IGLU
Interactive Graphical Library Utilities
versión 20071127

27 de noviembre de 2007

Índice

1. Introducción **1**

2. Uso general de Iglu **1**
2.1. Creación y compilación de un programa usando la librería Iglu 1

3. Uso de IgluDibujo **3**
3.1. Constructores 3
3.2. Objetos gráficos 4
3.2.1. Puntos 4
3.2.2. Líneas 6
3.2.3. Cuadrados 7
3.2.4. Rectángulos 7
3.2.5. Círculos 7
3.2.6. Arcos de círculo 7
3.2.7. Elipses 7
3.2.8. Arcos de elipse 8
3.2.9. Texto 8
3.3. Color 8
3.3.1. Variables de color 10
3.4. Máscara rectangular 11
3.5. Capas 11
3.6. Cambio de coordenadas 13
3.6.1. Arcos de círculo simulados 16
3.7. Acciones sobre la ventana: limpiar, congelar, pausa, grabar, reescalar/restaurar dimensiones 18

4. Uso de IgluGrafico **20**
4.1. Constructores 20
4.2. Títulos 21
4.3. Ejes 21
4.4. Gráficos de línea 21
4.4.1. Líneas 22
4.4.2. Símbolos 22
4.5. Gráficos de barras 24

1. Introducción

Este manual describe el uso de la librería Iglu (versión 1.1), que permite tener una salida gráfica desde un programa en C++ en un sistema Linux. Con Iglu es posible realizar dos tipos de tareas:

1. Dibujos simples (o muy complicados, dependiendo de la habilidad del programador), en base a objetos gráficos elementales (puntos, líneas, rectángulos, arcos de círculo), y con salida tanto a un terminal XWindows como a un archivo PostScript encapsulado.

2. Gráficos elementales, con ejes, títulos, y donde un conjunto de puntos dados es representado con determinados estilos de línea o símbolos.

Iglu es producto del esfuerzo mancomunado de (en orden de ingreso al proyecto): José Rogan, Xavier Andrade, Juan Alejandro Valdivia y Víctor Muñoz. El espíritu es disponer de una herramienta que permita graficar a partir de un programa en C++, para uso en investigación y docencia. Esperamos que esta herramienta —y este manual— sean útiles para sus potenciales usuarios.

2. Uso general de Iglu

2.1. Creación y compilación de un programa usando la librería Iglu

El siguiente código esquematiza un uso elemental de Iglu:

```
#include "iglu.h"

using namespace std;
using namespace iglu;

int main(){
   IgluDibujo v;
   v.plot_line(0,0,250,250);
```

```
   v.wait();
   return 0;
}
```

Primero, el header `iglu.h` permite acceder a los comandos gráficos disponibles en Iglu. Luego se crea una ventana de XWindows sobre la cual se ejecutarán los comandos gráficos posteriores. Por default, esta ventana se crea con un tamaño de 500 pixeles de ancho por 500 de alto, sin título (para otros constructores disponibles, consultar Sec. 3.1). El nombre de la ventana (en este caso `v`), puede ser, por supuesto, cualquier nombre de variable válido para C++.

A continuación vienen los comandos gráficos que se desean utilizar. En este caso, dibujamos una línea desde el pixel $(0, 0)$ al $(250, 250)$. La primera coordenada es el eje horizontal, la segunda el vertical, medidos desde el vértice inferior izquierdo. Es decir, ésta será una línea recta desde el vértice inferior izquierdo hasta el centro de la ventana `v`. (Más comandos gráficos en las Secs. 3.2, 3.3 y 3.6.)

Finalmente, la función `wait()` "congela" la imagen obtenida, hasta que el usuario presiona la tecla `Enter` o la barra de espaciado. En ese momento la ventana se cierra y, en este caso, el programa termina. (Para otras acciones realizables al término de un dibujo, ver Sec. 3.7).

El ejecutable, si el código fuente se encuentra en un archivo llamado `programa.cc`, se crea con la línea de comandos:

```
g++ -o programa programa.cc -I. -I<directorio_instalacion>/ \
   -L/usr/lib/X11 -L<directorio_instalacion> -liglu -lX11 -lm
```

done `<directorio_instalacion>` es el directorio en el cual se instaló Iglu.

Si está instalado el paquete `pkg-config`, basta con agregar `<directorio_instalacion>/lib/pkgconfig` a la variable de ambiente `PKG_CONFIG_PATH`. Por ejemplo, agregando en `~/.bashrc`, las líneas:

```
export PKG_CONFIG_PATH="${PKG_CONFIG_PATH}":<directorio_instalacion>/lib/pkgconfig
```

De este modo, la compilación se realiza con la siguiente línea:

```
 g++ -o programa ‘pkg-config --cflags iglu‘ programa.cc ‘pkg-config --libs iglu‘
```

En el ejemplo anterior se ha supuesto que el usuario desea construir un objeto `IgluDibujo`, a partir de los objetos gráficos disponibles en Iglu. Los comandos disponibles para `IgluDibujo` se describen en la Sec. 3.

Si el usuario desea representar conjuntos de datos en un gráfico, con ejes, con determinados estilos de síbolo y línea, y la posibilidad de agregar títulos, entonces un uso elemental sería:

```
#include "iglu.h"

using namespace std;
```

```
using namespace iglu;

int main(){
  IgluGrafico g;
  double x[3] = {.2,.5,.8}, y[3] = {.3,.1,.9};
  g.plot(x,y,3);
  g.wait();
}
```

En este caso se han creado dos vectores, uno de abscisas `x`, y otro de ordenadas `y`. Luego se grafican. Los comandos disponibles para `IgluGrafico` se describen en la Sec. 4.

3. Uso de IgluDibujo

3.1. Constructores

Los siguientes son los constructores posibles, ejemplificados con la creación de una ventana `v`:

`IgluDibujo v;`
: Crea una ventana de 500 pixeles de ancho por 500 pixeles de alto, sin título.

`IgluDibujo v(ancho,alto);`
: Crea una ventana de `ancho` pixeles de ancho, y `alto` pixeles de alto, sin título.

`IgluDibujo v("nombre");`
: Crea una ventana de 500 pixeles de ancho por 500 pixeles de alto, con título `"nombre"`.

