	Tuerce el sistema de coordenadas, pero únicamente el eje X especificado por el parámetro (ángulo).
skewY(ángulo)	Tuerce (sesga) el sistema de coordenadas, pero únicamente el eje X especificado por el parámetro (ángulo).

3.5. DOM de SVG

SVG es interactivo cuando se usa con un lenguaje de programación, como JavaScript. Sin embargo, el lenguaje necesita una forma de accesar a los elementos y atributos de SVG. El W3C resolvió este problema usando el DOM2 como base para el propio DOM de SVG. El DOM de SVG es una API que proporciona a cualquier lenguaje acceso a las partes del documento SVG, puede hacer, virtualmente, lo que sea: crear, eliminar, cambiar, o manipular los elementos.[26, p. 547]

El DOM de SVG extiende en algunas áreas el DOM2. E.g., el DOM de SVG extiende el soporte para más eventos de los que están definidos en el DOM2. Los eventos específicos de SVG no deben estar incluidos en el DOM2 porque entonces el DOM dejaría de ser una interfaz independiente.[26, p. 548]

3.5.1. Módulos importantes del DOM para SVG

Los módulos más importantes del DOM para SVG son[26, p. 548]:

- **Núcleo**. Este módulo permite el acceso a todos los elementos de un documento a través de un sistema jerárquico con estructura de árbol.
- **CSS**. Incluye diferentes maneras de accesar y cambiar estilos.
- **Eventos**. Proporciona acceso a diferentes eventos, como las acciones del ratón.
- **SVG**. Son interfaces específicas para accesar a los elementos de SVG.

3.5.2. Módulo *Núcleo* del DOM2

En el DOM, los elementos de un documento están divididos en objetos. Estos objetos están arreglados en una estructura de árbol. El árbol consiste de nodos, y cada nodo representa un elemento, atributo, evento u objeto.

El nodo de la parte superior del árbol es el elemento *svg*. También es llamado elemento *raíz* porque, en una representación visual, el elemento superior es la raíz.

El Listado 7 ilustra la jerarquía de nodos

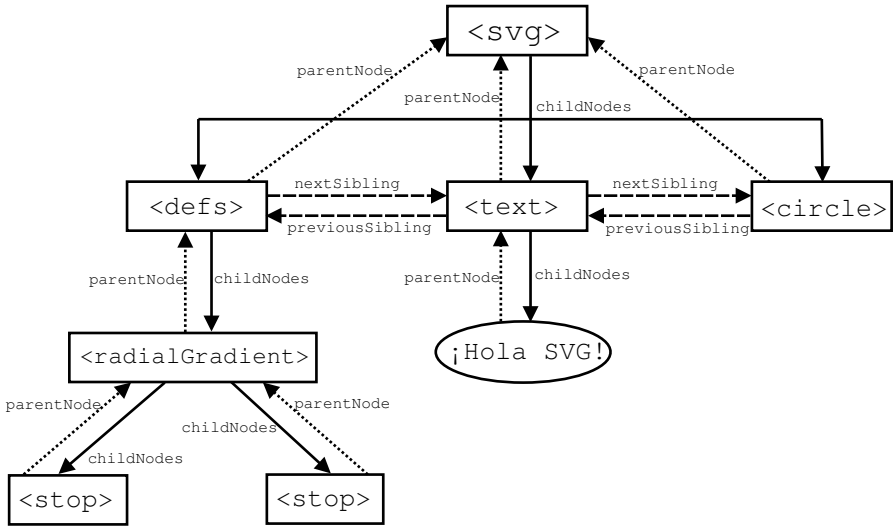
Listado 7 Un documento SVG

```
<?xml version="1.0" ?>
<svg width="300" height="150">
  <defs>
    <radialGradient id="gradiente"cx="70%" cy="30%" r="50%" fx="50%" fy="50%">
      <stop offset="0%" style="stop-color:green" />
      <stop offset="100% sytle="stop-color:black" />
    </radialgradient>
  </defs>
  <text x="15" y="40" id="text">
    ¡Hola SVG!
  </text>
  <circle id="circulo" cx="60" cy="80" style="fill:url(#gradiente)"/>
</svg>
```

En el módulo *núcleo*, cada nodo tiene propiedades. Con las propiedades, se puede recorrer la jerarquía del documento. La Tabla 3.7 muestra las propiedades de los nodos.[26, p. 549-550][22, p. 286]

La Figura 3.4 muestra la relación entre los diferentes elementos del Listado 7.

Figura 3.4: Relación entre elementos



Las diferentes propiedades sugieren el nombre de los métodos para accesar los nodos.

También se puede accesar directamente las propiedades del elemento dado. Sin embargo, SVG ofrece un método de accesar estas propiedades: sólo se antepone al atributo la palabra ***get*** y en mayúscula la

Tabla 3.7: Propiedades de los Nodos

Propiedad	**Descripción**
nodeName	Contiene el nombre del nodo
nodeValue	Contiene el valor del nodo (valor del atributo)
nodeType	Contiene un número correspondiente al tipo de nodo. Véase la tabla 1.1 en la página 10
parentNode	Una referencia al nodo padre del objeto actual, si hay alguno
childNodes	Contiene una lista con todos los nodos hijos del nodo actual. La lista es un arreglo. El índice comienza en 0.
firstChild	Referencia al primer nodo hijo del elemento, si hay alguno. Si un nodo no tiene hijos, el valor es 0
lastChild	Una referencia al último nodo hijo del elemento. Si un nodo no tiene hijos, el valor es 0
nextSibling	Referencia al siguiente hermano del nodo, en el mismo nivel (si es que existe).
previousSibling	Referencia al hermano anterior del nodo, en el mismo nivel (si es que existe).
ownerDocumnet	Señala al objeto **Document** en el que el elemento está contenido

siguiente letra. E.g., *firstChild* cambia a *getFirstChild()* y se obtiene el método. Del mismo modo funciona para asignar una propiedad, pero se antepone la palabra ***set***: firstChild cambia a setFirstChild(). [26, p. 551]

3.5.2.1. Diferentes tipos de nodos

En el DOM de SVG, existen los mismos tipos de nodos que en DOM[8], pero los más utilizados son:

- **Element**. Este tipo representa todos los elementos SVG, tales como figuras geométricas, gradientes, filtros, etc.
- **Attribute**. Este tipo cubre todos los atributos. Los atributos son, en la mayoría de los casos, hijos de elementos. Un atributo no puede tener hijos.
- **Text**. Este tipo contiene sólo texto. No puede tener hijos.
- **Document**. Este tipo representa el documento completo. Es la raíz del documento. Por otro lado, el elemento raíz es un hijo del nodo documento. Con el nodo documento, se pueden crear otros nodos.

3.5.2.2. Acceso a nodos

Accesando el documento

Para poder accesar a los elementos de SVG, lo primero que se debe hacer es accesar al propio documento SVG. Para accesar al documento se usa una variable (e.g. ***evt)*** para hacer referencia al objeto **Event**[9], a fin de tener acceso al evento que activo la ejecución del script. Si se llama a una función desde un *manejador de*

[8] Véase la sección 1.1 en la página 10

[9] Véase la sección 1.2.5.2 en la página 12

evento (*Event* es un argumento que cualquier elemento SVG que llama a una función recibe), una referencia a este objeto es enviada a la función llamada como parámetro.

Se deben completar dos pasos para obtener una referencia al documento SVG:

1. Llamar al método *getTarget()* o usar la propiedad *target* para obtener una referencia al elemento SVG que es donde se produjo el evento.

2. Llamar al método *getOwnerDocument()* o usar la propiedad *ownerDocument* para tener una referencia al objeto **SVGDocument** en el que está contenido el elemento.

Opcionalmente, para obtener una referencia al nodo raíz de la jerarquía de objetos del documento se llama al método *getDocumentElement()* o se usa la propiedad *documentElement* del objeto **SVGDocument**.

En resumen, el siguiente código asigna a la variable *SVGDoc* al actual documento SVG y a *SVGRoot* el elemento raíz de ese documento:

```
SVGDoc = evt.getTarget().getOwnerDocument();
SVGRoot = SVGDoc.getDocumentElement();
```

Es buena práctica de programación asignar estas variables como globales, desde que se carga el documento, para que haya acceso al documento SVG desde cualquier script. El Listado 8 muestra un ejemplo.

Listado 8 Variables globales SVGDoc y SVGRoot

```
<svg ... onload="init(evt)">
  <defs>
    <script type="text/javascript"><![CDATA[
      function init(evt){
        SVGDoc = evt.getTarget().getOwnerDocument();
        SVGRoot = SVGDoc.getDocumentElement();
      }
    ]]></script>
    ...
  </defs>
  ...
</svg>
```

Acceso a nodos

Para accesar los nodos existen diferentes formas:

- **Usando métodos y propiedades**[10]. E.g., usando la propiedad *nextSibling*, se accesa al nodo adyacente si existe. La propiedad regresa un objeto de tipo Nodo con el que se puede trabajar. Sin embargo, para lograr esto, de algún modo se debe accesar al nodo original "de la nada".

- **Usando *getElementById()*, *getElementsByTagName()*, y otros**. Estos métodos del objeto *Document* (o *SVGDocument*) regresan objetos nodo o listas de nodos.

  getElementById() toma el atributo *id* de un elemento como un parámetro y regresa una referencia al elemento. Por otro lado, *getElementsByTagName()* toma un nombre de etiqueta como parámetro (e.g., svg para <svg>) y regresa una lista de todos los elementos del documento actual que contienen ese nombre de etiqueta.

[10] Véase la tabla 3.7 en la página anterior

En la práctica, el acceso vía *getElementById()* es ampliamente usado, debido a que el acceso a un elemento en el documento a través del atributo *id* es muy práctico y directo. El acceso a través de la *interfaz Nodo* del DOM es complicada, debido a que requiere de una larga implementación para recorrer la estructura del árbol, además no ofrece muchas ventajas.

3.5.2.3. Modificación de nodos

Creación de nodos

El DOM soporta una variedad de métodos relacionados con la creación de nodos como parte del objeto **Document**, como muestra la Tabla 3.8.

Tabla 3.8: Métodos del DOM para crear nodos

Método	Descripción
createAttribute(*nombre*)	Crea un atributo para un elemento especificado por la cadena *nombre*.
createComment(*cadena*)	Crea un comentario de texto HTML/XML de la forma <!– *cadena* –>, donde *cadena* es el contenido del comentario.
createElement(*nombreDeEtiqueta*)	Crea un elemento del tipo especificado por el parámetro de cadena *nombreDeEtiqueta*.
createTextNode(*cadena*)	Crea un nodo de texto con el contenido *cadena*.

Inserción y adición de nodos

El objeto **Node** soporta dos métodos muy útiles para insertar contenido, **insertBefore(*newChild*, *referenceChild*)** y **appendChild(*newNode*)**. En el caso de **appendChild()**, se invoca como método del nodo al que se desea anexar un hijo, lo que añade el nodo referido por *newChild* al final de la lista de sus hijos. En el caso del método **insertBefore()**, se especifica delante de qué hijo queremos insertar *newChild* utilizando *referenceChild*.[22, p. 292]

Copia de nodos

Para copiar nodos existentes, el objeto **Node** proporciona el método **cloneNode()**, el cual creará una copia exacta del nodo actual y devolviendo una referencia del nodo copiado. El método toma solo un argumento de tipo lógico que indica si la copia debe incluir todos los hijos del nodo o sólo al elemento en sí. [18, p. 64][22, p. 294]

Eliminación y reemplazo de nodos

El objeto **Node** soporta el método **removeChild(*child*)** que se utiliza para eliminar un nodo de un padre dado especificado por la referencia ***child*** que se le pasa. Este método devuelve el objeto **Node** que ha eliminado.[18, p. 58-59][22, p. 295]

Además de eliminar un nodo, es posible reemplazar uno utilizando el método **replaceChild(*newChild*, *oldChild*)**, donde ***oldChild*** es el nodo a reemplazar con ***newChild***.[15, p. 59][22, p. 295]

Modificación de nodos de texto

El DOM también define varios métodos para actuar sobre los nodos de texto, éstos están resumidos en la Tabla 3.9. Un **textNode** tienen las propiedades **length** (que indica la cantidad de caracteres que contiene) y **data** para su valor.[22, p. 297]

Tabla 3.9: Métodos para manipulación de nodos de texto

Método	Descripción
appendData(*cadena*)	Añade la *cadena* pasada al final del nodo de texto
deleteData(*offset*, *cantidad*)	Borra la *cantidad* de caracteres empezando por el índice especificado por *offset*
insertData(*offset*, *cadena*)	Inserta el valor en la cadena empezando por el índice del carácter especificado por *offset*
replaceData(*offset*, *cantidad*, *cadena*)	Reemplaza la cantidad de caracteres de texto en el nodo empezando por *offset* con caracteres que corresponden a *cadena*
splitText(*offset*)	Divide el nodo de texto en dos piezas en el índice dado en *offset*. Devuelve la parte derecha de la división en un nuevo nodo de texto y deja la parte izquierda en el original
substringData(*offset*, *cantidad*)	Devuelve una cadena correspondiente a la subcadena que empieza por el índice *offset* y sigue hasta una *cantidad* de *caracteres*

3.5.2.4. Manipulación de atributos

El DOM soporta varios métodos de atributos para los elementos, donde se incluyen **getAttribute(attributeName)**, **setAttribute(attributeName, attributeValue)** y **removeAttribute(attributeName)**. En el DOM2 hay incluso un método de objetos **Node** muy útil, **hasAttributes()**, que se puede emplear para determinar si un elemento tiene atributos definidos.

3.5.3. Manipulación de CSS

El DOM2 CSS proporciona un módulo para todos los elementos de estilo de un documento SVG. El módulo consiste de diferentes interfaces, y la mayoría de las interfaces en el DOM de SVG empiezan con CSS.

Manipulación de estilo insertados en línea

El DOM2 añade soporte para manipular valores de CSS. La forma más básica de modificar estos valores es a través de la propiedad *style* o el método *getSytle()* que corresponde a la especificación de hojas de estilo insertado en línea para un elemento específico.

Por ejemplo,

```
<text id="miTexto">Esto es una prueba</text>
```

agregándole un estilo en línea:

```
<text id="miTexto" style="color:red">Esto es una prueba</text>
```

Manipulando las interfaces DOM mediante JavaScript:

```
t = document.getElementById("miTexto");
t.style.color = "green";
```

El aspecto clave es cómo asignar los diversos nombres de propiedades CSS a los nombres de propiedades con un guión como en *background-color*, lo que en JavaScript se convierte *backgroundColor*. En general, las propiedades CSS con guión se representan en el DOM como una sola palabra con la segunda palabra con mayúscula tipo *camelback* o *joroba de camello*.[22, p. 302]

E.g., la propiedad CSS *text-align* cambia a la propiedad DOM2 *textAlign*.

La interfaz *CSSStyleDeclaration* ofrece los siguientes métodos:

- ***getPropertyValue(p)*** que lee la propiedad *p* que se pasa como parámetro.
- ***setProperty(p, v)*** que asigna a la propiedad *p* el valor *v*.

De este modo se tiene:

```
t = documentgetElementById("miTexto");
t.getStyle().setProperty("color", "green");
```

3.5.4. Eventos SVG

SVG soporta el *modelo de eventos del DOM2*[11]. Los eventos se pueden vincular al estilo HTML mediante atributos o al estilo DOM mediante *oyentes de evento*.

Eventos UI

Estos eventos son lo que el usuario activa mediante dispositivos externos como el ratón o el teclado o dispositivos independientes (eventos lógicos UI). La Tabla 3.10 muestra el nombre del evento, una descripción y el nombre de atributo.[26, p. 560]

Eventos de ratón

Estos eventos son activados exclusivamente por dispositivos de "puntero". Moviendo y/o haciendo clic se puede activar uno de los eventos listados en la Tabla 3.11. [26, p. 561-562]

Mutaciones de evento

Las modificaciones en la estructura DOM son llamadas mutaciones; de este modo, los eventos de mutación son activados siempre que algo es cambiado en la estructura del documento, e.g., agregar o eliminar un nodo. Los eventos de mutación sólo pueden ser vinculados *mediante oyentes de evento*. [26, p. 562]

[11] Véase la sección 1.2.5.3 en la página 12

Tabla 3.10: Eventos UI

Evento	Descripción	Atributo
activate	Ocurre cuando un elemento es "activado" usando un dispositivo externo. Un usuario puede dar clic en un elemento o presionar una tecla.	onactivate
focusin	Ocurre cuando un elemento recibe el foco.	onfocusin
focusout	Ocurre cuando un elemento pierde el foco.	onfocusout

Tabla 3.11: Eventos de ratón

Evento	Descripción	Atributo
click	Ocurre cuando se presiona el botón del ratón sobre un elemento.	onclick
mousedown	Ocurre cuando el botón del ratón se presiona y se mantiene presionado sobre un elemento.	onmousedown
mousemove	Ocurre cuando el puntero del ratón se mueve sobre un elemento, estando dentro del elemento.	onmousemove
mouseout	Ocurre cuando el puntero del ratón deja un elemento.	onmouseout
mouseover	Ocurre cuando el puntero del ratón se mueve sobre un elemento.	onmouseover
mouseup	Ocurre cuando se deja de presionar el botón del ratón	onmouseup

Eventos específicos de SVG

La especificación de SVG añade nuevos eventos que no son parte del DOM2, debido a que SVG tiene capacidades especiales que no pueden ser parte del DOM2.

