# Percept Application

**version 0.6**

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8 Document System.

# Contents

# Chapter 1

# Percept User's Guide

*Percept* is an akronym for *P*ercept - *er*lang *c*oncurr*e*ncy *p*rofiling *t*ool.

It is a tool to visualize application level concurrency and indentify concurrency bottlenecks.

## 1.1 Percept

Percept, or Percept - Erlang Concurrency Profiling Tool, utilizes trace informations and profiler events to form a picture of the processes's and ports runnability.

### 1.1.1 Introduction

Percept uses `erlang:trace/3` and `erlang:system_profile/2` to monitor events from process states. Such states are,

- waiting
- running
- runnable
- free
- exiting

There are some other states too, `suspended`, `hibernating`, and garbage collecting (`gc`). The only ignored state is `gc` and a process is considered to have its previous state through out the entire garbage collecting phase. The main reason for this, is that our model considers the `gc` as a third state neither active nor inactive.

A waiting or suspended process is considered an inactive process and a running or runnable process is considered an active process.

Events are collected and stored to a file. The file can be moved and analyzed on a different machine than the target machine.

Note, even if percept is not installed on your target machine, profiling can still be done via the module percept_profile [page ??] located in runtime_tools.

## 1.1.2 Getting started

### Profiling

There are a few ways to start the profiling of a specific code. The command `percept:profile/3` is a preferred way.

The command takes a filename for the data destination file as first argument, a callback entry-point as second argument and a list of specific profiler options, for instance `procs`, as third argument.

Let's say we have a module called example that initializes our profiling-test and let it run under some defined manner designed by ourself. The module needs a start function, let's call it go and it takes zero arguments. The start arguments would look like:

```
percept:profile("test.dat", {test, go, []}, [procs]).
```

For a semi-real example we start a tree of processes that does sorting of random numbers. In our model below we use a controller process that distributes work to different client processes.

```
-module(sorter).
-export([go/3,loop/0,main/4]).

go(I,N,M) ->
    spawn(?MODULE, main, [I,N,M,self()]),
    receive done -> ok end.

main(I,N,M,Parent) ->
    Pids = lists:foldl(
        fun(_,Ps) ->
            [ spawn(?MODULE,loop, []) | Ps]
        end, [], lists:seq(1,M)),

    lists:foreach(
        fun(_) ->
            send_work(N,Pids),
            gather(Pids)
        end, lists:seq(1,I)),

    lists:foreach(
        fun(Pid) ->
            Pid ! {self(), quit}
        end, Pids),

    gather(Pids), Parent ! done.

send_work(_,[]) -> ok;
send_work(N,[Pid|Pids]) ->
    Pid ! {self(),sort,N},
    send_work(round(N*1.2),Pids).

loop() ->
    receive
        {Pid, sort, N} -> dummy_sort(N),Pid ! {self(), done},loop();
        {Pid, quit} -> Pid ! {self(), done}
    end.
```

```
dummy_sort(N) -> lists:sort([ random:uniform(N) || _ <- lists:seq(1,N)]).

gather([]) -> ok;
gather([Pid|Pids]) -> receive {Pid, done} -> gather(Pids) end.
```

We can now start our test using percept:

```
Erlang (BEAM) emulator version 5.6 [async-threads:0] [kernel-poll:false]

Eshell V5.6  (abort with ^G)
1> percept:profile("test.dat", {sorter, go, [5, 2000, 15]}, [procs]).
Starting profiling.
ok
```

Percept sets up the trace and profiling facilities to listen for process specific events. It then stores these events to the `test.dat` file. The profiling will go on for the whole duration until `sorter:go/3` returns and the profiling has concluded.