`IgluDibujo v(ancho,alto,"nombre");`
: Crea una ventana de `ancho` pixeles de ancho, `alto` pixeles de alto, y con título `"nombre"`.

En todos estos casos, `ancho` y `alto` deben ser números enteros positivos, y `"nombre"` es un arreglo de caracteres o un objeto tipo `string`.

3.2. Objetos gráficos

Los siguientes son los objetos gráficos disponibles en Iglu. En los ejemplos se ha supuesto que se ha creado una ventana `v` con alguno de los constructores de la Sec. 3.1.

3.2.1. Puntos

Existen tres maneras de dibujar puntos en Iglu.

1. Un punto individual, en la posición (x, y):

```
v.plot_point(x,y);
```

2. N puntos, con coordenadas (x_i, y_i), contenidas en arreglos `x[N]` e `y[N]`:

   ```
   v.plot_point(x,y,N);
   ```

   `x` e `y` pueden ser arreglos estáticos o dinámicos.

3. N puntos, con coordenadas (x_i, y_i), contenidas en vectores de la STL (Standard Template Library) x e y (¡de la misma longitud!):

   ```
   v.plot_point(x,y);
   ```

Por ejemplo, los cinco programas siguientes producirán el mismo dibujo: tres puntos en las posiciones $(50, 30)$, $(100, 70)$ y $(300, 100)$:

a) Con puntos individuales:

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
   IgluDibujo v;
   v.plot_point(50,30);
   v.plot_point(100,70);
   v.plot_point(300,100);
   v.wait();
   return 0;
}
```

b) Con arreglos estáticos (declarados e inicializados simultáneamente):

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
   IgluDibujo v;
   int x[3] = {50,100,300};
   int y[3] = {30,70,100};
   v.plot_point(x,y,3);
   v.wait();
   return 0;
}
```

c) Con arreglos estáticos (declarados e inicializados no simultáneamente):

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
   IgluDibujo v;
   int x[3], y[3];
   x[0] = 50; x[1] = 100; x[2] = 300;
   y[0] = 30; y[1] = 70; y[2] = 100;
   v.plot_point(x,y,3);
   v.wait();
   return 0;
}
```

d) Con arreglos dinámicos:

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
   IgluDibujo v;
   int * x = new int[3];
   int * y = new int[3];
   x[0] = 50; x[1] = 100; x[2] = 300;
   y[0] = 30; y[1] = 70; y[2] = 100;
   v.plot_point(x,y,3);
   v.wait();
   delete [] x;
   delete [] y;
   return 0;
}
```

e) Con vectores de la STL:

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
   IgluDibujo v;
   vector<int> x(3),y(3);
   x[0] = 50; x[1] = 100; x[2] = 300;
   y[0] = 30; y[1] = 70; y[2] = 100;
```

```
    v.plot_point(x,y);
    v.wait();
    return 0;
}
```

3.2.2. Líneas

Al igual que con los puntos (Sec. 3.2.1), existen tres modos de dibujar líneas:

1. Una línea desde el punto $(x1, y1)$ al punto $(x2, y2)$:

   ```
   v.plot_line(x1,y1,x2,y2);
   ```

2. Una línea formada por N puntos, con coordenadas (x_i, y_i), contenidas en arreglos `x[N]`, `y[N]` (estáticos o dinámicos):

   ```
   v.plot_line(x,y,N);
   ```

3. Una línea formada por N puntos, con coordenadas (x_i, y_i), contenidas en vectores `x` e `y` de la Standard Template Library (STL) (¡de la misma longitud!):

   ```
   v.plot_line(x,y);
   ```

Por ejemplo, el siguiente código crea dos rectas uniendo los puntos $(50, 30)$, $(100, 70)$ y $(300, 100)$:

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
   IgluDibujo v;
   v.plot_line(50,30,100,70);
   v.plot_line(100,70,300,100);
   v.wait();
   return 0;
}
```

En cuanto al uso de `plot_line` con arreglos o con vectores de la STL, basta modificar los ejemplos b)–e) de la Sec. 3.2.1, cambiando `plot_point` por `plot_line`. En todos los casos se obtendrá una línea conectando los puntos $(50, 30)$, $(100, 70)$ y $(300, 100)$.

3.2.3. Cuadrados

Los siguientes comandos dibujan un cuadrado con centro en el punto (x, y) y arista a, ya sea sólo su borde (`plot_square`) o lleno (`plot_square_filled`):

```
v.plot_square(x,y,a);
v.plot_filled_square(x,y,a);
```

3.2.4. Rectángulos

Los siguientes comandos dibujan rectángulos con vértice inferior izquierdo en $(x1, y1)$, y vértice superior derecho en $(x2, y2)$, ya sea vacíos o llenos:

```
v.plot_rectangle(x1,y1,x2,y2);
v.plot_filled_rectangle(x1,y1,x2,y2);
```

3.2.5. Círculos

Los siguientes comandos dibujan círculos (vacíos o llenos) con centro en (x, y) y radio r:

```
v.plot_circle(x,y,r);
v.plot_filled_circle(x,y,r);
```

3.2.6. Arcos de círculo

Los siguientes comandos dibujan arcos de círculo (vacíos o llenos) con centro en (x, y), radio r, comenzando en el ángulo θ_1 y terminando en el ángulo θ_2 (ángulos medidos en grados):

```
v.plot_arc_circle(x,y,r,theta_1,theta_2);
v.plot_filled_arc_circle(x,y,r,theta_1,theta_2);
```

3.2.7. Elipses

Los siguientes comandos dibujan elipses (vacías o llenas) con centro en (x, y), semieje horizontal r_x y semieje vertical r_y:

```
v.plot_ellipse(x,y,r_x,r_y);
v.plot_filled_ellipse(x,y,r_x,r_y);
```

3.2.8. Arcos de elipse

Los siguientes comandos dibujan arcos de elipse (vacías o llenas) con centro en (x, y), semieje horizontal r_x y semieje vertical r_y,comenzando en el ángulo θ_1 y terminando en el ángulo θ_2 (ángulos medidos en grados):

```
v.plot_arc_ellipse(x,y,r_x,r_y,theta_1,theta_2);
v.plot_filled_arc_ellipse(x,y,r_x,r_y,theta_1,theta_2);
```