El primer conjunto de eventos es muy específico de SVG, está relacionado con la *carga* y *despliegue* de los gráficos. La Tabla 3.12 muestras estos eventos.[26, p. 564]

Aparte de los eventos específicos de SVG, existen tres eventos de animación que se listan en la Tabla 3.13.[26, p. 565]

Tabla 3.12: Eventos específicos de SVG

Evento	Descripción	Atributo
SVGAbort	Ocurre cuando en la carga de un documento SVG se aborta la página antes de que todos los datos se hayan transmitido completamente.	onabort
SVGError	Ocurre cuando un error es encontrado, ya sea mientras se cargan los elementos o durante la ejecución de un script.	onerror
SVGLoad	Ocurre cuando el documento SVG se ha cargado completamente (incluyendo referencias y recursos externos) y ha sido analizado completamente.	onload
SVGResize	Ocurre cuando una página se redimensiona. Este evento sólo funciona con el elemento raíz <svg>.	onresize
SVGScroll	Ocurre cuando la página se desplaza en vertical o en horizontal o en ambos. Este evento sólo funciona con el elemento raíz <svg>.	onscroll
SVGUnload	Ocurre cuando el documento es descargado.	onunload
SVGZoom	Ocurre cuando se aplica un acercamiento/alejamiento al documento.	onzoom

Tabla 3.13: Eventos de animación

Evento	Descripción	Atributo
beginEvent	Ocurre cuando el elemento de animación comienza.	onbegin
endEvent	Ocurre cuando el elemento de animación termina. Si existen iteraciones, se activa después de la última iteración.	onend
repeatEvent	Ocurre cada vez que el elemento de animación es repetido; cada iteración activa el evento.	onrepeat

Capítulo 4

Biblioteca de objetos gráficos

Constantemente se tiene que ver con números, cantidades y comparaciones de magnitudes. Las estadísticas (hechos expresados mediante números) se consideran con frecuencia como "frías" o carentes de interés debido a que su significado no es aparentemente inmediato. El financiero observa los promedios de las acciones de Dow Jones, el gerente de ventas mantiene su vista sobre el volumen de ventas en dólares y el ama de casa está interesada ciertamente en las alzas y bajas del costo de la vida.

Si se presentan gráficamente los hechos numéricos, esto es, por medio de dibujos, la información aparece de inmediato ante nuestra vista y significa más que una sencilla tabulación de números. Tales dibujos son conocidos como *cartas, gráficas y diagramas.* Entre las gráficas más comunes se encuentran las *gráficas de barras, gráficas de líneas, gráficas de área* y *gráficas circulares.*

4.1. Cartas (Charts)

Algunas veces referidas como un gráfico de información. Una ***carta*** es un vehículo para consolidar y desplegar información para propósitos tales como análisis, planeación, monitoreo, comunicación, etc. Antes las cartas eran cosas tangibles como simples hojas de papel o tableros. Actualmente muchas cartas son generadas y desplegadas electrónicamente.

Existen cinco categorías principales de cartas: gráficas, mapas, diagramas, tablas y otras (aquellas que no encajan en ningunas de las cuatro categorías). A su vez, cada categoría principal esta dividida en subcategorías. Todos los gráficos de información individuales pueden estar incluidos en múltiples subcategorías dependiendo del criterio que sea utilizado, como la forma, formato, función, tipo de escalas, tipo de datos desplegados, uso, número de ejes, etc. Por ejemplo, una gráfica ampliamente utilizada para graficar la distribución de un conjunto de datos podría ser llamado histograma, gráfica de distribución, gráfica de columnas unidas, gráfico de barras, gráfica de dos ejes, gráfica de dos dimensiones, gráfica rectangular, gráfica cuantitativa, gráfica o carta. Estos términos son perfectamente correctos.[10, p. 71]

4.1.1. Gráficas (Graphs)

Una ***gráfica*** es una carta que gráficamente despliega relaciones cuantitativas entre dos o más grupos de información - por ejemplo, la relación entre ciudades y su población, la rapidez de un carro y su eficiencia, o el costo del dólar con respecto al tiempo. Las gráficas tienen combinación de uno, dos o tres ejes rectos o circulares utilizando una o más escalas cuantitativas. Esta definición claramente distingue las gráficas de otras cartas como los diagramas, tablas, cartas de texto, cartas proporcionales, cartas de ilustración, y los mapas. Debido a que las gráficas constituyen una de las principales categorías de cartas, las gráficas son frecuentemente referidas como cartas.[10, p. 164]

Las gráficas ofrecen muchas características importantes[10, p. 164]:

- Grandes cantidades de información pueden ser convenientemente y eficazmente revisadas.
- Generalmente los patrones de datos resaltan más claramente que en forma tabular.
- Desviaciones, tendencias, y relaciones son más notables.
- Comparaciones y proyecciones pueden ser hechas con mayor facilidad y precisión.
- Las anomalías en los datos se convierten en obvias.
- Los espectadores pueden más rápidamente determinar y asimilar la esencia de la información.
- Con ayuda de las gráficas se acortan las juntas y la toma de decisiones es más rápida.

4.1.1.1. Gráfica de barras vertical

También llamada gráfica de columnas. Las ***gráficas de barras verticales*** son una familia de gráficas que despliegan información cuantitativa por medio de una serie de rectángulos. Las gráficas de barras son frecuentemente utilizadas para comparar múltiples entidades o para mostrar cómo una o más entidades varían sobre el tiempo.

Cada barra representa un dato, y el conjunto completo de barras representa una serie de datos. Si el valor es positivo la parte superior de la barra se localiza con el valor que representa, si el valor es negativo la parte inferior de la barra se localiza con el valor que representa. La longitud y/o área pueden o no ser proporcionales con los valores que representan. Debido a que la parte superior o inferior de las barras puede ser muy pronunciado, este tipo de gráficas es una de las mejores para mostrar valores específicos. Debido a la naturaleza independiente de las barras, también es muy adecuado para representar datos discretos. Las gráficas de barras verticales normalmente tienen una escala cuantitativa lineal en el eje vertical y generalmente tienen una escala categórica o secuencial en el eje horizontal. [10, p. 80]

Gráfica de barras vertical simple

Cuando las barras despliegan sólo una serie de datos similar al de la Figura 4.1 A), es referida como una gráfica vertical simple. Las columnas pueden ser de cualquier ancho; sin embargo, las barras suelen ser uniformes en ancho a través de toda la gráfica. Lo mismo es cierto para los espacios entre las barras. La escala vertical es siempre cuantitativa, y valores negativos y positivos pueden ser trazados en la gráfica como se muestra en la Figura 4.1 B). Las escalas líneales casi siempre son utilizadas. Cuando ambos valores negativos y positivos son trazados, el resultado es llamado algunas veces gráfica de desviación. En una gráfica de barras verticales típica, el eje horizontal tiene una escala secuencial o categórica. [10, p. 80]

Zelazny sugiere que este tipo de gráficas debe contener a lo más siete u ocho elementos. Si se tienen más elementos, utilizar la gráfica de línea. [30, p. 36]

4.1.1.2. Gráfica de línea

También llamada gráfica de curva. Las gráficas de linea son una familia de gráficas que despliegan información cuantitativa por medio de líneas. Son extremadamente versátiles y, por lo tanto, son utilizadas extensamente. [10, p. 207]

En la Figura 4.2 se muestran los tres principales tipos de líneas utilizadas para conectar puntos/datos utilizados en las gráficas de líneas.

Figura 4.1: Gráfica de barras vertical simple

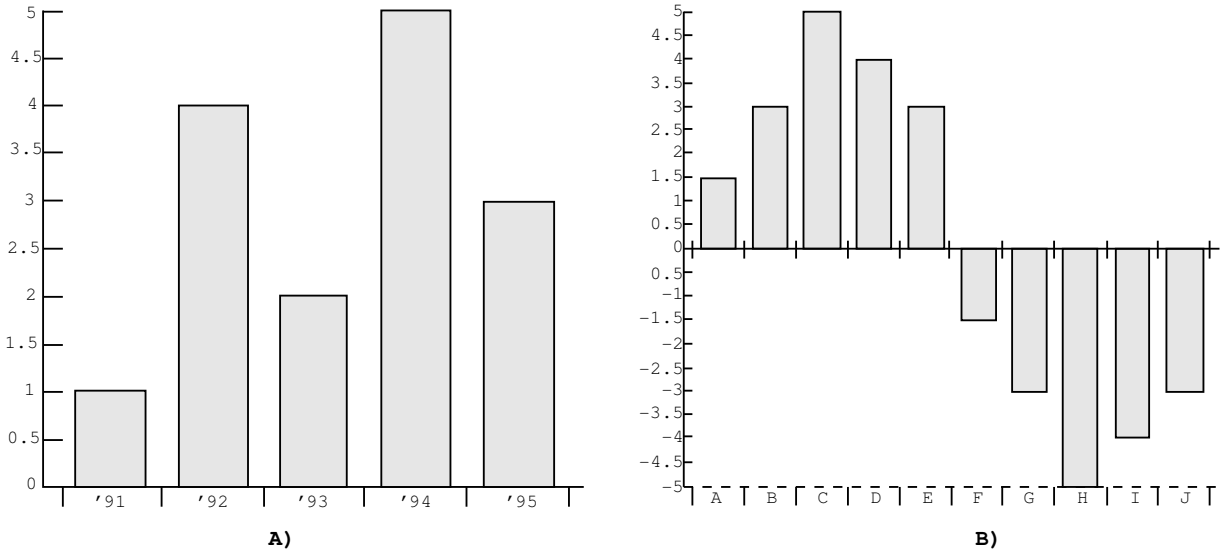
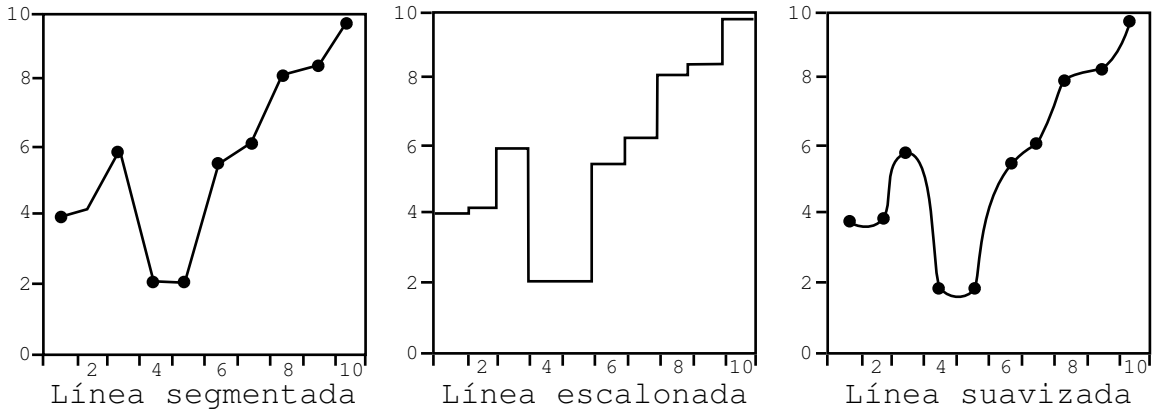


Figura 4.2: Tipos de líneas usados en las gráficas de línea



Gráfica de línea simple

Una gráfica de línea simple despliega sólo una serie de datos. Normalmente tiene una escala cuantitativa en el eje vertical y una escala categórica o secuencial en el eje horizontal. La Figura 4.3 muestra los dos tipos de escala. Con una escala categórica, un punto/dato es colocado directamente arriba de cada rotulo (categoría) en el eje horizontal. Todas las escalas deben ser mostradas en la escala a fin de que el espectador pueda interpretar correctamente la gráfica. Con escalas secuenciales, los puntos/datos frecuentemente no se localizan sobre los rótulos y en muchos casos los rótulos no se despliegan para cada valor. Valores positivos y negativos pueden ser trazados en el eje vertical como se muestra en la Figura 4.4. [10, p. 207]

4.1.1.3. Gráfica de área

Una gráfica de área tanto es como un proceso o técnica, como es un tipo de gráfica básica, ya que éstas son generadas llenando las áreas entre las líneas generadas por otro tipo de gráficas y el eje horizontal. Las gráficas de área generalmente no se utilizan para comunicar valores específicos. En su lugar, se utilizan muy frecuentemente para mostrar tendencias y relaciones, para identificar y/o agregar énfasis para especificar información en virtud del sombreado o el color, o para mostrar partes de un todo. [10, p. 10]

Al igual que en las gráficas de línea, hay tres tipos de curvas básicas utilizadas para generar gráficas de

Figura 4.3: Tipos de escalas en las gráficas de línea

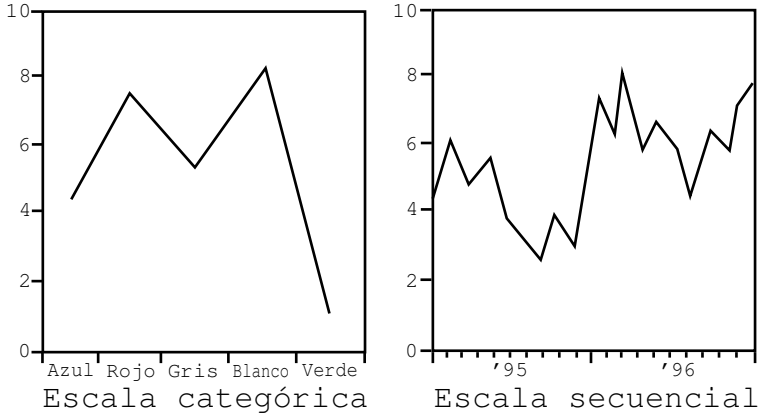
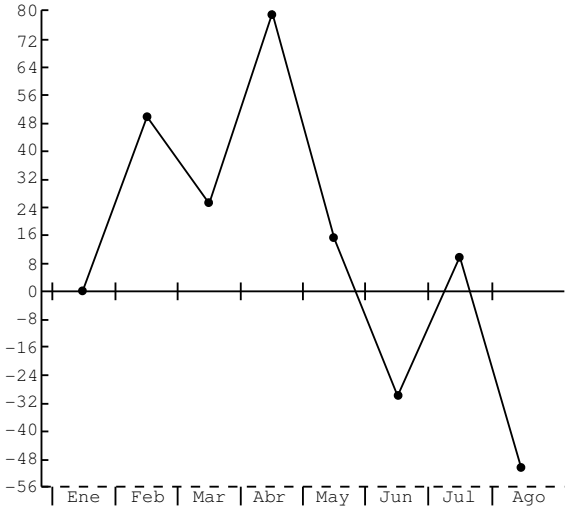


Figura 4.4: Valores negativos y positivos trazados en una gráfica de linea simple



área (segmentado, escalonado, suavizado). En la Figura 4.5 se muestran estos tres tipos de curvas.

Gráfica de área simple

También llamada gráfica de superficie o gráfica de silueta. Una gráfica de área simple típicamente tiene una escala cuantitativa en el eje vertical y una escala categórica o secuencial en el eje horizontal como se muestra en la Figura 4.6. Con una escala categórica, un punto/dato es colocado directamente arriba de cada rotulo (categoría) en el eje horizontal. Todas las escalas deben ser mostradas en la escala a fin de que el espectador pueda interpretar correctamente la gráfica. Con escalas secuenciales, los puntos/datos frecuentemente no se localizan sobre los rótulos y en muchos casos los rótulos no se despliegan para cada valor. Valores positivos y negativos pueden ser trazados en el eje vertical como se muestra en la Figura 4.7.

4.1.1.4. Gráfica de círculo

También llamada gráfica de pastel, gráfica de círculo dividido, gráfica circular de porcentaje, gráfica de sectores, sectograma, o gráfica segmentada. Las gráficas circulares son miembros de la familia de gráficas de área proporcional. Una gráfica circular consiste en un círculo dividido en segmentos (también llamados rebanadas o sectores). Cada segmento representa un dato en forma de porcentaje del total del círculo que es

Figura 4.5: Tipos de curvas usadas en las gráficas de área

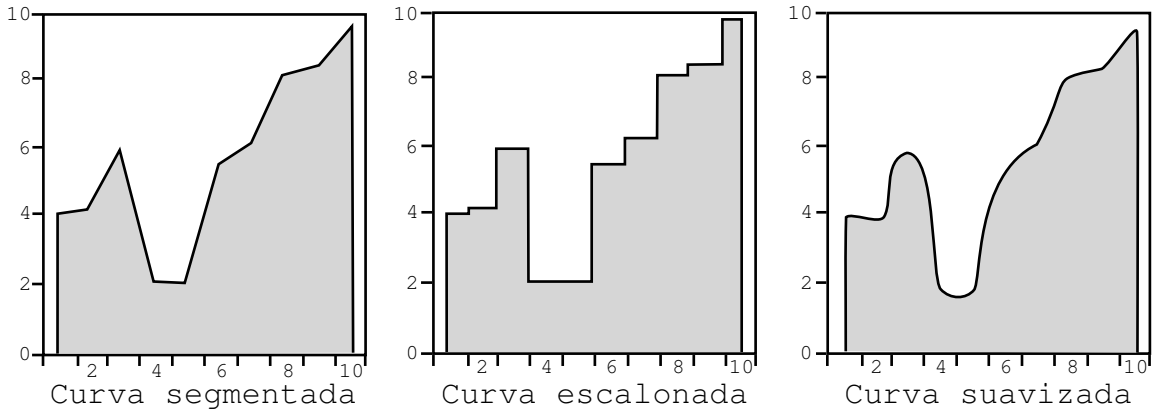
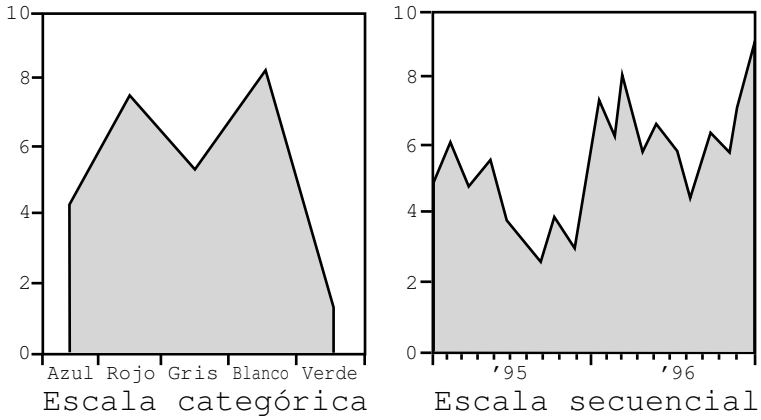