### Data viewing

To analyze this file, use `percept:analyze("test.dat")`. We can do this on any machine with Percept installed. The command will parse the data file and insert all events in a RAM database, `percept_db`. The initial command will only prompt how many processes were involved in the profile.

```
2> percept:analyze("test.dat").
Parsing: "test.dat"
Parsed 428 entries in 3.81310e-2 s.
    17 created processes.
    0 opened ports.
ok
```

To view the data we start the web-server using `percept:start_webserver/1`. The command will return the hostname and the a port where we should direct our favorite web browser.

```
3> percept:start_webserver(8888).
{started,"durin",8888}
4>
```

Overview selection Now we can view our data. The database has its content from `percept:analyze/1` command and the webserver is started.

When we click on the `overview` button in the menu percept will generate a graph of the concurrency and send it to our web browser. In this view we get no details but rather the big picture. We can see if our processes behave in an inefficient manner. Dips in the graph represents low concurrency in the erlang system.

We can zoom in on different areas of the graph either using the mouse to select an area or by specifying min and max ranges in the edit boxes.

Note:
Measured time is presented in seconds if nothing else is stated.

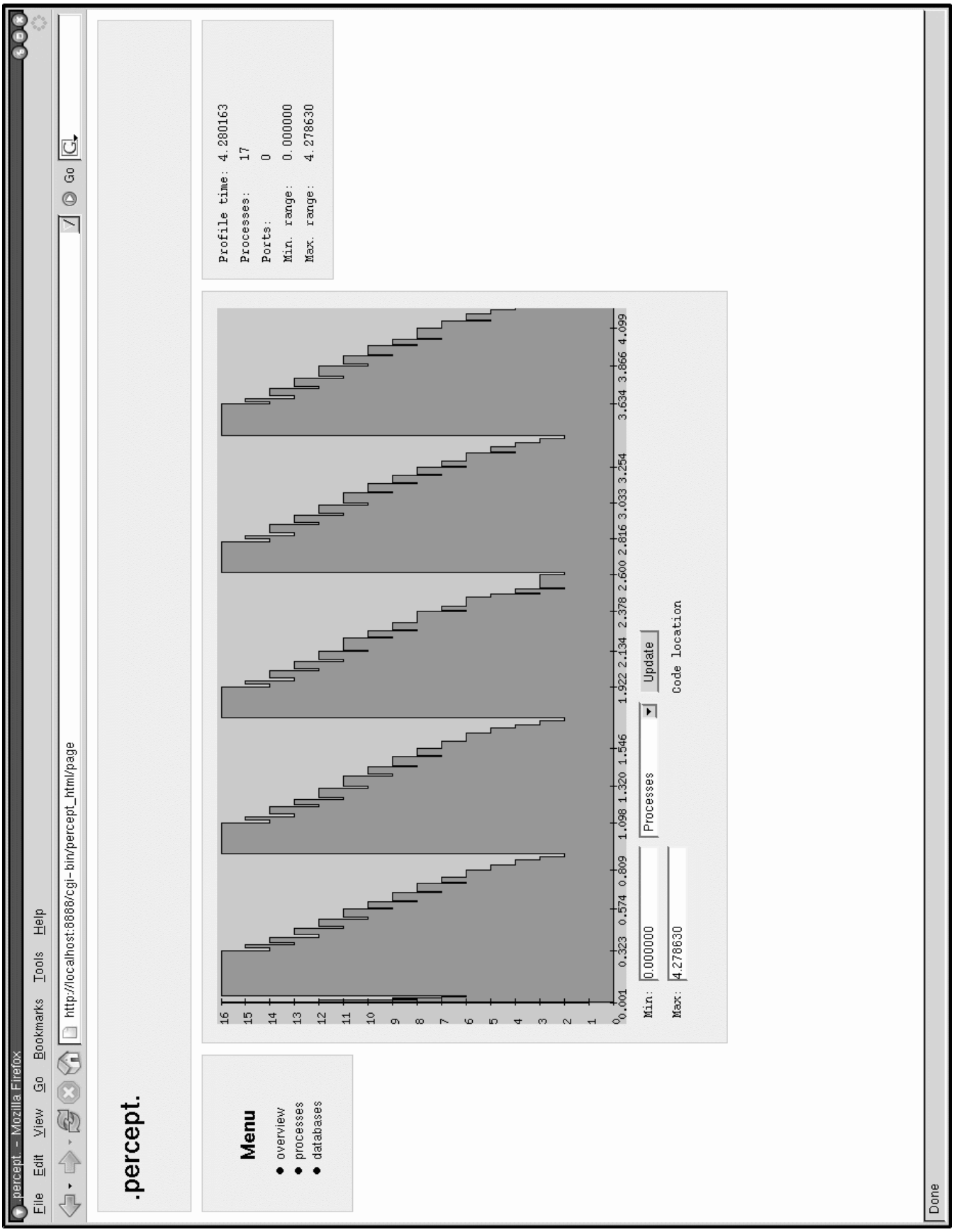


Figure 1.1: Overview selection

Processes selection To get a more detailed description we can select the process view by clicking the `processes` button in the menu.

The table shows process id's that are click-able and direct you to the process information page, a lifetime bar that presents a rough estimate in green color about when the process was alive during profiling, an entry-point, its registered name if it had one and the process's parent id.

We can select which processes we want to compare and then hit the `compare` button on the top right of the screen.

