UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Diseño y elaboración de una herramienta didáctica para el trazado de mapas geográficos.

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

NOMBRE DEL ALUMNO
Oscar Escamilla González

FACULTAD DE CIENCIAS
UNAM

TUTOR(A)
Lic. en C.C. Karla Ramírez Pulido

2007

Índice general

Introducción

La Cartografía constituye una herramienta fundamental para la percepción e interpretación del entorno donde vivimos, la cual nos ha permitido pronosticar y diagnosticar situaciones relevantes para el desarrollo de la sociedad; al mismo tiempo, constituye uno de los materiales base para ubicar y representar en el espacio cualquier fenómeno, ya sea natural o social que afecte nuestras vidas, así como realizar un mejor análisis de los eventos que se desarrollan a nuestro alrededor.

No se puede discutir la importancia que la cartografía ha adquirido en la vida cotidiana y su aplicación tampoco está relegada a minorías. Para concretar, el uso de mapas es un aspecto importante en la sociedad actual y es deseable brindar a los individuos conocimientos acerca de cartografía de una manera comprensible y fácil de asimilar.

Uno de los aspectos base de la cartografía son las proyecciones, con las cuales se elaboran los mapas. Las proyecciones son importantes ya que, como se verá más adelante, cada proyección tiene características que hacen a una u otra mejor para la elaboración de un mapa, dada la aplicación que se le quiera dar.

Retomando el punto de brindar al individuo conocimientos relacionados con las proyecciones cartográficas, el uso de herramientas computacionales es una práctica común, ya que facilitan esta tarea y ofrecen maneras sencillas y atractivas de mostrar este tema. En la actualidad existen muchos sistemas dedicados a mostrar al público los aspectos y características de las proyecciones utilizadas para la elaboración de mapas (ver sección 4.1). Estos sistemas tienen características interesantes y a su vez carecen de algunas deseables para cumplir con la tarea de mostrar los resultados de proyecciones cartográficas. Hablando en términos generales, las características que un sistema como los mencionados debería brindar son:

Poder ser utilizado a través de la Internet:
Uno de los objetivos principales de este tipo de sistemas es que sean utilizados con fines de divulgación, por lo tanto es deseable que puedan ser accedidos y ejecutados desde la Internet para tener un auditorio lo más amplio posible.

De fácil acceso y *software* adicional:
Debido a que el *software* estará publicado en la Internet, podrá ser ejecutado desde cualquier plataforma por lo que no se conoce en primera instancia el *software* con que cuenta el usuario, por lo tanto el sistema debe de ser lo más independiente posible en este aspecto para minimizar el conjunto de requisitos que el equipo del usuario debe cumplir.

Número de proyecciones y herramientas adicionales:
Debe contar con un número de proyecciones aceptable[1], además de ofrecer un conjunto de herramientas que le permitan al usuario comprender lo más posible acerca de la proyección.

Debe ser didáctico:
Debe facilitar la interpretación de los resultados y brindar mecanismos para que el usuario observe y experimente con las características de la proyección.

Intuitividad de la interfaz gráfica:
Como se encuentra dirigido a usuarios que pueden o no tener conocimientos acerca del tema y no estar familiarizados con el uso de aplicaciones de este tipo, el sistema debe ayudar a éste, a intuir como utilizarlo para que así ocupe el menor tiempo posible en aprender como manejarlo.

Modularidad y Extensibilidad:
Además es deseable que sea lo más modular y extensible posible para que pueda servir como base en el desarrollo de nuevos sistemas o se puedan agregar herramientas y funcionalidad.

A lo largo de este trabajo se expondrá el desarrollo de un sistema que trata de cubrir estos requisitos, como el marco teórico, el análisis de los requerimientos y la implementación del mismo.

[1] Al menos tantas como los sistemas ya existentes, las cuales son alrededor de ocho proyecciones (ver sección 4.1).

En el primer capítulo se presentará una introducción al marco teórico y los conceptos matemáticos básicos involucrados en el desarrollo de mapas. Los conceptos abarcarán la noción geométrica de proyección y las fórmulas matemáticas utilizadas para el cálculo de algunas proyecciones.

El segundo capítulo está relacionado con las tecnologías utilizadas para el desarrollo, además de dar una pequeña descripción de cada una de ellas incluyendo los beneficios que ofrecen. Al final de este capítulo se describirá brevemente la aplicación que se tomó como modelo y de la cual se tomaron los datos de entrada. Se describirá asimismo el proceso de conversión de los datos, que incluye tanto el diseño del nuevo formato de los datos, como el trabajo desarrollado para codificarlos.

Una vez mostrado el marco teórico (en particular las fórmulas matemáticas) y las tecnologías seleccionadas para la implementación, se expondrá el diseño general del sistema en el tercer capítulo. Después se profundizará en el diseño de cada módulo, explicando cuál es el funcionamiento deseado para cada uno. Durante esta explicación se describirán algunas de las consideraciones hechas para transformar y codificar los datos. Una vez que se muestre el diseño se procederá a mostrar la implementación de sus elementos más importantes.

En el último capítulo se mostrarán características de aplicaciones similares que se pueden encontrar en la Internet. Se compararán ciertas características tales como el número de proyecciones implementadas, la interfaz gráfica, dependencia de *software* y plataforma, entre otras.

Capítulo 1

Conceptos matemáticos y proyecciones cartográficas

1.1. Breve reseña histórica

La humanidad siempre ha intentado reconocer y adquirir el mayor conocimiento posible de su entorno. Conocer el clima, la fauna, la flora y la distribución de diversos recursos naturales siempre ha sido un factor prioritario para garantizar su subsistencia. Este conocimiento le ha permitido desplazarse, establecer su residencia, planear convenientemente la explotación de los recursos disponibles y adicionalmente le ha otorgado conciencia de saberse parte de ese entorno y su interacción con él. Un método rudimentario para almacenar y difundir este conocimiento es tener una representación gráfica, un dibujo de cómo está distribuido dicho entorno, es decir, un **mapa**. En los albores de la humanidad el desarrollo de mapas estaba dado en dos vertientes, uno el mapa instrumento, elaborado con propósitos prácticos, el otro, el mapa imagen, que es la representación conceptual y filosófica del medio en el que se desarrolla (¿Quién no ha visto ese mundo plano sostenido por elefantes encima de una tortuga (figura 1.1)?).

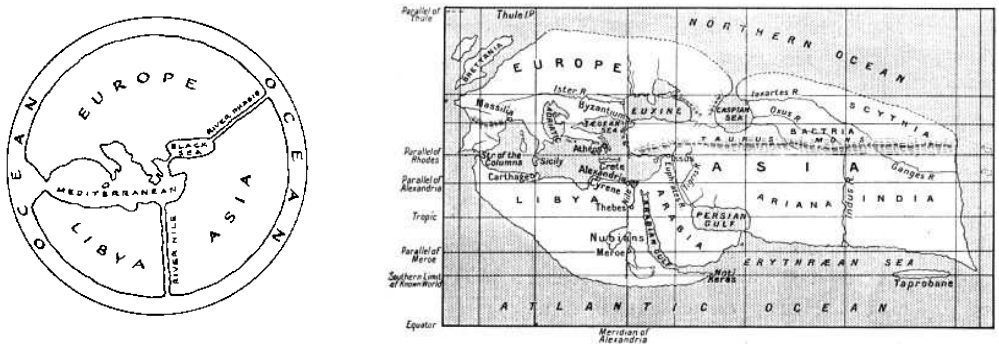
Los mapas más antiguos que se conocen fueron realizados por los babilonios hacia el 2500 a.C.[4] (figura 1.2). Estos mapas, eran simples dibujos tallados en arcilla y consistían, en su mayor parte, de mediciones de tierras para el cobro de impuestos. También se han encontrado en China mapas regionales más extensos, trazados en lienzos de seda, que datan del siglo II a.C. Además hay vestigios de entramados de fibra de caña en las Islas Mars-

Figura 1.1: Universo para el mundo hindú antiguo.

hall, en el sur del océano Pacífico, dispuestas de tal modo que muestran la distribución de las islas.

Figura 1.2: Tablilla babilónica.

Se cree que el primer mapa que representa el mundo conocido (figura 1.3.a) fue realizado en el siglo VI a.C. por el filósofo griego Anaximandro de Mileto. Tenía forma circular y representaba al mundo conocido agrupado en torno al mar Egeo y rodeado por el océano.



(a) Mapa de Anaximandro de Mileto

(b) Mapa de Eratóstenes de Cirene

Figura 1.3: Mapas de Anaximandro y Eratostenes

Uno de los mapas más famosos de la época clásica fue trazado por el geógrafo griego Eratóstenes de Cirene (figura 1.3.b) hacia el año 200 a.C. Representaba el mundo conocido desde Gran Bretaña al norte, la desembocadura del río Ganges al este, hasta Libia en el sur. Este mapa fue el primero en utilizar líneas paralelas transversales para señalar los puntos con la misma latitud. En el mapa también aparecen algunos meridianos, pero estos tenían una separación irregular. Hacia el año 150 a.C., el sabio griego Tolomeo[26] escribió *Geographia* (figura 1.4) el cual era una recopilación de mapas del mundo.

Éstos fueron los primeros mapas en los que se utilizó un método preciso basado en matemáticas. Aunque tenía muchos errores, como la extensión excesiva de la placa terrestre euro-asiática que en apariencia abarcaba la mayor parte del mundo, el trabajo de Tolomeo fue la base para los trabajos en cartografía 1200 años después.

El desarrollo de la cartografía en Roma durante el mismo periodo, es limitado en comparación con Grecia, lo único que cabe destacar son los itinerarios que señalaban rutas y horarios de ejércitos, caminos comerciales, etc. A partir del derrumbamiento del imperio Romano se produce en Europa un retroceso cultural, en el cual la cartografía se ve sumamente afectada ya que en ese momento desaparece la cartografía matemática, la cual es sustituida por otras técnicas basadas en hechos religiosos. Esto se ve reflejado en sus trabajos, ya que en todos, Jerusalén es representado como el centro del mundo (figura 1.5).

Aún así, son valiosos por su carácter estético y artístico que refleja una

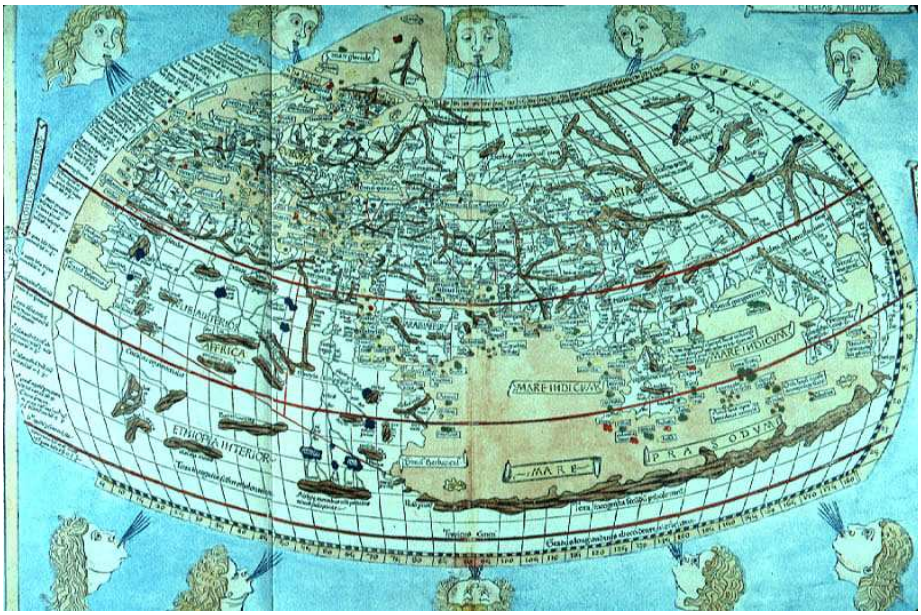


Figura 1.4: Mapa de Ulm. Pertenece a la *Geographia* de Tolomeo

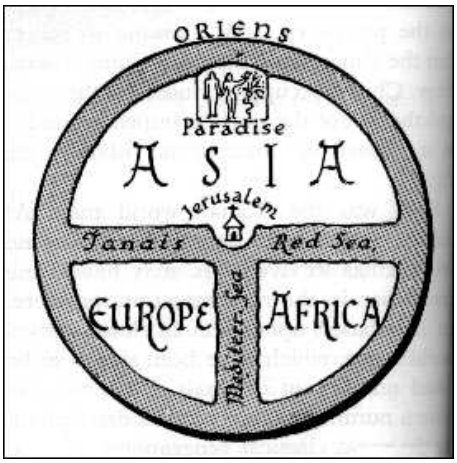


Figura 1.5: Ejemplo de un mapa basado en teología.

concepción teológica del mundo. Sin embargo, en la misma época, los navegantes árabes realizaron y utilizaron cartas geográficas de gran exactitud. Estos retomaron los conocimientos en cartografía de los griegos. El erudito árabe Al-Idrisi[27] realizó un mapa del mundo en 1154 (figura 1.6) basándose

en el trabajo de Tolomeo. Los navegantes del Mediterráneo comenzaron a elaborar cartas marítimas alrededor del siglo XIII, generalmente sin meridianos o paralelos, en cambio mostraban líneas de dirección entre los puertos más importantes. A estos mapas se les conoció como *portulanos*. Los portulanos eran libros con rutas marítimas a cada puerto de acuerdo con los ocho vientos más importantes: Tramontana, Grego, Levante, Laxaloch, Metzodi, Labetso, Poniente y Magistro[28].
El mundo islámico comenzó a producir su propia cartografía, convirtiéndose en los continuadores del desarrollo científico en este rubro de la era clásica. Entre los años 1271 y 1295 un explorador veneciano llamado Marco Polo[29] realizó un viaje a Asia y medio oriente, las historias de sus viajes despertaron el interés por estas regiones, lo que aumentó el intercambio comercial y cultural que ayudaron a que los avances cartográficos llegaran a Europa.

Figura 1.6: Mapa de Al-Idrisi

Los avances en la cartografía en Europa fueron posteriores ya que el espíritu de exploración no se despertó de nuevo hasta que las vías de comercio hacia Oriente se cerraron. Con el nacimiento de la imprenta se empezaron a difundir los mapas por toda Europa. En esta época se imprimieron los mapas de Tolomeo que durante los siguientes siglos, fueron una influencia fuerte para los cartógrafos europeos.

Se considera que el mapa realizado por un geógrafo alemán en 1507, Martín Waldseemüller, fue el primero en designar con el nombre de América al nuevo continente (figura 1.7). El nombre de América es un reconocimiento a Américo Vespucio, quien comenzó a trazar los mapas de sus viajes por

este continente por el año 1508. Entre otros exploradores que contribuyeron a la elaboración de mapas del nuevo continente tenemos a Juan Díaz de Solís, Martín Alonzo Pinzón, Nuñez de Balboa y Juan de la Costa[10]. Una contribución más para la época la constituyen dos documentos elaborados en 1525: los planisferios de los cardenales Salviatti y de Castiglione. Son importantes para la cartografía de la época ya que en ellos se basaron mapas posteriores.

Figura 1.7: Mapa de Waldseemüller

En 1570, Abraham Ortelius publicó el primer atlas moderno, *Orbis Terraum*, el cual disponía de un total de setenta mapas. En el siglo XVI los cartógrafos iban incorporando cada vez más información a sus mapas gracias a la contribución de navegantes y exploradores. Estos últimos encontraron un gran apogeo para esta actividad. Uno de los más reconocidos cartógrafos de la época es Gerardus Mercator. La proyección que consideró (figura 1.8) para sus mapas resultó ser de un valor incalculable para todos los navegantes. Con esta clase de mapas los marineros podían, utilizando una simple brújula, seguir con cierta precisión una ruta de viaje trazando una línea recta del punto de partida al punto de destino.

Los avances en la precisión de la cartografía en los años posteriores aumentaron considerablemente. Esto se debió a que se volvió a implantar y mejoró el sistema de latitud y longitud, además de los cálculos acerca del tamaño y de la forma de la tierra, lo que permitió medir con mayor precisión las deformaciones. Los primeros mapas que contenían información precisa, como

Figura 1.8: Mapa de Mercator

los ángulos de declinación magnética[25], se realizaron en la primera mitad del siglo XVII. Mapas más detallados como los que mostraban las corrientes marítimas se realizaron hacia 1665. En el mismo siglo se reafirmó a la cartografía como ciencia y los errores en los mapas empezaron a ser reducidos.

Hacia finales del siglo XVIII, una vez que decayó el espíritu explorador y nació un sentimiento de nacionalismo en el mundo, un importante número de países de Europa comenzaron a emprender estudios geográficos y topológicos detallados de su territorio. El mapa topológico de Francia fue publicado en 1793, con una forma cuadrada y de aproximadamente once metros cuadrados. Siguieron su ejemplo Gran Bretaña, España, Australia y Suiza por mencionar algunos. En los Estados Unidos de Norte América se organizó, en 1879, el *Geological Survey* con el fin de realizar mapas topológicos de gran escala en todo el país. En 1891, el Congreso Internacional de Geografía propuso cartografiar el mundo entero a la escala de 1:1,000,000. En el siglo XX, la cartografía ha experimentado una serie de innovaciones técnicas entre ellas la fotografía aérea que se desarrolló durante la I Guerra Mundial y que se utilizó de forma más general en la elaboración de mapas durante la II Guerra Mundial.

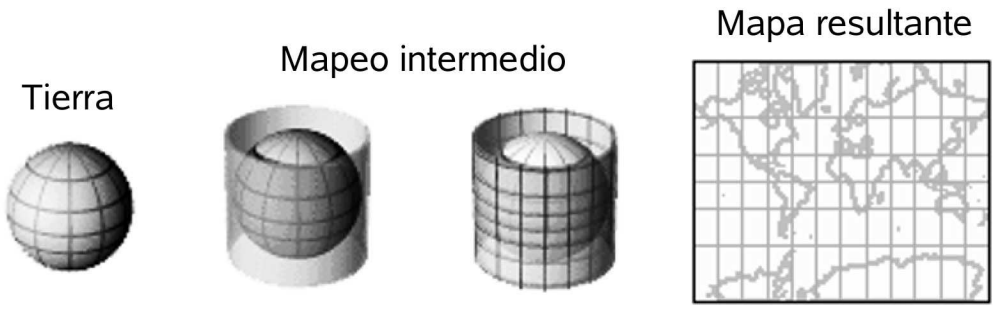


Figura 1.9: Técnica genérica de proyección

1.2. Conceptos matemáticos

Después de una reseña general de la historia de los mapas, podemos enfocarnos en los conceptos matemáticos que están detrás de la elaboración de mapas. En la mayoría de los mapas está involucrado el hecho de que tenemos un cuerpo en tres dimensiones (la Tierra en nuestro caso) y queremos representarlo en una forma plana como un papiro, una hoja, una carta náutica, etc. Formalizando lo anterior, lo que queremos es *proyectar* un objeto tridimensional en un plano. Es decir, hacer un mapeo de los puntos del objeto en tres dimensiones a puntos contenidos en un plano de acuerdo a ciertas *técnicas*. En estas técnicas está implícita la idea de que la Tierra es una esfera, aunque en realidad la forma de ésta es de un esferoide[1]. Históricamente estas técnicas se han basado en el concepto de un mapeo a una superficie intermedia como el cono, el cilindro y el plano mismo; estas superficies intermedias tienen la característica de que el mapeo a una forma plana no generará más distorsión. La técnica para hacer el mapeo es muy parecida para cada una de las superficies mencionadas, consistiendo básicamente en poner el objeto que se desea proyectar (la esfera) en contacto con otro (el cono, el cilindro o el plano) (figura 1.9) y se utiliza una serie de reglas para hacer el mapeo de cada punto de la superficie del primer objeto al otro. Muchas de estas reglas consisten en tener un punto como fuente de luz y tomar la sombra de cada punto del objeto. Esta es la idea analítica, pues en la práctica se utilizan modelos matemáticos.

[1]El radio ecuatorial es de aproximadamente 6378 km, mientras que el radio hacia los polos es de 6357 km. La diferencia es de sólo 0.00329 por ciento aproximadamente. Por lo que la aproximación, suponiendo la tierra esférica, es bastante aceptable en general.

Las proyecciones se clasifican en cónicas, cilíndricas y azimutales si es que se hizo el mapeo intermedio al cono, al cilindro o al plano respectivamente. Existen mapeos que sólo se pueden definir matemáticamente y que no caen en esta clasificación, por lo que se han hecho nuevas clasificaciones con nombres como *pseudocilíndricas* o *pseudocónicas*, por mencionar algunas.

Mencionamos "distorsión" en párrafos anteriores. Por supuesto que el hecho de tratar de pasar las características de la esfera a otra superficie crea deformación en ciertas propiedades, como por ejemplo la distancia entre puntos, el área, la forma, etc. Por lo tanto, la elección de la técnica con la que se realizará la proyección está ligada al uso que se le quiera dar al mapa resultante. Todos los tipos de proyección causan distorsiones y preservan algunas características del modelo original. Algunas preservan la distancia relativa (proyecciones equidistantes, figura 1.10), otras preservan el área (proyecciones equivalentes o de igual área, figura 1.11) y otras mantienen la forma (proyecciones conformes u ortomórficas, figura 1.12) de los objetos transformados. No existe la proyección ideal, todas preservan características a costa de distorsionar otras.

Lo más usual es intentar minimizar una de las distorsiones, por ejemplo, podríamos necesitar que ciertas distancias medidas en la superficie de la esfera se preserven. Obviamente, tratar de minimizar la distorsión en todas direcciones es inalcanzable. Sin embargo podríamos tratar que las distancias relativas se mantengan a lo largo de los meridianos, es decir, que el factor de escala sea constante (una proyección equidistante) a lo largo de los meridianos. Supongamos que tenemos un cuadrado cuyos lados tienen longitud uno. Si utilizamos la proyección con la característica que mencionamos, en la cual el factor de escala en los meridianos (líneas verticales) sea uno, pero que en las demás características (área, forma, etc.) se distorsione, podríamos tener algo como en figura 1.10.

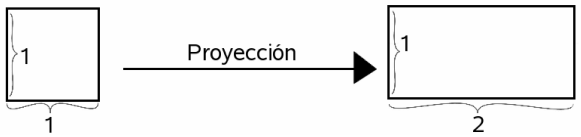


Figura 1.10: Preserva la distancia en los meridianos *(líneas verticales)*

Por otro lado, quizás lo que se desea es mantener el área, lo cual propicia que la forma y las distancias se distorsionen (figura 1.11). Esto nos da una

proyección equivalente. En nuestro ejemplo el área original es uno, y el área después de la proyección se mantiene a pesar de que la forma y las distancias se distorsionaron.