Figura 4.6: Tipos de escalas en las gráficas de área



la suma del conjunto de datos equivalente al 100 %. No se pueden desplegar valores negativos. Una gráfica circular tiene una escala alrededor de su circunferencia lo cuál la clasifica como una gráfica y, por lo tanto, se considera una gráfica de un único eje. La Figura 4.8 muestra este tipo de gráficas. El principal propósito de este tipo de gráficas es mostrar los tamaños relativos entre los componentes y con respecto a la gráfica total. Son utilizadas extensamente como herramientas de comunicación en presentaciones y publicaciones. [10, p. 73][10, p. 281]

Zelazny sugiere que este tipo de gráficas debe contener a lo más seis elementos. Si se tienen más de seis, seleccionar los cinco componentes más importantes y agrupar el resto en una sola categoría. [30, p. 28]

Figura 4.7: Valores negativos y positivos trazados en una gráfica de área simple

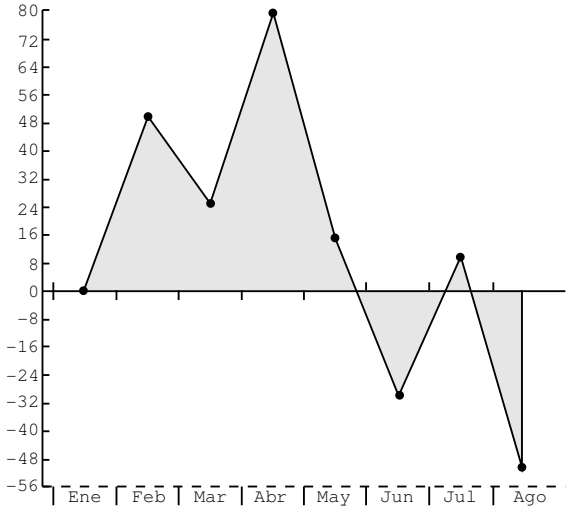
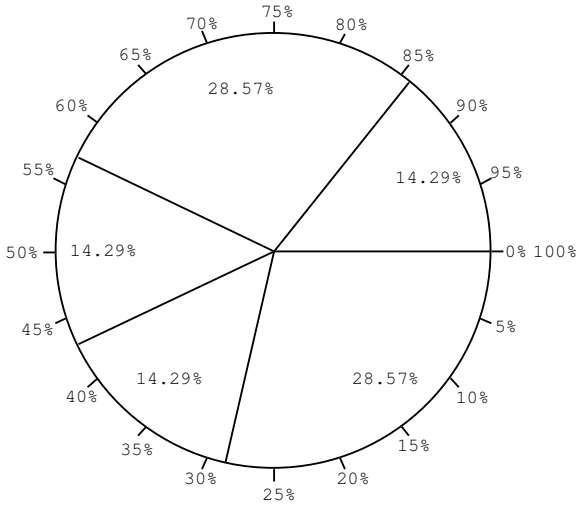


Figura 4.8: Gráfica circular



4.2. Análisis, diseño y codificación orientados a objetos

4.2.1. Análisis

El objetivo es desarrollar una biblioteca de objetos, la cual será llamada ***EzGraphSVG***, para la generación de gráficos de información para la Web. Esta biblioteca constará de cuatro módulos, uno para los *objetos comunes* y resto para *gráficas*. Con la ayuda del módulo de objetos comunes, se desarrollarán cuatro tipos de gráficas: de barras, de línea, de área, y de círculo. La biblioteca está dirigida principalmente a programadores, sin embargo, las gráficas podrán ser utilizadas por usuarios comunes siguiendo las instrucciones de uso.

4.2.1.1. Requerimientos

Los requisitos de la biblioteca están divididos en tres partes:

1. Una biblioteca de objetos comunes que implemente los diferentes componentes de una gráfica.

2. Cuatro tipos de gráficas:

   a) Gráfica de barras vertical

   b) Gráfica de línea.

   c) Gráfica de área.

   d) Gráfica de círculo

3. Compresión de la aplicación.

Requerimientos de las gráficas

- Las gráficas deben estar enfocadas para su uso en la Web y por lo tanto deben ser de transferencia rápida.
- Las gráficas se deben generar dinámicamente utilizando los recursos del cliente.
- Las gráficas estarán en su forma más simple.
- Ejes y cuadrícula.
- Escala cuantitativa en el eje vertical.
- Escala categórica en el eje horizontal.
- Un título (opcional).
- Dimensiones: ancho y alto (opcional).
- Ajuste personalizado o automático de la resolución de la gráfica.
- Todos los elementos de la gráfica serán en 2D.
- Todos los elementos de la gráfica usarán estilos CSS en línea.
- Los elementos barra, punto, o sector tendrán un color de relleno uniforme con una transparencia del 50 % y una línea de contorno del mismo color, pero sin transparencia.
- Al hacer clic en una barra, punto, o sector, según sea el caso, deberá aparecer un cuadro de información con los datos de ese elemento.
- El cuadro de información contendrá un botón de cierre, pero reemplazará al existente si se hace clic en otro elemento (barra, punto, o sector) y aparecerá en una zona apropiada. Además, si las cadenas son muy largas para contenerse dentro del cuadro, éstas serán truncadas.
- Los rótulos en el eje horizontal serán truncados si exceden su espacio.
- En el caso del gráfico de círculo, tendrá leyendas y si éstas son demasiadas y no caben dentro de la resolución de la gráfica, tendrá también flechas para el desplazamiento de las leyendas.
- Valores negativos y positivos se podrán graficar, excepto para el gráfico de círculo que sólo puede graficar valores positivos.

Compresión de la biblioteca

La aplicación debe ocupar el menor espacio de almacenamiento, para este fin se comprimirá con el estándar de compresión ***GZIP***.

4.2.1.2. Lenguaje de programación, API y formato de gráficos.

Como en los requisitos se especifica que los gráficos deben estar enfocados para la Web y que se deben generar dinámicamente utilizando los recursos del usuario, se utilizará:

1. **JavaScript** como lenguaje de programación por las siguientes razones:
   - Es un lenguaje dinámico de scripts orientado a objetos e independiente de la plataforma.
   - El núcleo del lenguaje está incrustado en los navegadores Web y, por lo tanto, utiliza los recursos del usuario.
2. La **API DOM** por las siguientes razones:
   - Es un estándar publicado por el W3C y lo soportan los navegadores Web modernos.
   - Permite examinar, modificar, agregar o quitar elementos del documento en tiempo de ejecución.
3. El formato **SVG** para gráficos por las siguientes razones:
   - Esta basado en XML.
   - Son gráficos vectoriales.

4.2.1.3. Tipo de datos

El tipo de datos que manejarán las gráficas será un arreglo de arreglos de JavaScript, estructurado de la siguiente forma:

```
[
  [Valor, Nombre, Color],
  [Valor, Nombre, Color],
  ...
  [Valor, Nombre, Color]
]
```

donde:

Valor es dato numérico finito positivo o negativo.

Nombre es una cadena entre comillas posiblemente vacía.

Color es una cadena entre comillas conforme a los estándares de color CSS. Este dato es opcional, si no se proporciona uno, se generará un color aleatoriamente en formato RGB.

Este arreglo, se tendrá que proporcionar como argumento al correspondiente método de una de las gráficas para crear todos sus elementos. Por supuesto, este arreglo no tiene que ser proporcionado como tal, puede ser una variable que contenga el arreglo o una referencia al arreglo.

4.2.1.4. Detección de errores

La detección de errores será en los datos de entrada para las gráficas. En cuanto se detecte un error el programa detendrá su ejecución, mandará un mensaje que indicará dónde se encontró el error y, por lo tanto, la gráfica no se creará. En el caso de que la aplicación se ejecute en Linux, los errores serán enviados a la salida estándar.

4.2.2. Diseño

4.2.2.1. Elementos que componen una gráfica

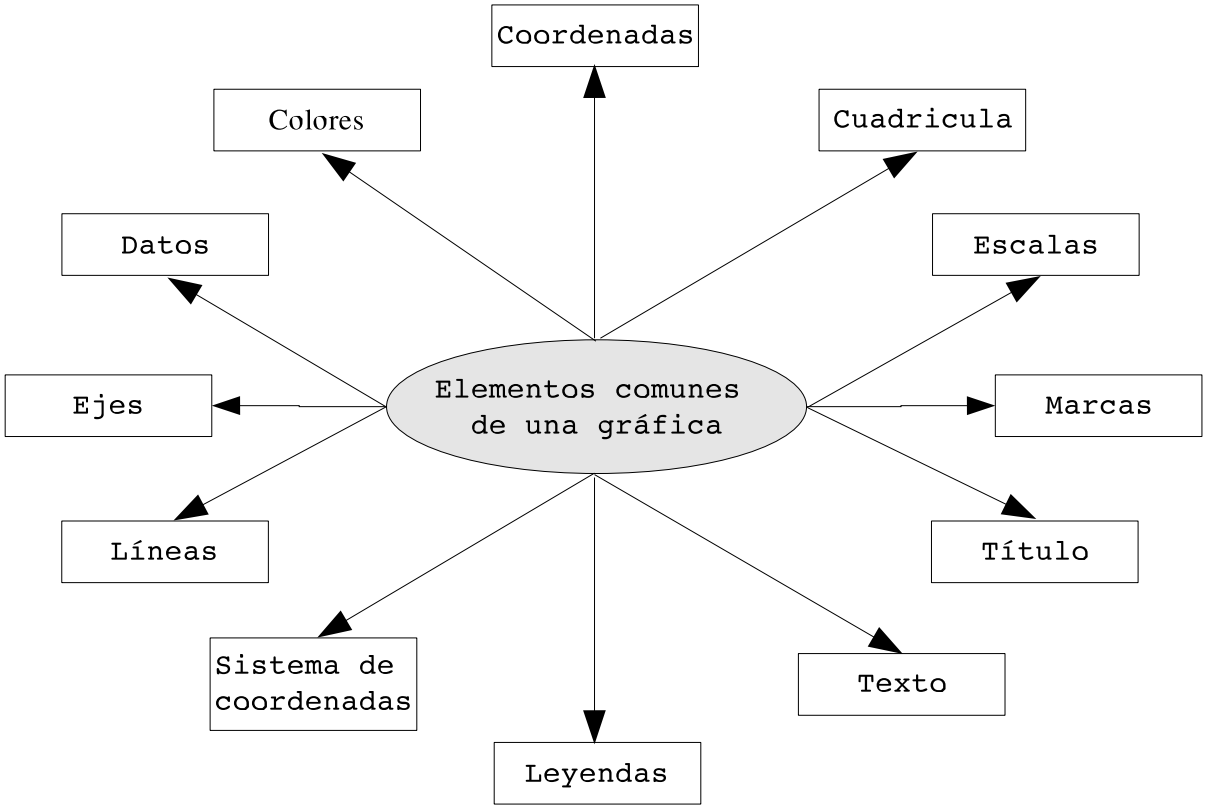
Antes de comenzar el diseño de las gráficas a nivel programación es conveniente revisar los elementos que las componen, debido a que éstos serán traducidos a objetos más adelante.

Para un mejor diseño, los elementos están separados en elementos comunes y elementos específicos.

Elementos comunes

La Figura 4.9 muestra los elementos comunes de una gráfica.

Figura 4.9: Elementos comunes de una gráfica



Elementos específicos

Las gráficas de barras, línea, área, y círculo tienen la siguiente lista de elementos específicos:

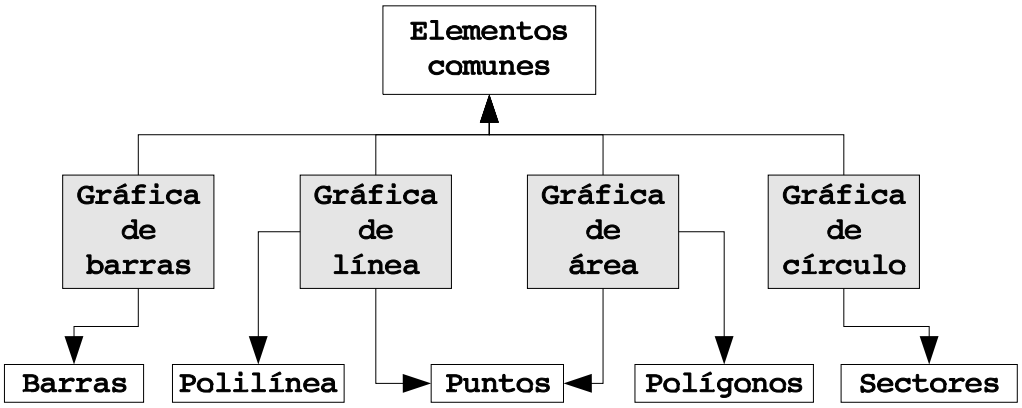
- Barras

- Polilínea
- Puntos
- Polígonos
- Sectores (o rebanadas)

Elementos en conjunto

La Figura 4.10 muestra todos los elementos que componen las gráficas.

Figura 4.10: Elementos de las gráficas de: barras, línea, área, y círculo

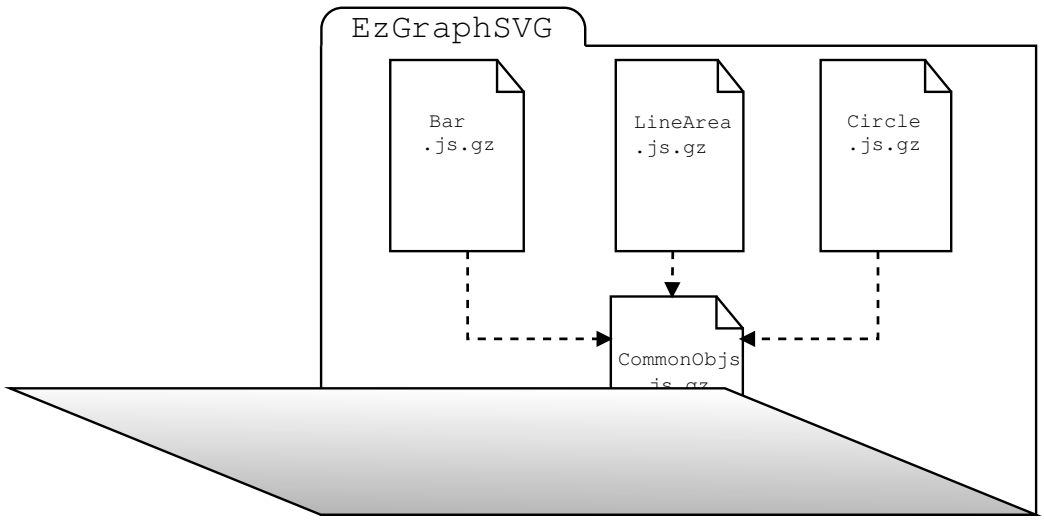


La biblioteca de objetos consta de los elementos comunes y los elementos específicos. Se organiza de este modo para facilitar el desarrollo de las gráficas.

Componentes físicos

La Figura 4.11 muestra cómo están organizados los componentes físicamente.

Figura 4.11: Componentes físicos



EzGraphSVG es el directorio que contiene los archivos.

CommonObjs.js.gz es el archivo que contiene los objetos comunes que componen una gráfica. El resto de los archivos depende de éste.

Bars.js.gz es el archivo que contiene el código de la clase SimpleVerticalBarGraph.

LineaArea.js.gz es el archivo que contiene el código de las clases SimpleLineGraph y SimpleAreaGraph.

Circle.js.gz es el archivo que contiene el código de la clase SimpleCircleGraph.

4.2.2.2. Diagrama de la jerarquía de objetos

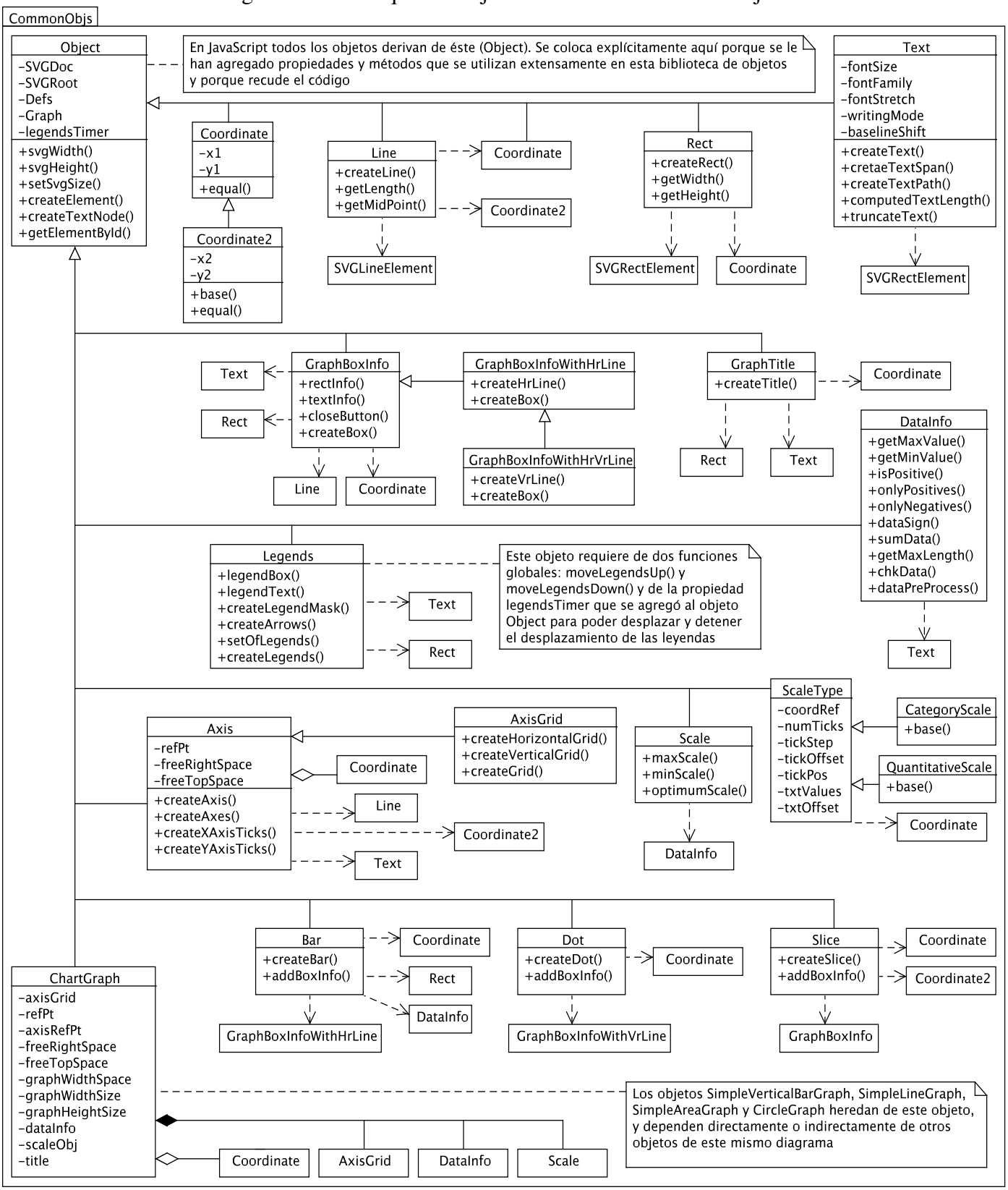
Debido a que el desarrollo de la biblioteca está orientado a objetos es muy apropiado aprovechar los diagramas UML[1] para modelar la jerarquía de objetos. Nuevamente, estos diagramas están divididos en dos: jerarquías de objetos de comunes y jerarquía de objetos de las gráficas.