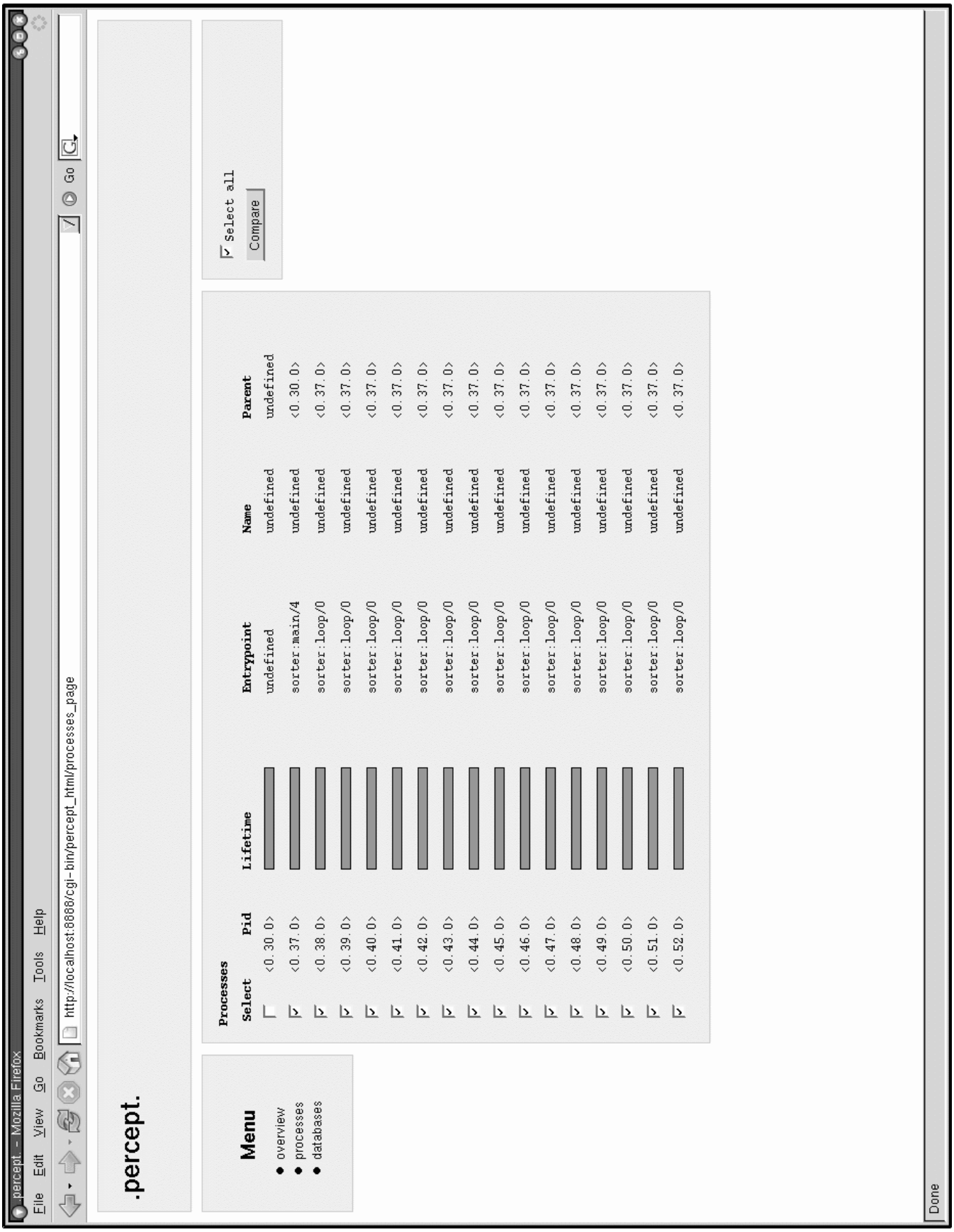


Figure 1.2: Processes selection

Compare selection The activity bar under the concurrency graph shows each process's runnability. The color green shows when a process is active (which is running or runnable) and the white color represents time when a process is inactive (waiting in a receive or is suspended).

To inspect a certain process click on the process id button, this will direct you to a process information page for that specific process.