3.2.9. Texto

El siguiente comando pone un texto `texto` (que puede ser un arreglo de caracteres o un objeto `string` de la STL), con el extremo izquierdo del texto en las coordenadas (x, y):

```
v.plot_text(x,y,texto);
```

Para un texto centrado respecto a las coordenadas (x, y):

```
v.plot_centered_text(x,y,texto);
```

3.3. Color

Todos los objetos gráficos anteriores son dibujados por defecto en color negro. Esto se puede cambiar con la función `color_set`:

```
v.color_set(r,g,b);
```

cambia el color de todos los objetos gráficos siguientes al color compuesto por una proporción r de rojo (*red*), g de verde (*green*) y b de azul (*blue*). r, g y b deben ser números reales (`double`) en el intervalo $[0, 1]$. (Esto corresponde a dar el código RGB del color.)

Por ejemplo, el siguiente código:

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
   IgluDibujo v;
   v.color_set(1,0,0);
   v.plot_line(0,0,200,200);
   v.color_set(1,0,1);
   v.plot_line(200,200,300,400);
   v.plot_circle(250,250,20);
   v.wait();
   return 0;
}
```

dibuja una línea roja entre los puntos $(0, 0)$ y $(200, 200)$, luego una línea violeta entre $(200, 200)$ y $(300, 400)$, y un círculo —también violeta— de radio 20 centrado en $(250, 250)$ (Fig. 1).

Un ejemplo más completo, mostrando todos los objetos gráficos disponibles y el uso del color, se obtiene con el siguiente código:

```
/*   ejemplo_objetos.cc */

#include "iglu.h"
```

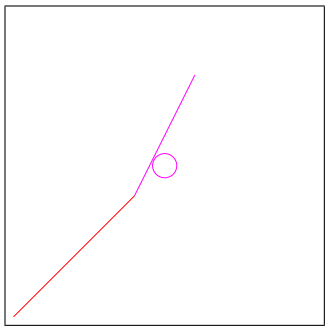

Figura 1: Ejemplo sencillo de uso del color.

```
using namespace std;
using namespace iglu;
int main(){
  IgluDibujo v("Objetos Graficos");
  v.plot_point(150,350);
  v.color_set(1,0,0);
  v.plot_point(200,400);
  v.color_set(0,0,1);
  v.plot_line(0,0,300,400);
  v.color_set(0,0,.9);
  v.plot_line(0,0,300,300);
  v.color_set(0,1,0);
  v.plot_rectangle(100,100,400,300);
  v.color_set(1,0,1);
  v.plot_arc_circle(300,400,50,0,180);
  v.plot_circle(300,300,25);
  v.color_set(0,1,1);
  v.plot_filled_arc_circle(400,100,50,0,30);
  v.plot_filled_circle(450,450,20);
  v.color_set(.5,.5,.5);
  v.plot_filled_rectangle(50,400,100,450);
  v.color_set(.5,0,.5);
  v.plot_square(250,50,10);
  v.plot_filled_square(250,70,20);
  v.wait();
}
```

El resultado se aprecia en la Fig. 2.

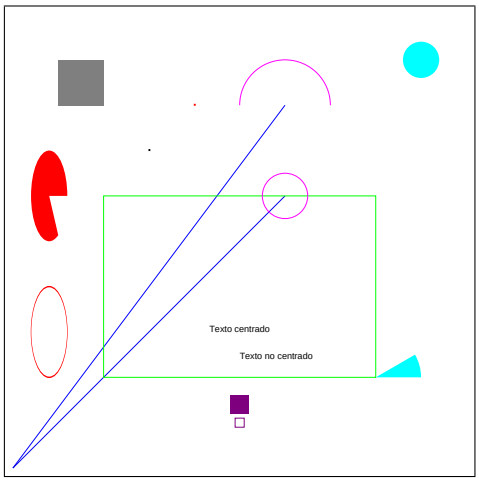

Figura 2: Un ejemplo completo de objetos y colores.

3.3.1. Variables de color

También es posible definir variables de C++ que representen un color:

```
IgluColor c = IgluColor(r,g,b);
```

define una variable `c` que corresponde a un color definido por sus valores `r`, `g` y `b`. En adelante, la variable `c` se puede usar para cambiar el color del dibujo. Por ejemplo, los códigos

```
v.color_set(r,g,b);
```

e

```
IgluColor c = IgluColor(r,g,b);
v.color_set(c);
```

son equivalentes.

3.4. Máscara rectangular

La función

```
clip_rectangle(xmin,ymin,xmax,ymax);
```

permite definir una región rectangular tal que todos los comandos gráficos sucesivos tendrán efecto visible sólo dentro de dicha región.

La figura 3 muestra un dibujo simple con y sin un `clip_rectangle` previo. El código para la primera figura es:

```
IgluDibujo g;
g.plot_line(0,0,500,500);
g.plot_line(500,0,0,500);
```

y para la segunda:

```
g.clip_rectangle(200,200,400,400);
g.plot_line(0,0,500,500);
g.plot_line(500,0,0,500);
```

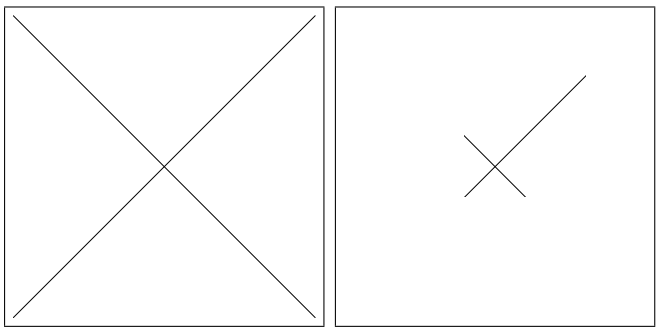

Figura 3: Efecto de una máscara rectangular.

3.5. Capas

Es posible distribuir los objetos gráficos dibujados en *capas*. Por defecto, un dibujo en Iglu tiene una sola capa, y los objetos aparecen en el dibujo en el orden en que fueron dibujados. En consecuencia, si dos objetos ocupan la misma region del dibujo, un objeto creado primero es ocultado por uno creado posteriormente. La figura 4 fue generada con el código:

```
IgluDibujo g;
g.color_set(0,1,0);
g.plot_filled_rectangle(200,200,400,400);
g.color_set(0,0,0);
g.plot_line(0,0,500,500);
g.color_set(0,0,1);
g.plot_filled_rectangle(10,10,120,120);
g.color_set(0,1,0);
g.plot_filled_rectangle(100,100,150,150);
```

Primero se dibujó el cuadrado verde grande, luego la línea negra, el cuadrado azul, y el cuadrado verde pequeño. Los objetos más recientes quedan por encima de los anteriores.

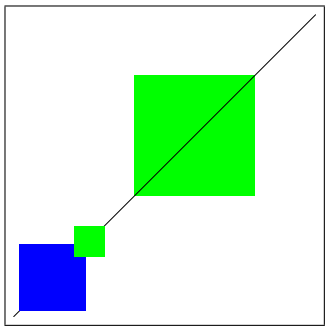

Figura 4: Dibujo con una capa. Los objetos más recientes ocultan a los anteriores.

Cuando se utilizan capas, en cambio, las capas inferiores son dibujadas antes que las superiores. Cada nuevo objeto puede ser ubicado en cualquiera de las capas existentes, y por tanto, distribuyéndolos en capas distintas, un objeto dibujado posteriormente puede quedar por debajo de uno anterior. De este modo es posible tener un control fino sobre la distribución de los objetos en el dibujo.

Iglu dispone de tres comandos para el manejo de capas. Si n es un número entero mayor o igual que 1:

`set_layers(n);`
: Declara que el dibujo actual contendrá n capas. Por defecto, el número de capas inicial es 1.

`use_layer(n);`
: Declara que todos los comandos gráficos sucesivos afectarán a la capa n. Por defecto, la capa utilizada inicialmente es la 1.

`clear_layer(n);`
: Borra los contenidos de la capa n.

La figura 5 es equivalente a la figura 4, pero usa el concepto de capas. El código es el siguiente:

```
IgluDibujo g;
g.set_layers(2);
g.color_set(0,1,0);
g.plot_filled_rectangle(200,200,400,400);
g.color_set(0,0,0);
g.plot_line(0,0,500,500);
g.color_set(0,0,1);
g.plot_filled_rectangle(10,10,120,120);
g.use_layer(2);
g.color_set(0,1,0);
```

```
g.plot_filled_rectangle(100,100,150,150);
g.use_layer(1);
g.color_set(0,0,1);
g.plot_filled_rectangle(140,140,170,170);
```

Inicialmente se indica que se usarán dos capas. Los primeros dos objetos gráficos son los mismos del ejemplo anterior: un cuadrado verde grande, una línea negra y un cuadrado azul. Como no se ha usado un `use_layer`, se utiliza la capa 1. Observar que dentro de una capa, se sigue usando la regla del ejemplo anterior: objetos creados después siempre aparecen encima de objetos creados antes. Luego se crea un cuadrado verde pequeño en la capa 2. Hasta este punto, el dibujo es igual al ejemplo anterior, porque la capa 2 es dibujada después de la capa 1, y el cuadrado verde pequeño aparece por encima del cuadrado azul. Finalmente se dibuja un cuadrado azul pequeño en la capa 1, de modo que, aunque fue creado después, aparece por debajo del cuadrado verde pequeño, pues está en una capa inferior.

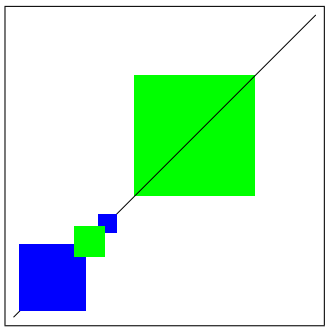

Figura 5: Dibujo con dos capas. El cuadrado verde pequeño está en la capa 2, y el resto de los objetos en la capa 1.

3.6. Cambio de coordenadas

Hasta el momento, todos los dibujos han sido hechos usando coordenadas enteras, con el origen en el vértice inferior izquierdo de la ventana. Esto no es necesario, sin embargo. Por ejemplo,

```
IgluDibujo v;
```

creará una ventana de 500 por 500 pixeles de ancho. Si se desea que la dirección horizontal represente, en vez del intervalo $[0, 500]$, el intervalo $[x_{\min}, x_{\max}]$, y la dirección vertical, en vez del intervalo $[0, 500]$, el intervalo $[y_{\min}, y_{\max}]$, entonces basta escribir:

```
IgluDibujo v;
v.map_coordinates(xmin,xmax,ymin,ymax);
```

En lo sucesivo, todos los comandos gráficos se referirán al nuevo sistema de coordenadas.

El siguiente código modifica el sistema de coordenadas para representar el intervalo $(10, 30)$ tanto en dirección vertical como horizontal, dibujando un triángulo en el nuevo sistema de coordenadas:

```
#include "iglu.h"
using namespace std;
using namespace iglu;
int main(){
  IgluDibujo v;
  v.map_coordinates(10,30,10,30);
  v.plot_line(15,15,30,15);
  v.plot_line(30,15,15,30);
  v.plot_line(15,30,15,15);
  v.wait();
}
```

El resultado está en la Fig. 6.

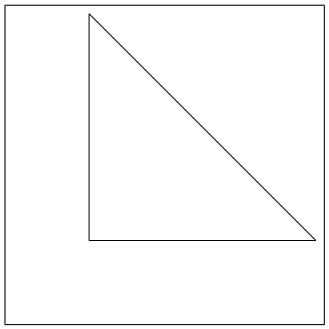

Figura 6: Un triángulo dibujado en un sistema de coordenadas modificado.

`map_coordinates` es particularmente útil para dibujar funciones matemáticas. El siguiente código dibuja el seno de x entre $x = 0$ y $x = 2\pi$ (Fig. 7):

```
/*   ejemplo_sen.cc */

#include "iglu.h"
#include <cmath>
using namespace std;
using namespace iglu;
int main(){
  IgluDibujo v("Funcion seno");
  const int N=100;
  double x[N], y[N];
```