Jerarquía de objetos comunes

La Figura 4.12 muestra el diagrama de la jerarquía de objetos comunes del módulo *CommonObjs*. Este diagrama sólo muestra los *atributos*, *métodos*, *generalizaciones*, *agregaciones* y *dependencias* de los objetos con el fin de tener un diagrama legible, además los objetos Coordinate, Coordinate2, Line, Rect y Text, que son muy utilizados por otros objetos, sólo muestran las características anteriormente mencionadas una vez. En la codificación hay documentación suficiente que explica la finalidad de las propiedades y métodos, así como los argumentos que reciben los métodos y lo que regresan.

[1] UML (Unified Modeling Language, Lenguaje Unificado de Modelado) proporciona un conjunto de elementos (símbolos, líneas y etiquetas) que permiten modelar sistemas de software. [5, p. 75]

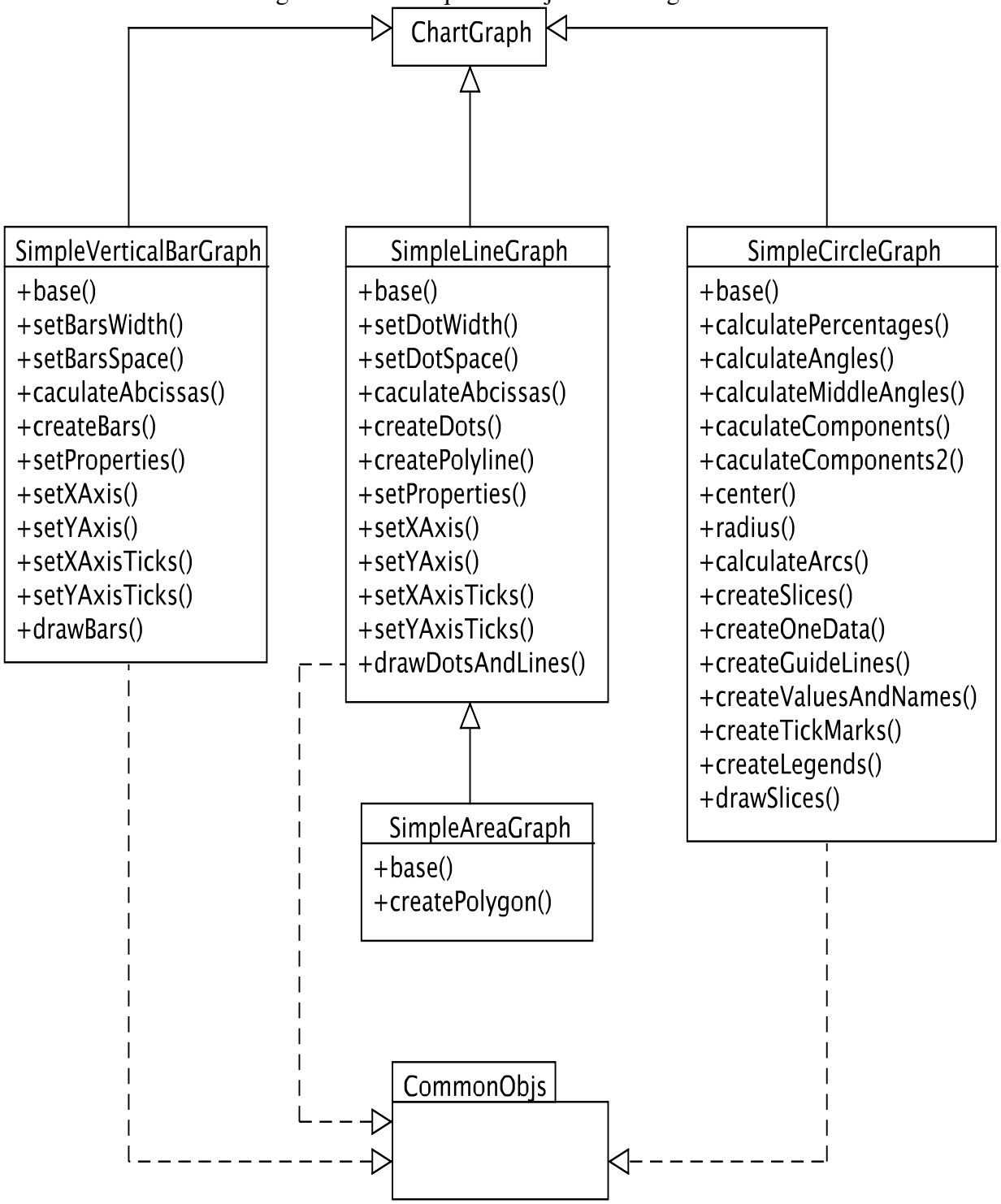
Figura 4.12: Jerarquía de objetos del módulo CommonObjs



Jerarquía de las gráficas

La Figura 4.13 muestra el diagrama de la jerarquía de objetos de las gráficas de barras, de línea, de área, y de círculo. Este diagrama también muestra únicamente los *atributos*, *métodos*, *generalizaciones*, *agregaciones* y *dependencias* de los objetos. Otras características están documentadas en la codificación.

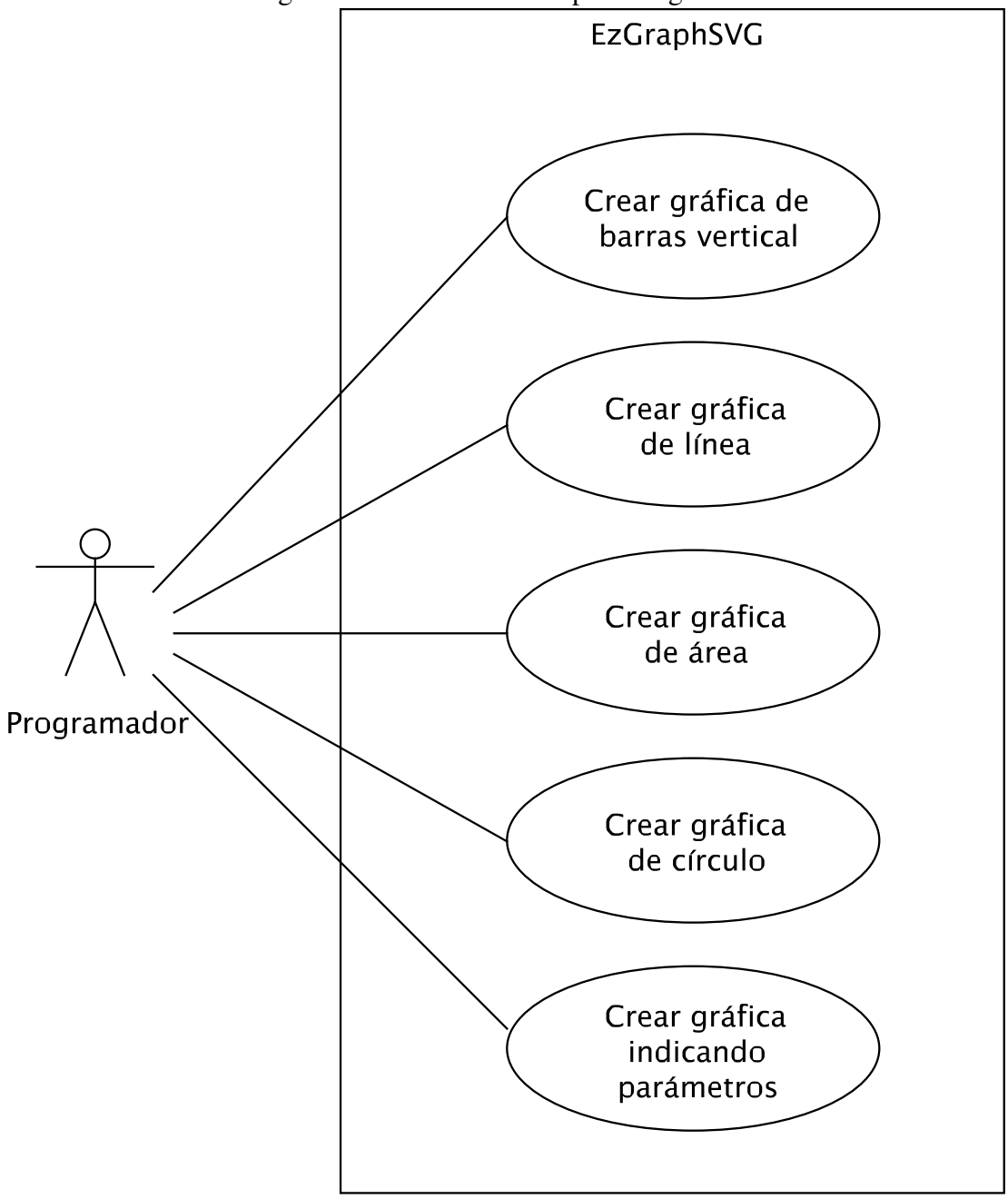
Figura 4.13: Jerarquía de objetos de las gráficas



4.2.2.3. Casos de uso

La Figura 4.14 muestra el diagrama los casos de uso de las diferentes gráficas.

Figura 4.14: Casos de usos para las gráficas



4.2.2.4. Plantilla para la utilización de los gráficos

El Listado 9 muestra una plantilla para generar cualquiera de los gráficos programados. Por supuesto, todos los comentarios pueden ser omitido. Esta plantilla sirve para *scripts* locales, los datos pueden ser suministrados por medio JavaScript en la misma plantilla o desde un archivo externo, en cuyo caso se tendría que indicar la ubicación de éste por medio de <script xlink:href=" "></script>. Esta plantilla también puede

ser combinada con *scripts del lado del servidor*[2] para suministrar los datos e inclusive para seleccionar el tipo de gráfica.

Si se desea incrustar cualquiera de las gráficas SVG en HTML, Adobe[12, download area] sugiere que se utilice la etiqueta <embed>[3] por cuestiones de seguridad.

Listado 9 Plantilla SVG

```
<svg onload="init(evt); usage(evt);">
<!-- BIBLIOTECA DE OBJETOS -->
<script xlink:href="EzGraphSVG/CommonObjs.js.gz"></script>
<!-- GRAFICO QUE SE USARA (Bars.js.gz, LineArea.js.gz, Circle.js.gz) -->
<script xlink:href="EzGraphSVG/..."></script>
<!-- SCRIPTS EXTERNOS ADICIONALES
<script xlink:href="..."></script>
.
.
.
<script xlink:href="..."></script>-->
<!-- SCRIPT PRINCIPAL -->
<script><![CDATA[
function usage(evt)     //Función para instanciar objetos y
{                       //llamar a su método de dibujo
   .
   .
   .
}
/*
   // CÓDIGO ADICIONAL
   .
   .
   .
*/
]]></script>
</svg>
```

El Listado 10 muestra las instrucciones de uso de la plantilla del Listado 9

[2] Véase el apéndice E en la página 97

[3] Este elemento es compatible con casi todos los exploradores, aunque no es estándar, especifica un objeto, normalmente en elemento multimedia, para incrustarlo en un documento HTML. [21, p. 724]

Listado 10 Instrucciones de uso

```
1. Indicar una de las siguientes biblioteca de la gráfica que se usará en el
atributo de xlink:href de la etiqueta <script></script>:
  -Bars.js.gz
  -LineArea.js.gz
  -Circle.js.gz
Por ejemplo:
  <script xlink:href="EzGraphSVG/Barjs.js.gz"></script>
2. Crear una instancia dentro de la definición de la función "usage" de
cualquiera de los siguientes objetos que se utilizará:
  SimpleVerticalBarGraph
  SimpleLineGraph
  SimpleAreaGraph
  CircleGraph
Por ejemplo:
 var graficaBarras = new SimpleVerticalBarGraph();
Ejemplo indicando parametros:
  var graficaBarras = new SimpleVerticalBarGraph("Barras");
ó
  var graficaBarras = new SimpleVerticalBarGraph(800,600,"Barras");

3. Llamar al método apropiado del objeto que se instanció para dibujar la gráfica
  drawBars(datos) para SimpleVerticalBarGraph
  drawDotsAndLines(datos) para SimpleLineGraph
  drawArea(datos) para SimpleAreaGraph
  drawSlices(datos) para CircleGraph

  Por ejemplo:
  graficaBarras.drawBars(datos)
  Donde "datos" es un "Arreglo de Arreglos" de JavaScript de la forma:
 [[VALOR, NOMBRE, COLOR], [VALOR, NOMBRE, COLOR], ..., [VALOR, NOMBRE, COLOR]]
  -VALOR es un dato numérico positivo o negativo, excepto para la
   "Gráfica de círculo" que sólo acepta valores positivos
  -NOMBRE es un cadena entre comillas (puede estar vacía)
  -COLOR es una cadena opcional que debe cumplir con las reglas CSS.
  Si no se proporciona el color se genera uno de la forma
  rgb(0-255,0-255,0-255)
```

El Listado 11 muestra el código para desplegar una gráfica de barras con el título “Puntos por partido por jugador”, además se indican los colores en los datos. En este Listado se han omitido los comentarios de la plantilla del Listado 9.

Listado 11 Jugadores.svg

```
<svg onload="init(evt); usage(evt);">
<script xlink:href="EzGraphSVG/CommonObjs.js.gz"></script>
<script xlink:href="EzGraphSVG/Bars.js.gz"></script>
<script><![CDATA[
function usage(evt)
{
   var datos = getData();  // Obtiene los datos a graficar
   var barras = new SimpleVerticalBarGraph("Puntos por partido por jugador");
   barras.drawBars(datos)
}
function getData()
{
   var names = ["Homer","Marge","Bart","Lisa","Maggie"];
   var points = [-4, 12, 24, 16, 5];
   colors = ["aqua","purple","silver","olive","teal"]
   var players = new Array();
   for( i=0; i<names.length; i++)
   {
      var player = new Array();
      player[0] = points[i];
      player[1] = names[i];
      player[2] = colors[i];
      players.push(player);  //Add an element
   }
   return(players);
}
]]></script>
</svg>
```

4.2.3. Codificación

La codificación tiene las siguientes características:

- Debido a que las variables SVGDoc, SVGRoot, Defs, y Graph se utilizan extensamente, están añadidas al objeto *Object* en el módulo CommonObjs, del cual todos los objetos de JavaScript derivan. Estas variables o propiedades tienen las siguientes finalidades:

  SVGDoc es la referencia al documento SVG.

  SVGRoot es la referencia al elemento raíz del documento, es decir, <svg>.

  Defs es la referencia al elemento <defs> dentro de <svg>.

  Graph es la referencia a un elemento <g> el cual se utiliza para agrupar todos los elementos de cualquier gráfica que se genere.

  legendsTimer es la referencia del temporizador para desplazar las leyendas.

- Las funciones/métodos svgWidth, svgHeight, setSVGSize, createElement, createTextNode, y getElementById también se utilizan extensamente y por eso ha sido agregados al objeto *Object*.

- Se utiliza mucho la técnica para convertir coordenadas cartesianas a coordenadas SVG, que está descrita en el apéndice A en la página 86
- Todos los métodos que crean elementos regresan una referencia del elemento que crean. Si se crean varios elementos, éstos se agrupan dentro de un elemento <g> de SVG y se regresa la referencia de este elemento.
- En el caso del gráfico de círculo, se utilizan leyendas para mostrar la información de cada sector. Sin embargo, si las leyendas son muchas y no caben dentro de la resolución de la gráfica, se aplica una mascara para mostrar sólo parte de las leyendas; también se dibujan flechas para poder desplazar las leyendas. Para el desplazamiento de las leyendas se emplean dos funciones globales (moveLegendsUp y moveLegendsDown) y una propiedad (legendsTimer) que está agregada al objeto *Object*. Se hizo de esta manera porque el desplazamiento se activa con el evento *mouseover* sobre las flechas cada 500 milisegundos mediante un temporizador, y se desactiva (se limpia el temporizador) cuando ocurre el evento *mouseout* sobre las flechas.

