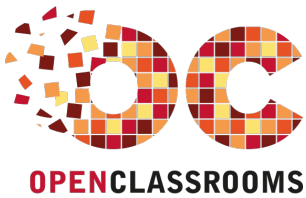
Erlang Graphical Drawer

Par rks`



www.openclassrooms.com

Sommaire

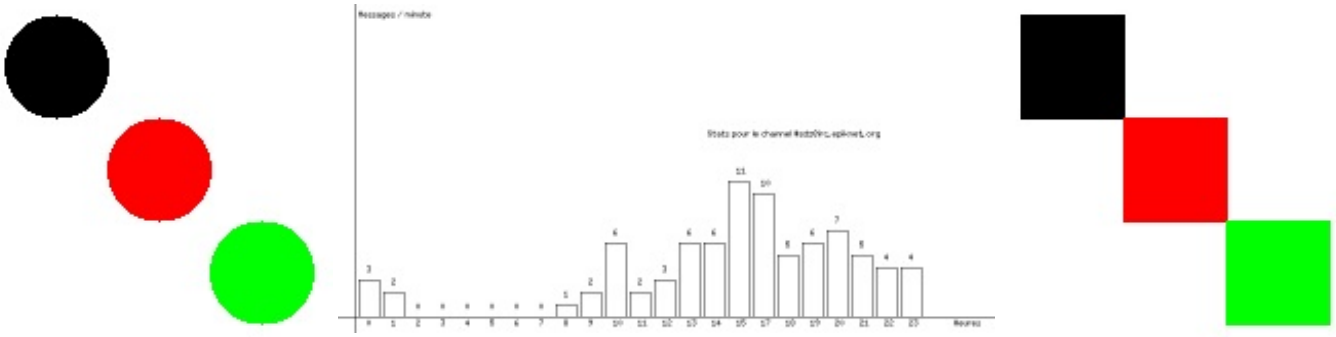
Erlang Graphical Drawer

Par rks`

Mise à jour : 01/01/1970

Dans ce tutoriel, je vais vous présenter le module egd : « erlang graphical drawer ». Un module faisant partie de la librairie standard d'Erlang et vous permettant - comme son nom l'indique - de tracer des graphiques. Il permet ensuite de les enregistrer au format PNG.

Avec ce module, vous pourrez aussi bien faire des dessins que de « vrais » graphiques représentant, par exemple, l'activité d'un channel IRC en fonction des périodes de la journée. Voyez plutôt :



Sommaire du tutoriel :

Présentation

Pour ceux qui se poseraient la question, je ne fais pas ce tuto [que] pour vous apprendre le fonctionnement d'un module relativement simple et d'ores et déjà détaillé dans la doc. L'intérêt de ce tuto est plutôt de vous mettre au courant de l'existence d'un tel module en Erlang, de vous le faire découvrir.
Ce module va donc vous permettre de réaliser des graphiques et des dessins *basiques* en Erlang.

Certains se demanderont probablement pourquoi faire ça en Erlang ; après tout, des graphiques, on peut très bien en faire en Python, en C, en OCaml, ... Il y a déjà le fait que pour ceux qui ne connaissent que l'Erlang, ce module est bien pratique, mais je doute que ces personnes ignorent l'existence de ce module ; ce n'est donc pas à eux que s'adresse ce tutoriel (du moins, je l'espère). L'autre point à envisager est de pouvoir utiliser ce module dans une application plus vaste qu'un simple programme faisant des graphiques. Mettons par exemple que vous ayez codé un bot (modulaire, cela va sans dire) en Erlang : il pourrait être intéressant de développer un module qui, à l'aide des informations réceptionnées par le module chargé de la gestion du protocole IRC, va pouvoir tracer des graphiques représentant l'activité d'un channel IRC en fonction du temps, le nombre de connectés à telle période de la journée, ...