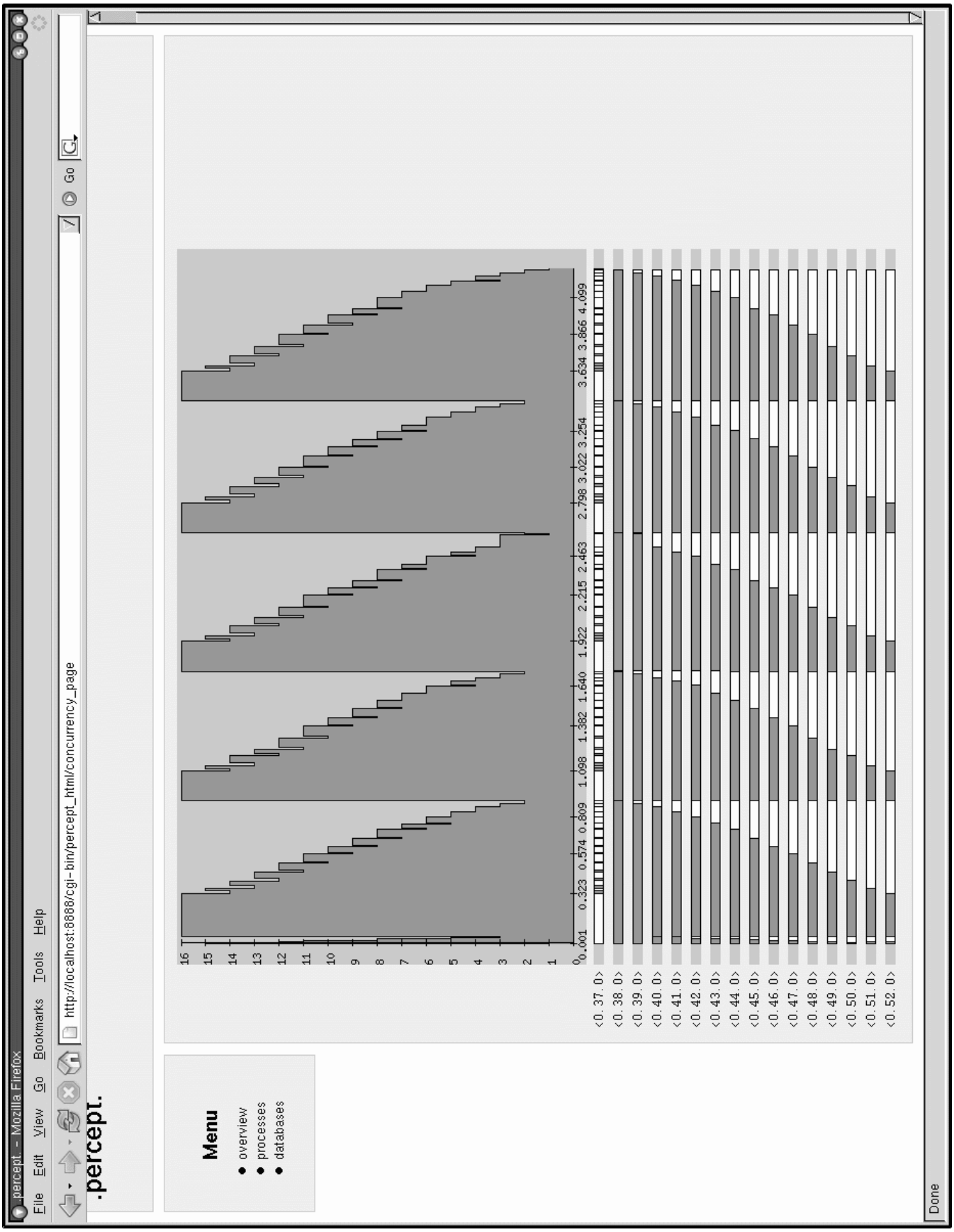


Figure 1.3: Processes compare selection

Process information selection Here we can some general information for the process. Parent and children processes, spawn and exit times, entry-point and start arguments.

We can also see the process' inactive times. How many times it has been waiting, statistical information and most importantly in which function.

The time percentages presented in process information are of time spent in waiting, not total run time.