4.3. Pruebas de rendimiento

4.3.1. Objetivo

Obtener parámetros de comparación entre los gráficos de información que generan EzGraphSVG y *OpenOffice Calc* (en formato JPG) exportado a HTML, lo cual son dos modos diferentes de publicar gráficos de información para la Web.

4.3.2. Características

Las características que se midieron son las siguientes:

- Porcentaje de CPU para generar y desplegar las gráficas. Para medirlo se utilizó la herramienta ***top***[4] de Linux.
- Espacio de almacenamiento que ocupan las gráficas. Para medirlo se utilizó el comando ***ls***[5] de Linux.
- Tiempo estimado de transmisión. Esta medición fue teórica, suponiendo una conexión vía módem recibiendo 46 Kbps (aproximadamente 5.6 Kb por segundo).

Las pruebas se realizaron en una computadora con las siguientes características:

- Procesador Pentium III a 500Mhz
- Monitor de 17” con resolución de 1280x1024 a 60Hz
- 512 de memoria RAM
- Sistema operativo Linux Mandrake 10.0 (Kernel 2.6)
- Navegador Web Mozilla 1.7

[4]El programa *top* proporciona una vista dinámica en tiempo real de un sistema en ejecución. Puede desplegar información resumida, así como una lista de los procesos que actualmente están siendo manejados por el kernel de Linux.

[5]El comando *ls* (LiSt) es similar a *dir* en DOS. Con la opción -l, imprime información adicional sobre los archivos, como el tamaño, permisos, propietario, etc.

- Plug-in Adobe SVGViewer 3.01 Beta Release 1

Las gráficas tienen las siguientes características

- Cada gráfica se generó con 8 y 24 datos.
- Para las gráficas de Barras, Línea y Área con 8 y 24 datos. se generaron dos conjuntos valores aleatorios [-1000000,1000000] y [-1, 1], ajustado a tres décimas, con el siguiente código de JavaScript:

```
(Math.random()*(valorExtremo))*( Math.pow(-1, Math.round(Math.random())+1)
```

- Para las gráficas de Círculo con 8 y 24 datos. se generaron dos conjuntos valores aleatorios: [0,1000000] y [0, 1], debido a que en ésta no se pueden graficar valores negativos.
- Cada gráfica fue generada a tres resoluciones: 1280x1024, 800x600, y 320x280.
- Las Tablas B.1, B.2, B.3 y B.4 en el apéndice B, muestran los datos que se utilizaron. Para el caso de la gráfica de círculo se tomaron todos los datos en valor absoluto.

4.3.3. Justificación

Se realizaron las pruebas de rendimiento para mostrar que los gráficos SVG generados por EzGraphSVG requieren de menos espacio de almacenamiento a comparación de los JPG generados por OpenOffice Calc, y por lo tanto, se transfieren más rápido a través de Internet.

Se utilizó OpenOffice Calc porque es un software OpenSource con características similares a MS Excel que corre bajo Linux y porque las pruebas fueran realizadas en este mismo sistema operativo.

Se utilizó JPG para la comparación porque es un formato ampliamente utilizado en la Web.

Los datos fueron generados aleatoriamente mediante JavaScript, sin embargo, no es un factor que afecte los resultados.

Se usaron valores comprendidos entre los rangos [-1, 1] y [-1000000, 1000000] ([0, 1] y [0, 1000000] en el caso de la gráfica de círculo) con el fin de tener valores extremos y ver como afectaban los cálculos dependiendo del número de dígitos utilizados en el proceso de generación de la gráfica.

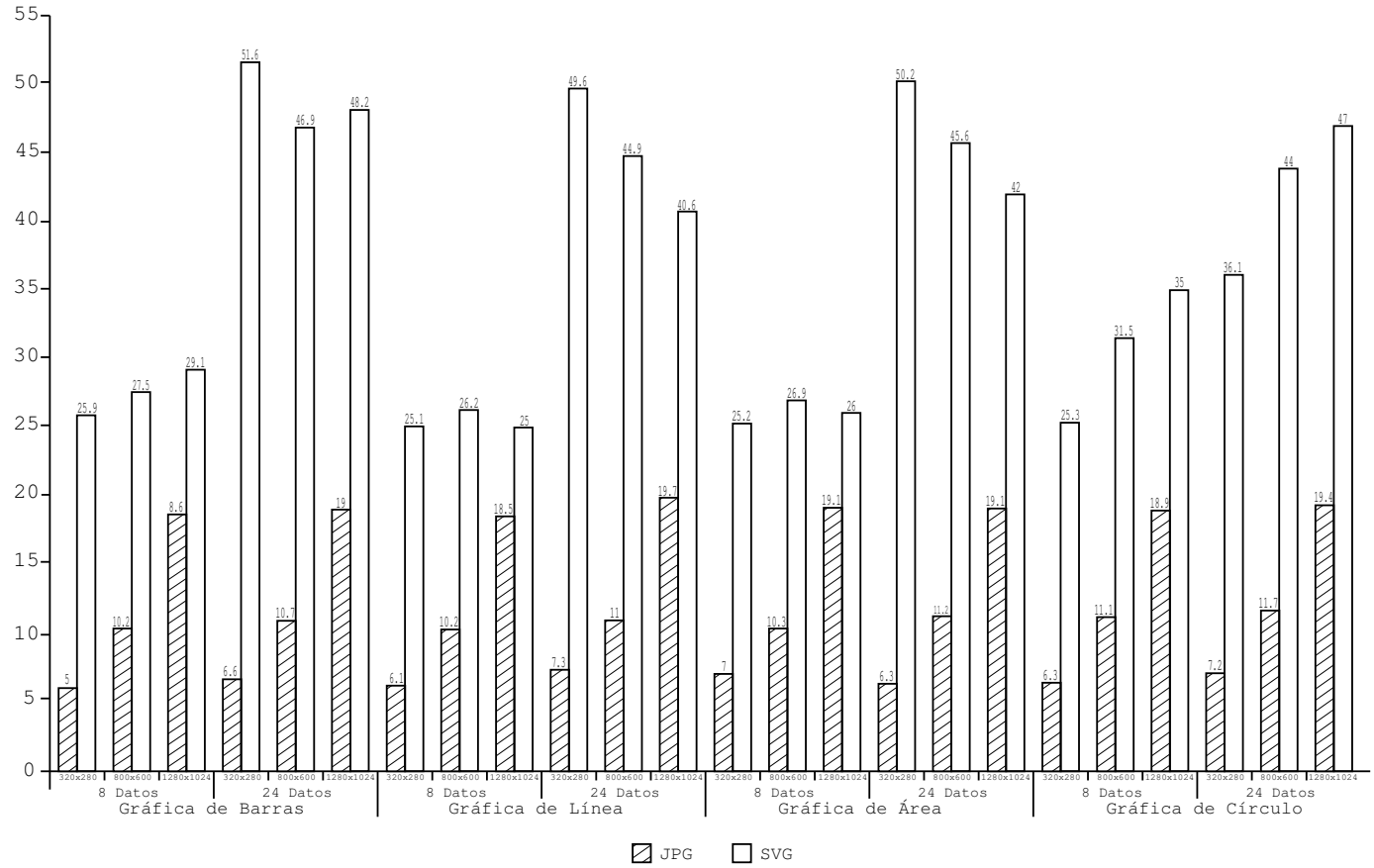
4.3.4. Resultados

4.3.4.1. Porcentaje de CPU

En el Apéndice B se encuentran las Tablas con los resultados obtenidos del porcentaje de CPU requerido para desplegar las gráficas en formatos SVG y JPG.

La Figura 4.15 en la página siguiente ilustra las comparaciones entre los formatos JPG y SVG de las distintas gráficas a diferentes resoluciones para 8 y 24 datos, con valores comprendidos entre [-1000000, 1000000] para las gráficas de barras, línea y área, y entre [0, 1000000] para la gráfica de círculo.

Figura 4.15: Porcentaje de CPU para generar las gráficas



A simple vista podría esperarse que cualquier gráfica en cualquier formato necesite de más CPU para desplegar las gráficas conforme aumentan su resolución. Con JPG sí ocurre, porque las gráficas se generaron previamente y no dinámicamente, sin embargo no todas las gráficas SVG presentan dicho comportamiento.

En el caso de las gráficas de línea y área con 24 datos, se nota que requieren menor CPU a mayor resolución, esto se debe a que pasan por un proceso de *truncamiento de cadena*[6] en los rótulos del eje horizontal.

Respecto a la gráfica de círculo, a mayor resolución y más datos requiere más CPU. Sin embargo, está gráfica también trunca cadenas, pero mediante máscaras y no mediante programación. De aquí se deduce que la técnica para truncar cadenas es más eficiente mediante máscaras.

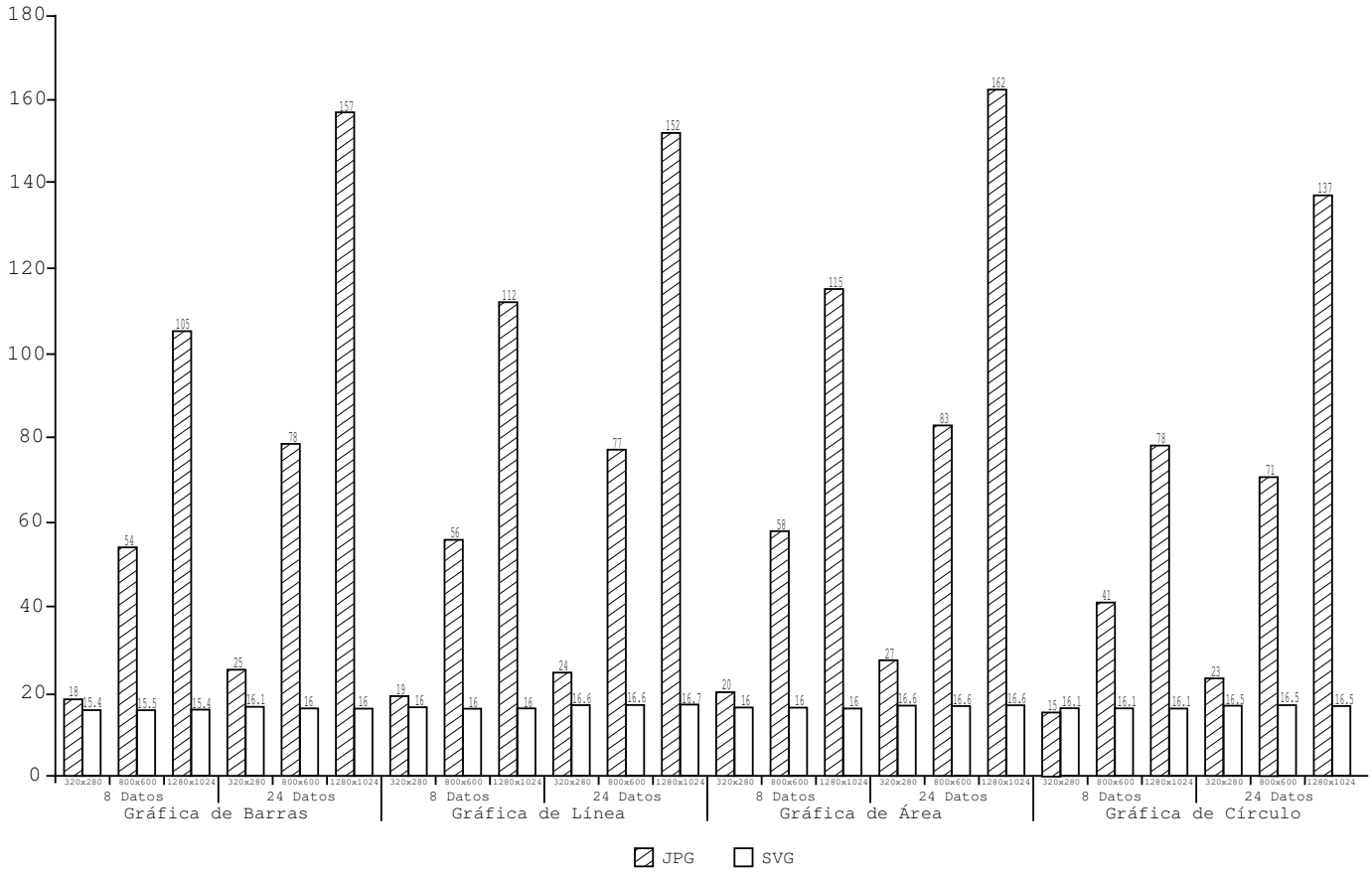
4.3.4.2. Espacio de almacenamiento

En el Apéndice B se encuentran las Tablas con los resultados obtenidos del espacio de almacenamiento para los formatos SVG y JPG.

La Figura 4.15 ilustra las comparaciones entre los formatos JPG y SVG de las distintas gráficas a diferentes resoluciones para 8 y 24 datos, con valores comprendidos entre [-1000000, 1000000] para las gráficas de barras, línea y área, y entre [0, 1000000] para la gráfica de círculo.

[6]Este proceso consiste, básicamente, en revisar el espacio disponible entre cada rótulo, si la cadena es más larga quita el último caracter y revisa si cabe la cadena, de lo contrario repite el proceso

Figura 4.16: Espacio de almacenamiento de las gráficas en KB



Esta gráfica no muestra ningún comportamiento extraño, los archivos JPG aumentan a mediada que aumenta su resolución, mientras que los archivos SVG mantienen un tamaño reducido, casi igual, para todas las gráficas a cualesquiera resoluciones.

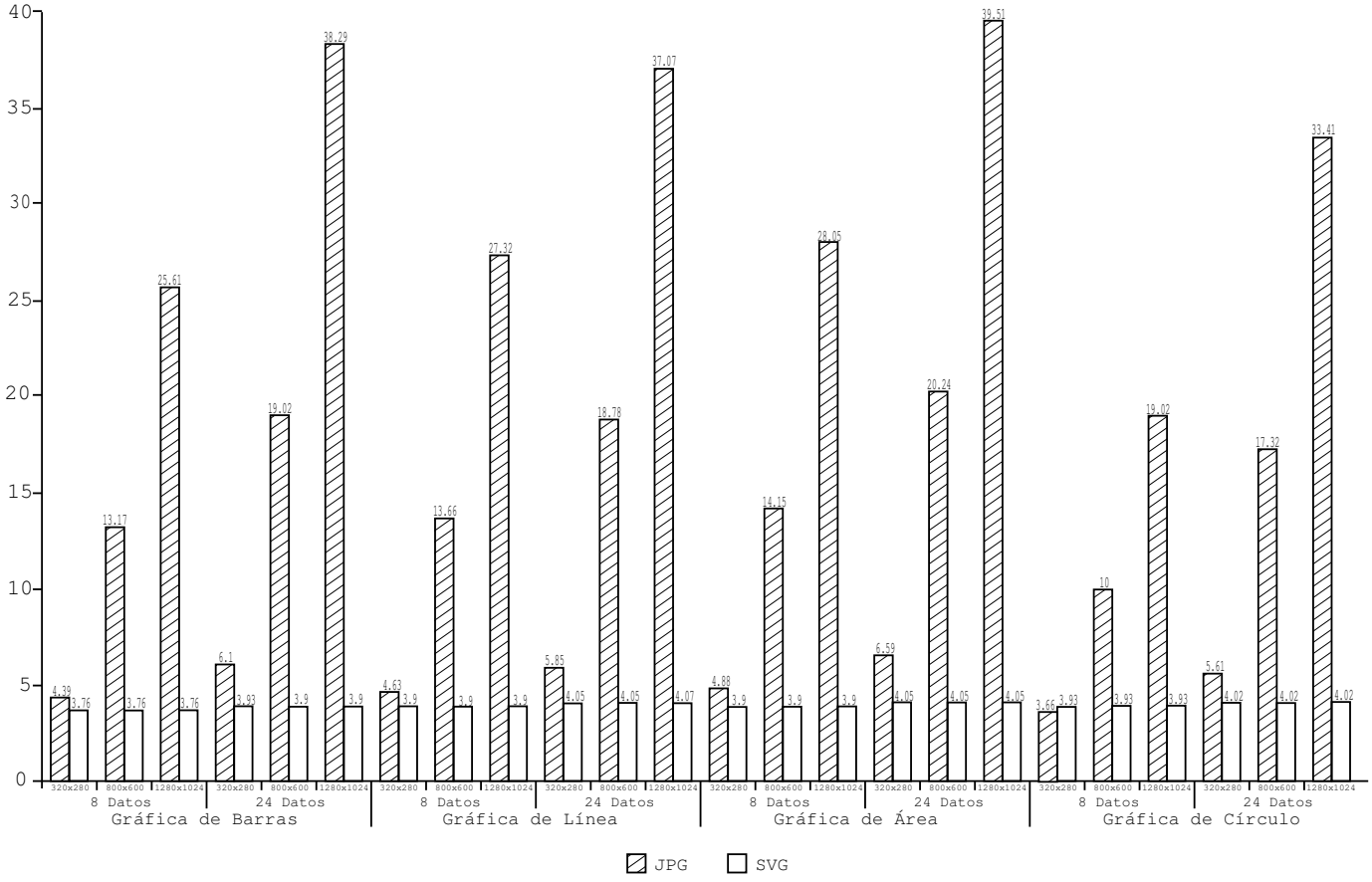
4.3.4.3. Tiempo estimado de transmisión

En el Apéndice B se encuentran las Tablas con los resultados obtenidos del tiempo estimado de transmisión para los formatos SVG y JPG.

La Figura 4.15 en la página anterior ilustra las comparaciones entre los formatos JPG y SVG de las distintas gráficas a diferentes resoluciones para 8 y 24 datos, con valores comprendidos entre [-1000000, 1000000] para las gráficas de barras, línea y área, y entre [0, 1000000] para la gráfica de círculo.

Esta gráfica tampoco muestra comportamientos extraños, el tiempo de transmisión es directamente proporcional al tamaño del archivo.