Dans ce tutoriel, je vais donc vous présenter la plupart des fonctions de ce module, puis nous verrons comment l'utiliser comme module d'une application plus élaborée.

Prise en main du module

Les bases

Dans cette partie, je vais vous présenter les bases qui vont vous permettre d'utiliser le module egd dans vos applications.

Prérequis

Information

Il faut savoir qu'egd considère toujours que l'origine d'un graphique est le point supérieur gauche, il faudra vous souvenir de cela dans la suite de ce tutoriel. À part cela, il n'y a rien de particulier à connaître, on peut commencer ! 

Types de base

- Les points : représentés dans la doc par le type point(), ce sont en fait de simples couples contenant des entiers. Des entiers **uniquement**. Si vous utilisez des nombres décimaux, cela va faire échouer votre programme. En Erlang, pour transformer des nombres décimaux en entiers, vous pouvez utiliser deux fonctions :
  1. « `@spec trunc(Number) -> int()` » : cette fonction va tronquer le nombre que vous lui passerez et renverra l'entier ainsi obtenu ;
  2. « `@spec round(Number) -> int()` » : celle-ci va retourner l'arrondi du nombre passé en argument.
- Les couleurs : représentées par le type color() (étonnant, n'est-ce pas ? ) sont encore une fois des tuples, contenant cette fois un triplet d'entiers contenus entre 0 et 255, vous allez ainsi représenter vos couleurs avec la notation RGB.
- L'image est, elle, représentée par le type egd_image() (eh oui, cette fois y a un préfixe, c'est plus long à écrire ) et elle contient en fait deux valeurs, la hauteur et la largeur.

En fait, j'ai un peu simplifié les choses quand je vous ai dit que le type color() était un tuple contenant 3 entiers : le type est en fait le résultat de la fonction egd:color/1 où l'argument est bien le tuple présenté précédemment. De même pour le type de l'image, il faut en fait passer la largeur et la hauteur en arguments de la fonction egd:create/2 qui va, elle, retourner le type image qui va être utilisé par la suite.

Parmi les types que vous venez de découvrir, certains vont vous servir dans la quasi-totalité des fonctions que vous allez rencontrer : c'est le type egd_image() et le type color(). Mais plutôt que d'en parler, voyez par vous-mêmes.

Relier deux points

C'est probablement la plus simple des fonctions, mais ce n'est pas la moins utilisée. Nous allons ici découvrir une fonction permettant de relier deux points du graphique :
« `@spec egd:lines(Image::egd_image(), Point1::point(), Point2::point(), Couleur::point()) -> ok` »
Bon ok, je ne vous apprends rien, vous auriez pu lire la doc pour savoir ça  (et pour tout dire, ça n'aurait pas forcément été une mauvaise idée).

À ce stade, vous devez être capables de créer votre image, de définir les couleurs que vous allez utiliser et désormais tracer des segments, voyons voir ça :

Code : Erlang

```
-module(graph). % oui, le nom est très recherché :)
-compile(export_all). % très utile lorsque vous exportez toutes vos fonctions
                      % et que vous en avez vraiment une liste considérable (ok,
                      % ce n'est pas le cas ici, mais tant pis, j'avais la flemme).

-define(height, 170). % définir de telles macros peut rendre votre code _bien_ plus
-define(width, 170).  % lisible, notamment pour des personnes
```

```
    tierces.

    init() ->
        MyImage = egd:create(?width, ?height),
        Black = egd:color({0, 0, 0}),
        Red = egd:color({255, 0, 0}),
        Green = egd:color({0, 255, 0}),
        drawLines(MyImage, [Black, Red, Green]).

    getPoints(X, Y, Nb) ->
        MaxX = X + (50 * Nb),
        MaxY = Y + (50 * Nb),
        Xs = list:seq(X, MaxX, 50),
        Ys = list:seq(Y, MaxY, 50),
        lists:zip(Xs, Ys).

    drawLines(Img, ColorList) ->
        Nb = length(ColorList),
        PointList = getPoints(0, Nb, {10, 10}, []),
        List = lists:zip(PointList, ColorList),
        lists:foreach(fun({{P1, P2}, Color}) -> egd:lines(Img, P1, P2,
    Color) end, List).
```

Ce code va en fait tracer autant de segments que vous définirez de couleurs ; dans le cas présent, ça va faire deux segments. Voyons comment observer le résultat !