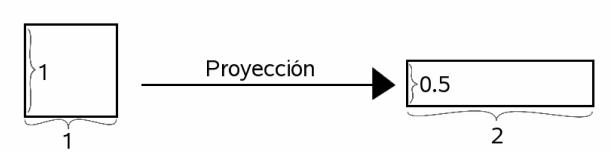


Figura 1.11: Preserva el área de las figuras

Por último, quizás queremos que se mantenga la forma, no importando que se distorsionen el área o las distancias, lo que dá lugar a una proyección ortomórfica o conforme (figura 1.12).

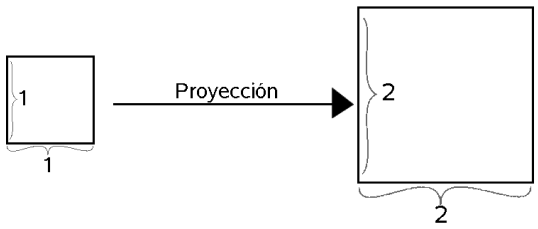


Figura 1.12: Preserva la forma

Al preservar la forma, las proyecciones ortomórficas preservan también los ángulos. En el ejemplo anterior, el ángulo entre los lados y la diagonal del cuadrado es 45^o, lo que se preserva en una proyección ortomórfica y es claro que este ángulo no se preserva en los otros dos tipos (figura 1.13). Por esta razón las proyecciones ortomórficas son usadas en el catastro de tierras y mapas a gran escala con fines de medición, ya que los ángulos medidos en la superficie se mantienen en la proyección, lo que permite la elaboración de cálculos.

Es importante señalar que, aunque estas propiedades son utilizadas para clasificar proyecciones, no representan a todos los posibles tipos, ya que existen muchos en los cuales no se preserva ninguna de estas características. En resumen, la mayoría de las proyecciones pueden clasificarse, primero

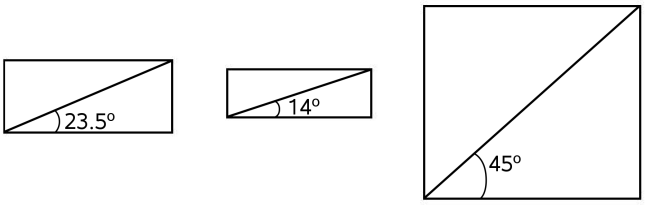


Figura 1.13: Diferencias entre los ángulos de los lados y la diagonal

por su mapeo intermedio (*cónicas*, *cilíndricas* y *azimutales*) y de manera secundaria por las características que preservan (*equidistantes, equivalentes* y *ortomórficas*).

Indicadores de Tissot

Los tipos de proyecciones están caracterizados por las propiedades que mantienen, pero, ¿cómo podemos, a priori, saber cuáles propiedades son las que preservan? En 1859 Nicolas-Auguste Tissot introdujo una herramienta importante para medir las distorsiones generadas por la proyección, que es denominada indicadores de Tissot. Los indicadores de Tissot son básicamente círculos sobre la superficie de la esfera puestos uniformemente. Estos círculos reflejan las distorsiones que genera la proyección de la siguiente manera.
Si el círculo resultante:

Tiene el mismo radio en todas direcciones: La proyección tiene una equivalencia total. Es decir, es equivalente en distancia, en forma y área.

Tiene un radio mayor pero constante en todas direcciones: Esta expresa que la proyección no distorsiona la forma, sólo el área y las distancias. Esto quiere decir que el factor de escala es constante. Esta constante es la relación entre el radio anterior y el radio producido por la proyección.

Es una elipse: Indica que existe una distorsión de las formas. La proporción de sus semiejes con respecto al radio original es la proporción de las deformaciones.

La deformación en el área está dada por la relación entre el área original y el área de la elipse resultante.

Ahora mencionemos las construcciones básicas de las proyecciones cónicas, cilíndricas y azimutales en general, analizando sus deformaciones con los indicadores de Tissot. Después tocaremos más a detalle algunas de las variantes de cada una, además de hablar de aquellas que no se pueden clasificar bajo estas características.

Se mostrarán las fórmulas para las definiciones matemáticas de cada proyección. Para esto hay que establecer ciertas convenciones:

1. Cada punto sobre la esfera se representará en coordenadas esféricas (R,λ, ϕ), que es el equivalente al sistema de coordenadas de latitud y longitud utilizado en los mapas, donde:
   - R es el radio de la esfera (la tierra en nuestro caso), el cual tomaremos como 1 sin pérdida de generalidad.
   - λ es la longitud del punto sobre la esfera (la Tierra).
   - ϕ es la latitud del punto sobre la esfera.
2. ϕ_1, ϕ_2 denotan a los paralelos estándar que se encuentran donde las superficies intermedias interceptan a la esfera.
3. λ_0 es el meridiano que se ubicará en el centro del mapa.
4. Las coordenadas resultantes estarán dadas en forma rectangular (x,y).

1.2.1. Proyecciones cónicas

De manera analítica, la proyección cónica más simple se logra poniendo en contacto un cono con la superficie de la esfera de forma que el cono sea tangente a la esfera. Obviamente el cono sólo hace contacto con la esfera a lo largo de un paralelo, al que denominaremos *paralelo estándar* de la proyección. Después, teniendo una fuente de luz dentro de la esfera, se proyectan los puntos de la superficie de la esfera en el cono. Una vez proyectados los puntos, se procede a cortar el cono a lo largo de un meridiano conveniente al cual nombraremos el *meridiano central*[2], para luego desdoblar el cono, con lo cual nos queda un semicírculo (figura 1.14).

El factor de escala es uno a lo largo del paralelo estándar y la distorsión aumenta a medida que la sección proyectada de la esfera se aleja de él.

[2]Este debe de ser el meridiano que quedará en el centro del mapa.

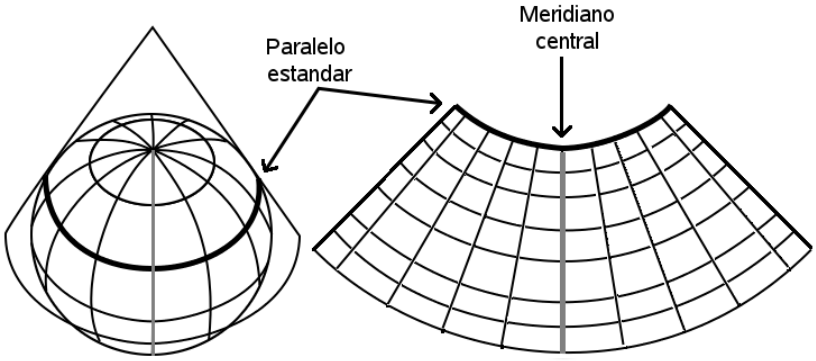


Figura 1.14: Ejemplo de una proyección cónica simple

El resultado de la proyección es que todos los meridianos se convierten en líneas radiales que salen desde uno de los polos (o de la punta del cono) y los paralelos se convierten en semicírculos con un polo como centro en común. Este polo es el mismo en el que convergen todos los meridianos.

Por otro lado podríamos tener un cono secante. En este caso tenemos dos *paralelos estándar*. El resultado de la proyección es similar, sólo que las propiedades se mantienen en los dos paralelos estándar. Otra diferencia es que la distorsión se reduce en las regiones que se encuentran entre los dos paralelos estándar.

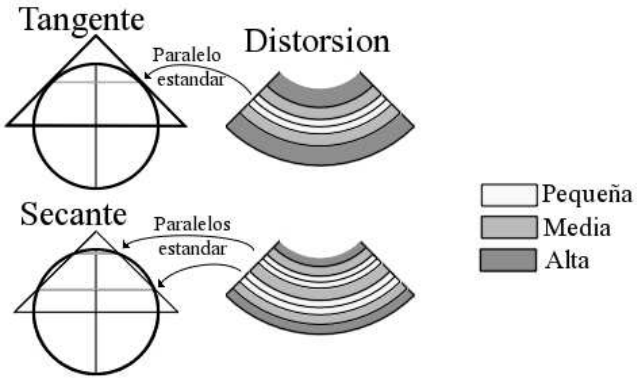


Figura 1.15: Comparación distorsiones en las proyecciones cónicas (*tangente y secante*).

Por último, hay que destacar que las proyecciones cilíndricas y azimutales

son un caso particular de ésta que acabamos de revisar. Las dos formas (el cilindro y el plano) son casos particulares de un cono; un cilindro es un cono recortado cuya punta está en el infinito y un plano es un cono cuya altura de la base a la punta es cero. Estas premisas se presentan sólo de forma teórica, ya que las fórmulas para el cono se vuelven inconsistentes al tratar de utilizar estos extremos.

Cónica simple

Es una proyección cónica que no mantiene ni la forma, ni el área. Los paralelos están separados a la misma distancia unos de otros.

Características

La escala se mantiene a lo largo de los meridianos y a lo largo de los paralelos estándar. El factor de escala se mantiene constante a lo largo de cualquier paralelo. Los meridianos se mapean como líneas radiales que salen desde un polo (la punta del cono). Los paralelos se mapean en semicírculos con un centro común (la punta del cono).

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$
\begin{aligned}
x &= \rho \cos\theta, \\
y &= \rho_0 - \rho \sin\theta, \\
\textit{donde} & \\
\rho &= (G - \phi), \\
\theta &= n(\lambda - \lambda_0), \\
\rho_0 &= (G - \phi_0), \\
G &= \frac{\cos\phi_1}{n} + \phi_1, \\
n &= \frac{\cos\phi_1 - \cos\phi_2}{\phi_1 - \phi_2}.
\end{aligned}
$$

Si tenemos que $\phi_1 = \phi_2$, las ecuaciones se reducen a :

$$
\begin{aligned}
G &= \cot\phi_1 + \phi_1, \\
n &= \sin\phi_1.
\end{aligned}
$$

En este caso ϕ_0 es una constante que debe ser colocada y centrada en la parte inferior del mapa. En realidad no afecta la apariencia del mapa.

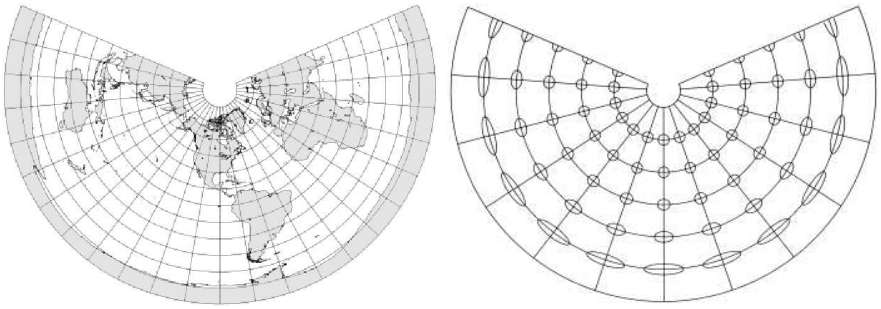


Figura 1.16: Proyección cónica simple

Lambert

Es una proyección cónica ortomórfica o conforme. Mantiene la forma, pero no el área, ni las distancias relativas.

Características

La escala aumenta conforme nos alejamos de la punta del cono sobre los meridianos. La distorsión es nula a lo largo de los paralelos estándar. La escala es constante a lo largo de cualquier paralelo.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$
\begin{aligned}
x &= \rho \sin n(\lambda - \lambda_0), \\
y &= \rho_0 - \rho \cos[n(\lambda - \lambda_0)], \\
&\textit{donde} \\
\rho &= F \cot^n(\frac{1}{4}\pi + \frac{1}{2}\phi), \\
\rho_0 &= F \cot^n(\frac{1}{4}\pi + \frac{1}{2}\phi_0), \\
F &= \frac{\cos\phi_1 \tan^n(\frac{1}{4}\pi + \frac{1}{2}\phi_1)}{n}, \\
n &= \frac{\ln(\cos\phi_1 \sec\phi_2)}{\ln[\tan(\frac{1}{4}\pi + \frac{1}{2}\phi_2)\cot(\frac{1}{4}\pi + \frac{1}{2}\phi_1)]}.
\end{aligned}
$$

El símbolo ϕ_0 se definió en la proyección anterior.

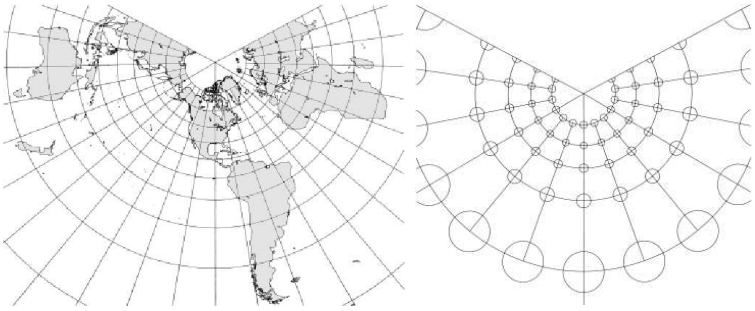


Figura 1.17: Proyección cónica de Lambert

1.2.2. Proyecciones cilíndricas

Para este tipo de proyección utilizaremos un cilindro tangente a la esfera. Por lo general el ecuador es la línea de contacto de la esfera y el cilindro. Los meridianos se transforman en líneas rectas verticales separadas uniformemente. Los paralelos se transforman en líneas rectas horizontales separadas cada vez más conforme se alejen del ecuador. Esto genera una cuadrícula rectangular. La escala es real a lo largo del ecuador y a medida que la región del mapa se aleja de este la distorsión aumenta. En el caso secante las características se mantienen. La diferencia radica en que tenemos dos líneas en las cuales se intercepta a la esfera (se tienen dos paralelos estándar), las propiedades del ecuador pasan a estas líneas. Además, como en el caso de las *proyecciones cónicas*, la distorsión disminuye entre estas dos líneas.

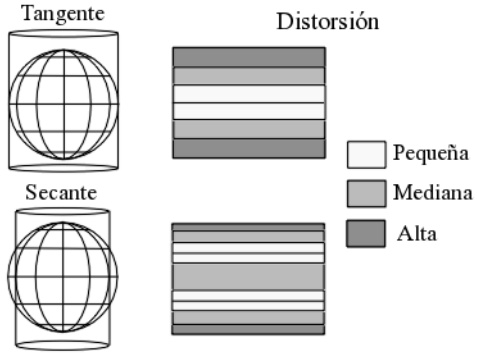


Figura 1.18: Comparación de distorsiones en proyecciones cilíndricas (*tangente y secante*)

Equirectangular

Ésta es una proyección cilíndrica equidistante en la cual el cilindro es secante a la esfera.

Características

Es equidistante a lo largo de los paralelos estándar y el ecuador. La distorsión en las distancias y en la forma se incrementa proporcionalmente a medida que la región se acerca a los polos. Los paralelos se transforman en líneas horizontales a distancia constante uno del otro. Los meridianos se transforman en líneas verticales ortogonales a los paralelos. El espacio entre ellos se incrementa a medida que se alejan del centro. Los polos se convierten en líneas rectas de igual longitud que el ecuador.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$X = R\lambda,$$
$$Y = R\phi.$$

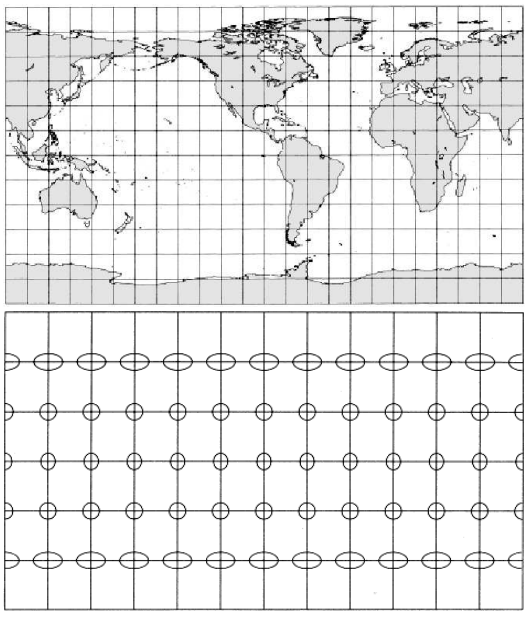


Figura 1.19: Proyección equirectangular

Mercator

Es una proyección cilíndrica conforme y el cilindro es secante a la esfera. Dado que es ortomórfica (conforme) mantiene los ángulos, por lo cual una línea recta trazada desde el punto de origen al punto destino, siempre corta a los meridianos en el mismo ángulo, lo que hace fácil seguir la trayectoria marcada con una brújula. Ésta es la razón de que fuera la proyección más importante para la navegación marítima durante el Renacimiento. Aunque esto tiene cierta complicación, ya que esta ruta no es la más corta, pues si mapeamos esta ruta de regreso a la esfera se transforma en un *loxodromo*[12] que es una especie de curva espiral sobre la superficie de la esfera (figura 1.20).

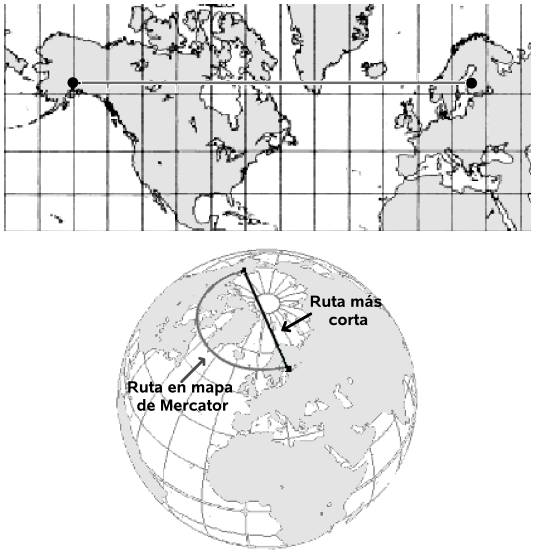


Figura 1.20: La ruta más corta sobre el mapa no es la ruta más corta en la esfera.

Características

La escala es verdadera (factor de escala uno) a lo largo del ecuador y de los paralelos estándar. La distorsión en la distancia y el área aumenta conforme se acerca a los polos, haciéndose infinita en éstos, por lo que los polos y regiones cercanas a éstos no pueden ser mostrados. El factor de escala es constante a lo largo de los paralelos. Los meridianos se transforman en líneas rectas verticales. Los paralelos se convierten en líneas rectas horizontales. Ninguno de los dos casos (paralelos o meridianos) mantienen una separación constante,

sino que esta separación aumenta a medida que se alejan del centro.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$x = R\lambda,$$
$$y = R \ln \tan(\frac{\pi}{4} + \frac{\phi}{2}).$$

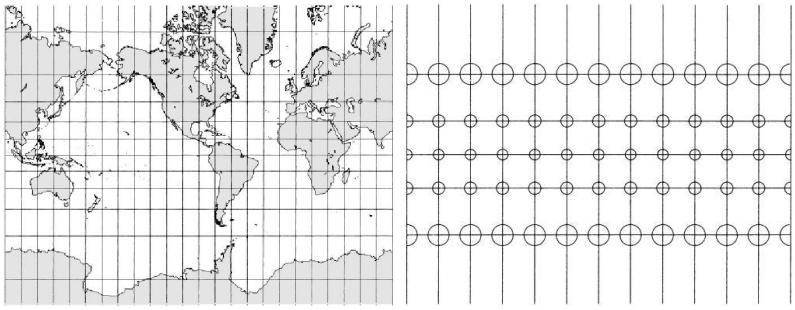


Figura 1.21: Proyección de Mercator

Cilíndrica equivalente

Como su nombre lo indica, ésta es una proyección cilíndrica que preserva el área. El cilindro utilizado para la proyección es tangente a la esfera en el ecuador y la fuente de luz se supone en un punto en el infinito (figura 1.22). Esta proyección fue desarrollada por Lambert en 1772. Es la más fácil de construir entre las proyecciones cilíndricas que preservan el área y es ampliamente usada en libros sobre cartografía. No es muy popular dado que la deformación en regiones cercanas a los polos es excesiva.

Característica

Los paralelos se transforman en líneas rectas horizontales, separadas a una distancia menor conforme se alejan del ecuador. Los meridianos se mapean en líneas rectas verticales a distancias iguales. Los polos se transforman en líneas de igual magnitud que los paralelos. La distorsión en la forma es nula en el ecuador, pero crece a medida que se acerca a los polos. El área se mantiene en todas direcciones del mapa.

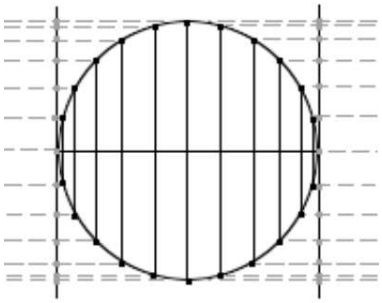


Figura 1.22: Método de una proyección cilíndrica equivalente.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$x = (\lambda - \lambda_0)\cos\phi_s,$$
$$y = \sin\phi \sec\phi_s.$$

En este caso ϕ_s es la "latitud estándar" de la proyección.