```
  v.map_coordinates(0,2*M_PI,-1.2,1.2);
  double dx = 2*M_PI/(N-1);
  for (int i=0;i<N;i++){
    x[i] = i*dx;
    y[i] = sin(x[i]);
  }
  v.plot_line(x,y,N);
  v.wait();
}
```

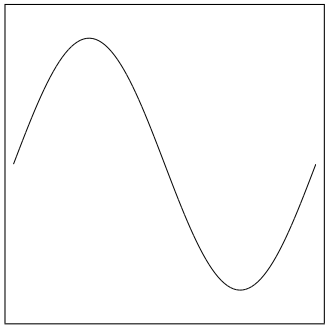

Figura 7: Función $\operatorname{sen} x$, para $x \in [0, 2\pi]$.

Es importante observar que, puesto que el uso de `map_coordinates` permite que los límites superior e inferior de las coordenadas horizontal y vertical sean arbitrarios, los argumentos de todas las funciones gráficas de la Sec. 3.2 no necesitan ser enteros, sino que pueden ser reales (`double`). Esto es cierto incluso si no ha habido un `map_coodinates` previo. Por ejemplo,

```
IgluDibujo v(100,100);
v.plot_line(0,0,50,50);
```

genera la misma línea que

```
IgluDibujo v(100,100);
v.plot_line(0.0,0.0,50.0,50.0);
```

Por su parte, es claro que el código

```
IgluDibujo v(100,100);
v.map_coordinates(0,1,0,1);
v.plot_line(.2,.2,.7,.7);
```

es válido, pues los 100 pixeles de ancho y alto deben representar ahora el intervalo $[0, 1]$, y por ende los argumentos de `plot_line` no son enteros. Pero también es aceptable

```
IgluDibujo v(100,100);
v.plot_line(0,0,40.3,50.8);
```

aun cuando no hay un `map_coordinates`. Por tanto, todos los ejemplos de la Sec. 3.2 se pueden repetir cambiando las coordenadas tipo `int` por coordenadas tipo `double`. [Los únicos que no pueden cambiar de `int` a `double` son, por supuesto, los argumentos numéricos de los constructores (Sec. 3.1), pues se refieren al tamaño en pixeles de la ventana.]

Nota:
`map_coordinates()` sólo puede utilizarse antes de dibujar el primer objeto gráfico. Si se intenta cambiar el mapeo de coordenadas después, Iglu envía un mensaje de advertencia al standard output, e ignora las nuevas coordenadas.

3.6.1. Arcos de círculo simulados

El uso de `map_coordinates()` introduce un nuevo problema. Consideremos el siguiente código:

```
IgluDibujo d;
d.map_coordinates(0,1,0,1);
d.plot_circle(.5,.5,.2);
```

Luego de un `map_coordinates()`, los argumentos entregados a todas las funciones de la Sec. 3.2 se refieren a las nuevas coordenadas. Por ejemplo, `plot_circle(.5,.5,.2)` dibujará un círculo en el centro de la ventana. Puesto que se usó el constructor default, la ventana creada es cuadrada (500 pixeles de alto y ancho), y por tanto el radio es igual a 2/10 del ancho o el alto.

¿Pero qué ocurre si se usa otro constructor, y la ventana ya no es cuadrada? Si se insiste en el mismo `map_coordinates`, la escala para coordenadas horizontales ya no es la misma que para coordenadas verticales, y el resultado será una elipse. Esto puede ser el comportamiento deseado a veces, pero en otras no. Debido a eso, se han implementado funciones que dibujar arcos de círculo "simulados", que fuerzan a Iglu a dibujar un círculo aun cuando las escalas verticales y horizontales no sean iguales.

La sintaxis es la siguiente, si se ha creado un objeto `IgluDibujo d`:

```
d.plot_fake_arc_circle(x,y,r,theta_1,theta_2,"eje");
d.plot_fake_circle(x,y,r,"eje");
d.plot_fake_filled_arc_circle(x,y,r,theta_1,theta_2,"eje");
d.plot_fake_filled_circle(x,y,r,"eje");
```

Estas funciones aceptan los mismos argumentos que los arcos de círculo usuales (Secs. 3.2.5, 3.2.6), y un argumento adicional, que es una cadena de texto que puede ser sólo `"x"` o `"y"`. Todas estas funciones dibujan en realidad arcos de *elipse*, donde el argumento `r` es el semieje en la dirección horizontal (`"x"`) o vertical (`"y"`), según se indique en `"eje"`, y el otro semieje es calculado de modo que, considerando el mapeo de coordenadas actual, el resultado sea un arco de círculo de radio `r`.

Tomemos como ejemplo el siguiente código, cuyo resultado se aprecia en la figura 8:

```
/* ejemplo_circulos_simulados.cc */

#include "iglu.h"
using namespace std;
using namespace iglu;

int main(){
  IgluDibujo d(700,500);
  d.map_coordinates(0,1,0,1);

  d.plot_arc_circle(.3,.5,.1,0,180);
  d.plot_arc_circle(.7,.5,.1,0,180);

  d.color_set(1,0,0);
  d.plot_fake_arc_circle(.3,.5,.1,0,180,"x");
  d.plot_fake_arc_circle(.7,.5,.1,0,180,"y");

  d.color_set(0,0,0);
  d.plot_circle(.3,.2,.05);
  d.plot_circle(.7,.2,.05);

  d.color_set(1,0,0);
  d.plot_fake_circle(.3,.2,.05,"x");
  d.plot_fake_circle(.7,.2,.05,"y");