Tracer et enregistrer votre graphique

Pour ce faire, vous allez avoir besoin de deux fonctions :

1. « `@spec egd:render(Image::egd_image()) -> binary()` » qui va se charger de « tracer » votre graphique ;
2. « `@spec egd:save(Binary::binary(), Filename::string()) -> ok` » ; vous devrez lui passer le résultat de la fonction egd:render/1 et indiquer le nom de l'image que vous souhaitez enregistrer.

Vous pouvez désormais modifier le code précédent afin qu'il enregistre votre image une fois que vous aurez fini de tracer votre graphique. Voici la nouvelle fonction init/0 :

Code : Erlang

```
    init() ->
        MyImage = egd:create(?width, ?height),
        Black = egd:color({0, 0, 0}),
        Red = egd:color({255, 0, 0}),
        drawLines(MyImage, [Black, Red]),
        ToSave = egd:render(MyImage),
        egd:save(ToSave, "<votre_dossier>/images/tutoriel.png").
```

Voilà, il ne vous reste qu'à baver (ou pas ) devant votre superbe graphique. 

Maintenant que vous savez tracer des traits et enregistrer vos graphiques, on va pouvoir commencer à regarder plus sérieusement ce module, parce que bon, tracer des segments, ça va bien deux minutes, et même si on peut tout faire avec, ce n'est vraiment pas le truc le plus pratique, ni le plus rapide au monde. Heureusement, les développeurs d'egd ont pensé à vous, et ils ont rajouté quelques fonctions assez utiles pour tracer, entre autres, des formes géométriques.

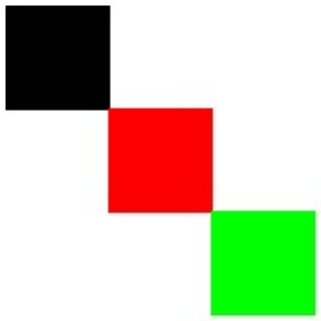
Des formes plus élaborées

1/ Des rectangles

Sans doute la forme la plus simple, vous pouvez tracer des rectangles simplement à l'aide des fonctions :

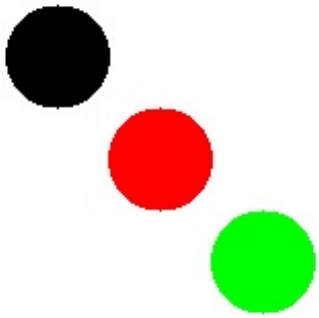
- « `@spec egd:rectangle(Image::egd_image(), Point1::point(), Point2::point(), Color::color()) -> ok` »
- « `@spec egd:filledRectangle(Image::egd_image(), Point1::point(), Point2::point(), Color::color()) -> ok` »

Qui vont respectivement vous créer des rectangles vides ou colorés. Point1 sera le point supérieur gauche du rectangle et Point2, le point inférieur droit. Bref, rien de bien compliqué. Vous pouvez essayer de modifier le code précédent pour qu'il vous trace des rectangles à la place des segments.



2/ Des ellipses

Vous ne disposez cette fois que d'une seule fonction :
« `@spec egd:filledEllipse(Image::egd_image(), Point1::point(), Point2::point(), Color::color()) -> ok` »
Les deux points sont ici les points supérieur gauche et inférieur droit du rectangle entourant l'ellipse. N'ayez crainte, vous ne verrez pas des rectangles autour de vos ellipses, c'est juste plus simple de se l'imaginer ainsi. Si vous voulez voir de quoi je parle, couplez cette fonction à la fonction egd:rectangle/4.