Figura 4.17: Tiempo estimado de transmisión considerando una transmisión de 46Kbps



Conclusiones

Después de lo investigado, desarrollado y de los resultados obtenidos se puede concluir que la biblioteca EzGraphSVG de esta tesis cumple con los siguientes factores de calidad:

Eficiencia. Como se mostró en la sección 4.3 en la página 78, las gráficas (barras, línea, área, círculo) SVG generadas dinámicamente consumen mucho procesador a comparación de las gráficas JPG estáticas, sin embargo el consumo de CPU para generar las gráficas a diferentes resoluciones es casi el mismo, a diferencia de los JPG, que aumenta a medida que la resolución de la gráfica aumenta.

El espacio de almacenamiento de la biblioteca es muy reducido a comparación de los JPG (16.7 KB el archivo SVG más grande contra 162 KB el archivo JPG más grande). Además el almacenamiento de las gráficas SVG a diferente resolución es el mismo y el almacenamiento de JPG aumenta a medida que aumenta su resolución. Esto implica que la transmisión por Internet es muy rápida de los archivos SVG, y aunándole el tiempo de despliegue en el cliente se puede concluir que la aplicación de esta tesis es muy eficiente.

Por otro lado, las grandes velocidades de los microprocesadores actuales, junto con el aumento considerable de las memorias centrales y las conexiones a Internet a alta velocidad, hacen que los recursos de consumo de CPU, almacenamiento y transmisión no sean actualmente fundamentales para la medida de la eficiencia. De cualquier forma, aunque se pueda transmitir y desplegar rápidamente las gráficas, SVG sigue teniendo características superiores a JPG, como la calidad visual, interactividad, selección de texto, zoom, pan, y espacio de almacenamiento reducido.

Portabilidad. EzGraphSVG ha sido probada en plataformas Linux y Windows utilizando el plug-in Adobe SVGView 3.01.

Corre sin cambio alguno en Windows utilizando los navegadores Opera 7.x, Mozilla 1.7.3 y FireFox 1.0. Debido a que I.E. 6.x JScript no cumple con la especificación de ECMAScript, la aplicación corre con unos ligeros cambios (variables y funciones globales) que están documentados en el código del módulo CommonObjs.js.gz, y aún así antes de desplegar cualquier gráfica manda un mensaje de error de *variable no definida.*

Corre sin ningún cambio en Linux utilizando los navegadores Opera 7.x, Mozilla 1.7.3 y FireFox 1.0

Por lo tanto, esta aplicación es completamente portable sin efectuar cambio alguno a diferentes plataformas con navegadores Web que implementen la especificación ECMAScript.

Verificabilidad. La verificabilidad no puede ser explícita, porque la aplicación SVG depende del plug-in y navegador Web, por lo tanto las pruebas de ensayo se aplican al navegador Web.

Integridad. La biblioteca no tiene capacidades de integridad debido a que JavaScript no puede encapsular propiedades y métodos (ambos son de ámbito público).

Facilidad de utilización. En diagrama de casos de uso en la página 74 muestra los posibles casos de EzGraphSVG. Para la utilización basta con crear un objeto del tipo de gráfica que se desee utilizar y

suministrar los datos a su correspondiente método, como se indico en la plantilla SVG en la página 75 y en las instrucciones de uso en la página 76.

El suministro de los datos pude ser mediante scripts cliente o servidor, extrayendo los datos de un archivo de texto, procesando XML o una RDBMS utilizando programación del lado del servidor.

Exactitud. EzGraphSVG cumple con los requisitos establecidos en la sección 4.2.1.1 en la página 66.

Extensibilidad. Gracias a que EzGraphSVG está desarrollada con el paradigma de Programación Orientado a Objetos y con ayuda de los diagramas de clases que se encuentran en las páginas 72 y 73 se pueden modificar los métodos de los objetos o crear nuevos objetos y métodos.

Reutilización. EzGraphSVG está diseñada de modo que se puedan reutilizar sus módulos para crear nuevas gráficas o construir variantes de gráficas, como por ejemplo:

- Histograma
- Dispersión
- Línea múltiple
- Área múltiple
- Barras agrupadas
- Barras horizontales
- Con efectos (3D, perspectiva, rotación, proyección de sombra, etc.)

Compatibilidad. EzGraphSVG puede ser combinada con otras tecnologías Web, tanto del lado del servidor como del cliente. Ver la aplicación de encuestas en el anexo E.3 en la página 99.

Discusiones

Actualmente la tendencia del desarrollo de sistemas computacionales es el diseño de éstos para ser utilizados vía Web, por lo que la biblioteca EzGraphSVG, reduce costos y tiempo en la generación de gráficos dinámicos. Además si las gráficas programadas no se adecuan a algún proyecto, se pueden reutilizar o modificar sus módulos para crear variantes o nuevas gráficas.

EzGraphSVG es un *framework* que se encarga de la parte de la presentación (gráficos) y la lógica (programación) dentro del flujo de trabajo. Además, cualquiera de las gráficas que se generan EzGraphSVG tienen las características de gráfico interactivo manejador de datos.

Indudablemente los mapas de píxeles nunca serán sustituidos totalmente, porque existen áreas donde son inmejorables, como por ejemplo, la fotografía digital. Los gráficos vectoriales deben utilizarse hasta donde se pueda y de ser necesario combinarse con los mapas de píxeles, a fin de tener lo mejor de ambos mundos.

SVG ya está en escena y programas comerciales como **Corel Draw** (www.corel.com) e **Illustrator** (www.adobe.com) tienen soporte para exportar sus formatos nativos a SVG. Tal vez no sustituyan sus formatos, pero ya están preparados para el nuevo estándar de gráficos vectoriales Web. Por otro lado, el proyecto **Inkscape** (www.inkscape.org), software OpenSource para gráficos vectoriales, ha tenido un enorme desarrollo y tiene como formato nativo SVG, así como el programa comercial **WebDraw** (www.jasc.com). Además, los proyectos **Batik** (xml.apache.org/batik), **X-Smiles** (www.xsmiles.org) y **Mozilla SVG** (www.mozilla.org /projects/svg) son los primeros navegadores Web que soportan nativamente SVG. Pero eso no es todo, **KDE** (www.kde.org) y **GNOME** (www.gnome.org) ya utilizan SVG para sus temas de escritorio, mediante las

bibliotecas *ksvg* y *librsvg* respectivamente. Y para sorpresa de todos, **BitFlash** (www.bitflash.com) es la primera compañía en crear software (*BitFlash Mobil SVG Player*) para visualizar gráficos SVG en dispositivos portátiles.

A SVG le espera un buen futuro porque se puede utilizar para gráficos estáticos, gráficos interactivos, gráficos dinámicos manejadores de datos, publicaciones impresas, temas de escritorio, presentaciones, animaciones y videojuegos sencillos.

SVG puede llegar a simplificar extraordinariamente el flujo de trabajo para la Web. En una aplicación única se podrá generar casi todo el contenido de las páginas, y convertirse en un formato universal: todos los programas podrán abrir todo tipo de archivos. Los gráficos SVG no serán, como hasta ahora, una versión de un gráfico que ha pasado por varias aplicaciones. Se ahorraran conversiones, pasos de un programa a otro, tareas de optimizar, cambiar de tamaño, etc.

Como profesionista, este trabajo me resultó muy útil porque enriqueció mis conocimientos en las tecnologías JavaScript, DOM, XML, DTD, XSLT, UML y, al mismo tiempo, aprendí una nueva tecnología de gráficos vectoriales para la Web: SVG.

A nivel programación, experimenté una codificación lenta porque no se puede depurar el código SVG generado dinámicamente mediante el DOM, el plug-in SVGView no lo permite y, por lo tanto, encontrar errores de de sintaxis y de lógica es muy engorroso. Además, cuando se utilizan acentos (en los comentarios) el plug-in falla y no genera nada, y lo peor del caso es que no indica a qué se debe ésto.

Durante el proceso de investigación noté el prominente panorama que le depara a XML como formato de intercambio y estructuras de datos, y día a día ha ido ganando terreno. Un gran número de de aplicaciones pueden ser desarrollados gracias a esta tecnología y a su creciente soporte de la API para el procesamiento del mismo en los lenguajes de programación modernos.

Cuando el lenguaje XML sea el estándar para la creación de páginas Web, los archivos SVG serán una parte más de las páginas, y no un accesorio que requiere de un plug-in.

Todo el código de la biblioteca que se desarrolló en esta tesis corresponde a software de Fuente Abierta (OpenSource), de modo que cualquier persona puede reutilizar libremente el código de cualquier forma que le sea útil, y por lo tanto, permite que para futuros trabajos e investigaciones esta biblioteca pueda ser expandida y mejorada agregando más tipos de gráficas y/o añadiendo efectos decorativos, o incluso incorporarla a los lenguajes del lado del servidor, y del mismo modo usarla en otras áreas.

EzGraphSVG falla con las siguientes condiciones anormales:

- Crear un objeto de cualquier tipo de gráfica y llamar más de una vez su método para dibujar.
- Crear más de un objeto de la misma gráfica y llamar a su método de dibujo por cada objeto creado.
- Crear más de un objeto de diferentes gráficas y llamar a su método de dibujo correspondiente por cada objeto.
- Cuando una cadena lleva algún acento en los parámetros que se pasan a las gráficas

Propongo que un futuro, cuando el plug-in Adobe SVGView soporte mutaciones de eventos, solucionar estas fallas implementado un mecanismo de mutación de eventos SVG aplicados al elemento *Graph*, que es el elemento <g> que agrupa todo el contenido gráfico que genera EzGraphSVG.

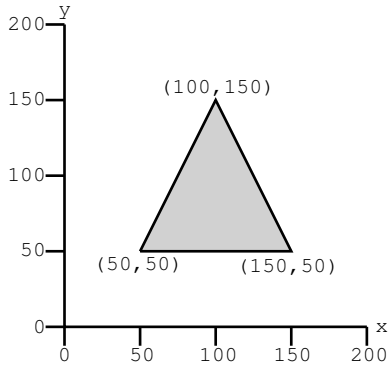
Por último, la información del proyecto EzGraphSVG está en la página Web **http://ezgraphsvg.objectis.net**

Apéndice A

Técnica para convertir coordenadas cartesianas a SVG

En el sistema de coordenadas cartesianas el punto de origen es (0,0). Suponiendo que solo se utiliza el primer cuadrante, el origen está en la esquina inferior izquierda del área de dibujo, y las coordenadas en el eje ***Y*** se incrementan de abajo hacia arriba. La Figura A.1 muestra las coordenadas de un triángulo en coordenadas cartesianas.

Figura A.1: Triángulo dibujado con coordenadas cartesianas



Debido a que en el *sistema de coordenadas de SVG*[1] las coordenadas de el eje ***Y*** incrementan de arriba hacia abajo. Las coordenadas tienen que ser recalculadas. En lugar de hacer eso a mano, se puede utilizar una secuencia de transformaciones para que SVG realice todo el trabajo. Primero, convertir la figura a SVG, con las coordenadas exactamente como se muestran en el Listado 12 (incluir también los ejes como una referencia). La figura aparecerá de cabeza. Nótese que la imagen en la Figura A.2 no está invertida de izquierda a derecha, debido a que la dirección de las coordenadas del eje ***X*** es la misma en coordenadas cartesianas y coordenadas SVG.

Para terminar la conversión realice los siguientes pasos:

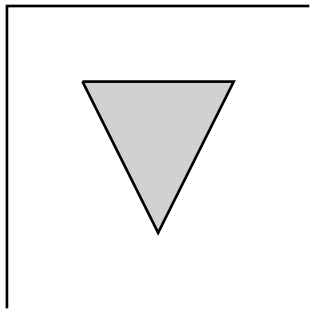
1. Encontrar la coordenada máxima "***Y***" en el dibujo original. En este caso 200.
2. Encerrar la figura entera en un elemento <g>.
3. Aplicar una transformación que mueva el sistema de coordenadas hacia abajo por el valor máximo "***Y***".

[1] Véase la sección 3.2.2 en la página 44

Listado 12 Uso directo de coordenadas cartesianas

```
<svg width="200" height="200">
    <!--ejes-->
    <line x1="0" y1="0" x2="200" y2="0" />
    <line x1="0" y1="0" x2="0" y2="220" />
    <!--triángulo-->
    <polygon points="50 50, 100 150, 150 50"
       style="fill: gray; stroke: black" />
</svg>
```

Figura A.2: Resultado en coordenadas SVG



```
transform = "translate(0, yMax)"
```

4. La siguiente transformación será escalar el eje "y" por un factor de -1, lo que ocasionará que se invierta la figura con respecto al eje "y".

```
transform = "translate(0, yMax) scale(1, -1)"
```

El Listado 13 incorpora esta transformación, produciendo un rectángulo como en coordenadas cartesianas.

Listado 13 Transformación de coordenadas cartesianas

```
<svg width="200" height="200">
  <g transform = "translate(0, yMax) scale(1, -1)">
    <!--ejes-->
    <line x1="0" y1="0" x2="200" y2="0" />
    <line x1="0" y1="0" x2="0" y2="220" />
    <!--triángulo-->
    <polygon points="50 50, 100 150, 150 50"
       style="fill: gray; stroke: black" />
  <g>
</svg>
```

Apéndice B

Datos de las pruebas de rendimiento

B.1. Valores de los datos

Tabla B.1: 8 datos con valores [-1000000, 1000000]

Dato 0	Dato 1	Dato 2	Dato 3	Dato 4	Dato 5	Dato 6	Dato 7
-103941.32	-637447.75	-90278.95	18903.66	932415.9	886445.7	206480.94	783720.88

Tabla B.2: 8 datos con valores [-1, 1]

Dato 0	Dato 1	Dato 2	Dato 3	Dato 4	Dato 5	Dato 6	Dato 7
0.72	0.79	-0.87	-0.94	-0.59	0.15	-0.24	0.8

Tabla B.3: 24 datos con valores [-1000000, 1000000]

Dato 0	Dato 1	Dato 2	Dato 3	Dato 4	Dato 5	Dato 6	Dato 7
-472516.05	92149.8	-707860	837682.7	-358142.75	-927044.96	686878.64	-598740.06
Dato 8	Dato 9	Dato 10	Dato 11	Dato 12	Dato 13	Dato 14	Dato 15
465192.08	-834105.58	-7894.91	115427.53	-17140.07	915535.1	-219399.6	75034.67
Dato 16	Dato 17	Dato 18	Dato 19	Dato 20	Dato 21	Dato 22	Dato 23
223898.31	-493848.9	379266.84	240485.38	660061.07	265466.43	-775927.23	-394609.91

Tabla B.4: 24 datos con valores [-1, 1]

Dato 0	Dato 1	Dato 2	Dato 3	Dato 4	Dato 5	Dato 6	Dato 7
-0.76	0.17	0.3	0.44	0.82	0.18	-0.42	-0.42
Dato 8	Dato 9	Dato 10	Dato 11	Dato 12	Dato 13	Dato 14	Dato 15
0.03	-0.33	0.11	-0.62	0.58	-0.88	0.03	-0.87
Dato 16	Dato 17	Dato 18	Dato 19	Dato 20	Dato 21	Dato 22	Dato 23
0.71	-0.27	-0.27	0.88	-0.31	0.23	-0.82	-0.51

B.2. Resultados de porcentaje de CPU

Porcentaje de CPU para desplegar las gráficas con 8 datos

Tipo de gráfica	Resolución	Rango de los valores	JPG CPU %	SVG CPU %
Barras	640x480	[-1000000, 1000000]	6	25.9
	800x600		10.2	27.5
	1280x1024		18.6	29.1
	640x480	[-1,1]	7.1	26.5
	800x600		10.3	28.4
	1280x1024		18.7	30.7
Línea	640x480	[-1000000, 1000000]	6.1	25.1
	800x600		10.2	26.2
	1280x1024		18.5	25
	640x480	[-1,1]	6.3	25.8
	800x600		9.9	27.3
	1280x1024		18.7	26
Área	640x480	[-1000000, 1000000]	7	25.2
	800x600		10.3	26.9
	1280x1024		19.1	26
	640x480	[-1,1]	6.9	26.2
	800x600		10.4	28.4
	1280x1024		19.3	17.2
Círculo	640x480	[-1000000, 1000000]	6.3	25.3
	800x600		11.1	31.5
	1280x1024		18.9	35
	640x480	[-1,1]	6.8	25.4
	800x600		11.1	31.4
	1280x1024		18.9	34.8

Porcentaje de CPU para desplegar las gráficas con 24 datos

Tipo de gráfica	Resolución	Rango de los valores	JPG CPU %	SVG CPU %
Barras	640x480	[-1000000, 1000000]	6.6	51.6
	800x600		10.7	46.9
	1280x1024		19	48.2
	640x480	[-1,1]	6.6	51.7
	800x600		10.2	46.8
	1280x1024		18.5	49.9
Línea	640x480	[-1000000, 1000000]	7.3	49.6
	800x600		11	44.9
	1280x1024		19.7	40.6
	640x480	[-1,1]	7	50
	800x600		10.9	44.8
	1280x1024		19.4	40.4
Área	640x480	[-1000000, 1000000]	6.3	50.2
	800x600		11.2	45.6
	1280x1024		19.1	42
	640x480	[-1,1]	6.5	50.1
	800x600		10.9	46
	1280x1024		18.9	41.4
Círculo	640x480	[-1000000, 1000000]	7.2	36.1
	800x600		11.7	44
	1280x1024		19.4	47
	640x480	[-1,1]	7.4	36.1
	800x600		11.5	43.3
	1280x1024		19.7	46.6