  d.wait();
  d.save("ejemplo_circulos_simulados.eps");
}
```

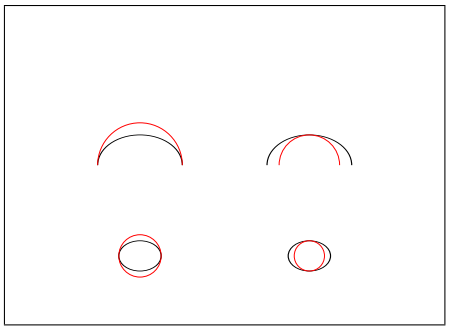

Figura 8: Arcos de círculo y círculos completos "simulados".

En primer lugar se ha creado un dibujo en una ventana de XWindows distinta de la default, no cuadrada (700 pixeles de ancho por 500 de alto), y con un mapeo de coordenadas

entre 0 y 1 en ambas direcciones. Por lo tanto, una unidad horizontal y una unidad vertical corresponden a distinto número de pixeles. Primero se dibuja un arco de círculo en el color default (negro) de radio 0.1 (tercer argumento de `plot_arc_circle`). En este caso, eso significa $700 \cdot 0.1 = 70$ pixeles de ancho, y $500 \cdot 0.1 = 50$ pixeles de alto, y el resultado es una elipse, como se aprecia en la figura 8 (panel superior). Luego, en rojo, se dibujan los mismos arcos, pero ahora como arcos de círculo simulados. Observar que los argumentos de `plot_fake_arc_circle` son los mismos que los de `plot_arc_circle`, más un parámetro adicional que primero es `"x"` y luego `"y"`. El resultado son dos arcos rojos, uno donde el valor `r=0.1` es interpretado como el radio del arco en la dirección horizontal (figura 8, panel superior izquierdo, centrado en `x=0.3`, `y=0.5`), y en el otro interpretado como el radio del arco en la direción vertical (figura 8, panel superior izquierdo, centrado en `x=0.7`, `y=0.5`). A pesar de que las escalas horizontal y vertical son distintas, los arcos rojos aparecen como circulares.

En el panel inferior de la figura 8 el procedimiento anterior se ha repetido, pero con círculos completos.

3.7. Acciones sobre la ventana: limpiar, congelar, pausa, grabar, reescalar/restaurar dimensiones

Una vez realizado un dibujo sobre una ventana `v`, se pueden realizar las siguientes acciones sobre ella:

`v.clean()`
: Limpia la ventana y borra de memoria todos los objetos gráficos contenidos en ella.

`v.wait()`
: Congela la ventana actual, hasta que el usuario presiona la tecla `Enter` o la barra de espaciado. En todos los ejemplos de las secciones anteriores hemos incluido un `wait()` al final, para examinar el resultado del dibujo antes de que se cierre.

`v.wait(t)`
: Congela la ventana actual durante t segundos. t puede ser cualquier número real positivo (`double`).[1] Esto permite realizar animaciones básicas. Por ejemplo, para dibujar una onda viajera sinusoidal podemos utilizar el siguiente código:

```
/*   ejemplo_pelicula.cc */

#include "iglu.h"
#include <cmath>
using namespace std;
using namespace iglu;
```

[1] Valores negativos son equivalentes a $t = 0$.

```
int main(){
  IgluDibujo v("Pelicula");
  const int N=100, Nt=200;
  double x[N], y[N];
  v.map_coordinates(0,2*M_PI,-1.2,1.2);
  double dx = 2*M_PI/(N-1), dt = 1, t=0;
  for (int j=0;j<Nt;j++){
    v.clean();
    t += dt;
    for (int i=0;i<N;i++){
      x[i] = i*dx;
      y[i] = sin(x[i]-.1*t);
    }
    v.plot_line(x,y,N);
    v.wait(.03);
  }
  v.wait();
}
```

En cada iteración se dibuja la función seno, desplazada una cierta fase proporcional a t ($-0.1t$ en este caso), y se espera 3 centésimas de segundo antes de dibujar el siguiente cuadro.

`v.save("archivo.eps")`

Graba el contenido actual de la ventana `v` en el archivo `archivo.eps`. El archivo es guardado en formato PostScript encapsulado, pero la extensión no es agregada automáticamente, de modo que es tarea del usuario indicarla explícitamente.

Si `archivo.eps` existe, es sobreescrito.

`v.save_safe("archivo.eps")`

Igual que `save()`, pero no sobreescribe el archivo si éste existe, sino que graba a un archivo `archivo.eps.1`. Si éste existe, cambia el nombre a `archivo.eps.2`, y así sucesivamente, hasta encontrar un nombre válido.

Reescalar/restaurar dimensiones

Con el mouse se pueden modificar el tama no de la ventana. El dibujo contenido en la ventana es reescaladado proporcionalmente, para ajustarse a las dimensiones actuales de la ventana.

Para recuperar las dimensiones iniciales basta pulsar la tecla `r`.

4. Uso de IgluGrafico

El propósito de un objeto tipo `IgluGrafico` es representar datos sobre ejes coordenados. Diversos parámetros de cómo tales datos son representados (estilo de línea, símbolos, etc.) son modificables por el usuario. Por ahora, Iglu soporta sólo gráficos bidimensionales.

Todos los comandos de la Sec. 3 (excepto los constructores, Sec. 3.1) están disponibles en un `IgluGrafico`. En esta sección revisamos los comandos exclusivos para gráficos.

Observación importante: `IgluGrafico` fue introducido recientemente en Iglu, y por lo tanto hay una larga lista de características a mejorar. Por ejemplo, el tamaño de los títulos, mayor variedad de estilos de línea, etc. Confiamos en poder mejorar estos y otros aspectos en futuras versiones.

4.1. Constructores

Un gráfico `g` se crea con el constructor default