Avec les fonctions que l'on a vues dans les deux parties précédentes, vous devriez être en mesure de tracer bon nombre de graphiques, mais il vous faut savoir écrire du texte, et découvrir les quelques options avancées d'egd. Après quoi vous pourrez faire à peu près tout ce que vous voulez.

3/ Écrire du texte

La police

Pour écrire du texte, il va tout d'abord falloir définir la police que vous souhaitez utiliser ; les polices disponibles par défaut se trouvent dans le répertoire : `$ERLANG/lib/percept-$version/priv/Fonts`, où $ERLANG est le dossier où vous avez installé Erlang. Pour pouvoir sélectionner une police, il faut tout d'abord indiquer son adresse à l'aide de la fonction : filename:join/1 (cf. la doc pour plus d'informations). Voilà comment faire utiliser, par exemple, la police 6x11_latin1 :

Code : Erlang

```
Filename = filename:join([code:priv_dir(percept), "fonts",
"6x11_latin1.wingsfont"])
```

Une fois que vous avez l'adresse du fichier, il suffit juste de le charger à l'aide de egd_font:load/1 :

Code : Erlang

```
Font = egd_font:load(Filename)
```

Écrire

Une fois que vous avez défini votre police, il ne vous reste plus qu'à écrire votre texte. Pour cela, il vous suffit d'utiliser la fonction :
« `@spec egd:text(Image::egd_image(), Point::point(), Text::string(), Color::color()) -> ok` »
Où Point est le point supérieur gauche de la première lettre de votre texte.

Les options de render

Le type de l'image

Vous avez vu précédemment que la fonction egd:render/1 vous permettait d'enregistrer vos images au format png, cette fonction est en fait un lien vers egd:render/2, mais le 2e argument est directement mis à png. Si jamais vous ne voulez pas utiliser le png, egd:render/2 vous permet de choisir le type de votre image entre bitmap et... png. Les deux types disponibles sont donc :

- png
- raw_{bitmap}

Opaque ou... ?

Eh oui, vous pouvez aussi spécifier si le fond de votre image sera opaque ou non, cela à l'aide des options de egd:render/3 : le 3e argument est de type : `render_option()`, c'est-à-dire un des deux tuples suivants : 

1. `{render_engine, opaque}`
2. `{render_engine, alpha}`

Mise en application

Vous avez découvert dans la partie précédente les fonctions principales du module egd. Dans cette partie, nous allons mettre cela en application en développant un module pour bot IRC, qui va afficher les statistiques d'un chan donné en fonction des heures de la journée.

La base de notre module

Nous allons tout d'abord créer le module, et dans la fonction d'initialisation créer notre image ainsi que des axes pour notre graphique.

Code : Erlang

```
-module(graphs).
-export([init/0]).

-define(height, 400).
-define(width, 800).

init() ->
   Black = egd:color({0, 0, 0}),
   Img = egd:create(?width, ?height),
   % On trace les axes
   egd:line(Img, {0, (?height - 20)}, {?width, (?height - 20)},
Black), % (Ox)
   egd:line(Img, {20, 0}, {20, ?height}, Black), % (Oy)
   loop(Img).
```

Tout d'abord, quelques explications sur ce code. En effet, vous vous demandez peut-être pourquoi je définis la hauteur de l'axe des abscisses à `(?height - 20)` au lieu de 20, voire tout simplement 0. Eh bien, si vous vous souvenez, je vous ai dit au début de la partie précédente que pour egd, l'origine du repère était le point supérieur gauche de l'image ; or, on ne voit pas souvent des graphiques dont les données sont représentées de haut en bas, si ? On prend donc la hauteur de l'image, c'est-à-dire le point le plus bas, et on lui soustrait 20, pour que l'axe soit placé à 20 pixels de hauteur du bas de l'image.