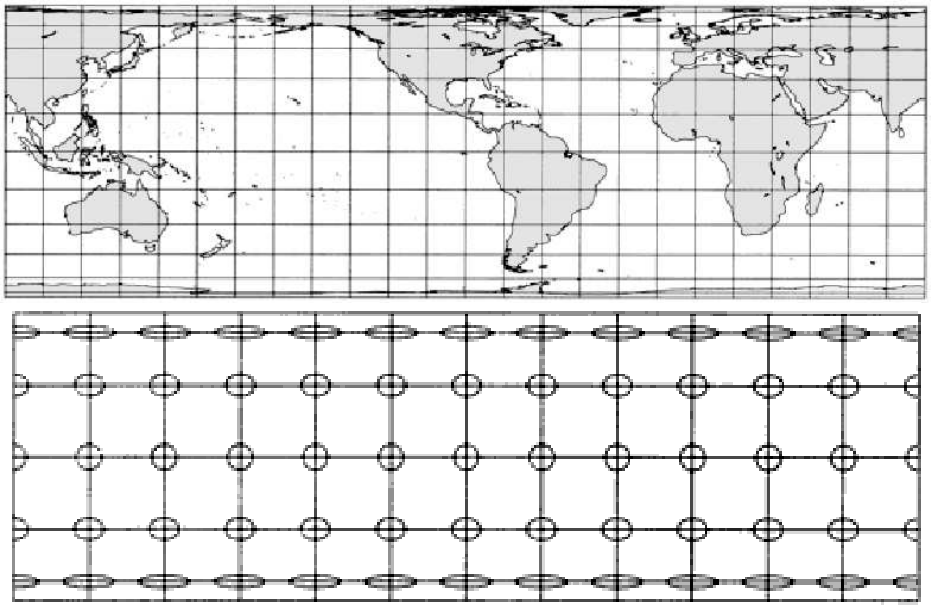


Figura 1.23: Proyección cilíndrica equivalente

Cabe destacar que proyecciones cilíndricas históricamente importantes

son casos particulares de ésta. Algunos ejemplos son:

Proyección	ϕ_s
Cilíndrica de Albers	0^o
Cilíndrica de Behrmann	30^o
Tristan Edwards	$37{,}383^o$
Peters	$44{,}138^o$
Ortográfica de Gall	45^o
Balthasart	50^o

1.2.3. Proyecciones azimutales

Una proyección azimutal se forma colocando un plano en contacto con la esfera y formulando una serie de reglas para la transferencia de los rasgos de una superficie a otra. En la versión en la que el plano es tangente a la esfera, la región de contacto es un solo un punto; esto causa que la distorsión en el factor de escala sea proporcional a la distancia a este punto. En todas las proyecciones azimutales el punto de tangencia es el centro de un mapa circular.

En el aspecto normal, es decir cuando el punto de tangencia (o *azimut*) es uno de los polos, todos los meridianos son líneas rectas radiales al punto de tangencia. Los paralelos son círculos concéntricos cuyo centro es el polo que hace contacto con el plano.

Otro aspecto importante de este tipo de proyecciones es el que las direcciones desde el azimut a cualquier otro punto se mantienen o tienen un factor de escala constante, lo que implica que la ruta más corta en la esfera entre el azimut y algún otro punto, es una línea recta en el plano obtenido. Esto ha hecho que este tipo de proyección sea el predilecto para los mapas de navegación aérea.

Azimutal equivalente o proyección azimutal de Lambert

Es una proyección azimutal en la cual el plano es tangente a la esfera. Ésta conserva las propiedades de las proyecciones azimutales. Todas las direcciones y las distancias son correctas desde el azimut. La distorsión aumenta conforme la región se encuentra más alejada del azimut.

Características

El resultado al mapear los paralelos y meridianos depende del lugar donde se

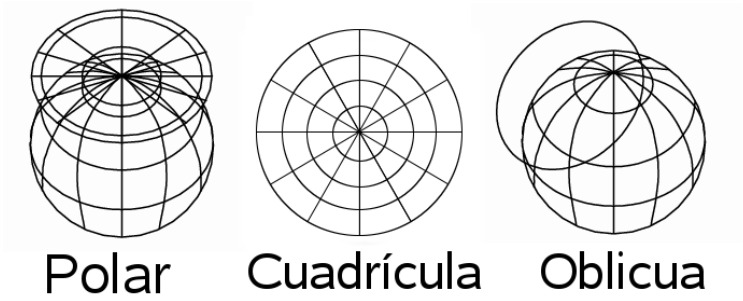


Figura 1.24: Proyección azimutal

encuentre el azimut. En el caso de que el punto de tangencia no sea un polo, los meridianos se transforman en curvas que inciden en uno o dos puntos (los polos). En la forma normal[3] la distancia entre los meridianos va disminuyendo a medida que la región se aleja del azimut. Sobre los paralelos el factor de escala se mantiene constante, pero conforme el paralelo está más alejado del azimut este factor se incrementa.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$x = k\cos\phi\sin(\lambda - \lambda_0),$$
$$y = k[\cos\phi_1\sin\phi - \sin\phi_1\cos\phi\cos(\lambda - \lambda_0)],$$

donde

$$k = \sqrt{\frac{2R}{1 + \sin\phi_1\sin\phi + \cos\phi_1\cos\phi\cos(\lambda - \lambda_0)}}.$$

Azimutal estereográfica

Es una proyección ortomórfica lo que produce que el área se distorsione conforme nos alejamos del azimut. En ésta el plano puede ser tangente o secante[4] a la esfera. Se tiene un punto como centro de luz que se encuentra del lado contrario del azimut. Se trazan líneas que pasan por el centro de luz y cada uno de los puntos que se desean mapear. La intersección de esta línea con el plano es el punto correspondiente a la proyección.

[3]La forma normal es cuando el azimut es uno de los polos.

[4]Esto depende del paralelo estándar y el meridiano central que se escojan.

Figura 1.25: Proyección azimutal equivalente

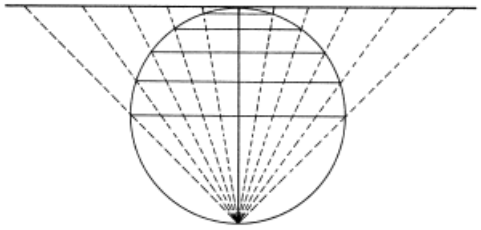


Figura 1.26: Método de una proyección estereográfica

Características

El resultado de la proyección cuando el azimut es uno de los polos es el mismo que en una proyección equivalente, con la diferencia de que la distorsión en la forma se traduce como distorsión en el área. La forma se mantiene pero el factor de escala aumenta conforme nos alejamos del azimut. Los paralelos se mapean en círculos concéntricos, los meridianos en líneas rectas que radian desde el azimut, las distancias a lo largo de los meridianos aumentan conforme la región se encuentra más alejada del azimut. A lo largo de los paralelos el factor de escala se mantiene constante.

Cuando el azimut no es uno de los polos, tanto los meridianos como los paralelos se transforman en curvas a distancias iguales, con la única diferencia que los primeros inciden en los polos.

La definición matemática de las coordenadas rectangulares para esta pro-

yección es:

$$x = k\cos\phi\sin(\lambda - \lambda_0),$$
$$y = k[\cos\phi_1\sin\phi - \sin\phi_1\cos\phi\cos(\lambda - \lambda_0)],$$

donde

$$k = \frac{2R}{1 + \sin\phi_1\sin\phi + \cos\phi_1\cos\phi\cos(\lambda - \lambda_0)}.$$

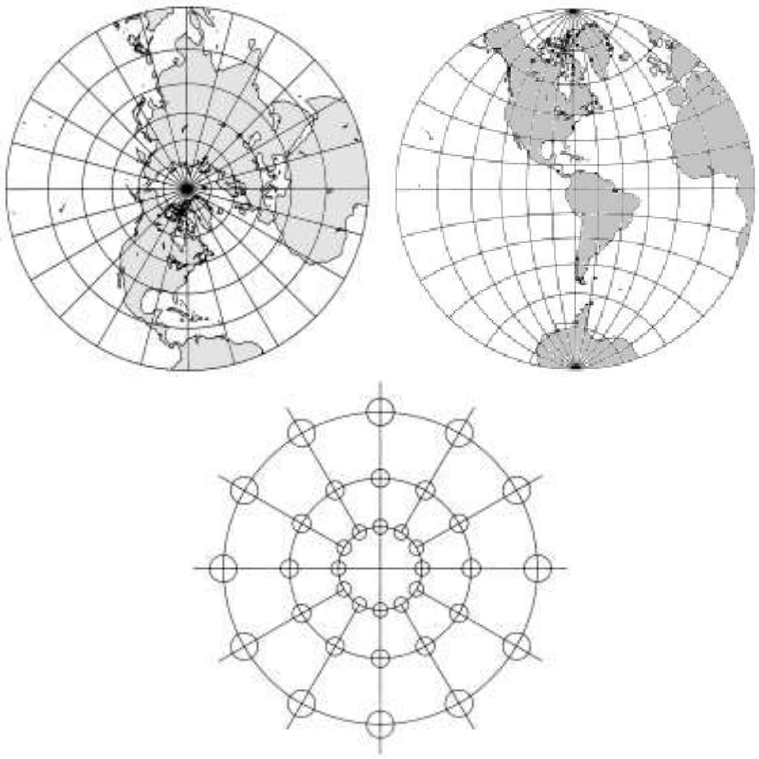


Figura 1.27: Proyección estereográfica

Ortogonal

En este tipo de proyección azimutal no se conserva ninguna propiedad, no preserva la forma, ni el área, ni las distancias. Por esta razón es poco usada

para fines científicos, ya que además de no conservar propiedades, sólo muestra una mitad de la esfera a la vez. Esta proyección se hizo popular desde que se tomaron fotografías de la Tierra en el espacio durante los sesentas ya que es muy parecida a la apariencia de la tierra desde el espacio. Esta proyección se construye a partir de un plano tangente, con una fuente de luz puesta en el infinito y aplicando el procedimiento de la proyección estereográfica.

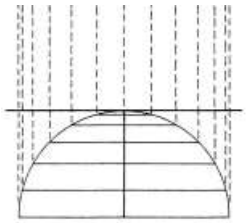


Figura 1.28: Proyección ortogonal

Característica

Cuando la proyección es normal (cuando el azimut es uno de los polos) los paralelos se transforman en círculos concéntricos; los meridianos se transforman en líneas rectas que radian desde el azimut; el factor de escala se decrementa conforme se aleja del azimut y la distorsión aumenta. Si el azimut no es uno de los polos, los meridianos se transforman en líneas curvas que inciden en uno de los polos (el otro no es visible en la proyección) y están espaciados irregularmente. Los paralelos se mapean en un segmento de elipse que, al igual que los meridianos, están espaciados irregularmente. Los grados de distorsión en el área y las distancias son los mismos que en la forma normal.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$x = \cos\phi \sin(\lambda - \lambda_0),$$
$$y = \cos\phi_1 \cos\phi - \sin\phi_1 \cos\phi \cos(\lambda - \lambda_0).$$

1.2.4. Proyecciones pseudocilíndricas

En todas la proyecciones cilíndricas existe una fuerte distorsión en latitudes altas llegando a ser extrema en los polos, tornándose éstos en líneas rectas

Figura 1.29: Proyección ortogonal

o distorsionarse al grado de no ser posible mostrarlos. En las *proyecciones pseudocilíndricas* se tiene como objetivo reducir estas distorsiones. Este tipo de proyecciones se basa en el mismo principio de mapear los rasgos de la esfera en un cilindro, con la diferencia de que se tienen varios de ellos cortando a la esfera en distintas latitudes.

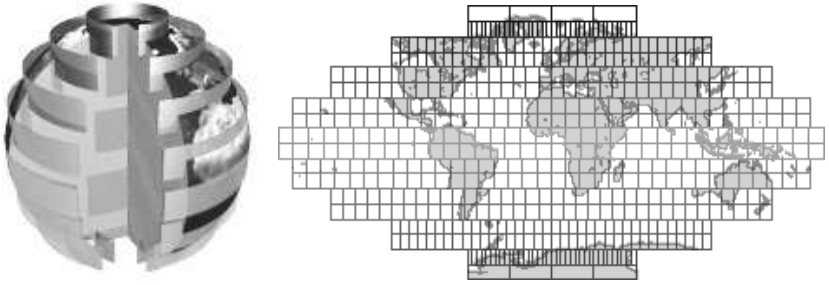


Figura 1.30: Proyecciones pseudocilíndricas

La diferencia más significativa entre este tipo de proyecciones y las cilíndricas consiste en que los meridianos no se mapean en líneas rectas perpendiculares a los paralelos, sino que se transforman en líneas curvas que inciden en los paralelos en distintos ángulos.

Sinosoidal simple

Es la proyección pseudocilíndrica más antigua que se sigue utilizando. La característica que se preserva es el área, dejando de lado la distorsión en las formas y en la distancia.

Característica

Los paralelos se mapean como líneas rectas paralelas entre sí, separadas a distancias iguales. Los meridianos se mapean como líneas curvas que inciden en los polos, con excepción del meridiano central que es una línea recta ortogonal a los paralelos. Todos los meridianos están separados a distancias iguales. Además la longitud del meridiano central mapeado es la mitad que la del ecuador. El factor de escala es uno a lo largo de los paralelos y en el meridiano central. La distorsión en la forma es bastante severa en otros meridianos distintos al meridiano central.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$x = (\lambda - \lambda_0)\cos\phi,$$
$$y = \phi.$$

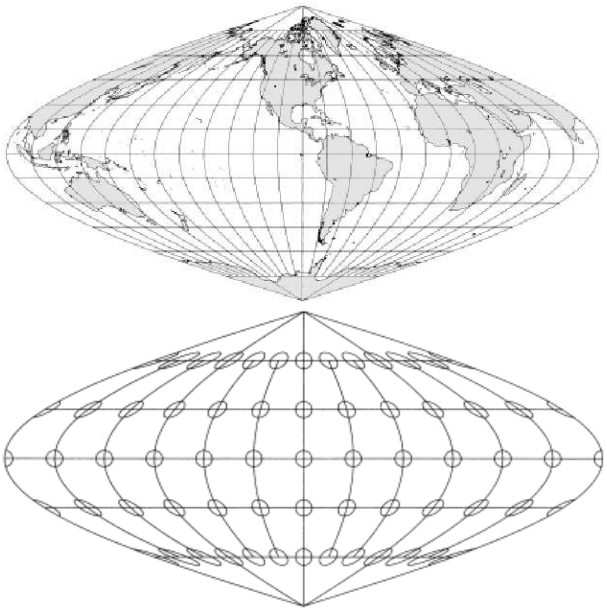


Figura 1.31: Proyección sinosoidal simple

Eckert IV

Esta proyección tradicionalmente se ha utilizado para mostrar cambios climáticos, corrientes marítimas, etc.

Características

El factor de escala es uno a lo largo de los paralelos $40^o30'$ Norte y $40^o30'$ Sur. Además es constante a lo largo de cualquier paralelo y es simétrica entre paralelos iguales de signo distinto[5]. La distorsión en la forma es nula en los paralelos $40^o30'$ Norte, $40^o30'$ Sur y en el meridiano central. El meridiano central se transforma en un línea recta ortogonal al ecuador, los demás meridianos son líneas curvas cuya concavidad es hacia el meridiano central. Los paralelos son líneas rectas perpendiculares al meridiano central, separadas cada vez más a medida que están a distancia mayor del ecuador. Los polos se mapean en líneas rectas cuyo largo es la mitad del largo del ecuador.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$x = \frac{2}{\sqrt{\pi(4+\pi)}}(\lambda - \lambda_0)(1+\cos\theta),$$
$$y = 2\sqrt{\frac{2}{\pi(4+\pi)}}\sin\theta.$$

donde θ es la solución de:

$$\theta + \sin\theta\cos\theta + 2\sin\theta = (2+\tfrac{1}{2}\pi)\sin\phi.$$

Esta ecuación puede ser aproximada por el método de Newton[14] utilizando $\theta_0 = \frac{\phi}{2}$ para obtener :

$$\Delta\theta = -\frac{\theta+\sin\theta\cos\theta+2\sin\theta-(2+\frac{1}{2}\pi)\sin\phi}{2\cos\theta(1+\cos\theta)}$$

1.2.5. Proyecciones pseudocónicas

En este tipo de proyecciones al igual que en las proyecciones cónicas, los paralelos se transforman en semicírculos con un centro en común, pero los meridianos no son ya líneas rectas radiales, sino que se transforman en líneas curvas en el mapa resultante.

[5] 10^oN tiene el mismo factor de escala que 10^oSur.

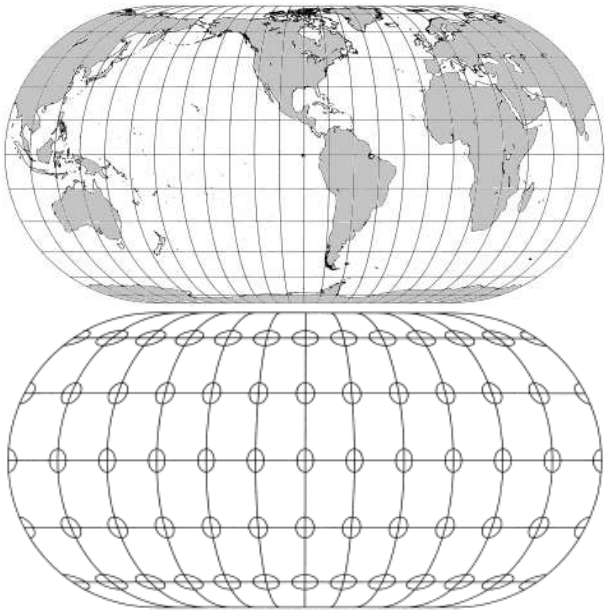


Figura 1.32: Proyección Eckert IV

Bonne

Es una proyección equivalente que no conserva la forma. Usada principalmente para mapas topográficos a gran escala, fue desarrollada rudimentariamente por Tolomeo, pero se reconoce a Rigobert Bonne[13] como el creador de esta proyección. A principios del siglo XIX fue utilizada para un mapa topográfico de Francia.

Características

El factor de escala es uno a lo largo del meridiano central y a lo largo de cada paralelo; la distorsión en la forma es nula a lo largo del paralelo y el meridiano central[6], el meridiano central se mapea a una línea recta en el centro del mapa; los meridianos se transforman en curvas complejas las cuales conectan puntos que están a igual distancia en los paralelos; todos los meridianos inciden en los polos; los paralelos se mapean en arcos concéntricos a distancia constante uno del otro[7] a lo largo del meridiano central. Los polos se convierten en puntos en los extremos superior (el polo norte) e inferior (el

[6]En otras regiones no es del todo conforme, pero la distorsión en la forma es tan pequeña que es aceptable.

[7]Esta distancia es la misma que tienen en la esfera original.

polo sur).

Esta proyección puede convertirse en otras haciendo que el paralelo estándar sea:

Proyección	Paralelo estándar
Werner	Es uno de los polos
Sinosoidal	Es el ecuador

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$\begin{aligned} x &= \rho \sin\theta, \\ y &= \cot\phi_1 - \rho\cos\theta, \\ \textit{donde} & \\ \rho &= \cot\phi_1 + \phi_1 - \phi, \\ \theta &= \frac{(\lambda - \lambda_0)\cos\phi}{\rho}. \end{aligned}$$

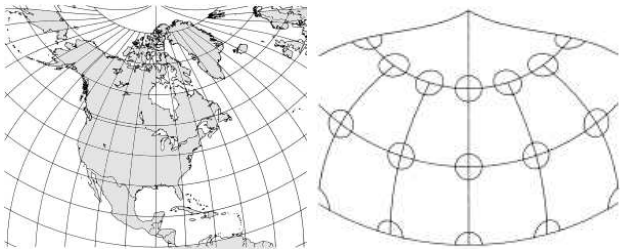


Figura 1.33: Proyección Bonne

Werner

Es una proyección en la cual no se conserva la forma, pero se mantiene el área. Es un caso especial de la proyección Bonne, ya que se toma uno de los polos como paralelo estándar.

Características

La escala es 1 a lo largo del meridiano central y sobre todos los paralelos. La distorsión en la forma es nula sólo en el meridiano central. El meridiano

central se mapea en una línea recta mientras que el resto de los meridianos presentan el mismo comportamiento de la proyección Bonne. Los paralelos se transforman al igual que la proyección Bonne, en arcos concéntricos. Uno de los polos (generalmente el polo norte) en un arco que comparte su centro con los paralelos.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$\begin{aligned} x &= \rho \sin\theta, \\ y &= \cot\phi_1 - \rho\cos\theta, \\ \textit{donde} & \\ \rho &= \cot\phi_1 + \phi_1 - \phi, \\ \theta &= \frac{(\lambda - \lambda_0)\cos\phi}{\rho}, \\ \phi_1 &= (+/-)90^o. \end{aligned}$$

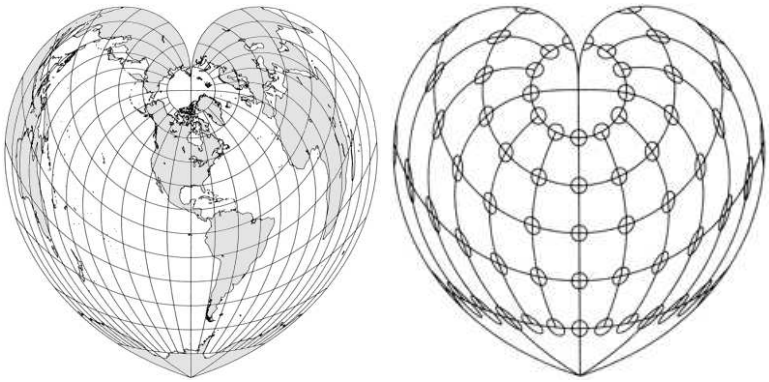


Figura 1.34: Proyección Werner

1.2.6. Proyecciones misceláneas

Además de las presentadas anteriormente existen muchos otros tipos de proyecciones. Para ejemplificar algunas tenemos las siguientes:

Hammer Aitoff

La proyección Hammer Aitoff pertenece a un tipo de proyecciones llamadas "azimutales modificadas". Es una modificación a la proyección azimutal de Lambert. La modificación consiste en reducir a la mitad los valores de latitud de cada punto y duplicar el valor de la longitud. Aunque parece una proyección pseudocilíndrica no lo es, ya que tiende a deformar en menor grado las regiones cercanas a los polos que en una proyección pseudocilíndrica común.

Características

La escala decrece a lo largo del meridiano central y el ecuador a medida que se alejan del centro. La distorsión en la forma es moderada y es mucho menor (comparada con una proyección pseudocilíndrica) cerca de los polos. El meridiano central se transforma en una línea recta cuya longitud es la mitad de la del ecuador. Los demás meridianos son curvas cóncavas hacia el meridiano central, separadas a una distancia igual sobre el ecuador.

El ecuador se mapea en una línea recta horizontal. Los otros paralelos se mapean en líneas curvas cóncavas hacia los polos.

La definición matemática de las coordenadas rectangulares para esta proyección es:

$$x = \frac{2R\sqrt{2}\cos\phi\sin(\frac{\lambda}{2})}{D},$$
$$y = \frac{R\sqrt{2}\sin\phi}{D},$$
donde
$$D = \sqrt{1+\cos\phi\cos(\lambda/2)}.$$

Proyección armadillo

Es una proyección que asemeja la tierra mapeada a una porción de una dona, parecida también a la forma de un armadillo enroscado parcialmente (de ahí su nombre). fue presentada por Erwin J. Raisz de la universidad de Harvard en 1943.