```
IgluGrafico g;
```

Esto crea una ventana de 710 pixeles de ancho, 500 de alto, con ejes titulados `"x"` (abscisas) y `"y"` (ordenadas), ambos entre 0 y 1 (figura 9):

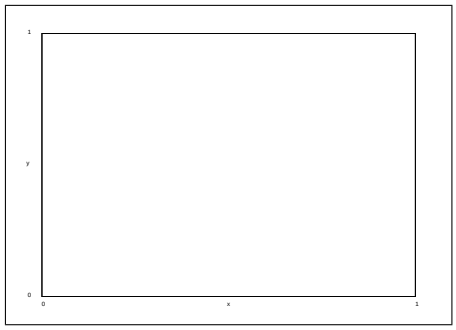

Figura 9: Gráfico creado por el constructor `IgluGrafico()`.

En las secciones que siguen suponemos que se ha creado un objeto `IgluGrafico g`.

4.2. Títulos

Se puede agregar/modificar un título al gráfico o a los ejes:

`g.title("titulo")`
Agrega/modifica un título al grafico.

`g.x_title(.ejе x")`
Agrega/modifica un título al eje x.

`g.y_title(.ejе y")`
Agrega/modifica un título al eje y.

4.3. Ejes

La función `axes()` controla el rango de los ejes. Existen tres llamadas posibles:

`g.axes(xmin,xmax,ymin,ymax)`
Modifica los ejes del gráfico de modo que abarcan el rango $[x_{\min}, x_{\max}]$ para las abscisas e $[y_{\min}, y_{\max}]$ para las ordenadas. Este comando puede darse en cualquier etapa del gráfico: antes de graficar la primera serie de datos, entre dos series de datos, o luego de graficar todos los datos.

`g.axes(AXES_AUTO)`
Los ejes son calculados automáticamente de modo que todos los puntos graficados sean visibles. Normalmente, uno sabe el rango de las abscisas, pero las ordenadas son calculadas de alguna forma, y no son conocidas a priori. Esta llamada permite despreocuparse de definir a priori los ejes, entregándole el trabajo al graficador.[2]

`g.axes(AXES_FIX)`
Lo contrario a lo anterior. Los ejes no son calculados automáticamente, y deben ser declarados explícitamente en la forma `axes(xmin,xmax,ymin,ymax)`. Es el comportamiento por defecto, y por lo tanto normalmente no se usa (de hecho, en los ejemplos a continuación no aparece).

4.4. Gráficos de línea

La función `plot()` grafica las ordenadas y versus las abscisas x, conectando los puntos por una línea sólida negra.

Si x e y están dados por arreglos estáticos o dinámicos de dimensión n, entonces la llamada a la función es:

```
g.plot(x,y,n);
```

Si son vectores de la STL,

```
g.plot(x,y);
```

Alternativamente, los puntos podrían ser representados por símbolos aislados, o por una superposición de símbolos y líneas. Esto corresponde a los *estilos de línea* y *estilos de punto* disponibles, y que pueden ser modificados. Al modificar uno de estos parámetros de estilo, todos los gráficos de línea siguientes son afectados, hasta la siguiente modificación de estilo. El estilo por defecto es una línea sólida negra. Las siguientes funciones modifican los estilos de un gráfico de línea:

[2] En la versión actual de Iglu, este uso de `axes()` normalmente no da resultados muy estéticos, pero lo presentamos porque puede ser útil de todos modos.

4.4.1. Líneas

`g.line_set(tipo_linea)`
El tipo de línea se cambia a uno dado por la variable `tipo_linea`, la cual puede ser:

`g.line_set(LINE_NONE)`
No se dibuja línea entre puntos consecutivos.

`g.line_set(LINE_SOLID)`
Se dibuja una línea sólida entre puntos consecutivos.

Valor por defecto: `LINE_SOLID`.

`g.line_color_set(r,g,b)`
Da el color de la línea de acuerdo a su código RGB. (Por defecto: $r = g = b = 0$, negro.)

4.4.2. Símbolos

`g.symbol_set(tipo_simbolo)`
El tipo de símbolo se cambia a uno dado por la variable `tipo_simbolo`, la cual puede ser:

`g.symbol_set(SYMBOL_NONE)`
No se dibuja símbolo sobre cada punto.

`g.symbol_set(SYMBOL_DOT)`
Cada punto es dibujado como un punto (pequeño).

`g.symbol_set(SYMBOL_CIRCLE)`
Cada punto es dibujado como un círculo abierto.

`g.symbol_set(SYMBOL_FILLED_CIRCLE)`
Cada punto es dibujado como un círculo lleno.

Valor por defecto: `SYMBOL_NONE`.

`g.symbol_color_set(r,g,b)`
Da el color del símbolo de acuerdo a su código RGB. (Por defecto: $r = g = b = 0$, negro.)

`g.symbol_size_factor_set(f)`
Cambia el tamaño del símbolo. El nuevo tamaño es el predefinido, multiplicado por el factor f, de modo que $f > 1$ aumenta y $f < 1$ disminuye el tamaño. f puede ser cualquier número real no negativo.

El siguiente código ilustra usos posibles de estos cambios de estilo:

```
/* ejemplo_estilos_linea.cc */

#include "iglu.h"
using namespace std;
using namespace iglu;

int main(){
  IgluGrafico g;
  g.axes(-.5,2,0,4);
  double x[3] = {-.2,.5,1.8}, y[3] = {.3,.1,3.9};
  double X[3] = {-.1,.25,1.4}, Y[3] = {.6,.2,3.1};
  double xx[3] = {0,.25,1.3}, yy[3] = {.2,.8,2.1};
  double XX[3] = {-.4,.2,1.8}, YY[3] = {.5,3,1};
  g.plot(x,y,3);
  g.line_set(LINE_NONE);
  g.symbol_set(SYMBOL_DOT);
  g.plot(X,Y,3);
  g.symbol_color_set(1,0,0);
  g.symbol_set(SYMBOL_CIRCLE);
  g.plot(xx,yy,3);
  g.line_color_set(0,0,1);
  g.line_set(LINE_SOLID);
  g.symbol_set(SYMBOL_FILLED_CIRCLE);
  g.symbol_size_factor_set(2);
  g.plot(XX,YY,3);
  g.wait();
  g.save("ejemplo_estilos_linea.eps");
}
```

El resultado se encuentra en la figura 10.

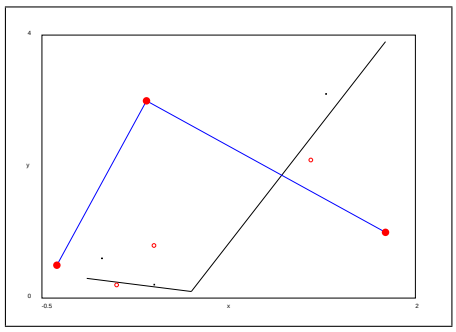

Figura 10: Distintas combinaciones de estilos de línea.

Primero se crean cuatro series de datos. Cada una será graficada con distintos estilos.

La primera (`x,y`) es graficada en el estilo default: puntos unidos por una línea sólida negra. La segunda (`X,Y`) es graficada sin línea, con un símbolo tipo `DOT`. Para la tercera serie (`xx,yy`), el símbolo se cambia a círculos abiertos (`CIRCLE`) de color rojo. Finalmente, antes de graficar la cuarta serie (`XX,YY`) se indica que se usará una línea sólida azul, y el tipo de símbolo se cambia a círculo lleno. Observar que como no se cambió el color del símbolo, sigue en vigencia el último cambio anterior, de modo que el resultado son círculos llenos rojos. Lo que sí se cambió en la última serie de datos fue el tamaño del símbolo, de modo que los círculos llenos tienen dos veces el radio de los círculos abiertos.

4.5. Gráficos de barras

También es posible presentar series de datos en forma de gráfico de barras. La función a usar es `bar()`.

Si x e y son arreglos de dimensión n que contienen las abscisas y ordenadas, respectivamente, entonces la llamada a función es