La fonction loop/1

Cette fonction va nous servir à recevoir les requêtes du core de notre application et les interpréter pour tracer, ou effacer, des graphs. Je vais ici utiliser la même convention que dans mon tutoriel sur le remplacement à chaud **[#]**, libre à vous de la suivre ou non, je vous préviens juste, des fois que certains se posent des questions. Voyez plutôt :

Code : Erlang

```
loop(Image) ->
   receive
      {From, quit} ->
         io:format("Quitting module: graphs...~n");

      {From, render, [Filename]} ->
             ToSave = egd:render(Img),
```

```
                egd:save(ToSave, Filename),
                egd:destroy(Img),
                init();

        {From, trace, [List]} ->
            case drawGraph(Image, List) of
                ok -> From ! ok;
                {error, Why} ->
                    io:format("Error: ~p...~n", [Why]),
                    From ! {error, Why}
            end,
            loop(Img)
    end.
```

Regardons ce code de plus près : le premier message ne devrait pas vous poser de problème, du moins je l'espère ; le second est un peu plus intéressant : lorsque qu'on reçoit le message nous ordonnant d'enregistrer l'image à l'adresse Filename, on met en marche la procédure habituelle, à savoir : « render + save », puis on fait appel à la fonction destroy, fonction que vous ne connaissez pas encore. C'était la dernière fonction qu'il vous restait à découvrir, elle nous sert ici à détruire notre image, puisqu'une fois enregistrée, on ne va plus s'en servir. Ensuite, on fait appel à la fonction init/0 au lieu de relancer la boucle. L'image précédente ayant été détruite, il faut en recréer une nouvelle, et plutôt que de tout refaire soi-même, autant passer par la fonction init, qui est là pour ça.

Le troisième et dernier message filtré est celui dans lequel on reçoit l'ordre de tracer le graphique. Ici, l'argument List sera une liste de valeurs exploitable par la fonction drawGraph. Si vous vouliez améliorer, ou plutôt diversifier ce système, vous pourriez faire :

Code : Erlang

```
{From, trace, [Type, List]} ->
    case Type of
        lines -> ToSend = analyse(drawLines(Img, List));
        rects -> ToSend = analyse(drawRects(Img, List));
        %% ...
    end,
    From ! ToSend,
    loop(Img)
```

Avec comme fonction analyse :

Code : Erlang

```
analyse(ok) ->
    ok;
analyse({error, Why}) ->
    io:format("Error: ~p...~n", [Why]),
    {error, Why}.
```

L'architecture concernant la communication avec le reste de votre application est désormais fonctionnelle, il ne reste qu'à coder les fonctions, ou du moins une des fonctions pour ma part, vous servant à tracer des graphiques. Je vais ici vous montrer la fonction drawRects, libre à vous de faire les autres (ou non).

Code : Erlang

```
drawRects(Img, []) ->
    {error, "Liste vide"};
drawRects(Img, List) ->
    Font = egd_font:load(filename:join([code:priv_dir(percept),
"fonts", "6x11_latin1.wingsfont"])),
    Color = egd:color({0, 0, 0}),
```

```
      drawRects2(Img, Color, Font, 25, List).

drawRects(Img, Color, F, X, List) ->
   case List of
      [] -> ok;
      drawRects(Img, Color, F, X, LIst) ->
 case List of
    [] -> ok;
    [{Hour, Total, Nb}|Total] ->
       P1 = {X, (?height - 20)},
       Middle = round(Total/Nb),
       Height = Middle * 15,
       P2 = {(X+25), (?height - (Height+20))},
       egd:rectangle(Img, P1, P2, Color),
       egd:text(Img, {(X+10), (?height - (Height+40))}, F,
integer_to_list(Middle), Color),
       egd:text(Img, {(X+10), (?height - 20)}, F,
integer_to_list(Hour), Color),
       drawRects(Img, Color, F, (X+30), Tail)
 end.
```