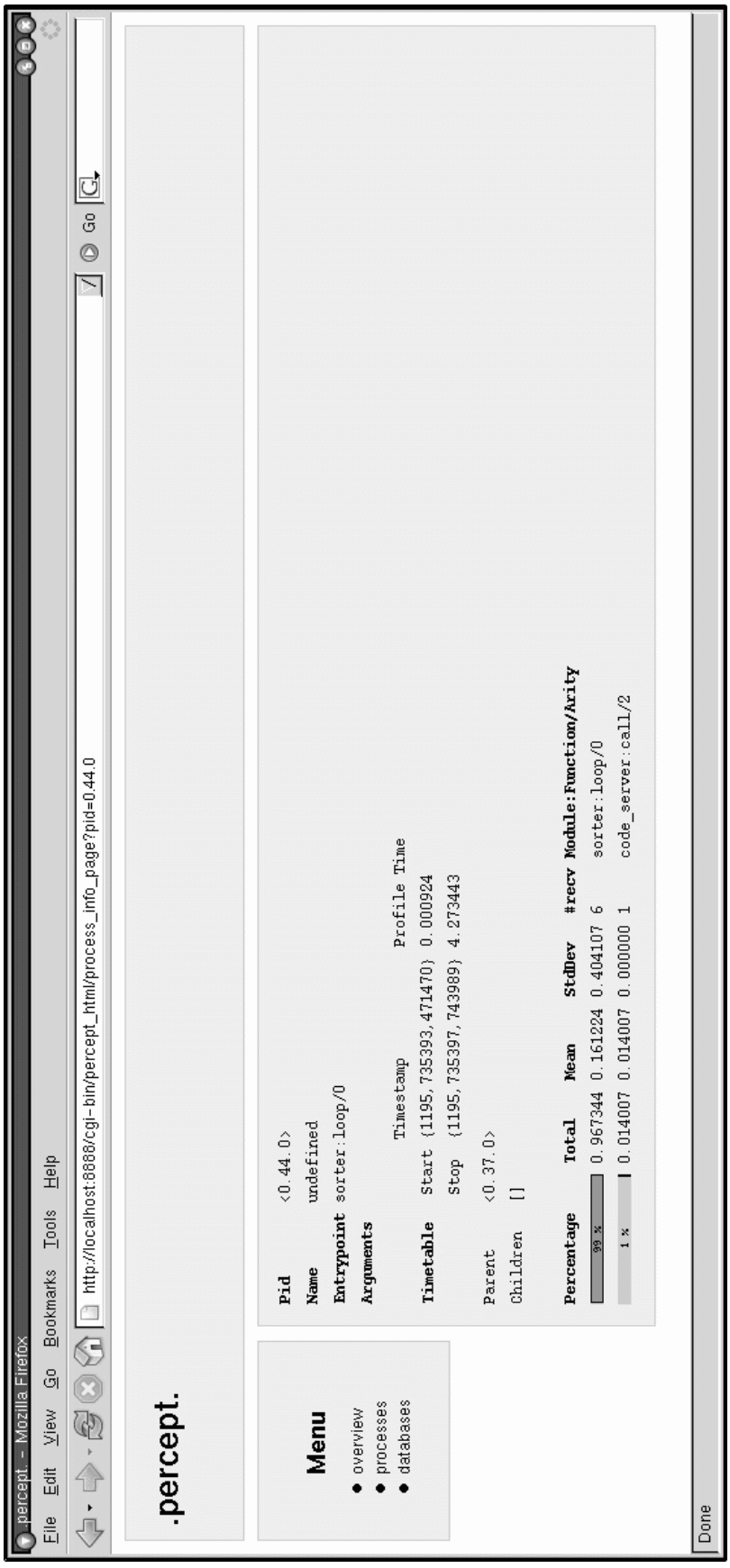


Figure 1.4: Process information selection

## 1.2 egd

### 1.2.1 Introduction

The egd module is an interface for the gd library and is used by Percept to generate dynamic graphs to its web pages.

Please see libgd.org[1] for futher details about the gd library.

The foremost purpose for this module is to enable users to generate images from erlang code and/or datasets and to send these images to either files or web servers.

### 1.2.2 File example

Drawing examples:

```
-module(img).

-export([do/0]).

do() ->
    Im = egd:create(200,200),
    White = egd:color(Im, white),
    Green = egd:color(Im, green),
    Blue = egd:color(Im, blue),
    Red = egd:color(Im, red),
    Black = egd:color(Im, black),
    Yellow = egd:color(Im, {255,255,0}),

    egd:fill(Im, {10,10}, White),

    % Line and fillRectangle

    egd:filledRectangle(Im, {20,20}, {180,180}, Red),
    egd:line(Im, {0,0}, {200,200}, Black),

    egd:save(egd:image(Im, gif), "/home/egil/test1.gif"),

    % Fill border

    egd:fill(Im, {15,5}, Blue),
    egd:fill(Im, {115,100}, Green),

    egd:save(egd:image(Im, gif), "/home/egil/test2.gif"),

    % Pacman filledArc
    egd:filledArc(Im, {100,100}, 100,100, 28,332, Yellow, [arc]),
    egd:filledArc(Im, {100,100}, 100,100, 28,332, Black, [arc, no_fill, edged]),

    egd:save(egd:image(Im, gif), "/home/egil/test3.gif"),

    % Text
```

[1] URL: http://www.libgd.org

```
{W,H} = egd:fontSize(Im, giant),
String = "egd says hello!",
Length = length(String),
egd:text(Im, giant, {round(100 - W*Length/2), 200 - H - 5}, String, Black),
egd:save(egd:image(Im, gif), "/home/egil/test4.gif"),

egd:destroy(Im).
```