Características

No preserva la forma ni el área. Ésta es de las que más tierra presenta en

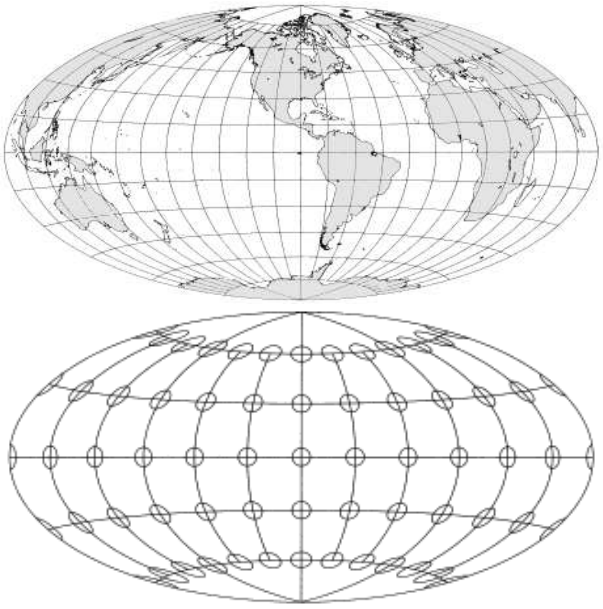


Figura 1.35: Proyección HammerAitoff

proporción a los océanos; la región antártica no puede ser mostrada. El polo norte se transforma en un arco elíptico; el meridiano central (10^o o 15^o este regularmente) se convierte en una línea recta. Los otros meridianos se mapean en arcos elípticos cóncavos en dirección al meridiano central. Los paralelos son arcos elípticos cóncavos hacia el polo norte. El factor de escala decrece a medida que la región se aleja del meridiano central. La deformación en la forma es moderada en las partes centrales (cerca del meridiano central) e incrementa hacia los extremos. La definición matemática de las coordenadas rectangulares para esta proyección es:

$$\phi_s = -\arctan\left[\frac{\cos\left(\frac{\lambda-\lambda_0}{2}\right)}{\tan 20^o}\right],$$

Si $\phi_s < \phi$ el punto no es visible, de otro modo :

$$x = (\lambda + \cos\phi)\sin\left(\frac{\lambda - \lambda_0}{2}\right),$$
$$y = \frac{\lambda + \sin 20^o - \cos 20^o}{2} + \sin\phi\cos 20^o \cos\left(\frac{\lambda - \lambda_0}{2}\right).$$

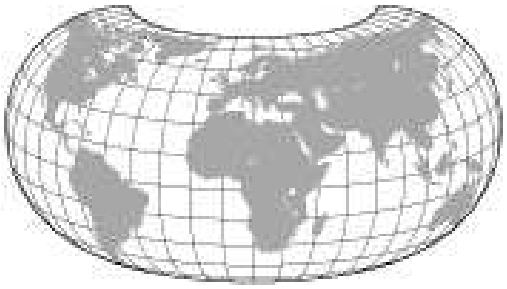


Figura 1.36: Proyección armadillo

Capítulo 2

Tecnologías utilizadas y aplicación original

2.1. XML "eXtended Markup Language"

XML proviene de un lenguaje que inventó IBM[19] alrededor de los años setentas. El lenguaje de IBM se llama GML (*General Markup Language*) y surgió por la necesidad que tenían en la empresa de almacenar y procesar grandes cantidades de información de diversos temas generada por sus investigaciones.

Esta idea de lenguaje agradó mucho a ISO[20] y en 1986 empezaron a trabajar para estandarizar el lenguaje, lo que dió surgimiento a *SGML*[15].

En 1989, un usuario de una red (Tim Berners-Lee[30]), que había conocido el lenguaje de etiquetas (Markup) y el hipertexto creó un nuevo lenguaje llamado *HTML*[18]. Este lenguaje fue aceptado rápidamente y varias compañías se dedicaron a competir haciendo el visor de HTML más poderoso y definiendo etiquetas como mejor les conviniera. A consecuencia de esto el lenguaje creció sin control, haciendo poco compatible los documentos entre navegadores.

A partir de 1996 y hasta la fecha, una organización llamada *W3C*[16] ha intentado poner orden en este rubro, estableciendo etiquetas y reglas para lograr un estándar.

Desde 1998, la W3C trabaja en el desarrollo de XML, para lograr solucionar las carencias de HTML como:

- El contenido está mezclado con el aspecto que se le quiere dar al docu-

mento.

- Es difícil compartir la información contenida en los documentos.
- La forma en la que se muestra la información depende del visor que se utilice.

XML es un meta-lenguaje que permite definir la estructura que poseen los datos almacenados y la manera en que dicha estructura es codificada. Esto, como veremos más adelante, se logra mediante una simple definición de etiquetas (en DTD o Schema). Posteriormente, cuando los datos son recuperados, las etiquetas permiten una interpretación uniforme de su estructura, independiente de la aplicación que los lea o la plataforma donde son recuperados. La simplicidad de uso de XML ha contribuido a su notable popularidad actual.

Es importante señalar que XML no hace más que lo descrito, es decir, no permite hacer ningún tipo de manipulación directamente a los datos, sólo es una forma de almacenar y definir la codificación de almacenado.

2.1.1. DTD "*D*ocument *T*ype *D*efinition"

El propósito de un **DTD** es definir la estructura de un documento XML y los elementos válidos dentro de él. Esto se logra mediante la definición de etiquetas que estructurarán el documento, además del tipo de información dentro de cada etiqueta[1]. Un DTD se puede declarar dentro del propio documento o como una referencia externa.

Veamos un ejemplo de DTD (figura 2.1). Deseamos declarar una versión abstracta de un correo electrónico, para lo que tendríamos que declarar la dirección o direcciones a las cuales debe llegar, la dirección del autor, si se tiene que redirigir la respuesta a alguna otra dirección, si lleva archivos adjuntos, etc.

Dado este pequeño ejemplo de la estructura de un DTD procederemos a definir las estructuras que formarán el DTD y su sintaxis en el documento.

Sintaxis

En el DTD los elementos que estarán en nuestro documento XML se declaran de la siguiente forma:

[1]Que podrían ser otras etiquetas o datos (texto o números).

```
<?xml version="1.0"?>
<!ELEMENT email
          (Para+, ResponderA?, De, Tema, Adjunto*,
          (MensajeHTML|MensajeTXT)
  )>

<!ELEMENT Para              (#EMPTY )>
<!ELEMENT ResponderA        (#EMPTY )>
<!ELEMENT De                (#EMPTY )>
<!ELEMENT MensajeHTML       (#PCDATA)>
<!ELEMENT MensajeTXT        (#CDATA )>
<!ELEMENT Tema              (#CDATA )>
<!ELEMENT Adjunto           (#ANY)>

<!ATTLIST Para direccion CDATA #REQUIRED>
<!ATTLIST Para nombre    CDATA #IMPLIED>
<!ATTLIST De direccion   CDATA #REQUIRED>
<!ATTLIST De nombre      CDATA #IMPLIED>
<!ATTLIST ResponderA direccion   CDATA #REQUIRED>
```

Figura 2.1: Un DTD para definir correos electrónicos

```
<!ELEMENT nombre-elemento (contenido)>
```

nombre-elemento: Es el nombre del elemento que estamos declarando. En nuestro ejemplo en la línea 2 declaramos un elemento llamado *email*, el cual tiene como contenido lo declarado en las líneas 3 y 4.

contenido: Es el tipo de datos que va a tener el elemento.

El contenido de un elemento puede ser:

EMPTY: Si el elemento es vacío. En este caso las etiquetas de la forma <nombre−elemento> </nombre−elemento> se pueden sustituir por <nombre−elemento/>. En el ejemplo (figura 2.1) en la línea 7, se puede observar una declaración de este tipo.

#PCDATA o #CDATA: Indican que el contenido es texto que puede o no ser analizado a su vez. El valor **PCDATA** especifica que el texto será analizado, mientras que el valor **CDATA** indica que los datos deben ser tomados literalmente. Por ejemplo en las líneas 10 y 11 tenemos los dos tipos, esto es porque en *MensajeTXT* sólo queremos texto y en *MensajeHTML* queremos que existan etiquetas de HTML que serán interpretadas para darle formato al texto.

ANY: Si el valor puede ser de cualquier tipo. Por ejemplo, un archivo adjunto (línea 13), entonces puede ser cualquier cosa.

Elemento hijos: Estos se deben definir en un lista de elementos separados por "," o bien cada uno por separado. Cabe mencionar que de la primera forma, el orden en la lista indica también el orden de aparición dentro del elemento. En la figura 2.1, en las líneas 3 y 4, la definición de *email* tiene una lista de hijos. En este caso dentro del elemento <email> deben aparecer los elementos <Para>, <ResponderA>, <De>, <Tema>, <Adjunto>, <Mensaje>[2] en ese orden.

De estos últimos podemos definir el número de presencias en el elemento padre de la siguiente forma.

[2]Debe ser alguno de los dos tipos de Mensaje (HTML ó TXT)

<!ELEMENT nombre-elemento (contenido)>: Definimos que el elemento deberá aparecer una vez. En la línea 3 definimos que <*De*> debe aparecer una vez dentro de un elemento email.

<!ELEMENT nombre-elemento (contenido+)>: Definimos que el elemento deberá aparecer al menos una vez. En un <*email*> debemos tener por lo menos una dirección de destino (línea 3).

<!ELEMENT nombre-elemento (contenido*)>: Definimos que el elemento puede aparecer las veces que sean. Podríamos tener un <*email*> que no tenga <*Adjuntos*> o que tenga uno o más (línea 3).

<!ELEMENT nombre-elemento (contenido?)>: Definimos que el elemento deberá aparecer una vez o ninguna. Esto lo vemos en la línea 3 de nuestro ejemplo, con la definición del contenido de <*email*> en la parte de <*ResponderA*>.

En la declaración de contenido podemos mezclar cualquiera de éstos, excepto *EMPTY* o *ANY*.

La sintaxis para declarar los atributos de un elemento es:

```
<!ATTLIST nombre-elemento
        nombre-atributo tipo-atributo valor-omision>
```

Ahora veamos por separado cada uno de sus elementos:

nombre-elemento: Es el elemento al que se desea agregar un atributo.

nombre-atributo: Es el identificador del atributo a agregar.

tipo-atributo: Es la clase de dato del atributo el cual puede ser de los siguientes tipos:

CDATA	El valor del atributo es texto y no se tratará de interpretar.
(eval1|eval2 |...)	Sólo puede tomar alguno de los listados (eval1 o eval2, etc).
ID	El valor del atributo tiene que ser único (un identificador), ningún otro elemento del mismo tipo puede tener el mismo valor en el atributo.
IDREF	Es el ID de otro elemento.
IDREFS	Es una lista de ID.
NMTOKEN	Es un nombre válido de XML.
NMTOKENS	Es una lista de nombres válidos de XML.
ENTITY	El valor del atributo es una entidad (un ENTITY). Es un valor constante predefinido.
ENTITIES	El valor del atributo es una lista de entidades.
NOTATION	El valor del atributo es el nombre de una Notación (cm, mm, etc).
xml	Es el valor predefinido.

valor-omisión: Asigna el valor por omisión del atributo o puede agregar condiciones, las cuales están descritas en la siguiente tabla.

#DEFAULT *valor*	Asigna el valor por omisión del atributo.
#REQUIRED	Obliga que a este atributo siempre se le asigne un valor.
#IMPLIED	Declara que el atributo puede aparecer o no.
#FIXED *value*	Le da un valor fijo al atributo y no se puede modificar.

Podemos ver ejemplos de éstos entre las líneas 15 y 19 dentro de la figura 2.1

En la figura 2.1 se muestran documentos XML que se ajustan a la descripción establecida por el DTD de la figura 2.1.

```
<email>
        <Para direccion=oescamil@gmail.com
                nombre="Oscar Escamilla González" />
        <ResponderA direccion=otra@servidor.com/>
        <De direccion=dir@servidor.com/>
        <Tema>Ninguno en particular</Tema>
        <MensajeTXT>
                Un texto
        </MensajeTXT>
</email>
```

```
<email>
        <Para direccion=oscar@gmail.com />
        <De direccion=dir@servidor.com
                nombre="Panchito Acosta"/>
        <Tema> Ninguno en particular</Tema>
        <MensajeHTML>
                <font color="#09af19">
                        Nada en particular
                </font>
        </MensajeHTML>
</email>
```

Figura 2.2: Ejemplos de documentos *email*

2.1.2. XML Schema

Al igual que un DTD, XML Schema tiene como propósito definir la estructura de un documento XML. Trata de cubrir las carencias de un DTD agregando más características como:

- Es extensible para adición de nuevas características.
- Está escrito en XML.
- Soporta espacios de nombres.
- Hace un mejor manejo de los tipos de datos.

Este poder adicional en XML Schema viene acompañado de una complejidad mayor en las descripciones. En el trabajo que aquí se presenta no fue necesario integrarlo; por ello no se profundizará más en el tema.

2.2. Estilos de intérpretes para XML

Una vez que se tiene el documento XML se desea poder trabajar con éste y como se mencionó en la sección anterior, XML no proporciona dicha característica, por lo que se recurre al uso de un intérprete o analizador para obtener la información y procesarla adecuadamente. Existen dos formas típicas de realizar el análisis de un documento XML, mediante SAX o DOM. Éstas son dramáticamente diferentes en el estilo de realizar su tarea, como veremos a continuación.

2.2.1. DOM (*D*ocument *O*bject *M*odel)

DOM es una interfaz independiente del lenguaje y la plataforma, además de que permite a programas y scripts acceder y actualizar dinámicamente el contenido, estructura y estilo de un documento XML. El documento puede ser procesado y el resultado de ese proceso puede ser incorporado de nuevo dinámicamente al documento.

DOM es una especificación de W3C[16] acerca de las características y funciones que debe tener un *analizador* de este tipo. En general lo que hace un analizador del estilo DOM, es formar un árbol jerárquico[3] y almacenarlo en memoria. A partir de este árbol los datos pueden ser recuperados recorriéndolo.

En nuestro ejemplo (figura 2.2) se generará un árbol muy parecido al mostrado en la figura 2.3

DOM tiene tres niveles distintos en su especificación.

Nivel 1. Especifica las interfaces mínimas de bajo nivel que puedan representar cualquier documento.

Nivel 2. Soporta espacios de nombres, CSS[4] y eventos[5], con lo que se per-

[3]Esto es muy natural dada la estructura de un documento XML.

[4]Cascading Style Sheets[32]. Es una especificación del aspecto del documento al ser presentado.

[5]Llamaremos eventos a acciones con el teclado, ratón, etc.

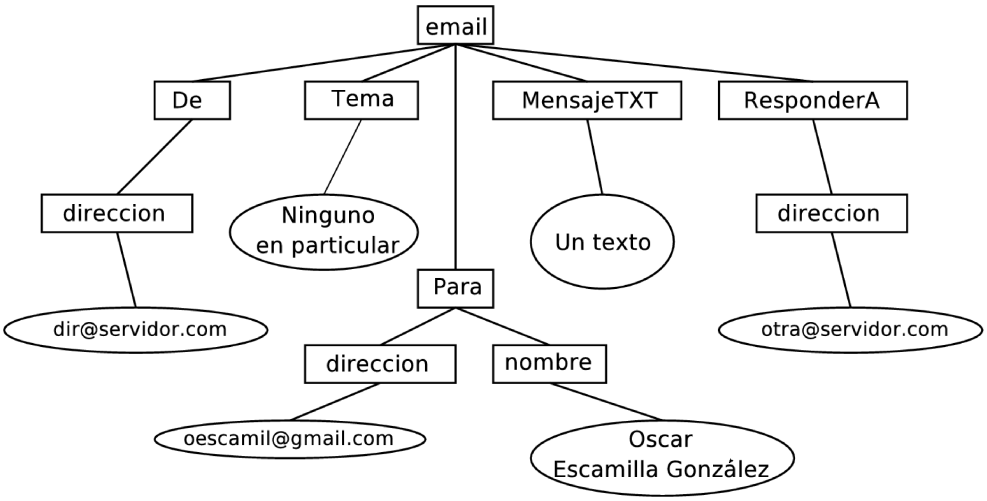


Figura 2.3: Árbol de un documento XML email

mite un XML más dinámico.

Nivel 3. Extiende las funciones relacionadas con eventos ya que permite definir **manejadores de eventos**. Describe el flujo de eventos a través de la estructura del árbol sintáctico que genera DOM y provee la información contextual básica de cada evento.

La ventaja de este estilo de analizador es que la estructura de datos ya está construida lo que permite agregar, eliminar o alterar elementos de forma simple. Sin embargo para documentos muy grandes, el tiempo de ejecución y memoria utilizada para formar estas estructuras podría no ser excesivo. De igual forma, si lo que nos interesa es sólo un segmento pequeño del documento o bien la información es irrelevante, por ejemplo sí sólo se requiere contar los elementos, este estilo desperdicia muchos recursos en cuanto a memoria y tiempo de procesamiento.

2.2.2. SAX (*S*imple *A*PI for *X*ML)

SAX al igual que DOM especifica una manera de acceder a la información almacenada en un documento XML para cualquier lenguaje o plataforma. Sin embargo SAX resuelve el problema desde una perspectiva diferente. A diferencia de DOM, un analizador SAX no construye una estructura, sino que

<MensajeTXT>	⟵ *Inicio Etiqueta*
Un texto	⟵ *Contexto*
</MensajeTXT>	⟵ *Fin etiqueta*
<ResponderA	⟵ *Inicio Etiqueta*
direccion=otra@servidor.com	⟵ *Contexto*
/>	⟵ *Fin Etiqueta*

Figura 2.4: Eventos SAX

notifica acerca de eventos durante el análisis[6] (figura 2.4). Con estos eventos y su contexto[7] la aplicación puede entonces procesar la información del documento. La desventaja de SAX es que una vez que se procesa un evento el analizador lo descarta y no hay forma de recuperar eventos anteriores directamente, esta es la razón por la que este modelo de analizador es para lectura solamente en contraste con DOM. Para documentos extensos o búsquedas de pequeñas secciones de información en proporción al documento, SAX tiene un mejor manejo de recursos.

2.3. JAVA

Java fue diseñado por James Gosling de Sun Microsystems en 1990. Estaba pensado para funcionar como software para dispositivos electrónicos como calculadoras, microondas, la televisión interactiva, entre otros. Curiosamente, el lenguaje fue diseñado antes de que comenzara la era World Wide Web, entorno en el que ha adquirido popularidad.

Como dato curioso, inicialmente Java se llamó OAK (roble en inglés), aunque tuvo que cambiar debido a que dicho nombre ya estaba registrado por otra empresa. Se dice que el nombre original fue OAK debido a la existencia de este árbol en los alrededores del lugar de trabajo del equipo de desarrolladores.

Java ganó aceptación mundial con gran rapidez debido principalmente a

[6]Como el hecho de que empezó una etiqueta, terminó el documento, etc.

[7]Con contexto nos referimos a la información correspondiente a cada etiqueta, como sus atributos, su nombre, etc.

que es un lenguaje orientado a objetos (basado en POO[8]), independiente de la plataforma y con un ambiente de trabajo en red sencillo con la ayuda de JDK (*Java Development Kid*). El JDK contiene además un gran número de herramientas para construir interfaces gráficas para el usuario y los *applets* que son pequeñas aplicaciones que se pueden agregar directamente en documentos HTML. La *independencia de la plataforma* es una razón importante para el éxito de Java. Sus programas se compilan y generan código (*bytecode*) para una máquina virtual (denominada JVM o *Java Virtual Machine*). Este código puede ser ejecutado en cualquier computadora que tenga un intérprete; además los intérpretes de Java cuentan con "JIT" (*Just In time*), lo cual quiere decir que el código en *bytecode* se compila de nuevo para generar código nativo de la plataforma correspondiente, y así aumentar la velocidad en ejecución. Con el uso de Java los desarrolladores de software se liberan de la carga de hacer portátiles sus programas y pueden concentrarse en la producción de software que se podrá ejecutar en cualquier computadora que tenga una JVM.

Aquí se resumen algunas de las características por las que Java se ha hecho tan popular:

Orientación a objetos: Java es casi totalmente orientado a objetos[9]. Todos los métodos se encuentran encapsulados en alguna clase e incluso los tipos primitivos tienen clases que los encapsulan.

Simplicidad: La sintaxis de Java se parece mucho a la de C y C++, lo que la hace fácil de aprender, aunque Java es mucho más simple y pequeño que C++. En Java no existen encabezados de archivos (archivos .h), aritmética de apuntadores, preprocesado, herencia múltiple y sobrecarga de operadores. Además realiza la recolección automática de basura, lo cual beneficia al desarrollador pues éste no tiene que preocuparse por el manejo explícito de la memoria.

Compactabilidad: Los intérpretes y las bibliotecas básicas están diseñadas para ser pequeños. La versión más compacta puede utilizarse para controlar pequeños dispositivos (celulares, PDAs, etc). El intérprete de Java y el soporte básico, se mantienen pequeños al empaquetar por separado otras bibliotecas.

[8]POO.- Paradigma Orientado a Objetos.

[9]Como Java tiene iteraciones, condicionales y tipos primitivos como enteros y reales, hay mucha gente que lo considera un lenguaje híbrido.

Portabilidad: El código fuente se compila para una arquitectura independiente de la plataforma (para la JVM) y se puede ejecutar en cualquier computadora con un intérprete de Java.

Trabajo en Red: Java tiene elementos integrados para el trabajo en red (*java.net*), aplicaciones Web (*applets*), aplicaciones cliente-servidor (*servlets*), además de contar con acceso remoto a bases de datos (*JDBC*).

Trabajo en XML: Las bibliotecas de Java cuentan con una amplia gama de clases para crear, leer y manipular documentos XML. Cabe mencionar que el primer analizador para XML del modelo SAX fue desarrollado en este lenguaje.

Interfases Gráficas: Las bibliotecas *java.awt* y *java.swing* facilitan la escritura de programas orientados a eventos[10] con interfaces gráficas. Contiene una gran gama de objetos para desarrollar este tipo de aplicaciones como botones, áreas de texto, creación y despliegue de imágenes, manejo y abstracción de una gran cantidad de eventos, como movimientos del ratón, escritura con el teclado, etc.