Bon, il convient ici de parler un peu plus précisément du format des données. La liste contient des tuples qui renferment chacun trois entiers ; le premier représente l'heure de la journée à laquelle ont été postés les messages. Le second représente le nombre total de messages postés, et le troisième en combien de temps (en minutes) ils ont été postés. Grâce à ces deux chiffres, on va pouvoir établir un moyenne de messages / minute pour chaque heure de la journée. Vous remarquerez de plus que je ne me suis pas occupé de la création de cette liste ici, tout d'abord car cela n'a que peu d'intérêt et ensuite, parce qu'il vaut mieux faire cela dans un module distinct pour avoir un code plus facilement compréhensible.

Le code final

Le code précédent, ou plutôt les morceaux de code  , ne sont pas vraiment les meilleurs possibles, notamment le filtrage effectué dans la fonction loop, la fonction drawRects risque d'être redondante si l'on ajoute d'autres types de graphiques. Vous pourrez ainsi factoriser votre code si vous faites une fonction draw/3 qui prend en paramètre :

1. l'image ;
2. le type de graphique ;
3. les données à exploiter.

De plus, il existe une fonction très pratique, disponible de base, dans à peu près tous les langages fonctionnels : fold left ; en Erlang, cette fonction est nommée foldl (erl -man lists pour plus d'information). Voici le nouveau code une fois que ces modifications auront été effectuées :

Secret (cliquez pour afficher)