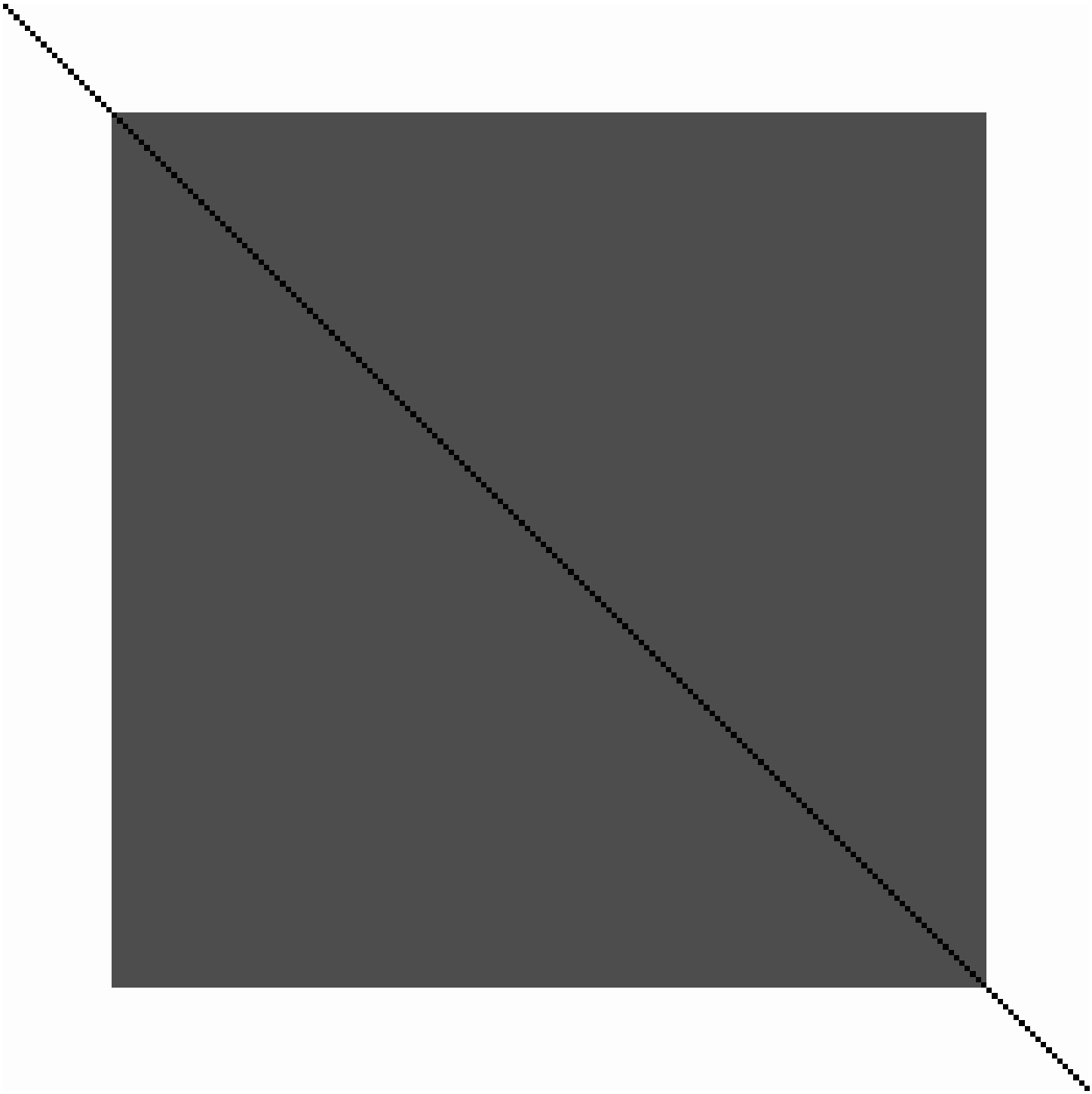

Figure 1.5: test1.gif