Internacionalización: Java es uno de los primeros lenguajes de programación cuya codificación nativa es *UNICODE*. Java cuenta con herramientas para la manipulación de carácteres *UNICODE*, soporte para fecha, hora local, etc.

Hilos de ejecución (*Threads*): Tiene soporte para la construcción, manejo y comunicación entre múltiples hilos de ejecución.

2.4. Datos base de los mapas

2.4.1. Aplicación original

El doctor Michael Barot[3] construyó una aplicación en C++ para desarrollar mapas y proyecciones. Este programa toma como entrada un archivo con la definición de los contornos de continentes y lagos, dando como resultado un archivo en formato postscript. Fue elaborado con propósitos didácticos y para el desarrollo de un video[23] producido en conjunto por el Instituto de Matemáticas de la UNAM y TV UNAM.

[10] En JAVA estos eventos también están encapsulados en objetos.

El archivo antes mencionado fue obtenido a través del portal de la CIA[11] de los Estados Unidos de Norte América; este archivo es público y se puede descargar desde la página de la CIA [24]. Dicho archivo contiene la definición de los contornos, pero no son polígonos completos, son pequeños segmentos que al unirlos forman el contorno completo. Cada desplazamiento está definido por un punto inicial (con coordenadas X, Y) seguido por una lista de desplazamientos (donde desplazamiento se entiende como la cantidad que se le debe sumar al punto anterior en su coordenada X o en su coordenada Y para llegar al siguiente).

El archivo está estructurado de la siguiente manera:

Un encabezado. Su longitud es de 47 bytes de los cuales los tres primeros son una cadena de tres carácteres ("cil"), los siguientes tres marcan un número. Un byte más es utilizado para marcar un salto de línea. Los siguientes treinta y dos bytes son comentarios con un salto de línea al final. Los últimos ocho, dos enteros de 4 bytes, son el número de segmentos y el máximo número de desplazamientos que puede tener un segmento.

Una lista de Segmentos. La definición de los segmentos es de la siguiente manera. Los primeros 24 bytes definen 4 enteros de 5 bytes donde los primeros 2 números definen la latitud máxima y mínima del segmento, los siguientes 2 representan la longitud máxima y mínima. Le siguen 4 bytes que marcan el número de desplazamientos a leer. Cada desplazamiento es de 4 bytes de longitud, son dos números que marcan el desplazamiento en X y el desplazamiento en Y respectivamente. Una vez que se leyó la cantidad indicada de desplazamientos los siguientes bytes se tiene que tomar como la definición de un nuevo segmento.

2.4.2. Decodificación y codificación

En esta sección se hablará del diseño elaborado y del trabajo hecho para generar un documento XML a partir de los datos mencionados en la sección anterior. Este XML tiene la finalidad de ofrecer un documento público que contenga la información de los contornos de los continentes, islas y lagos sobre la superficie de la Tierra mediante polígonos, brindando las ventajas de un documento XML.

[11] CIA - Central Intelligence Agency.

Diseño XML (documento *Polygons*)

Para lograr una mayor versatilidad y un mejor manejo en los datos, se decidió codificarlos en XML manteniendo la idea de polígonos, solo que en lugar de tener segmentos separados, el nuevo formato consta de polígonos completos y cerrados.

A continuación se presenta el DTD que define la estructura para la nueva codificación.

2.5.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE Polygons [
3   <!ELEMENT Polygons (Polygon+)>
4   <!ATTLIST Polygons units
5           (degrees | minutes | seconds) #REQUIRED>
6   <!ELEMENT Polygon (InitPoint, (Points | Deltas))>
7   <!ATTLIST Polygon Orientation
8           (CtrClockwise | Clockwise) #REQUIRED>
9   <!ELEMENT InitPoint EMPTY>
10  <!ATTLIST InitPoint x CDATA #REQUIRED>
11  <!ATTLIST InitPoint y CDATA #REQUIRED>
12  <!ELEMENT Points (Point+)>
13  <!ELEMENT Point EMPTY>
14  <!ATTLIST Point x CDATA #REQUIRED>
15  <!ATTLIST Point y CDATA #REQUIRED>
16  <!ELEMENT Deltas (Delta+)>
17  <!ELEMENT Delta EMPTY>
18  <!ATTLIST Delta x CDATA #REQUIRED>
19  <!ATTLIST Delta y CDATA #REQUIRED>
20 ]>
```

Figura 2.5: Un DTD para definir los contornos de lagos y continentes.

Este DTD describe un documento del tipo *Polygons* (línea 2). Estos documentos estarán compuestos de una lista de elementos de tipo *Polygon* (línea 3). El elemento *Poligons* tiene un atributo *units* (línea 4) el cual indica el tipo de unidad correspondiente a los valores de cada punto en los elementos *Polygon*, ya sean grados, minutos o segundos. Los elementos *Polygon* tienen

un atributo describiendo su orientación (línea 7) y dos elementos hijos (línea 6) que corresponden a un punto inicial (*InitPoint*) y una lista de puntos ó de desplazamientos (*Points* ó *Deltas* respectivamente).

El atributo *Orientation* nos indica si el polígono representa un lago o una masa continental mediante la convención de que la tierra siempre está del lado derecho, al recorrer el polígono.

El hijo *InitPoint* indica dónde empieza el polígono. Los elementos *Point* de la lista de puntos nos indican la secuencia de puntos que constituyen al polígono. Para los elementos *Delta*, es importante el punto inicial ya que estos elementos nos indican cuánto tenemos que desplazarnos del punto anterior para llegar al actual, es decir, si tenemos:

```
<InitPoint x=0 y=0>
  <Deltas>
    <Delta x=10 y=10/>
    <Delta x=10 y=10/>
  </Deltas>
```

Nuestro polígono consistirá de los puntos $P_1 = (0,0)$, P_2=(10,10) y P_3=(20,20), donde P_2 es el punto inicial desplazado diez unidades sobre x y diez unidades sobre y y P_3 es el punto P_2 desplazado diez unidades sobre x y diez unidades sobre y. [12].

El objetivo de tener desplazamientos (*Deltas* línea 6) ó puntos (*Point* línea 6) es tener diversidad en la forma de representación; así, si alguien quiere utilizar nuestro DTD tienen dos opciones. El resto de los tipos no se explican, dado que a partir de la figura 2.5 es claro su significado.

Traducción a XML (*Polygons*)

Antes de profundizar en el diseño y desarrollo relacionado con la codificación de los datos en un archivo XML, es importante destacar que esta parte del desarrollo del sistema fue realizada por el *Doctor José de Jesús Galaviz de Casas* (jose@pateame.fciencias.unam.mx), la *Lic. Adriana Ramirez Vigueras*(tita@puemac.matem.unam.mx) y *Gildardo Bautista Cano* (gil@puemac.matem.unam.mx) quienes realizaron un árduo y excelente trabajo relacionado con la planeación, organización, análisis y conversión de los datos del archivo mencionado en la sección 2.4.1 **"Aplicación Original"**.

[12]Hay que aclarar que también como convención los polígonos empiezan y terminan en el mismo punto.

Cabe mencionar que gracias a este trabajo se logró que tanto la implementación como la interfaz gráfica reflejaran un manejo adecuado y poco distorsionado de los mapas. Dicho trabajo merece esta distinción pues sin su colaboración no habría sido posible realizar la implementación y por consiguiente la obtención de tan buenos resultados en este trabajo.

Ya con el DTD y el formato del archivo original, se procedió a decodificar y codificar de nuevo en XML los datos. Este trabajo consistió en tres etapas.

Recodificación: Dado que el formato de los datos estaba comprimido, se tradujo a flotantes para almacenarlos como tales en el documento XML.

Separación: Se distinguió entre los polígonos que estaban completos, es decir, aquellos segmentos que iniciaban y terminaban en el mismo punto. Éstos fueron alrededor de 90 y en su mayoría eran islas y lagos.

Con los demás, se utilizó un algoritmo ingenuo para unirlos y crear los polígonos enteros. Este consistió en buscar los vecinos más cercanos utilizando como referencia los puntos extremos de cada segmento y unirlos. Después de este proceso todavía existían muchos segmentos que el programa había omitido o existían polígonos mal formados, por lo que se procedió a la corrección manual de estos problemas. Esto es bastante ineficiente pero este proceso sólo se realizó una vez, por lo que el costo por la ineficiencia comparado con el tiempo de implementación resultó ser la mejor opción.

Traducción. Una vez que se tenía la definición correcta de los polígonos se empezó un proceso de eliminación de puntos y polígonos. Muchos de los polígonos eran demasiado pequeños para ser útiles por lo que se eliminaron. En general esto ocurría con islas y lagos, por lo que se decidió eliminar aquellos polígonos que consistieran de menos de 1250 puntos. Se tomó esta decisión ya que para el nivel de detalle que se pretendía alcanzar con el documento XML, estos polígonos serían demasiado pequeños.

El resultado final fue un documento XML de tipo *Polygon* con un tamaño de 31 megabytes. Este archivo para fines prácticos es demasiado grande para manipularse adecuadamente, por lo que se decidió reducir el número de puntos tomando uno de cada dieciséis de cada polígono. Con esto se logró reducir el tamaño del documento a 2.1 megabytes conservando una calidad aceptable en el detalle de los contornos.

Capítulo 3

Descripción del sistema

Los trabajos de cartografía, en especial los relacionados con la elaboración de mapas, constituían una labor intensa y tardada que, en general, no lograba los resultados de precisión esperados. Cada mapa era elaborado artesanalmente, lo que hacía que no hubiera dos iguales y además dificultaba la amplia difusión de los mapas. La invención de la imprenta y otros mecanismos de impresión ayudó a reducir estos inconvenientes y con el desarrollo de las computadoras y dispositivos electrónicos casi se ha alcanzado la desaparición de estos problemas (principalmente de tiempo y precisión).

3.1. Introducción

El sistema expuesto en este trabajo se llama JAMM[1]. Es un sistema implementado en Java y está diseñado para ofrecer un conjunto de bibliotecas para desarrollar proyecciones de la Tierra de manera sencilla. Ofrece también una interfaz gráfica que puede ser utilizada en red (un *applet*) para visualizar y configurar cada tipo de proyección. Este sistema fue desarrollado pensando en ofrecer al usuario la oportunidad de experimentar y visualizar los resultados de un conjunto de proyecciones predefinidas. Ofrece a los programadores avanzados un conjunto de bibliotecas con las cuales se puedan crear nuevos tipos de proyecciones, además de contar con la biblioteca de la interfaz gráfica que permite desplegarlas y manipular los parámetros de las mismas ofreciendo una manera rápida y sencilla de observar los resultados, lo único que se debe hacer es extender e implementar una cla-

[1] JAMM son las siglas para **J**ust **A**nother **M**ap **M**aker.

se (*jamm.gui.TransformacionToGUI o jamm.gui.TransformacionAdapter*) y agregarla a un objeto JAMM (*jamm.gui.JAMM*). Además el sistema incluye el archivo XML[21] que contiene la descripción de los polígonos que representan a los contornos de continentes y lagos sobre la superficie de la tierra.

3.2. Diseño

JAMM está dividido en tres módulos principales:

Módulo 1.- Analizador de un documento XML *Polygons[21]*: como ya mencionamos, esta parte se encarga de analizar un documento XML en el cual se encuentra contenida la descripción de los contornos de continentes y lagos, para su interpretación y manipulación posterior. Este analizador construye estructuras que definen polígonos con base en los datos contenidos en el documento XML.

Módulo 2.- Motor de proyecciones: éste se encarga de hacer la transformación de los puntos, reconstruir los polígonos y hacer cambios sobre ellos para desplegarlos. Cabe mencionar que esta parte es independiente de la anterior y de la interfaz gráfica. Se acopla a los demás módulos a través de la clase *jamm.tranformacion.Poligono*.

Módulo 3.- Interfaz gráfica: ésta se encarga de desplegar los resultados, capturar datos como el tipo de proyección que se desea utilizar y los parámetros de la misma.

3.2.1. Analizador

En el capítulo anterior se mencionó que existen dos modelos de analizadores de XML, el modelo utilizado en este trabajo es SAX. Se tomó esta decisión debido a que el árbol sintáctico generado por un analizador de estilo DOM sería muy pesado, en cuanto a espacio en memoria se refiere, para nuestro problema en particular. El número estimado de puntos es de 66,000. Esto generaría un árbol con al menos 66,000 hojas, más toda la estructura contextual jerárquica asociada; por lo que resulta innecesariamente costoso utilizar un analizador de este tipo.

Se utilizó el modelo SAX para generar estructuras, porque si bien mantiene los 66,000 puntos, no guardará el resto de la información contextual acerca de éstos.

Las funciones de este módulo del sistema (figura 3.1) son el asignar un documento para ser analizado, comenzar el análisis del mismo y la entrega del resultado.

Documento: se puede indicar el documento que se desea procesar por medio de la asignación de su URL.

Proceso: una vez que se tiene asignado el documento se pide a este módulo que lo analice y construya las estructuras que contendrán los datos.

Resultados: después de procesado el documento, regresa el resultado del proceso, el cual será entregado en forma de un arreglo de objetos de tipo *Polygon*. Adicionalmente, se le puede pedir el tipo de unidad en el que están codificados los puntos de cada *Polygon*. (Ver Sección 2.4.2).

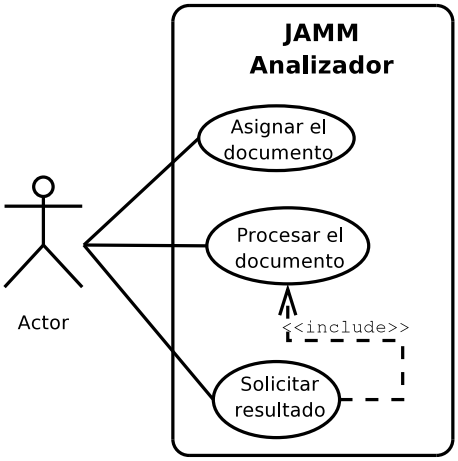


Figura 3.1: Caso de uso del analizador

3.2.2. Motor de proyecciones

Todas las proyecciones trabajan sobre puntos, se les aplica una transformación (fórmula) bajo ciertas restricciones implícitas de la misma. Con este diseño, el motor de proyecciones es un programa que trabaja sobre un conjunto de *polígonos*. A cada uno de sus puntos les aplica una *fórmula*.

Esta fórmula cambiará[2] sin modificar todo el proceso, ya que el aplicar la fórmula a cada punto es una pequeña parte, comparada con el proceso de transformación y modificación adecuada del polígono.

Con respecto a la modificación adecuada de los polígonos, cabe mencionar que existen casos en los que un polígono al ser transformado puede fragmentarse. Si esto ocurre, el sistema realizará las modificaciones pertinentes (crear los polígonos necesarios y agregar o quitar puntos).

Los fragmentos resultantes de la transformación se pueden agrupar en tres casos:

- **El fragmento es todo el polígono original.**
  Se refiere a un polígono que no se fragmentó después de la transformación, por lo que no se realiza un proceso extra con él. Estos polígonos se identifican fácilmente porque el punto final del fragmento es igual a su punto inicial (figura 3.2).

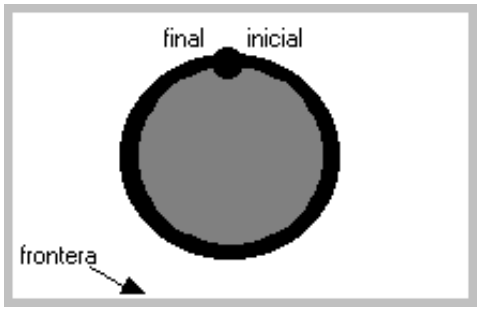


Figura 3.2: Polígono completo

- **El inicio de un fragmento es el fin de otro.**
  En este caso se necesita unir los dos fragmentos para generar uno solo. En el lado izquierdo de la figura 3.3 se muestra este caso. Se tienen dos fragmentos de polígono (A y B). El punto inicial de A es igual al punto final de B, por lo tanto estos dos fragmentos deben unirse en uno nuevo (el polígono C, lado derecho de la figura 3.3), cuyo punto inicial es el punto inicial de B y su punto final es el punto final de A.

[2] Al igual que las restricciones implícitas de la misma.

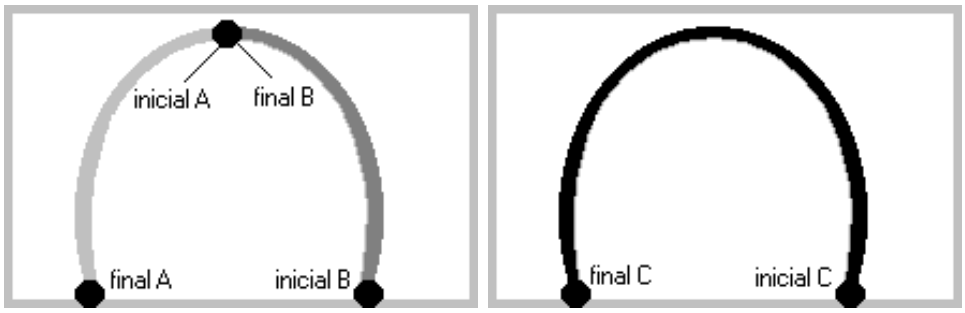


Figura 3.3: Unión de fragmentos

- **El fragmento no contiene a otro.**
  Cuando un polígono se fragmenta y los fragmentos resultantes no cumplen con las características mencionadas en las definiciones anteriores, los extremos de los fragmentos (el punto inicial y el punto final de cada uno) están muy cerca de la frontera del mapa, por lo tanto se deben agregar todos los puntos de la frontera que estén entre estos dos puntos.

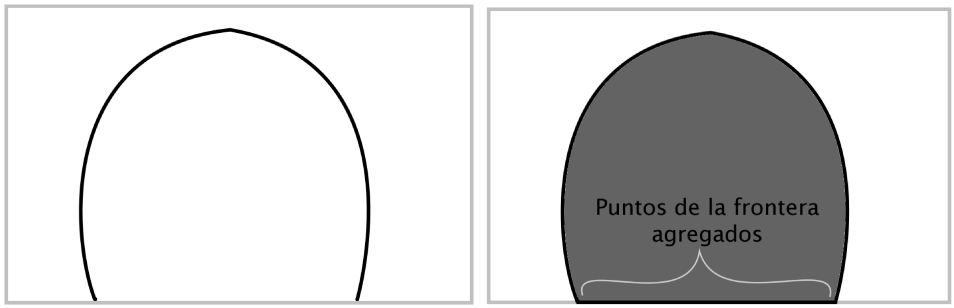


Figura 3.4: Agregar puntos de la frontera

- **El fragmento contiene a otros.**
  Cuando un polígono tiene sólo contenciones simples[3] el nuevo polígono se forma de la unión de éste con todos los polígonos que contiene y las porciones de la frontera que se encuentra entre ellos (ver figura 3.5).

  Cuando existen contenciones no simples, los polígonos se deben separar por profundidad, es decir, un polígono tiene que unirse con los que él

[3]Decimos que un polígono tiene sólo contenciones simples si sólo contiene polígonos que no contienen a su vez a otros.

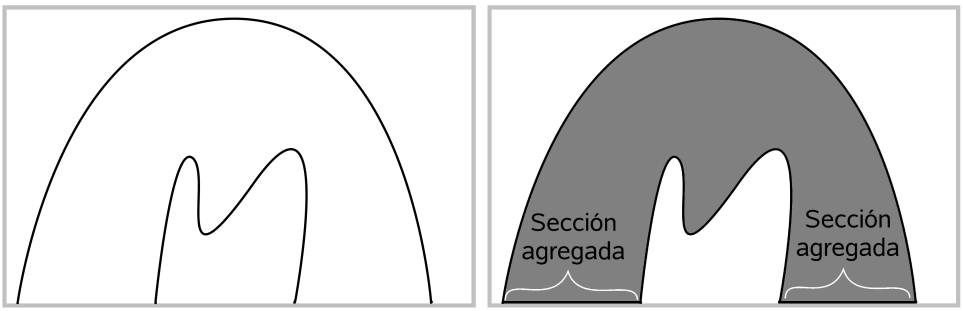


Figura 3.5: Unión simple

contenga directamente. Los demás deben ser procesados de nuevo bajo todas las reglas anteriores, descartando los segmentos que ya han sido unidos.

Para entender bien esto veamos un ejemplo. En la figura 3.6 del lado izquierdo tenemos cinco fragmentos (A, B, C, D, E). En este caso uniríamos los fragmentos A, B y C, dando como resultado el polígono F y los fragmentos D y E (lado derecho figura 3.6).

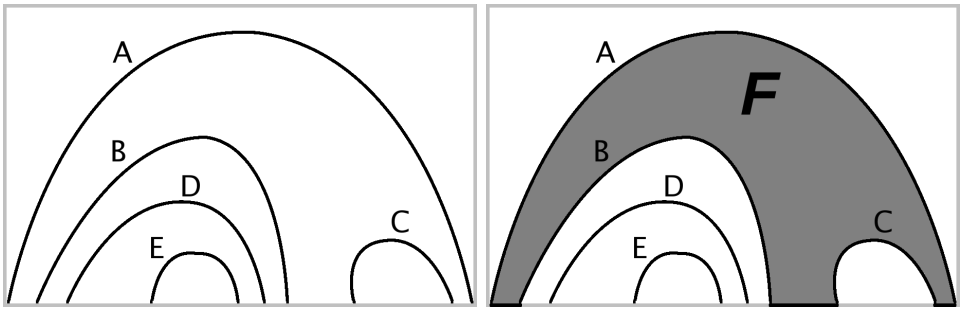


Figura 3.6: Contenciones no simples

Entonces descartamos los fragmentos que ya unimos y procedemos con los restantes (D y E) como se muestra al lado izquierdo de la figura 3.7. Por lo que el resultado de ésto, son dos polígonos independientes como se muestra en la figura 3.8.

Retomando la idea de la *fórmula* como una transformación, podemos hacer varias observaciones. La fórmula a aplicar y los puntos que se mapean en la frontera dependen de la transformación, por lo que estos aspectos tendrán

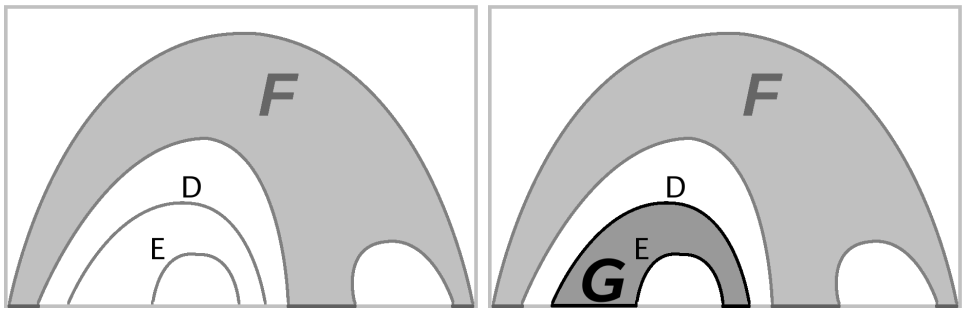


Figura 3.7: Contenciones no simples

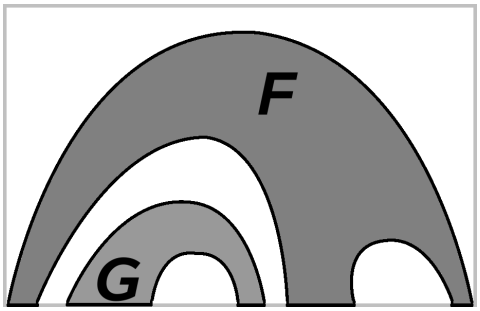


Figura 3.8: Resultado

que ser especificados dentro de cada transformación. Esto nos permite definir una clase abstracta para las transformaciones. En la figura 3.9 se presenta el diagrama de clases en UML[22] de *Transformacion*. El valor *toRadians*

jamm.transformacion.Transformacion
`+toRadians: double = PI/180`
`+apply(point:java.awt.geom.Point2D): java.awt.geom.Point2D` `+getBorder(): jamm.transformacion.Poligono`

Figura 3.9: Clase Abstracta *Transformacion*

ofrece una forma sencilla para transformar grados a radianes o viceversa, multiplicando o dividiendo por $\frac{\pi}{180}$ respectivamente.

El método *apply* también es abstracto y recibirá un objeto de la clase *Point2D*[4], el cual contendrá la longitud y latitud del punto que se quiere

[4]Se encuentra dentro del paquete java.awt.geom en las bibliotecas estándar de Java.

transformar, mismas que estarán expresadas en grados.

getBorder es un método que regresa la frontera de la transformación, es decir los puntos de los extremos. Comúnmente ésta se encuentra compuesta por el conjunto de puntos (±180,Y), (X,±90), donde:

$$X \in [-180, 180],$$
$$Y \in [-90, 90].$$

Continuando, el motor de proyecciones está pensado para ofrecer un comportamiento como el que se representa en el caso de uso de la figura 3.10. En el cual se puede asignar la fórmula a evaluar, las unidades de los valores de cada punto[5] y posteriormente se realiza la transformación. El proceso de transformación implica el manejo de los segmentos y polígonos resultantes con base en los casos antes expuestos.

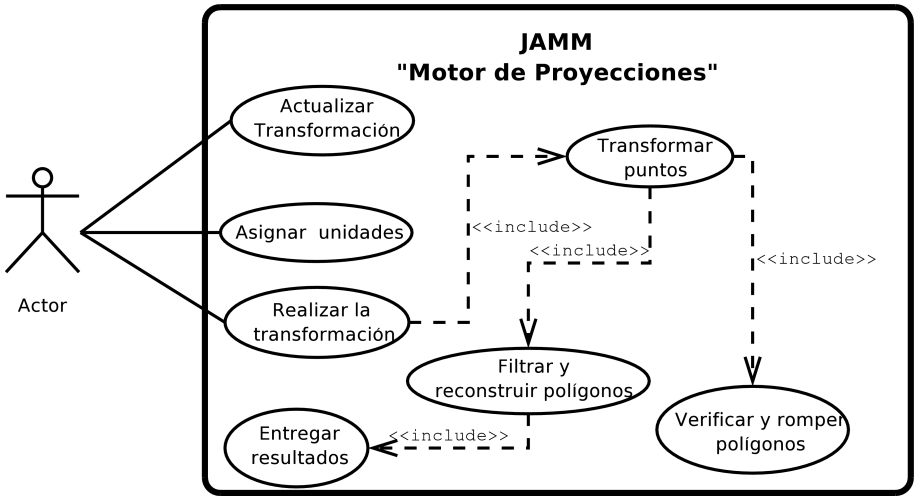


Figura 3.10: Caso de uso del motor de proyecciones

3.2.3. Interfaz gráfica

La interfaz gráfica está diseñada con la finalidad de que el usuario pueda modificar el tipo de proyección, los parámetros de la misma, o bien, realizar la transformación de los puntos con base en la proyección, todo esto de manera independiente, desplegando gráficamente el resultado (ver figura 3.11).

[5]Estas unidades pueden ser *grados, minutos o segundos.*

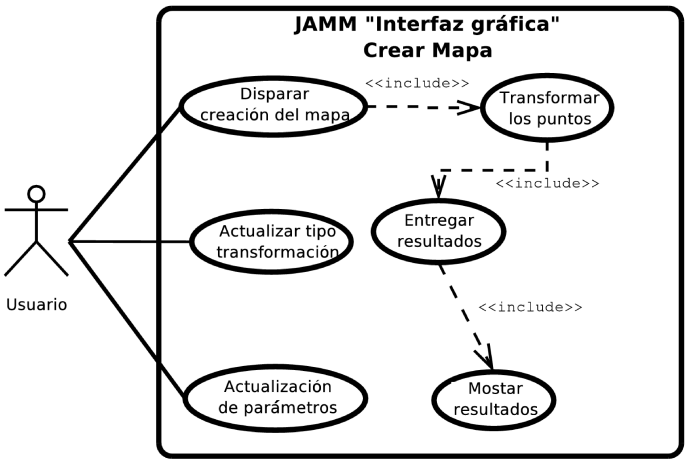


Figura 3.11: Caso de uso de la interfaz gráfica

Con el propósito de poder cambiar los parámetros de la transformación por medio de la interfaz gráfica, es necesario agregar más funciones a nuestra clase abstracta *Transformacion*. Con este fin definimos una nueva clase abstracta *TransformacionToGUI*, la cual nos permitirá obtener más información acerca de la transformación (figura 3.12).

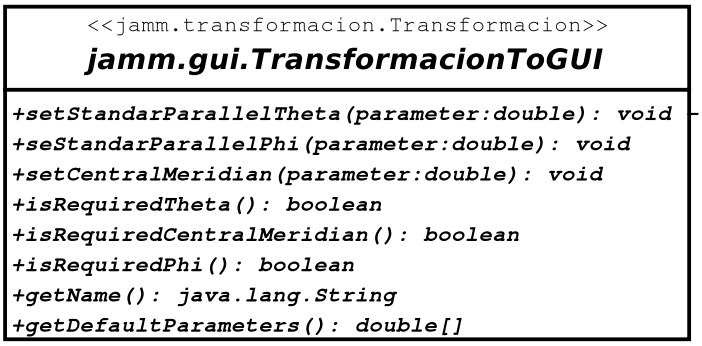


Figura 3.12: Clase Abstracta *TransformacionToGUI*

Con los métodos *setStandarParallelTheta, setStandarParallelPhi y setCentralMeridian*, se asignan cada uno de los parámetros (los dos paralelos estándar y el meridiano central respectivamente).

Cabe mencionar que todas las transformaciones utilizan un máximo de tres parámetros. Con el objetivo de brindar una mayor versatilidad a la inter-

faz gráfica necesitamos conocer los parámetros que utiliza la transformación, para ello definimos tres métodos *isRequiredTheta, isRequiredPhi e isRequiredCentralMaridian*. Éstos nos dicen si dentro de la proyección existen parámetros que la modifican como el paralelo estándar *theta*, el paralelo estándar *phi* o el *meridiano central* respectivamente.

El método *getName* regresa una cadena de carácteres, para que la interfaz gráfica pueda etiquetarla y mostrar esta etiqueta en un menú emergente, con el fin de que el usuario pueda identificarla y seleccionarla.

3.3. Implementación

3.3.1. Analizador

Como se mencionó en la sección anterior esta parte es la encargada de analizar el documento XML y regresar el resultado del análisis. Esto se logra por medio de la clase *jamm.parser.Parseo* (figura 3.13). *Parseo* entrega el resultado en un arreglo de objetos de tipo *jamm.transformacion.Poligono* (ver figura 3.14) por medio de alguno de los siguientes métodos: *getPoligonos* o *parser*.

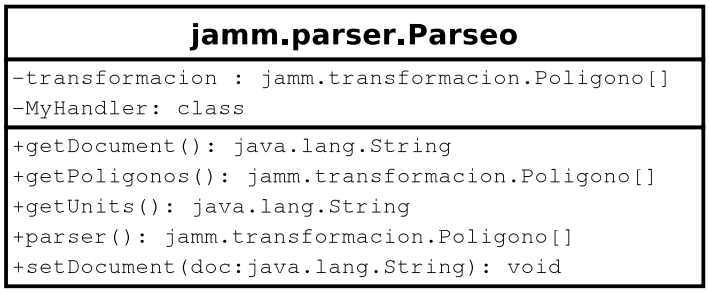


Figura 3.13: Clase *Parseo*

El analizador está implementado utilizando la clase *javax.xml.parsers.SAXParser*. Esta clase se encarga de leer y analizar el documento pasando los eventos a un objeto de la clase *org.xml.sax.helpers.DefaultHandler*. Se extendió esta clase dentro de *Parseo* como un elemento privado llamado *MyHandler* y se sobreescribieron los métodos necesarios para darle el comportamiento deseado a ciertas funciones, como las encargadas de detectar los

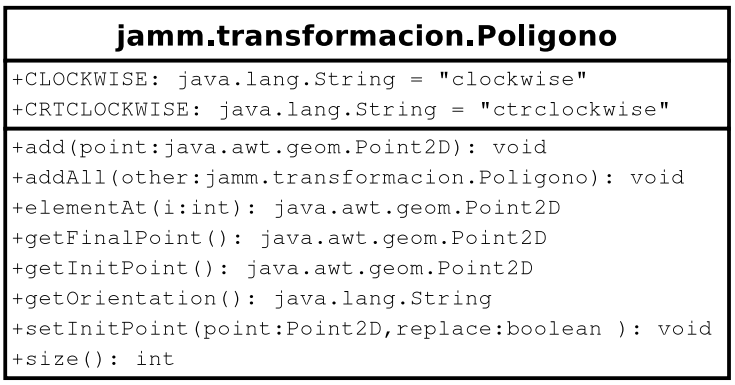


Figura 3.14: Clase *Poligono*

eventos de errores, de inicio y fin de un elemento; por esta razón no se sobreescribieron todos los métodos ya que sólo necesitábamos de estos eventos.

De igual manera, no era necesario implementar el reconocimiento de inicio para todos los elementos del documento XML. A continuación se muestra el código completo del método *startElement*, que es invocado por la implementación de la clase *SAXParser* al encontrar el inicio de un elemento. Después del código se muestran los elementos relevantes y las acciones respectivas a tomar al detectar el inicio de cada uno.

Método *startElement* en MyHandler