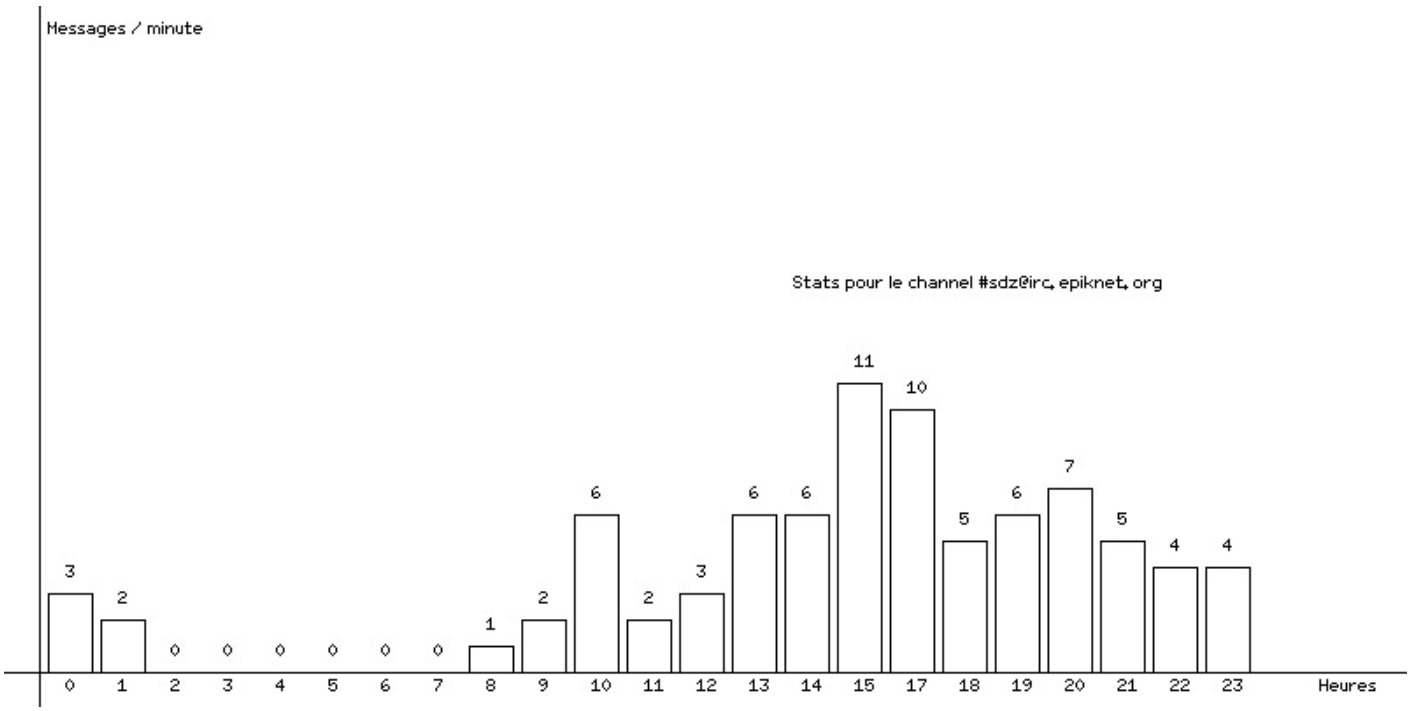
Code : Erlang

```
-module(graphs).
-export([init/0]).

-define(height, 400).
-define(width, 800).

init() ->
   Black = egd:color({0, 0, 0}),
   Img = egd:create(?width, ?height),
   % On trace les axes
   egd:line(Img, {0, (?height - 20)}, {?width, (?height - 20)},
Black), % (Ox)
   egd:line(Img, {20, 0}, {20, ?height}, Black), % (Oy)
   loop(Img).

loop(Image) ->
   receive
```

```
            {From, quit} ->
               io:format("Quitting module: graphs...~n");

            {From, render, [Filename]} ->
                    ToSave = egd:render(Img),
                    egd:save(ToSave, Filename),
                    egd:destroy(Img),
                    init();

            {From, trace, [Type, List]} ->
               case draw(Image, Type, List) of
                  ok -> From ! ok;
                  {error, Why} ->
                     io:format("Error: ~p...~n", [Why]),
                     From ! {error, Why}
               end,
               loop(Img)
      end.

   draw(_, _, []) ->
      {error, "Liste vide"};
   draw(Img, Type, List) ->
      Font = egd_font:load(filename:join([code:priv_dir(percept),
"fonts", "6x11_latin1.wingsfont"])),
      Color = egd:color({0, 0, 0}),
      case Type of
         rects -> lists:foldl(fun(A, Last) -> drawRect(Img, Color,
Font, Last, A) end, 25, List)
         %% etc.
      end.

   drawRect(Img, Color, F, X, {Hour, Total, Nb}) ->
      P1 = {X, (?height - 20)},
      Middle = round(Total/Nb),
      Height = Middle * 15,
      P2 = {(X+25), (?height - (Height+20))},
      egd:rectangle(Img, P1, P2, Color),
      egd:text(Img, {(X+10), (?height - (Height+40))}, F,
integer_to_list(Middle), Color),
      egd:text(Img, {(X+10), (?height - 20)}, F,
integer_to_list(Hour), Color),
      X + 30.
```

Vous pourrez remarquer que la fonction analyse/1 est ainsi devenue obsolète, aucune raison donc de la placer dans ce code. Le module graphique peut désormais être intégré à une de vos applications, quelle qu'elle soit. 

Voici un exemple de l'utilisation de ce module couplé avec un bot IRC :



Voilà, vous savez désormais à peu près tout ce qu'il faut savoir pour manier le module egd. J'espère que la lecture de ce tutoriel aura été agréable. Si jamais vous avez des problèmes, n'hésitez pas à passer sur le forum pour poser votre question !

Notes

Ce tuto a été écrit, non pas en zCode, mais en mdown, un petit langage de mise en forme agréable à utiliser (et qui peut produire du zCode), conçu par rz0 ; vous pourrez trouver une comparaison entre la syntaxe mdown et le zCode sur cette page.

Merci à Cygal, bluestorm et lastsseldon et tous les autres de #sdz pour leur relecture et leurs conseils.

Un tutoriel signé PHM