B.3. Resultados de almacenamiento

Espacio de almacenamiento (en kilobytes) de las gráficas con 8 datos

Tipo de gráfica	Resolución	Rango de los valores	JPG almacenamiento	SVG almacenamiento
Barras	640x480	[-1000000, 1000000]	18	15.4
	800x600		54	15.4
	1280x1024		105	15.4
	640x480	[-1,1]	18	15.4
	800x600		50	15.4
	1280x1024		95	15.4
Línea	640x480	[-1000000, 1000000]	19	16
	800x600		56	16
	1280x1024		112	16
	640x480	[-1,1]	17	16
	800x600		47	16
	1280x1024		90	16
Área	640x480	[-1000000, 1000000]	20	16
	800x600		58	16
	1280x1024		115	16
	640x480	[-1,1]	18	15.9
	800x600		49	15.9
	1280x1024		93	15.9
Círculo	640x480	[-1000000, 1000000]	15	16.1
	800x600		41	16.1
	1280x1024		78	16.1
	640x480	[-1,1]	16	16.1
	800x600		43	16.1
	1280x1024		82	16.1

Espacio de almacenamienot (en kilobytes) de las gráficas con 24 datos

Tipo de gráfica	Resolución	Rango de los valores	JPG almacenamiento	SVG almacenamiento
Barras	640x480	[-1000000, 1000000]	25	16.1
	800x600		78	16
	1280x1024		157	16
	640x480	[-1,1]	23	15.9
	800x600		71	15.9
	1280x1024		134	15.9
Línea	640x480	[-1000000, 1000000]	24	16.6
	800x600		77	16.5
	1280x1024		152	16.7
	640x480	[-1,1]	23	16.5
	800x600		69	16.5
	1280x1024		129	16.5
Área	640x480	[-1000000, 1000000]	27	16.6
	800x600		83	16.6
	1280x1024		162	16.6
	640x480	[-1,1]	25	16.5
	800x600		75	16.5
	1280x1024		140	16.5
Círculo	640x480	[-1000000, 1000000]	23	16.5
	800x600		71	16.5
	1280x1024		137	16.5
	640x480	[-1,1]	24	16.4
	800x600		73	16.4
	1280x1024		142	16.4

B.4. Resultados de transmisión

Tiempo estimado de transmisión (en segundos) de las gráficas con 8 datos

Tipo de gráfica	Resolución	Rango de los valores	JPG tiempo	SVG tiempo
Barras	640x480	[-1000000, 1000000]	4.88	3.9
	800x600		14.5	3.9
	1280x1024		28.05	3.9
	640x480	[-1,1]	4.39	3.88
	800x600		11.95	3.88
	1280x1024		22.68	3.88
Línea	640x480	[-1000000, 1000000]	6.59	4.05
	800x600		20.24	4.05
	1280x1024		39.51	4.05
	640x480	[-1,1]	6.1	4.02
	800x600		18.29	4.02
	1280x1024		34.15	4.02
Área	640x480	[-1000000, 1000000]	3.66	3.93
	800x600		10	3.93
	1280x1024		19.02	3.93
	640x480	[-1,1]	3.9	3.93
	800x600		10.49	3.93
	1280x1024		20	3.93
Círculo	640x480	[-1000000, 1000000]	5.61	4.02
	800x600		17.32	4.02
	1280x1024		33.41	4.02
	640x480	[-1,1]	5.85	4
	800x600		17.8	4
	1280x1024		34.63	4

Tiempo estimado de transmisión (en segundos) de las gráficas con 24 datos

Tipo de gráfica	Resolución	Rango de los valores	JPG tiempo	SVG tiempo
Barras	640x480	[-1000000, 1000000]	25	16.1
	800x600		78	16
	1280x1024		157	16
	640x480	[-1,1]	23	15.9
	800x600		71	15.9
	1280x1024		134	15.9
Línea	640x480	[-1000000, 1000000]	24	16.6
	800x600		77	16.6
	1280x1024		152	16.7
	640x480	[-1,1]	23	16.5
	800x600		69	16.5
	1280x1024		129	16.5
Área	640x480	[-1000000, 1000000]	27	16.6
	800x600		83	16.6
	1280x1024		162	16.6
	640x480	[-1,1]	25	16.5
	800x600		75	16.5
	1280x1024		140	16.5
Círculo	640x480	[-1000000, 1000000]	23	16.5
	800x600		71	16.5
	1280x1024		137	16.5
	640x480	[-1,1]	24	16.4
	800x600		73	16.4
	1280x1024		142	16.4

Apéndice C

Código de la aplicación

C.1. EzGraphSVG

El código de ***EzGraphSVG*** está en el directorio del mismo nombre en el CD que acompaña esta tesis, los scripts están comprimidos y tienen los siguientes nombres:

- La biblioteca de objetos comunes se llama CommonObjs.js.gz
- La gráfica de barras se llama Bars.js.gz
- La gráfica de línea y área se llama LineaArea.js.gz
- La gráfica de circulo se llama Circle.js.gz

C.2. Aplicación de encuestas

El código de esta aplicación está en en el directorio ***encuesta***, y contiene los siguientes archivos:

- configura_encuestas.sql
- voto.html
- procesa_encuestas.php
- CommonObjs.js.gz
- Bars.js.gz

C.3. Contenido adicional del CD

C.3.1. Navegadores Web

- Windows
    - Mozilla 1.73
    - FireFox 1.0
    - Opera 7.5

- Linux
  - Mozilla 1.7
  - FireFox 1.0
  - Opera 7.5

C.3.2. Adobe SVGView plug-in

- Windows
  - Adobe SVGView 3.01
- Linux
  - Adobe SVGView 3.01 beta3

C.3.3. GhostView

- Windows
  - GhostView
  - GhostScript

Tesis en formato PS y PDF

- tesis.ps
- tesis.pdf

C.4. Cómo instalar el plug-in SVGViewer para Mozilla 1.x, FireFox 1.0 y Opera 7.x

El plug-in SVGView se instala automáticamente para Explorer 6.x en Windows, y en Mozilla 1.x en Linux. Para los demás exploradores hay realizar pasos adicionales.

A continuación se muestra cómo instalar el plug-in en ambos sistemas operativos; en los dos casos primero hay realizar la instalación predeterminada para poder copiar unos archivos que el plug-in instala.

C.4.1. Windows

- Copiar los archivos NPSVG3.dll y NPSVG3.zip a la carpeta de plugins de Opera, Mozilla o FireFox.

C.4.2. Linux

- Copiar el archivo libNPSVG3.so a la carpeta de plugins de Opera o FireFox.

Apéndice D

Galería de capturas

Este apéndice contiene capturas de las gráficas de barras, línea, área y círculo corriendo en Linux Mandrake 10 y Window98, con diferentes navegadores Web.

Cada gráfica fue generada con valores aleatorios. El Listado 14 muestra el código, pero se tienen que agregar algunos datos según el tipo de gráfica, como se muestra en las instrucciones de uso en la página 76.

Listado 14 Listado para generar gráficas con valores aleatorios

```
<svg onload="init(evt); usage(evt);">
<script xlink:href="EzGraphSVG/CommonObjs.js.gz"></script>
<script xlink:href="EzGraphSVG/..."></script>
<script><![CDATA[
function usage(evt){
  var grafica = new // Tipo de gráfica
  grafica.draw // Método de dibujo con el parámetro: generarDatos()
}
function generarDatos() {
  var numDatos = Math.round( Math.random()*(20-5) ) + 5; // Num. aleatorios
  var datosArray = new Array(numDatos);
  for(var i=0; i<numDatos; i++)
    datosArray[i] = datos(i);
  return(datosArray);
}
function datos(i){
  var array = new Array(2);
  // Valores positivos y negativos
  array[0] = (Math.random()*10000)*(Math.pow(-1, Math.round(Math.random())+1));
  //array[0] = Math.random()*(10000-100)+100; // Valores positivos
  array[1] = "Dato " + i;
  return(array);
}
]]></script>
</svg>
```

Figura D.1: Gráfica de barras (Opera7.5 en Windows98)

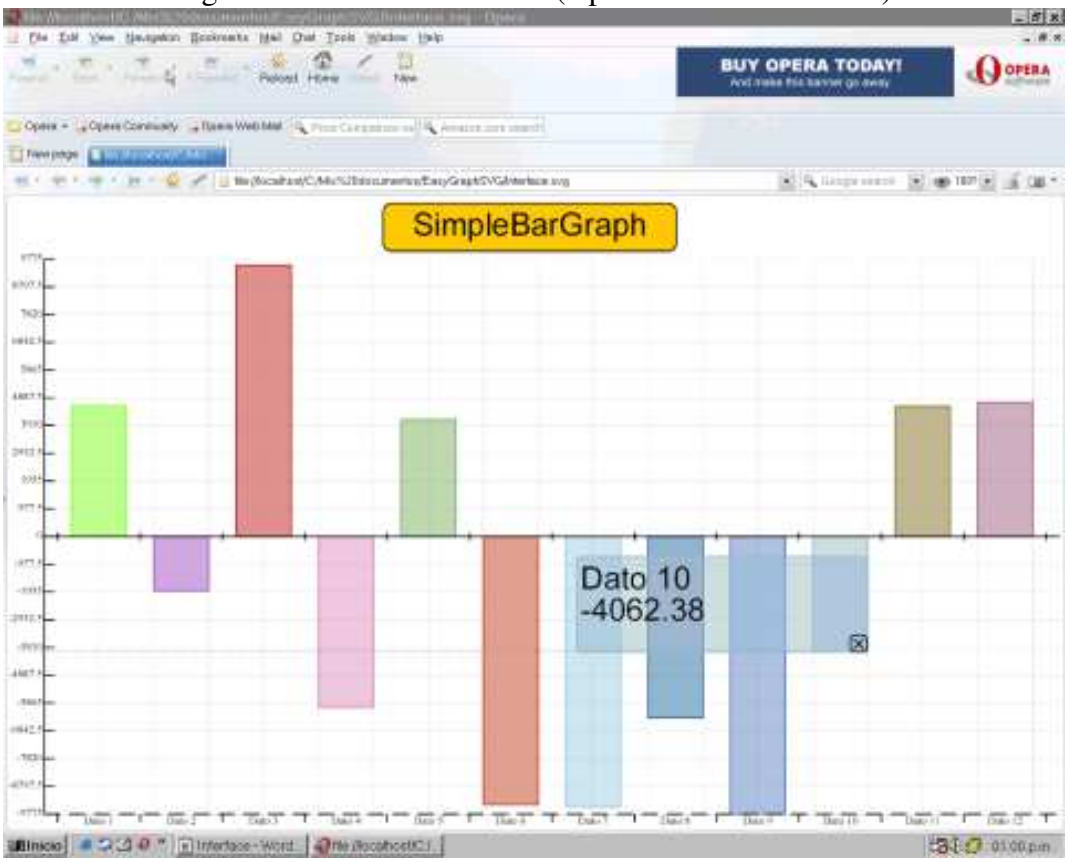


Figura D.2: Gráfica de barras (IE6 en Windows98)

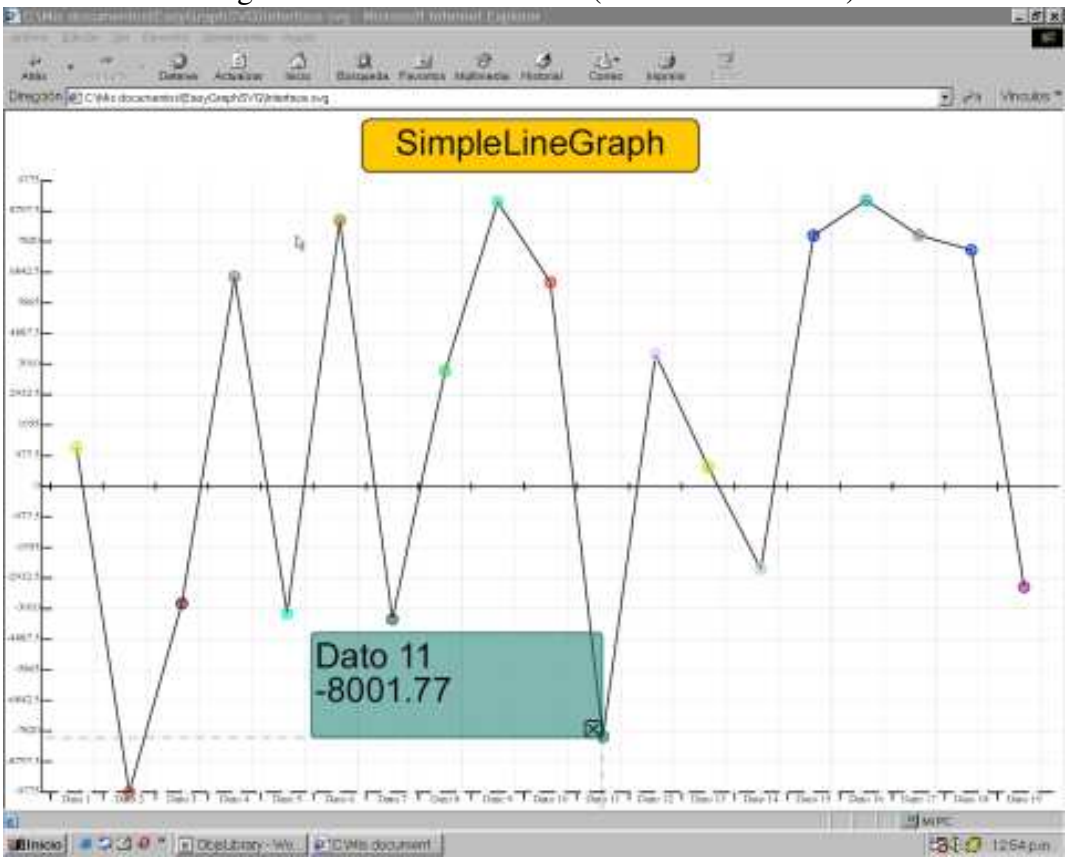


Figura D.3: Gráfica de línea (FireFox1.0 en Linux)

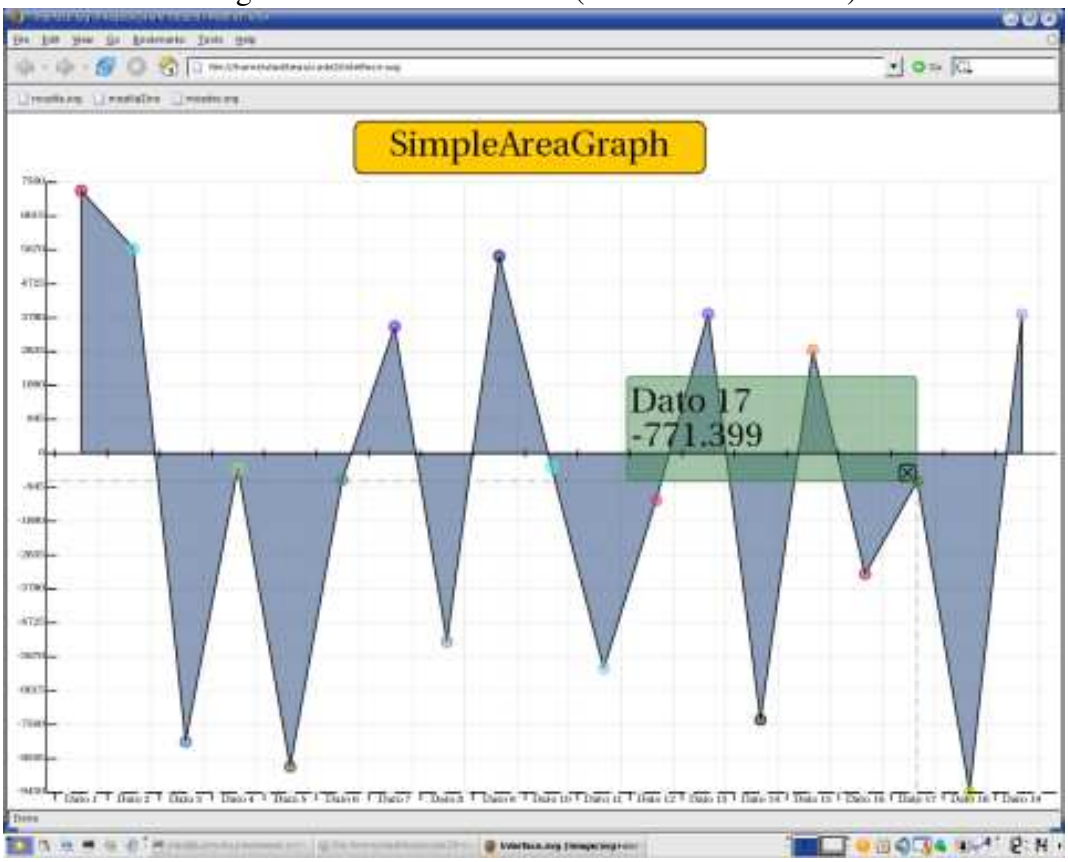
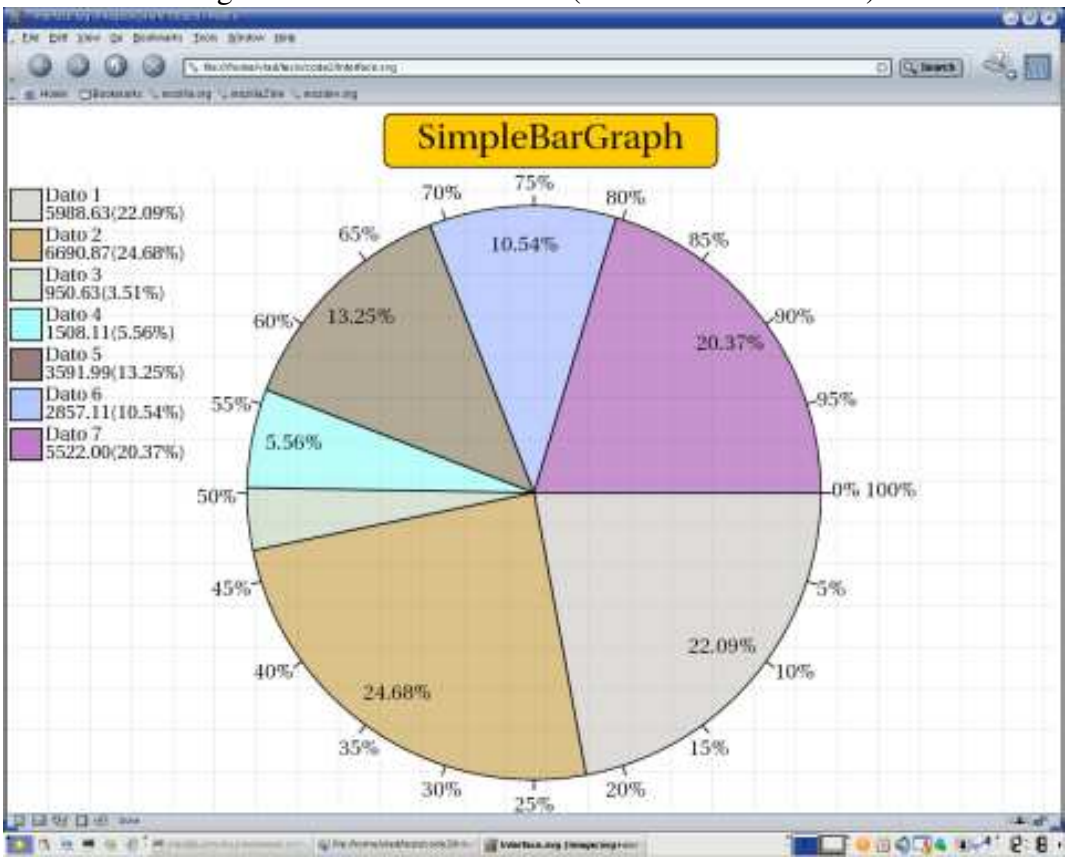