```
1  private class MyHandler extends DefaultHandler{
2  /* Para encontrar el índice del elemento */
3  final String[] opcion=
4   {"delta","point","polygon","initpoint","polygons"};
5  Poligono actual;
6  double[] ini=new double[2];
7  public void startElement(String namespaceURI,
8   String localName,
9   String qualifiedName,
10  Attributes attributes)throws SAXException {
11  /*
12    Obtenemos el nombre verdadero entre localName
13    y qualifiedName
14  */
15  String name=
16   getName(localName, qualifiedName).toLowerCase();
```

```
17  int op=-1;
18  /* Buscamos el índice */
19  for(int i=0;i<opcion.length;i++){
20   if(name.equals(opcion[i])){
21    op=i;
22    break;}}
23  switch(op){
24   case 0: /* <Delta x='X' y='Y'> */
25    double equis=
26     Integer.parseInt(attributes.getValue(0));
27    double ye=
28     Integer.parseInt(attributes.getValue(1));
29    ini[0]=(ini[0]+equis);
30    ini[1]=(ini[1]+ye);
31    actual.add(new Point2D.Double(ini[0],ini[1]));
32    break;
33   case 1: /* <Point x='X' y='Y' > */
34    ini[0]=(Integer.parseInt(attributes.getValue(0)));
35    ini[1]=(Integer.parseInt(attributes.getValue(1)));
36    actual.add(new Point2D.Double(ini[0],ini[1]));
37    break;
38   case 2: /* <Polygon Orientation='X' > */
39    actual=new Poligono();
40
41    if((attributes.getValue(0).toLowerCase())
42                        .equals("clockwise")){
43     actual.setOrientation(actual.CLOCKWISE);
44    }else{
45     actual.setOrientation(actual.CTRCLOCKWISE);
46    }
47    break;
48   case 3:/* <InitPoint x='X' y='Y'> */
49    ini[0]=Integer.parseInt(
50     attributes.getValue(0));
51    ini[1]=Integer.parseInt(
52     attributes.getValue(1));
53
54    actual.setInitPoint(
55     new Point2D.Double(ini[0],ini[1]),true);
56    break;
```

```
57    case 4: /* <Polygons units='X'> */
58      /*
59        Units es una variable global de Parseo y
60        guarda las unidades del polígono
61      */
62      units=attributes.getValue(0).trim().toLowerCase();
63      break;
64    }
65  }
66  ⋮
```

A continuación se detalla cada uno de los tipos de datos que el método *startElement* maneja:

<Polygons>: (línea 57) Para obtener el atributo *units* y así determinar las unidades en las que están codificados los valores de cada punto en los elementos *<Polygon>*.

<Polygon>: (línea 38) Al detectar el inicio de este elemento construimos un nuevo objeto de la clase *jamm.transformacion.Poligono* y le asignamos la orientación, es decir, el valor del atributo *Orientation* del elemento *<Polygon>* actual. Este nuevo objeto *Poligono* se asigna a la variable *actual* (definida en la línea 5), para su manipulación posterior.

<InitPoint>: (línea 48) Al iniciar el elemento se actualizan los valores del arreglo *ini* (definido en la línea 6) con el de los atributos del elemento *<InitPoint>*. Con estos nuevos valores se crea un nuevo objeto *java.awt.geom.Point2D* y se asigna como el punto inicial del polígono que se está procesando y que está asignado a la variable *actual*.

<Point>: (línea 33) Al detectar este elemento creamos un nuevo objeto *java.awt.geom.Point2D* con los atributos del elemento *<Point>* y lo agregamos al polígono asignado a la variable *actual*.

<Delta>: (línea 24) Cuando el comienzo de este elemento es detectado se deben sumar a los valores del punto anterior los valores de los atributos de este elemento, en nuestro caso, a los valores almacenados en el arreglo *ini* y se construye un nuevo objeto *Point2D*. Después de esto, se agrega el nuevo punto al polígono *actual* y se actualizan los valores del punto anterior, es decir, los valores del arreglo *ini*. El motivo por

lo que esto funciona es muy similar al caso del elemento <*InitPoint*> ya que sabemos que antes de detectar el inicio del elemento <*Delta*>, se debió detectar el inicio de <*InitPoint*> o de algún otro elemento <*Delta*>, por lo que siempre el valor del punto anterior está almacenado en el arreglo *ini*.

El inicio de los demás elementos como <*Points*> *o* <*Deltas*>, es irrelevante dado que sólo nos sirven para el control de la estructura en la construcción del documento XML.

Por último <*Polygon*> es el único elemento que nos interesa detectar si alcanzó su final. En el momento que se detecta este evento se sabe que se terminó de recolectar toda la información del polígono y se puede empezar a analizar y almacenar la de uno nuevo.

La información de los otros elementos es recuperada en su totalidad al detectar su inicio, por lo que no es necesario detectar el evento de fin de elemento para ellos.

Método *endElement*

```
0  ⋮
1  public void endElement(String namespaceUri,
2                         String localName,
3                         String qualifiedName)
4      throws SAXException {
5    /* Obtenemos su verdadero nombre */
6   String name=getName(localName,qualifiedName).toLowerCase();
7   if(name.equals("polygon")){
8    Point2D fin=(Point2D)(actual.getInitPoint()).clone();
9    actual.add(fin);
10   poligonos.add(actual);
11   /* Donde se almacenan los Polígonos */
12  }
13 }
14 ⋮
```

Como se observa en el código anterior, al final del objeto polígono se agrega un punto igual al punto inicial (línea 9), esto es porque los polígonos definidos en el documento XML deben tener como punto final un punto con las mismas coordenadas que el punto inicial, a consecuencia de ésto el programa se asegurará que sea un polígono cerrado, ya que errores de

redondeo (en especial en el caso de los elementos <*Delta*>) podrían causar diferencias entre estos puntos a pesar de que en el documento se cumpla la condición.

3.3.2. Motor de proyecciones

La parte más interesante del motor de proyecciones se encuentra en la clase *jamm.transformacion.Proyeccion*, representada por el método *public Poligono[] apply(Poligono poligono)* y el método *public Poligono[] applytoGrid(Poligono poligono)*.

Antes de exponer los dos métodos mencionados, se explicarán las clases y métodos auxiliares que son utilizados dentro de la implementación de *apply* y *applytoGrid*.

Método *fixed*

```
0 public Point2D fixed(Point2D p)
1 {
2   return  new Point2D.Double(p.getX()*unit,p.getY()*unit);
3 }
```

El método *fixed* toma como argumento un objeto de la clase *Point2D* y convierte los valores de sus atributos, para que sus unidades estén en términos de grados. Devuelve un elemento *java.awt.goem.Point2D* con las unidades correctas para la proyección[6]. Las unidades para los valores de los puntos se especifican mediante el método *seUnits* de la clase *Proyeccion*

Método *calculaSeparacion*

```
0 protected void calculaSeparacion()
1 {
2  Poligono frontera=transformacion.getBorder();
3  Point2D punto1=frontera.getInitPoint();
4  Point2D punto2=frontera.elementAt(frontera.size()/2);
5
6  deltaSeparacionN=punto1.distance(punto2)/10.0;
7  deltaSeparacionM=deltaSeparacionN/2.0;
8 }
```

[6]El método *apply* de la clase *jamm.transformacion.Transformacion* necesita que las unidades los puntos estén en grados.

La clase *jamm.transformacion.Proyeccion* detecta si un polígono se debe fragmentar cuando el último punto proyectado se haya separado *"demasiado"* del anterior. Pero la pregunta a responder es, ¿cómo detectar cuándo es "*demasiado*"?. El programa trata de solucionar este problema calculando la distancia entre dos puntos, lo que nos da una buena referencia acerca de las dimensiones del mapa resultante. Estos puntos son el punto inicial y un punto que esté a la mitad en la frontera. La frontera se determina por medio del objeto asignado a la variable global *transformacion*[7], dentro de la clase *Proyeccion*.

Se utiliza esta distancia para calcular el valor de las variables *deltaSeparacionN* y *deltaSeparacionM*, las cuales representan la distancia máxima permitida para que dos puntos se separen antes de fragmentar un polígono ya sea para una proyección normal o una proyección aplicada a polígonos que integran la cuadrícula del mapa respectivamente. Esto se debe a que existen diferencias entre el comportamiento de estos dos tipos de polígonos. Los polígonos que representan los lagos y continentes siempre son polígonos cerrados; en cambio los polígonos que integran la cuadrícula del mapa son polígonos abiertos[8].

Los valores 10.0 y 2.0 (en las líneas 6 y 7) se calcularon empíricamente y son los que mejores resultados presentaron; aunque dada la manera en que estos valores se calcularon, podrían originar errores como el de fragmentar más de lo debido un polígono o no separarlo.

Lo anterior ocurre porque existen proyecciones en las cuales los puntos se separan muy rápido a medida que están más cerca de ciertas regiones o en casos extremos se expanden al infinito. Esta situación se puede prevenir con un buen manejo de la frontera y una revisión de los puntos a transformar. Este problema desaparece si la frontera se define de tal manera que ninguno de sus puntos caiga en las regiones antes mencionadas; esto es similar a restringir los puntos a proyectar a un conjunto. Por ejemplo, en la **Proyección de Mercator** (Sección 1.2.2) ocurre lo descrito anteriormente en los puntos muy cercanos a los polos, es decir, todos los puntos cuya latitud sea mayor a 85^o e inferior a -85^o, por lo tanto se debe restringir la proyección a que ninguno de los puntos cumpla lo mencionado, en otras palabras, los puntos de la frontera deben ser:

[7]Se puede cambiar el valor de esta variable por medio del método *setTransformacion* de la clase *jamm.parser.Parseo*.

[8]Es decir que su punto inicial y su punto final no tienen las mismas coordenadas.

$$[\pm 180, Y]\ Y \in [-85, 85],$$
$$[X\ , \pm 90]\ X \in [-180, 180].$$

Además el método *apply* de la clase *jamm.transformacion.Transformacion* (ver la figura 3.9), está diseñado para lanzar un mensaje (la excepción *IndeterminatePointException*) indicando que el punto que recibió como parámetro no se puede calcular. En el ejemplo de la proyección de Mercator, al implementar el método *apply* de la clase *jamm.transformacion.Transformacion*, se podría verificar que la latitud del punto esté en -85^o y 85^o[9].

Los dos conceptos (restringir la proyección y redefinir la frontera) obviamente no son iguales, pero dado que todo lo que no esté dentro de la frontera o no se pueda calcular no será tomado en cuenta, el resultado es similar.

Aunque este método no es utilizado por *apply* directamente, los valores que este método actualiza (*deltaSeparacionN* y *deltaSeparacionM*) son indispensables ya que con base en estos se toma la decisión de fragmentar un polígono. Este método se invoca cada vez que se cambia la transformación con el método *setTransformacion* de la clase *jamm.transformacion.Proyeccion*.

Método *daIndicesCercanos*

```
0  private int[] daIndicesCercanosF(Poligono adherir){
1   Point2D iniAdherir=adherir.getInitPoint();
2   Point2D finAdherir=adherir.getFinalPoint();
3   double minIni=Double.MAX_VALUE;
4   double minFin=minIni;
5   int minIdIni=-1;
6   int minIdFin=-1;
7
8   for(int i=frontera.size()-1;i>0;i--){
9    Point2D punto=frontera.elementAt(i);
10   double dist=finAdherir.distance(punto);
11   if(minFin>dist){
12    minIdFin=i;
13    minFin=dist;
14   }}
15
16   for(int j=1;j<frontera.size();j++){
```

[9]De esta manera está implementada la proyección de Mercator en el sistema.