```
g.bar(x,y,n);
```

Si x e y son vectores de la STL, entonces la llamada es

```
g.bar(x,y);
```

Como para los gráficos de líneas, es posible modificar algunos parámetros de estilo asociados al gráfico de barras. El estilo por defecto es dibujar sólo el contorno de una barra, con línea sólida negra, sin separación entre barras contiguas. Las siguientes funciones modifican este comportamiento, y tienen efecto desde la siguiente serie de datos graficada, hasta una nueva modificación.

`g.bar_set(tipo_barra)`
: El tipo de barra se cambia a uno dado por la variable `tipo_barra`, la cual puede ser:

  `g.bar_set(BAR_NONE)`
  : No se dibujan barras.

  `g.bar_set(BAR_OPEN)`
  : Se dibujan los contornos de cada barra.

  `g.bar_set(BAR_FILLED)`
  : Se dibujan barras sólidas.

  Valor por defecto: `BAR_OPEN`.

`g.bar_color_set(r,g,b)`
: Da el color de la barra de acuerdo a su código RGB. (Por defecto, $r = g = b = 0$, negro.)

`g.bar_space_factor_set(f)`

Cambia la separación entre barras adyacentes. f debe ser un número real entre 0 y 1, donde $f = 0$ representa separación nula (no hay espacio entre barras adyacentes), y $f = 1$ representa separación igual a la distancia entre abscisas consecutivas (es decir, la barra colapsa a una línea vertical).[3]

Valor por defecto: $f = 0$.

El siguiente código presenta un ejemplo de gráfico de barras, con el uso de distintos estilos. El resultado es la figura 11.

```
/* ejemplo_barras.cc */

#include "iglu.h"
#include <cmath>
using namespace std;
using namespace iglu;

int main(){
  IgluGrafico g;
  int M=17, N=20;
  double dx = 1.3;
  g.axes(-M-1,-M+N*dx+1,-int(sqrt(-M+N*dx))-2,int(sqrt(-M+N*dx))+2);
  vector<double> x_none(2),y_none(2),x_red(2),y_red(2),
    x_filled(2),y_filled(2),x(N-6),y(N-6);
  for (int i=0;i<N;i++){
    double abscisa =-M+i*dx;
    double ordenada =(i%2)?-sqrt(i):sqrt(i);
    if (i<2){
      x_none[i] = abscisa;
      y_none[i] = ordenada;
    }
    else if (i<4){
      x_red[i-2] = abscisa;
      y_red[i-2] = ordenada;
    }
    else if (i<6){
      x_filled[i-4] = abscisa;
      y_filled[i-4] = ordenada;
    }
```

[3]La descripción anterior tiene sentido si las abscisas son equiespaciadas. IgluGrafico no hace ningún intento por verificar si lo son, y para calcular la separación entre barras se consideran sólo los primeros dos datos de la serie a graficar. Debido a ello, el uso de `bar_space_factor_set()` puede tener efectos inesperados si las abscisas no están equiespaciadas.

```
    else {
      x[i-6] = abscisa;
      y[i-6] = ordenada;
    }
  }
  g.line_set(LINE_NONE);
  g.symbol_set(SYMBOL_CIRCLE);
  g.plot(x_none,y_none);
  g.plot(x_red,y_red);
  g.plot(x_filled,y_filled);
  g.plot(x,y);
  g.line_set(LINE_SOLID);
  g.symbol_set(SYMBOL_NONE);


  g.bar_space_factor_set(.7);
  g.bar_set(BAR_NONE);
  g.bar(x_none,y_none);
  g.bar_set(BAR_OPEN);
  g.bar_color_set(1,0,0);
  g.bar(x_red,y_red);
  g.bar_set(BAR_FILLED);
  g.bar(x_filled,y_filled);
  g.bar_set(BAR_OPEN);
  g.bar_color_set(0,0,0);
  g.bar(x,y);
  g.wait();
  g.save("ejemplo_barras.eps");
}
```

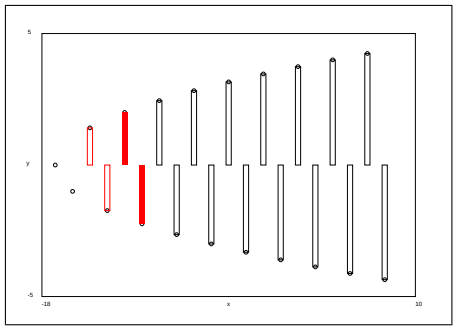

Figura 11: Ejemplo gráfico de barras, con distintos estilos.

La idea es graficar cuatro series de datos, que están sobre las curvas $y = \sqrt{x}$, $y = -\sqrt{x}$.

Primero se indica que las abscisas parten de $M = 17$ y que se graficarán $N = 20$ puntos, con abscisas separadas en $dx = 1.3$. Los ejes se definen en el rango adecuado para que todos los puntos sean visibles en el gráfico. A continuación se definen los cuatro vectores que contendrán cada serie de datos.

Ahora se grafican los datos. Primero, como referencia, se grafican todas las series de datos con círculos abiertos en cada punto. Después se indica que las barras tendrán un factor de espaciado 0.7, de modo que serán más bien delgadas. Y finalmente se despliega cada serie de datos con un estilo distinto: la primera (`x_none,y_none`), sin barras; la segunda (`x_red,y_red`), con barras abiertas rojas; la tercera (`x_filled,y_filled`), con barras sólidas rojas; la cuarta (`x,y`), con barras abiertas negras.