Figura D.4: Gráfica de barras (Mozilla1.7.3 en Linux)



Apéndice E

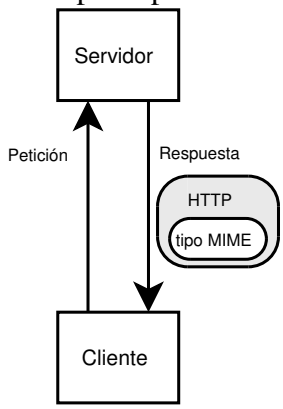
Programación SVG del lado del servidor

E.1. Tipos MIME

Los servidores Web envían archivos, incluyendo archivos SVG, que solicitan los clientes, los cuales intentan desplegar esos archivos. Para los clientes, debe haber un mecanismo para identificar que tipo de contenido llega. La solución para este problema es usar los tipos ***MIME*** (***M**ultipurpose **I**nternet **M**ail **E**xtensions*, *Extensiones de correo Internet de propósito múltiple*) del documento.

MIME es un estándar para el intercambio de correo que soporta datos gráficos, de audio, de vídeo y de otros binarios. Los tipos MIME son parte del protocolo HTTP, lo que significa que un tipo MIME es enviado en el flujo HTTP, como se ilustra en la Figura E.1. [26, p. 691]

Figura E.1: El principio cliente/servidor



Los tipos MIME deben ser configurados tanto en el navegador Web como en el servidor Web. En el caso del navegador Web, el plug-in de SVGView de Adobe hace está tarea automáticamente.

El tipo MIME de SVG es ***image/svg+xml*** con extensión ***svg*** o ***svgz***. [26, p. 692]

E.2. SVG con los más importantes lenguajes del lado del servidor

Cuando está configurado correctamente un sitio Web para archivos SVG, todos los datos servidos desde el servidor que tengan la extensión ***svg*** o ***svgz*** tendrán el tipo MIME correcto, satisfaciendo el navegador del cliente si el plug-in está instalado.

Supóngase que un script Perl del lado del Servidor genera código SVG. Este script se ejecuta llamando http://nombreservidor/nombrescript.pl. Sin embargo, cuando se generan archivos SVG desde el servidor, se

necesita configurar el tipo MIME manualmente, debido a que los scripts del lado del servidor no tienen la extensión ***svg*** o ***svgz.***

Los lenguajes del lado del servidor tienen capacidades para lectura/escritura de archivos, procesamiento XML, interacción con RDBMS, etc., y gracias a esas capacidades la Web a crecido enormemente.

E.2.1. PHP

Cuando se utiliza PHP, el paso más importante es configurar el tipo MIME correcto, mediante la función ***header()***, el cual debe ser llamado antes de que la primera salida de texto sea enviada al cliente [26, p. 696]:

```
header("Content-type: image/svg+xml");
```

Pero hay otra cuestión. La primera línea de un documento XML, y por lo tanto de SVG también, es:

```
<?xml version="1.0" ?>
```

Entonces el script mínimo necesario para generar archivos SVG desde PHP es:

```
<?php
  header("Content-type: image/svg+xml");
  echo("<?xml version='1.0' ?>");
?>
<svg ... >
   ...
  <?php
     //Código adicional PHP
  ?>
  ...
</svg>
```

E.2.2. ASP/ASP.NET

No importa si se utiliza ASP o ASP.NET, el tipo MIME de respuesta se configura asignando a la propiedad ***ContentType*** del objeto ***Response*** [26, p. 697]:

```
Response.ContentType = "image/svg+xml"
```

Entonces el script mínimo necesario para generar archivos SVG desde ASP (con extensión de archivo *asp*) o desde ASP.NET (con extensión de archivo *aspx*) es:

```
<% Response.ContentType = "image/svg+xml"%>
<?xml version="1.0" ?>
<svg ... >
   ...
  <%
     //Código adicional ASP
 %>
  ...
</svg>
```

E.2.3. Perl

Con Perl, se necesita enviar el encabezado completo HTTP cuando se crea el script [26, p. 697]:

```
print "Content-type: image/svg+xml\n";
```

Entonces el script mínimo necesario para generar archivos SVG desde Perl es:

```
#!/usr/bin/perl
print "Content-type: image/svg+xml\n";
print <<'EOF'
<?xml version="1.0" ?>
<svg ... >
  ...
  //Código adicional Perl
  ...
</svg>
EOF
```

E.2.4. JSP

Se puede configurar la respuesta del tipo MIME desde JSP usando la siguiente línea [26, p. 698]:

```
<%@ page contentType="image/svg+xml"%>
```

A diferencia de otros lenguajes y/o tecnologías, se puede poner esta directiva virtualmente en cualquier parte de la página, aún después de la primera salida SVG. Sin embargo, es buen estilo usar esta directiva tan pronto como sea posible.

Entonces el script mínimo necesario para generar archivos SVG desde JSP:

```
<%@  page contentType="image/svg+xml"%>
<?xml version="1.0" ?>[1]
<svg ... >
   ...
  <%
     //Código adicional JSP
 %>
  ...
</svg>
```

E.3. Aplicación de encuestas

Este aplicación sirve para emitir encuestas desde un página Web y mostrar el resultado del total de las encuestas en forma gráfica usando. Esta encuesta ilustra un ejemplo práctico de la utilización de EzGraphSVG.

La idea general es presentar información en forma gráfica cuando el usuario haga clic sobre el botón "Mostrar resultados". Para este fin se añade su voto a la base de datos, se obtienen todos los votos de la base de datos y se genera la gráfica de los resultados actuales.

[1] Andrew Watt [26, p. 698] menciona que se debe incluir esta línea, sin embargo en en Tomcat 5.0.25 no funciona.

El código de está aplicación se encuentra en el CD que acompaña está tesis, en el directorio **encuesta**. Las Figuras E.2 y E.3 muestran la entrada y salida de esta aplicación.

Figura E.2: Formulario de la encuesta

Figura E.3: Resultado de la encuesta

Apéndice F

Glosario de términos

Este glosario contiene una lista de términos y expresiones utilizados principalmente en el área de graficación por computadora.

canal alfa (alpha chanel): Un término que es habitual en la terminología de gráficos. Representa la transparencia de una imagen. Una imagen puede tener una gama de valores alfa, desde completamente transparentes a completamente opacos. [8, p. 5]

CSS: (Cascading Style Sheets, Hojas de Estilo en Cascada). Es un medio de especificar propiedades como el color o fuentes para elementos de documentos ***HTML/XML***. SVG utiliza muchas de las mismas definiciones de propiedades como CSS2 y el estilo puede ser aplicado usando hojas de estilo externas o en línea. [26, p. 1061]

entrelazado (interlaced): Es una opción para dar la "ilusión" de un tamaño más pequeño en los formatos JPEG **progresivo** o el GIF **entrelazado**. Esto es muy útil en los gráficos Web. Cuando se descarga el archivo, primero se muestra una resolución muy deficiente, y gradualmente mejora a medida que se va descargando el archivo. Cuando se usa esta opción, los archivos se guardan de modo distinto, pero siguen pareciendo iguales cuando están completamente cargados. [8, p. 79]

escala de grises (grayscale): Este modelo de color es un método que sirve para mostrar los modelos de color en blanco y negro. Tiene un solo canal (***value***, véase ***HSV***). Esto le permite mostrar 255 sombras de grises, siendo 0 el negro y 255 el blanco. [15, p. 194][8, p. 7]

gráficos: (del gr. *graphikós*, derivado del gr. *grapho*) son la representación de ideas, descripciones, operaciones y demostraciones que se logran por medio de figuras, dibujos o signos con el fin de transmitir la información de un modo meramente visual.

HSV: El modelo de color es similar a ***RGB***, pero cada uno de sus canales representa el *matiz(hue)*, la *saturación(saturation)* y el *valor(value)*. Es un enfoque más intuitivo para describir el color. [8, p. 7][8, p. 191-192]

- **Hue.** Este canal determina el color real de la imagen. Utiliza una rueda de color de 360 grados para determinar el color. El rojo está en 0 grados, por ejemplo, y el cían está en 180 grados.
- **Saturation.** Determina la pureza de un color. Está representado por un porcentaje. Un color con una saturación total es un color puro. Un píxel que no tenga saturación no tendrá color, sera ***escala de grises***.

- **Value.** Este canal representa la luminosidad de una imagen. También es un valor porcentual. Un *píxel* con un valor total será claro; mientras que un *píxel* con 0 de luminosidad sera negro.

indexado (indexed): Este modelo trabaja con un valor fijo de color para cada ***píxel.*** Cada color utilizado en una determinada imagen se pone en un *mapa de color* o *tabla de color*, el cual es parte de un archivo de imagen indexada. Esta tabla de color es usada para mapear el color apropiado a cada píxel que será desplegado.[15, p. 189][8, p. 189-191]

modelos de color: Los modelos de color se usan para clasificar y estandarizar el color. ***RGB***, ***HSV***, ***Escala de grises*** e ***Indexado*** son de los modelos más populares. [15, p. 188]

pan: Es la operación de mover el área visible de un ***canvas*** SVG horizontalmente, verticalmente o diagonalmente. [26, p. 1064]

profundidad de color (bit depth): Un término matemático utilizado para describir el número de colores disponibles en una imagen. Una resolución de 8 bits significa que la pantalla puede mostrar 256 colores distintos al mismo tiempo. Debido a que el ojo humano sólo puede ver alrededor de 10 millones de colores, 24 bits de color es ideal. Véase la Tabla F.1. [8, p. 5]

Tabla F.1: Profundidades de bits y colores disponibles

Profundidad de bits	Colores disponibles
1	2
2	4
4	16
8	256
16	65,536
24	16,777,216

raster: Formación rectangular que consiste en un conjunto de líneas horizontales compuestas por *píxeles* que se utilizan para formar una imagen en un CRT (Cathodic Ray Tube, Tubo de Rayos Catódicos). [11, p. 15][4, p. 1]

render: Es la creación de una representación bidimensional de un objeto basado en las propiedades de su forma y superficie (e.g., una fotografía para imprimir o para desplegar en el monitor). [24, p. 759]

RGB: Significa los tres colores utilizados en el modelo RGB: *red(rojo), green(verde) y blue (azul)*. Es un modelo *aditivo*, donde cada canal se añade para componer cualquier color. En este modelo, cada color se representa como un conjunto de tres valores: un valor para el rojo, un valor para el verde y un valor para el azul. Cada valor tiene un rango entre 0 y 255. Si todos los valores son 0, el color resultante es negro; si todos los valores son 255, el color es blanco. RGB puede representar más de 16 millones de colores (***TrueColor***). 255 valores de rojo x 255 valores de verde x 255 valores de azul = 16,777,216 colores. [8, p. 6-7][8, p. 188-189]

suavizado (anti-aliasing): Una técnica diseñada para suavizar el efecto escalonado que resulta de dibujar primitivas gráficas en una imagen ***raster***. [24, p. 752][8, p. 5]

viewport: Es la región rectangular en la cual el contenido se le aplica ***render***. [26, p. 1067]

zoom: Es la operación de acercar o alejar el ***viewport***.

Bibliografía

[1] CAGLE, Kurt. *SVG Programming: The Graphical Web*. Apress, USA, 2002.

[2] CORTÉS Ferreyra, Gonzalo. *World Wide Web ¡Espectacular!* Alfaomega Grupo Editor, México, 1997.

[3] DEMEL, John T.; MILLER, Michael J. *Gráficas por computadora*. McGraw-Hill, España, 1990.

[4] EISENBERG, J. David. *SVG Essentials*. O'Reilly, USA, 2002.

[5] FLYNT, John P. *Software Engineering*. Thomson Course Technology, USA, 2004.

[6] FOLEY, James et al. *Computer Graphics Principles and Practice. 2nd edition*. Addison-Wesley, USA, 1997.

[7] FRANKLIN, Derek; MAKAR, Jobe. *Macromedia Flash MX ActionScript avanzado Entrenando con Macromedia*. Pearson Educación, México, 2002.

[8] HARDFORD, Alex. *GIMP*. Prentice Hall, Madrid, 2000.

[9] HAROLD, Elliotte Rusty. *Processing XML with Java*. Addison-Wesley, USA, 2003.

[10] HARRIS, Robert L. *Information Graphics: a comprehensive illustrated reference*. Oxford University Press, USA, 1999.

[11] HILL, F. S. Jr. *Computer Graphics using OpenGL. 2nd edition*. Prentince Hall, USA, 2001.

[12] Adobe Systems Incorporated. SVG Zone [en línea]. Technical report, Adobe Systems Incorporated, 2001.

[13] ISSI Camy, Lázaro. *La biblia de JavaScript*. Anaya Multimedia, Madrid, 2002.

[14] JOYANES Aguilar, L.; ZAHONERO MARTÍNEZ, I. *Estructuras de Datos Algoritmos, abstracción y objetos*. McGrawHill, Madrid, 1998.

[15] KYLANDER, Olof S.; KYLANDER, Karin. *GIMP The Official Handbook*. Coriolis, USA, 1999.

[16] LAKER, Micah. *SAMS Teach Yourself SVG in 24 Hours*. SAMS Publishing, USA, 2002.

[17] MANGER, Jason J. *Fundamentos de JavaScript*. McGraw-Hill Osborne, México, 1997.

[18] MARINI, Joe . *The Document Object Model: Processing Structured Documents*. McGraw-Hill/Osborne, USA, 2002.

[19] MORRISON, Michael, et al. *XML Al descubierto*. Prentice Hall, España, España, 2000.

[20] Netscape Communication Corp. *Core JavaScript Guide 1.5 [en línea]*. Disponible en Web: http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide.

[21] POWELL, Thomas A. *HTML 4. Manual de referencia*. Osborne McGraw-Hill, España, 2001.

[22] POWELL, Thomas; SCHNEIDER, Fritz. *JavaScript Manual de referencia*. McGraw-Hill/Osborne Media, España, 2002.

[23] RILEY, Sean. *Game programming with Python*. Charles River Media, USA, 2004.

[24] ROOSENDAL, Ton et al. *Blender 2.3 Guide*. Blender Foundation, Amsterdam, 2004.

[25] WALL, Kurt et al. *Programación en Linux. Al descubierto*. Prentice Hall. 2a edición, Madrid, 2001.

[26] WATT, Andrew et al. *SVG Unleashed*. SAMS Publishing, USA, 2002.

[27] WELLING, Luke; THOMSON, Laura. *PHP and MySQL Web Develpment. 2nd edition*. Sams Publishing, USA, 2003.

[28] WILFRED, Ashish et al. *Proyectos Profecionales PHP*. Anaya Multimedia, Madrid, 2002.

[29] WYKE, R. Allen; THOMAS, B. Donald. *Fundamentos de programación en Perl*. Osborne McGraw-Hill, Colombia, 2002.

[30] Gene ZELAZNY. *Say it with charts. 4th edition*. McGraw-Hill, USA, 2000.

Colofón

Esta tesis se terminó de imprimir en Marzo del 2005. Fue realizada por el autor utilizando sólo software OpenSource. Se utilizó el sistema operativo Linux distribución Mandrake 10. Fue escrita y editada en LyX[1] 1.3.3 usando LaTeX[2] 3.4.5 y TeX[3] (Web2C 7.4.5) 3.14159. La mayoría de las Figuras fueron creadas con Skencil[4] 0.6.16 (antes Sketch), pero también con Dia 0.92.2, OpenOffice[5] Draw 1.1 y Umlet[6] 4.5. Para administrar la bibliografía en BibTeX se utilizó JabRef[7] 1.6. El logo EzGraphSVG fue creado en Inkscape 0.41 [8]. Para impresión y distribución en formato digital se exporto de DVI a PDF. La portada se hizo en la imprenta TESIS VZ en Rep. de Cuba 99, Desp. 9 Col. Centro, C.P. 06010 México D.F, y ahí mismo fue empastada.

[1] http://www.lyx.org
[2] http://www.latex-project.org
[3] http://www.tug.org
[4] http://www.skencil.org
[5] http://www.openoffice.org
[6] http://www.umlet.com
[7] http://jabref.sourceforge.net
[8] www.inkscape.org