```
17      Point2D punto=frontera.elementAt(j);
18      double  dist=iniAdherir.distance(punto);
19      if(minIni>dist){
20       minIdIni=j;
21       minIni=dist;
22      }}
23     int res[] ={minIdFin,minIdIni};
24     return res;
25  }
```

Este método busca los índices de los puntos más cercanos al punto inicial y al punto final del polígono *adherir* sobre la frontera de la proyección. El método supone que el polígono que se recibe como parámetro es abierto, es decir, es un fragmento y por lo tanto el índice de los puntos más cercanos será distinto en cada caso (para el punto final y el punto inicial). Conocer estos índices es importante ya que como se mencionó en secciones anteriores, cuando un polígono se fragmenta, tanto el punto inicial como el final de los polígonos resultantes quedarán muy cerca de la frontera y conociendo los puntos donde el polígono haría contacto con ella, se pueden determinar los puntos que serán agregados en caso de ser necesario.

Método *PegaPuntosAFrontera*

```
0  private  Poligono pegaPuntosAFrontera (Poligono actualTmp)
1  {
2   PoligonoWithIndexs actual=new PoligonoWithIndexs(actualTmp);
3   Point2D ini1=actual.getInitPoint();
4   Point2D fin1=actual.getFinalPoint();
5   if(ini1.getX()==fin1.getX() && ini1.getY()==fin1.getY())
6    return actual;
7   int[] indices=daIndicesCercanosF(actual);
8   Point2D ini=frontera.elementAt(indices[1]);
9   Point2D fin=frontera.elementAt(indices[0]);
10  actual.setFin(indices[0]);
11  actual.setIni(indices[1]);
12  actual.setInitPoint(ini,false);
13  actual.add(fin);
14  return actual;
15 }
```

El método *pegaPuntosAfrontera* agrega los puntos más cercanos de la

frontera como puntos extremos del polígono, para que así el punto final e inicial del mismo se encuentren sobre la frontera de la proyección. En caso de que el polígono sea cerrado, es decir, que su punto inicial y su punto final sean el mismo, no se realiza ninguna modificación. Cabe mencionar que este método encapsula el polígono en un objeto de la clase *PoligonoWithIndexs* (figura 3.15).

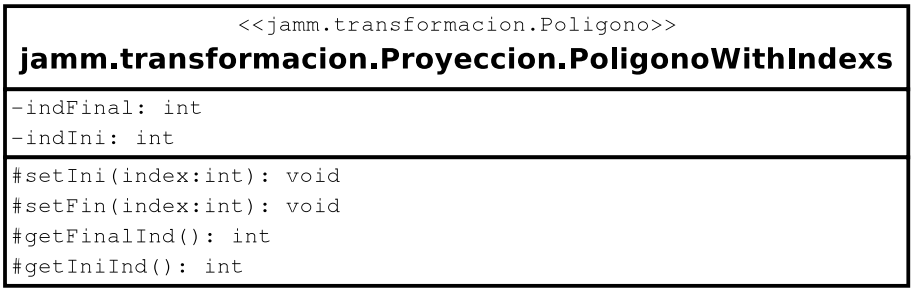


Figura 3.15: PoligonoWithIndexs

Esta clase extiende la clase *jamm.transformacion.Poligono* y ofrece además la funcionalidad de guardar la posición de los puntos extremos del polígono sobre la frontera de la proyección. Estas posiciones o índices serán utilizados en el proceso de agregar los puntos necesarios de la frontera al polígono, así como en el proceso para unir polígonos conforme a los casos expuestos en la Sección 3.2.2.

Método *unePorInicialFinal*

```
0  private Poligono[] unePorInicialFinal(Poligono[] source)
1  {
2   Vector fuente=new Vector();
3   for (int i=0;i<source.length;i++) {
4    fuente.add(source[i]);
5   }
6   if(!fuente.isEmpty()){
7    Poligono anterior=(Poligono)fuente.lastElement();
8    Point2D inicialA=anterior.getInitPoint();
9    Point2D finA=anterior.getFinalPoint();
10   for(int i=0;i<fuente.size()-1;i++){
11    Poligono tmp=(Poligono)fuente.elementAt(i);
12    Point2D puntoFin=tmp.getFinalPoint();
13    Point2D puntoIni=tmp.getInitPoint();
```

```
14      if(puntoIni.distance(finA)<deltaSeparacionN){
15       /* Es el punto Final */
16       fuente.remove(tmp);
17      anterior.addAll(tmp);
18      break;}
19      if(inicialA.distance(puntoFin)<deltaSeparacionN){
20       /* Es el punto Inicial */
21       fuente.remove(anterior);
22       tmp.addAll(anterior);
23       anterior=tmp;
24       break;
25      }}}
26   Poligono[] tmp3=new Poligono[fuente.size()];
27   fuente.toArray(tmp3);
28   return tmp3;
29  }
```

El método *unePorInicialFinal* busca y une dos polígonos que cumplan que el punto inicial de uno sea punto el final del otro. Como consecuencia de esto, en cada llamada a este método, a lo más se unirán dos polígonos y el número de polígonos en el arreglo resultante será, a lo más, uno menos que en el arreglo de entrada.

Método *buscaDentro*

```
0   private int[] buscaDentro(PoligonoWithIndexs adherir,
1                             Vector fuente,int i,int j)
2  {
3   for(int k=0;k<fuente.size();k++){
4    PoligonoWithIndexs tmpFig=
5      (PoligonoWithIndexs)fuente.elementAt(k);
6    if (i==tmpFig.getIniInd()){
7     /* Encontré el inicio de uno */
8     fuente.remove(k);
9     adherir.addAll(tmpFig);
10    int[] res2=new int[2];
11    res2[0]=adherir.getFinalInd();
12    res2[1]=adherir.getIniInd();
13    return res2;
14   }
15  }
```

```
16
17  /* No unimos, no cambiamos */
18  int[] res={i,j};
19  return res;
20  }
```

Este método verifica si el punto de la frontera (indicado por el argumento *i*) que se acaba de agregar al polígono *fuente*, es igual al punto de inicio de alguno de los polígonos restantes (los cuales se encuentran almacenados en el Vector *fuente*). La verificación se hace a través de los índices de los puntos en la frontera y por ende el objeto *adherir* es de la clase *jamm.transformacion.Proyeccion.PoligonoWithIndexs* que almacena estos índices. En caso de encontrar un polígono que cumpla con dicho requerimiento, se elimina del Vector *fuente*, se une al polígono *fuente* y se realizan los ajustes necesarios, como cambiar los índices correspondientes al punto inicial y final dentro de la frontera. Si no encuentra ningún punto, no realiza acción alguna y regresa los mismos índices que obtuvo de los parámetros.

Método *pegaFrontera*

```
0  private void pegaFrontera(final PoligonoWithIndexs adherir,
1                            final Vector fuente)
2  {
3   Point2D iniAdherir=adherir.getInitPoint();
4   Point2D finAdherir=adherir.getFinalPoint();
5   if(iniAdherir.distance(finAdherir)==0)
6    return;
7   int i= adherir.getFinalInd();
8   int j= adherir.getIniInd();
9   String orientacion=adherir.getOrientation();
10  int suma=1;
11  if(!frontera.getOrientation().equals(
12                              adherir.getOrientation())){
13   suma=1;  //recorremos en el mismo sentido
14  }else{
15   suma=-1;   //recorremos en el sentido contrario
16  }
17
18  /*
19   Si tenemos que recorrer la mayoría de la frontera
20   algo hicimos mal. Prevenimos posibles errores y
```

```
21    no agregamos la frontera
22   */
23   if (j-i >= (frontera.size()-10) )
24      return;
25
26   while(i!=j){
27    adherir.add(frontera.elementAt(i));
28    res=buscaDentro(adherir,fuente,i,j);
29    i=res[0];
30    i+=suma;
31    if(suma==1 && i==frontera.size())
32       i=0;
33
34    if(suma==-1 && i== -1)
35     i=frontera.size()-1;
36    }
37  }
```

El método *pegaFrontera* agrega la porción de la frontera que está entre el punto inicial y el punto final del polígono *adherir*, además de unirlo con alguno de los polígonos restantes almacenados en el Vector *fuente*.

En principio el método revisa que se pueda agregar parte de la frontera, esto se logra mediante la comparación del punto inicial y final del polígono. Si éste es cerrado, no se realiza acción alguna. En caso contrario, comprueba la orientación del polígono para decidir en que sentido se deben obtener los puntos de la frontera que serán agregados. Después se examina que no se esté agregando toda la frontera como medida de seguridad para no errar en la construcción del polígono. Si no se detectan errores se comienzan a agregar los puntos de la frontera, cotejando si cada punto agregado es el inicio de algún otro polígono mediante el método *buscaDentro*. Después de agregar y avanzar en los índices, se realiza una verificación de los mismos, dado que los puntos de la frontera se manejan como una lista circular.

A continuación se presenta un pequeño ejemplo gráfico de lo que realiza este método:

En la figura 3.16 se representa un polígono que se fragmentó en 3 segmentos, *adherir, A y B*. En el método *pegaFrontera* se está procesando el polígono *adherir* mientras *A y B* están almacenados en el Vector *fuente*.

Dado que *adherir* no es un polígono cerrado pasa exitosamente la verificación de la línea 5 del método *pegaFrontera* y continúa el proceso (línea

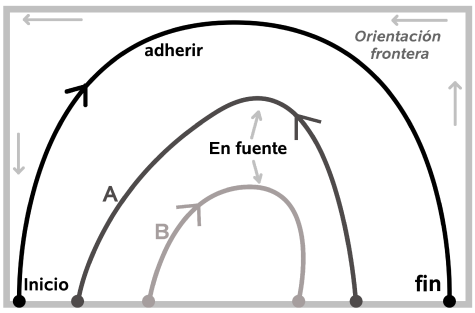


Figura 3.16: Ejemplo

11) con la decisión de cómo recorrer la frontera. Dado que su orientación es distinta a la de la frontera, se agregarán los puntos necesarios recorriendo la frontera en sentido inverso, tal y como se muestra en el lado izquierdo de la figura 3.17.

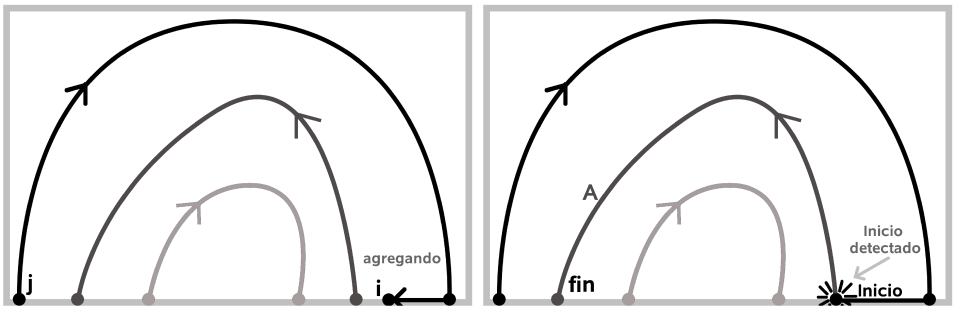


Figura 3.17: Agregando frontera

Cada vez que se agrega un punto nuevo se invoca el método *buscaDentro* (línea 28) para poder detectar el inicio de otro polígono, como se muestra en el lado derecho de la figura 3.17. Al detectar el inicio de A (línea 6 del método *buscaDentro*) el método *buscaDentro* elimina a A del Vector *fuente*, lo agrega a *adherir* y regresa los índices nuevos para continuar agregando el resto de los puntos necesarios de la frontera. Esto se ilustra en la parte izquierda de la figura 3.18.

Finalizamos este proceso cuando los índices son iguales, lo que significa que el polígono se ha cerrado.

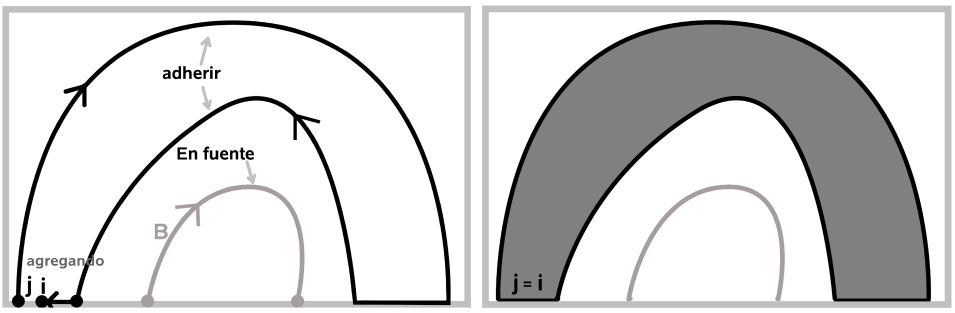


Figura 3.18: Agregando frontera

Método *une*

```
0  protected Poligono[] une(Poligono[] fuente)
1  {
2   int size=fuente.length;
3   fuente=unePorInicialFinal(fuente);
4   /* Unimos por puntos extremos */
5   while(size>fuente.length){
6    size=fuente.length;
7    fuente=unePorInicialFinal(fuente);
8   }
9   /*
10    Pegamos los extremos de los polígonos resultantes
11    a la frontera
12  */
13  Vector fuenteTmp=new Vector();
14  for (int i=0;i<fuente.length;i++) {
15   fuenteTmp.add(pegaPuntosAFrontera(fuente[i]));
16  }
17  Vector res=new Vector();
18  /*
19    Agregamos la frontera y unimos polígonos si
20    es necesario.
21  */
22  while(!fuenteTmp.isEmpty()){
23   PoligonoWithIndexs tmp=
```

```
24              (PoligonoWithIndexs)fuenteTmp.remove(0);
25    pegaFrontera(tmp,fuenteTmp);
26    res.add(tmp);
27   }
28
29   /*
30     Regresamos el resultado como arreglo
31   */
32   Poligono[] tmp3=new Poligono[res.size()];
33   res.toArray(tmp3);
34   return tmp3;
35  }
```

El método *une* se encarga de realizar todos los ajustes necesarios después de transformar un polígono. Este método recibe un arreglo de polígonos que es el resultado de la fragmentación después de la transformación de un polígono[10]. Los posibles ajustes después de la transformación son:

- Unir polígonos con base en sus puntos extremos (con el método *unePorInicialFinal*).
- Agregar segmentos de la frontera y unir polígonos según las reglas dadas en la Sección 3.2.2 (por medio del método *pegaFrontera*).

Método *apply*

```
0  public Poligono[] apply(Poligono poligono){
1   Vector poligonos=new Vector();
2   Poligono newPoligono=new Poligono();
3   Point2D anterior=null;
4   newPoligono.setOrientation(poligono.getOrientation());
5   for(int j=0;j<poligono.size();j++){
6    Point2D tmpPoint=poligono.elementAt(j);
7    try{
8    tmpPoint=fixed(tmpPoint);
9    tmpPoint=rotacion.apply(tmpPoint);
10   tmpPoint=transformacion.apply(tmpPoint);
11   if(anterior==null)
12     anterior=tmpPoint;
```

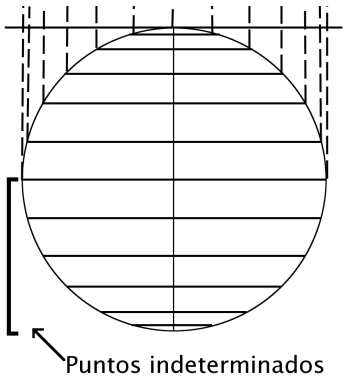
[10]El arreglo puede ser de un solo polígono si el original no se fragmento.

```
13    if(tmpPoint.distance(anterior)> deltaSeparacionN){
14      if(newPoligono.size()>minimoDePoligonos)
15       poligonos.add(newPoligono);
16     newPoligono=new Poligono();
17     newPoligono.setOrientation(poligono.getOrientation());
18     newPoligono.setInitPoint(tmpPoint,false);
19     anterior=tmpPoint;
20     }else{
21      newPoligono.add(tmpPoint);
22      anterior=tmpPoint;
23     }}catch(IndeterminatePointException e){} }
24  Poligono[] res=new Poligono[0];
25  if(newPoligono.size()>minimoDePoligonos)
26   poligonos.add(newPoligono);
27  Poligono[] tmp=new Poligono[poligonos.size()];
28  poligonos.toArray(tmp);
29  res=une(tmp);
30  return res;
31 }
```

apply es el método principal en la transformación de un polígono, éste se encarga de rotar y realizar la transformación de cada punto, decidir cuándo fragmentar e invocar los métodos para los ajustes posteriores a la transformación.

A partir de la línea 8 y hasta la línea 10, se ajustan las unidades del punto que actualmente se encuentra en proceso, se rota y se aplica la transformación. En la línea 13 se verifica si el punto actual no está demasiado separado del punto que se transformó anteriormente. Si la separación es demasiada, se procede a fragmentar, guardando el segmento del polígono que hasta el momento se ha transformado y finalmente se toma uno nuevo (línea 16). Antes de guardar el polígono se verifica que la cantidad de puntos en él sea significativa, es decir, que la cantidad de puntos sea mayor a la variable *minimoDePoligonos* (línea 14), de no ser así se desecha. En caso de que la separación no exceda los límites, se agrega al polígono que actualmente se está procesando, como se puede ver en la línea 21.

Dentro de la proyección, existen puntos que no se pueden transformar. Por ejemplo en la **Proyección Ortogonal** (Sección 1.2.3), esto ocurre con todos los puntos que se encuentran detrás de otro. Estos puntos no aparecerán en el mapa resultante, por lo tanto no están determinados.



En caso de que no se pueda calcular el punto, el método *apply* de la clase *Transformacion* arrojará una excepción (*IndeterminatePointException*). Al detectar esto (en la línea 23) la clase *Proyeccion* descarta el punto y continúa con el procesamiento de los puntos restantes.

Cuando se termina de transformar todos los puntos, se verifica que el último polígono tenga un número significativo de puntos (línea 25) o es desechado. Al final todos los *Poligonos* no descartados se pasan al proceso de ajuste, es decir, se llama al método *une* con los polígonos resultantes como parámetro (línea 29).

Método *applyToGrid*

```
0  private Poligono[] applyToGrid(Poligono poligono){
1                              ⋮
2      /*
3        Esta parte del código es idéntica a la del método
4        ``apply'', por lo que se omite.
5      */
6                              ⋮
7      /* Separación para la cuadrícula */
8     if(tmpPoint.distance(anterior)> deltaSeparacionM){
9      /* La cuadrícula necesita menos puntos */
10     if(newPoligono.size()>minimoDePoligonos/2)
11      poligonos.add(newPoligono);
12     newPoligono=new Poligono();
13     newPoligono.setOrientation(poligono.getOrientation());
14     anterior=null;
15    }else{
```

```
16                                  ⋮
17      /*
18        Esta parte del código es idéntica a la del método
19        ``apply'', por lo que se omite.
20       */
21                                  ⋮
22      /* La cuadrícula necesita menos puntos */
23      if(newPoligono.size()>minimoDePoligonos/2)
24       poligonos.add(newPoligono);
25      Poligono[] tmp3=new Poligono[poligonos.size()];
26      poligonos.toArray(tmp3);
27      res=tmp3;/* Sin ajustes extra */
28      return res;
29  }
```

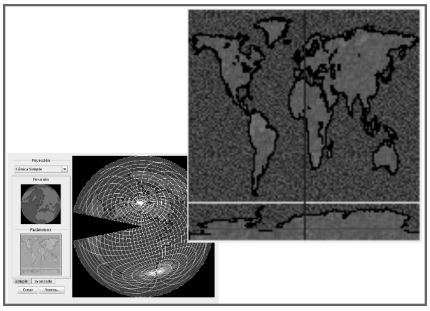
El método *applyToGrid* realiza las mismas operaciones que el método *apply*, con la diferencia de que la separación máxima entre dos puntos es distinta[11], además de que el número de puntos considerados para que un segmento sea significativo es menor.

3.3.3. Interfaz gráfica

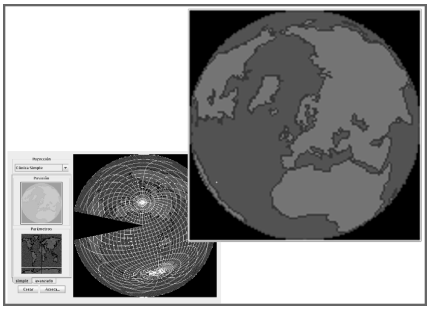
Se desarrollaron tres componentes gráficos en el sistema para facilitar el ingreso de los parámetros y dar una visión más intuitiva de lo que significa cada uno. Por ejemplo, tenemos un componente que con base en los polígonos originales del documento XML, construye una representación de la tierra en tres dimensiones[12] y es utilizado para obtener la posición deseada de la tierra por el usuario para construir el mapa (figura 3.19b). Existe otro (figura 3.19a) cuya función es capturar los datos de los parámetros para la transformación (los paralelos estándar o el meridiano central). Este utiliza una proyección equirectangular para desplegar un mapa, brindando al usuario una representación gráfica de los parámetros. Y por supuesto se desarrolló otro componente (figura 3.19c) para mostrar los resultados de la transformación, tanto los polígonos, como los meridianos y paralelos transformados.

[11] En *apply* es de *deltaSeparacionN* y en este método es *deltaSeparacionM*.

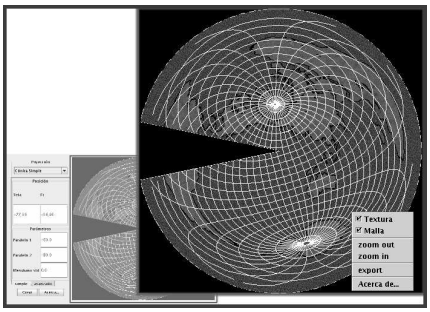
[12] Esto no es completamente cierto dado que no existe un motor de gráficos en 3D, sino que se utiliza el motor de proyecciones para generar una proyección ortogonal.



(a) Paralelos estándar y meridiano central.



(b) Posición de la tierra.



(c) Mapa creado con la proyección.

Figura 3.19: Componentes gráficos para captura de parámetros.

A continuación se muestra al sistema *jamm* después de realizar una proyección cónica la cual tiene como meridiano central el meridiano de Greenwich y cuyos paralelos estándar son 56° y 75° este.

El sistema ofrece dos maneras de introducir los parámetros para la proyección dada. Por omisión, *jamm* muestra y deja modificar gráficamente los parámetros (figura 3.20), pero también brinda la posibilidad de introducir valores exactos al presionar la pestaña "avanzado" (figura 3.21).

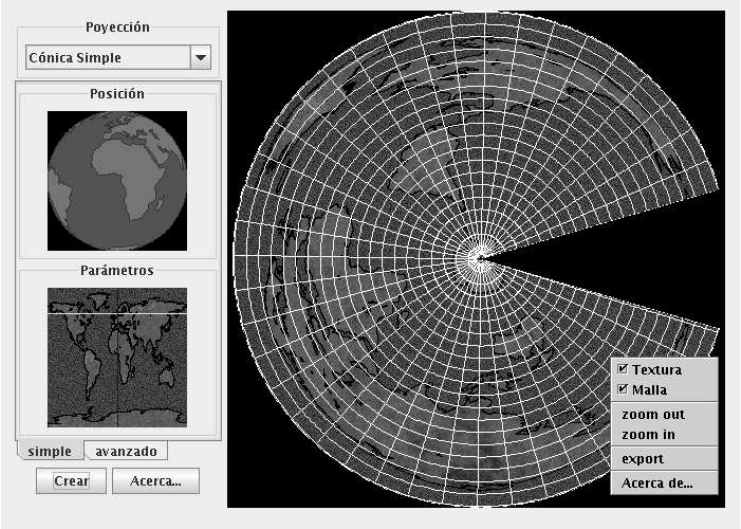


Figura 3.20: Interfaz gráfica de JAMM.

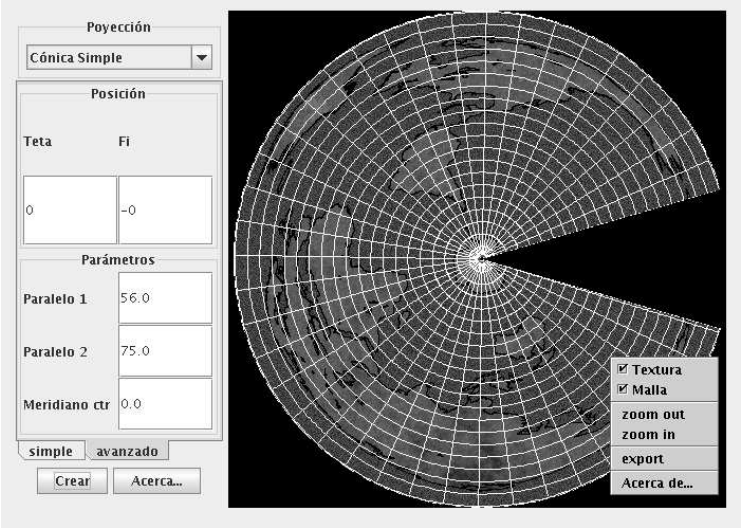


Figura 3.21: Parámetros exactos.

Por otro lado el sistema de la posibilidad de desplegar el mapa resultante de varias maneras. Se puede seleccionar la manera de desplegarlo de un menú emergente (figura 3.22) que aparece al hacer click con el botón derecho del ratón sobre el mapa.

Figura 3.22: Menú emergente de *jamm*.

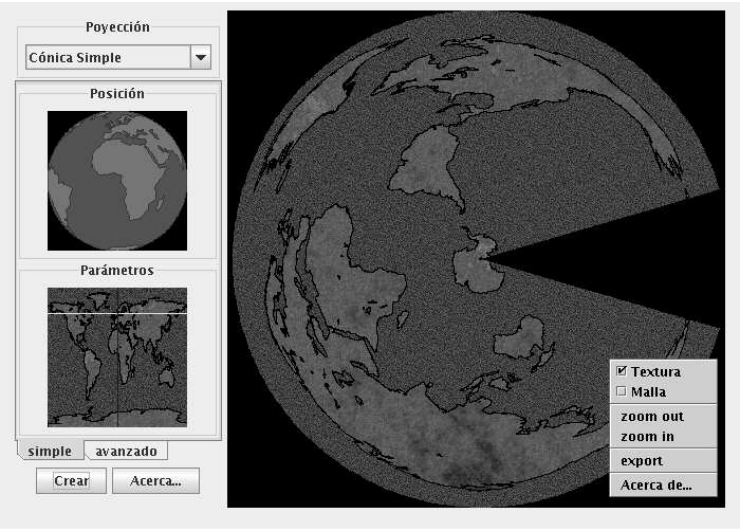


Figura 3.23: Mapa sin paralelos y meridianos.

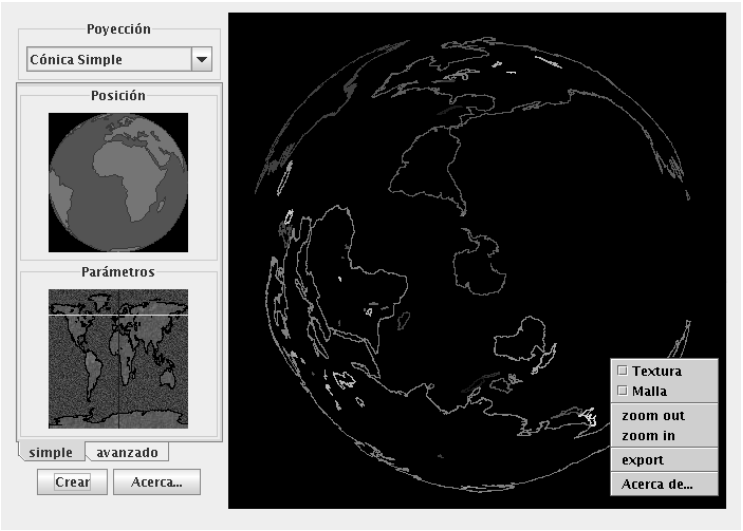


Figura 3.24: Mapa sin texturas.

Por último, el sistema permite hacer acercamientos al mapa a través del mismo menú desplegable.

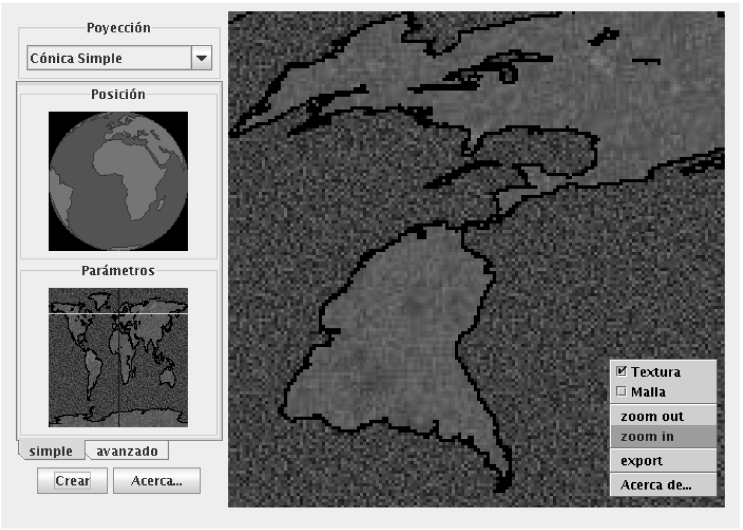


Figura 3.25: Ejemplo de acercamiento a un mapa.

3.4. Trabajo a futuro

Se puede extender el sistema JAMM para que pueda especificar el documento XML del cual se desea extraer los datos si en el futuro se construyen documentos XML distintos, como por ejemplo, alguno que describa la superficie de la luna, de otros planetas o de otros polígonos en general.
También se le pueden agregar los indicadores de Tissot (ver Sección 1.2) de cada proyección, definiendo los polígonos que corresponden a cada círculo o bien implementando algún algoritmo que los calculase para después pasárselos al motor de proyecciones y mostrarlos.
Se podría hacer la presentación de los continentes integrando divisiones políticas y etiquetar con sus nombres los países y estados (en el caso de la Tierra) o divisiones y etiquetas más detalladas de los polígonos. Esto sería por medio de una captura mucho más detallada de la información y una estructuración del documento XML, para darle cabida a la nueva información. Además se tendría que volver a implementar el analizador y agregar funciones a la interfaz gráfica para la manipulación y despliegue de esta nueva información.

Capítulo 4

Comparativo y conclusiones

4.1. Comparativo con otras aplicaciones

Comparemos ahora las cualidades de JAMM, en cuanto a la funcionalidad ofrecida al usuario final [1], con otros sistemas de software libre que se pueden encontrar en la Internet.

<table>
<tr><td colspan="4">Matthew's Map Projection Software
http://www.users.globalnet.co.uk/~arcus/mmps/</td></tr>
<tr><td>Autor</td><td>Objetivo</td><td>Plataforma</td><td>Dependencias</td></tr>
<tr><td>Matthew Arcus</td><td>Ilustrativo</td><td>Unix
Windows</td><td>ImageMagic
Un compilador de C</td></tr>
<tr><td colspan="2">Interfaz gráfica</td><td colspan="2">Proyecciones</td></tr>
<tr><td colspan="2">No cuenta con interfaz gráfica, todos los datos de entrada se toman directamente de la línea de comandos.</td><td colspan="2" rowspan="6">Azimutal
Bonne
Equirectangular
Estereográfica
Gnomónica
Mollweide
Mercator
Ortografica
Sinusoidal</td></tr>
<tr><td colspan="2">Datos de entrada</td></tr>
<tr><td colspan="2">Está basado en la transformación de una imagen en formato ppm[2].</td></tr>
<tr><td colspan="2">Características Relevantes</td></tr>
<tr><td colspan="2">La calidad de la imagen resultante depende del tamaño de la imagen original. Además despliega la sombra que se genera sobre la tierra a una hora determinada.</td></tr>
</table>

[1] El usuario de la interfaz gráfica.
[2] Portable Pixel Image[31]

Online Map Creation (OCM) http://www.aquarius.geomar.de/omc/make_map.html			
Autor	***Objetivo***	***Plataforma***	***Dependencias***
Martin Weinelt	Ilustrativo Técnico	Unix Windows MacOS	Un navegador con soporte para imágenes y conexión a la Internet.
Interfaz gráfica		***Proyecciones***	
Es una forma HTML que se encarga de recopilar los parámetros para las proyecciones.		Azimutal de Lambert Azimutal equidistante Cilíndrica equidistante Estereográfica Mercator Ortográfica	
Datos de entrada			
Definición del contorno de los continentes através de puntos.			
Características Relevantes			
Es solo un *front-end* para una biblioteca de C (GMT) especializada en datos cartográficos. Da la opción de crear el mapa y guardarlo como un *postscript*[3] o lo despliega como una imagen *jpg*[4] en el navegador.			

Java map proyection of the world http://www.btinternet.com/~se16/js/oldmapproj.htm			
Autor	***Objetivo***	***Plataforma***	***Dependencias***
Henry Bottomley	Didáctico	Unix Windows MacOS	Navegador, java
Interfaz gráfica		***Proyecciones***	
Interfaz gráfica simple. Se puede escoger las proyección en un menú desplegable y los parámetros son obtenidos por el teclado.		Azimutal Azimutal estereográfica Cilíndrica de igual área Gnomónica Mercator Mollweide Ortográfica Sinusoidal Triangular de igual área	
Datos de entrada			
Utiliza una imagen para obtener los datos de los puntos.			
Características Relevantes			
El mapa resultante no tiene una buena definición, se ven muy marcados los *pixels* como al agrandar una imagen pequeña.			

[3] Consulte **http://es.wikipedia.org/wiki/PostScript/**.
[4] Consulte **http://www.image-formats.com/jpeg.htm**.

PUEMAC Sección Mapas http://interactiva.matem.unam.mx/mapas/html/polo.html			
Autor	***Objetivo***	***Plataforma***	***Dependencias***
José Galaviz Gildardo Bautista	Didáctico	Unix Windows MacOS	Navegador, java
Interfaz gráfica		***Proyecciones***	
Tienen un interfaz gráfica simple. Presenta el mapa resultante en blanco y negro.		Equirectangular Estereográfica	
Datos de entrada			
Utiliza una imagen para obtener los datos de los puntos.			
Características Relevantes			
El mapa resultante no tiene una buena definición aunque su desempeño es rápido.			

Versamap Mapping & Cartography Software http://www.versamap.com/			
Autor	***Objetivo***	***Plataforma***	***Dependencias***
versamap	Didáctico Técnico	Windows	Ninguna
Interfaz gráfica		***Proyecciones***	
Interfaz gráfica simple, todos los datos se obtienen por medio de campos de texto.		Albers de igual área Azimutal de áreas iguales Azimutal equidistante Cónica conforme Cónica simple Equirectangular Estereográfica Gnomónica Hammer Aitoff Mercator Ortográfica Robinson	
Datos de entrada			
Definición del contorno de los continentes a través de puntos.			
Características Relevantes			
Ofrece la opción de guardar el mapa resultante en distintos formatos, además de permitir poner etiquetas en puntos específicos. Además gráfica las divisiones políticas.			

<table>
<tr><td colspan="4">JAMM (Just Another Map Maker)
http://interactiva.matem.unam.mx/mapas/html/proyecta.html</td></tr>
<tr><th>Autor</th><th>Objetivo</th><th>Plataforma</th><th>Dependencias</th></tr>
<tr><td>Oscar Escamilla González
Gildardo Bautista
Adriana Ramírez
José Galaviz</td><td>Didáctico</td><td>Unix
Windows
MacOS</td><td>Navegador, java</td></tr>
<tr><th colspan="2">Interfaz gráfica</th><th colspan="2">Proyecciones</th></tr>
<tr><td colspan="2">Existe una representación gráfica interactiva de la tierra, la cual permite cambiar la posición de la misma para la transformación. Además cuenta con otra donde especifican gráficamente los parámetros. La entrada de los parámetros es de dos formas posibles, por medio de cajas de texto o por medio de los elementos gráficos antes mencionados. También hay que mencionar que despliega el mapa de distintas formas como, sólo los contornos, rellenos con textura, con o sin la malla, con la representación de meridianos y paralelos.</td><td colspan="2" rowspan="6">Armadillo
Azimutal de Lambert
Azimutal equivalente
Azimutal estereográfica
Bonne
Cónica simple
Equirectangular
Hammer aitoff
Mercator
Ortogonal
Sinusoidal simple
Werner</td></tr>
<tr><th colspan="2">Datos de entrada</th></tr>
<tr><td colspan="2">Utiliza un archivo XML con la definición de los contornos de los continentes y lagos.</td></tr>
<tr><th colspan="2">Características Relevantes</th></tr>
<tr><td colspan="2">Permite hacer acercamientos al mapas sin volver a crear la proyección.</td></tr>
</table>

Como se puede apreciar, JAMM respecto a los demás programas similares existentes:

- Ofrece un mayor número de proyecciones.
- Posee una interfaz gráfica atractiva e intuitiva.
- Su dependencia con software adicional es mínima.
- Es independiente de la plataforma.
- Tiene una calidad aceptable en el mapa producido y ofrece una función para hacer acercamientos, además de que la representación de los datos de entrada es mucho más versátil.

Conclusiones

El contar con un sistema computacional que ayude a visualizar los resultados de ciertas proyecciones para construir mapas de la superficie terrestre resulta muy útil para mostrar al público las características de éstas, además de que ayuda a comprender como afectan los parámetros en el mapa resultante.

Por otro lado, tener una buena base de bibliotecas para desarrollar nuevas aplicaciones (como las que ofrece el sistema) con la menor cantidad de trabajo podría ayudar a investigadores y desarrolladores de *software* a verificar, experimentar y conjeturar nuevos resultados. Por ejemplo, se podría utilizar el motor de transformaciones para implementar proyecciones más generales fuera del contexto de la cartografía. Por otro lado, dado que el diseño del sistema es modular, se pueden desarrollar nuevas aplicaciones con base en las bibliotecas del motor de proyecciones, las del analizador sintáctico o en ambas.

Además el sistema JAMM es escalable, dado que el diseño permite añadir nuevos tipos de proyecciones de forma sencilla utilizando el motor de proyecciones e implementando la clase *jamm.transformacion.Tranformacion* o si se desea trabajar con la interfaz gráfica, se debe implementar la clase *jamm.gui.TransformacionToGUI*.

Por otro lado el sistema JAMM logra cumplir los siguientes objetivos:

Ofrecer interacción e intuitividad.
Los componentes gráficos como el que despliega los resultados, el de captura de parámetros y el encargado de designar la posición del globo terrestre en la proyección proporcionan interactividad e intuitividad a la aplicación, al mismo tiempo que brindan un atractivo visual al sistema, con lo que se cumple el cuarto objetivo planteado en la introducción.

Ser didáctico.
Logra ser didáctico al ofrecer una manera sencilla de experimentar con los resultados de distintos tipos de proyecciones y con los cambios en los parámetros de cada una. Gracias a que el sistema tiene un buen desempeño tanto en la transformación como en el cálculo y despliegue de los resultados gráficos, se brinda una manera fácil de observar las diferencias entre los resultados de distintos tipos de proyecciones o los efectos al cambiar parámetros de las mismas. Asimismo la capacidad del sistema para mostrar acercamientos permite apreciar los resultados con mayor detalle.

Tener una audiencia amplia y diversa.
Dado que el sistema está implementado en Java, puede ser presentado a una audiencia amplia y diversa ya que se aprovecha la independencia que mantiene con la plataforma y *software* adicional; además los sistemas operativos utilizados con más frecuencia brindan soporte a los programas hechos en Java. Una ventaja adicional es que puede ser ejecutado dentro de una página WEB. Con esto se cumple con los otros dos objetivos planteados en la introducción.

Índice de figuras

Índice de figuras

Bibliografía

[1] John P.Snyder. **Flatterning the Earth** The University Of Chicago Press, Chicago 1993

[2] Ignacio Gutiérrez Pérez, Artículo 157
http://www.cartesia.org/articulo157.html
noviembre 2004

[3] Página personal de Michael Barot
http://www.matem.unam.mx/barot/
noviembre 2004

[4] PUEMAC
http://interactiva.matem.unam.mx/
noviembre 2004

[5] Proyecciones cartográficas
http://cablemodem.fibertel.com.ar/rubenro/novedades1/7380.shtml
noviembre 2004

[6] Nociones Básicas sobre Proyecciones Cartográficas
http://nivel.euitto.upm.es/∼mab/tematica/htmls/proyecciones.html
noviembre 2004

[7] *Map Proyections*
http://www.fes.uwaterloo.ca/crs/geog165/mapproj.htm
noviembre 2004

[8] La Tierra (Apuntes de Geología General)
http://plata.uda.cl/minas/apuntes/Geologia/geologiageneral/ggcap01b.htm
noviembre 2004

[9] *3-D Software Map Projection*
http://www.3dsoftware.com/Cartography/USGS/MapProjections/
noviembre 2004

[10] Página del conocimiento
http://webs.sinectis.com.ar/mcagliani/
noviembre 2004

[11] *Map Projections*
http://mathworld.wolfram.com/MapProjections.html
noviembre 2004

[12] *Map Projections Loxodrome*
http://mathworld.wolfram.com/Loxodrome.html
noviembre 2004

[13] *Carlos A. Furuti's Home page*
http://www.progonos.com/furuti/index.html
noviembre 2004

[14] Método de Newton
http://mathworld.wolfram.com/NewtonsMethod.html
noviembre 2004

[15] *Standard Generalized Markup Language*
http://www.w3.org/MarkUp/SGML/
enero 2005

[16] *World Wide Web Consortium*
http://www.w3c.org/
enero 2005

[17] *HyperText Markup Language*
http://www.w3c.org/MarkUp/
enero 2005

[18] *HyperText Markup Language History*
http://www.w3c.org/People/Raggett/book4/ch02.html
enero 2005

[19] *International Business Machines*
http://www.w3c.org/MarkUp/
enero 2005

[20] *International Organization for Standardization*
http://www.iso.org
enero 2005

[21] Documento XML de tipo *Polygons*
http://interactiva.matem.unam.mx/~oescamil/instituto/mapas/polygons.xml
noviembre 2004

[22] Tutorial de UML
http://www.dcc.uchile.cl/~psalinas/uml/introduccion.html
noviembre 2004

[23] Aventuras Matemáticas IMATEM
http://www.matem.unam.mx/aventuras/
noviembre 2004

[24] *Central Intelligence Agency*
http://www.cia.gov/
abril 2005

[25] Declinación magnética.
http://www.iesgaherrera.com/fiqui/declmag.pdf
abril 2005

[26] Cartografía griega.
http://www.mgar.net/var/cartgrie.htm
abril 2006

[27] Biografía Al-Idrisi.
http://es.wikipedia.org/wiki/Al-Idrisi
abril 2006

[28] NAVEGACIÓN - VIENTOS
http://www.mgar.net/mar/viento.htm
abril 2006

[29] PUEMAC
http://puemac.matem.unam.mx/mapas/html/cronologia/1100-1300.html
abril 2006

[30] Tim Berners-Lee
http://www.el-mundo.es/navegante/personajes/bernerslee.html
junio 2006

[31] PPM Format
http://netpbm.sourceforge.net/doc/ppm.html
noviembre 2006

[32] Cascading Sheet Style
http://www.w3c.org/CSS/
diciembre 2006

Imágenes del primer capítulo.

Universo para el mundo hindú antiguo.(figura 1.1)
http://www.infidelguy.com/heaven_sky.htm

Tablilla babilonica. (figura 1.2)
http://interactiva.matem.unam.mx/mapas/html/cronologia/6300-300.html

Mapa de Anaximandro de Mileto. (figura 1.3.a)
www.veaseademas.com/archivos/anaximandro.gif

Mapa de Eratóstenes de Cirene. (figura 1.3.b)
http://interactiva.matem.unam.mx/mapas/html/cronologia/ 6300-300.html

Mapa de Ulm. Pertenece a la *Geographia* de Tolomeo (figura 1.4)
http://academic.emporia.edu/aberjame/map/h_map/h_map.htm

Ejemplo de un mapa basado en teología. (figura 1.5)
http://www.princeton.edu/h̃is291/T-O_Map.html

Mapa de Al-Idrisi (figura 1.6)
http://interactiva.matem.unam.mx/mapas/html/cronologia/1100-1300.html

Mapa de Waldseemüller (figura 1.7)
http://academic.emporia.edu/aberjame/map/h_map/h_map.htm

Mapa de Mercator (figura 1.8)
http://academic.emporia.edu/aberjame/map/h_map/h_map.htm