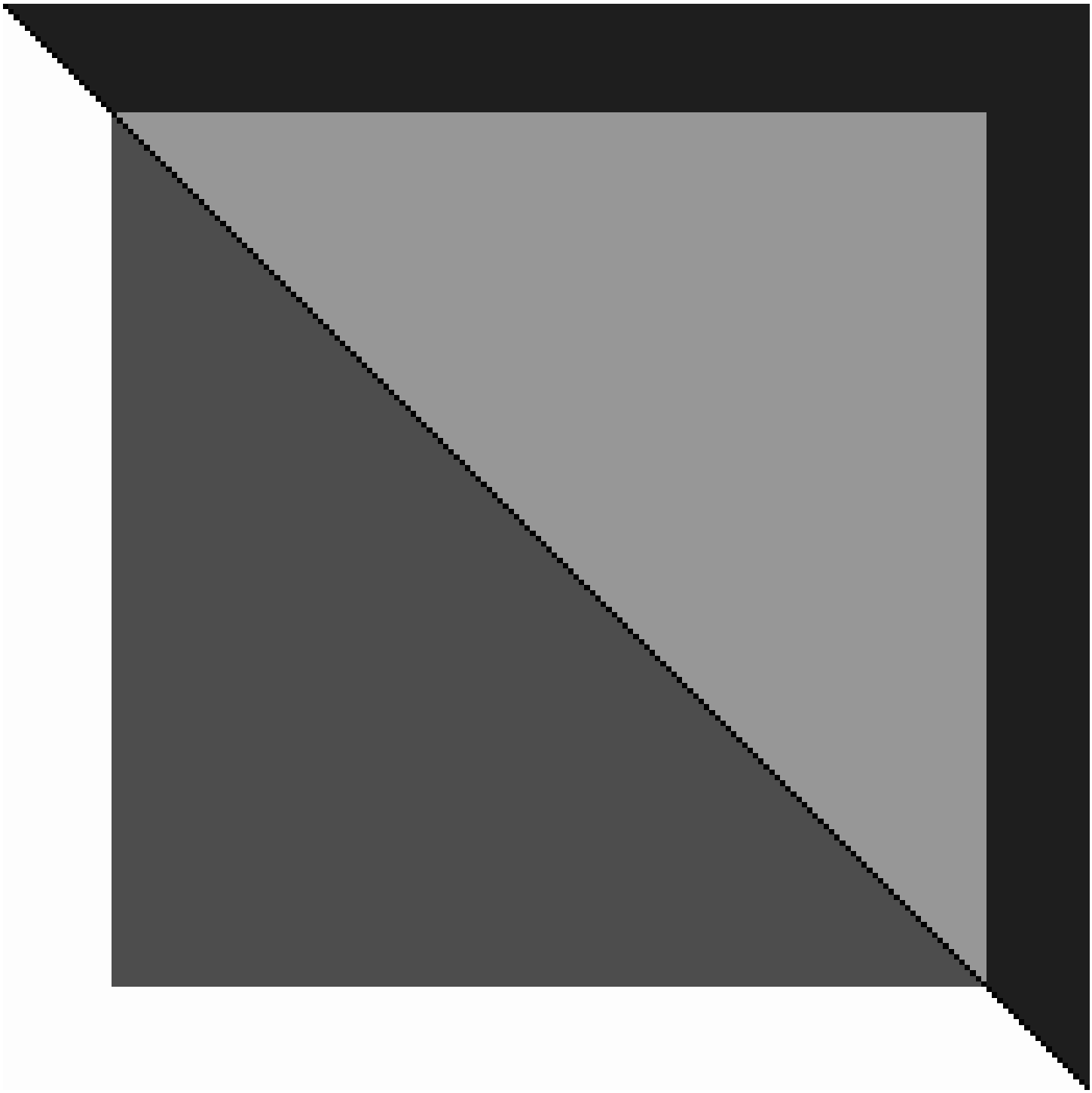

Figure 1.6: test2.gif

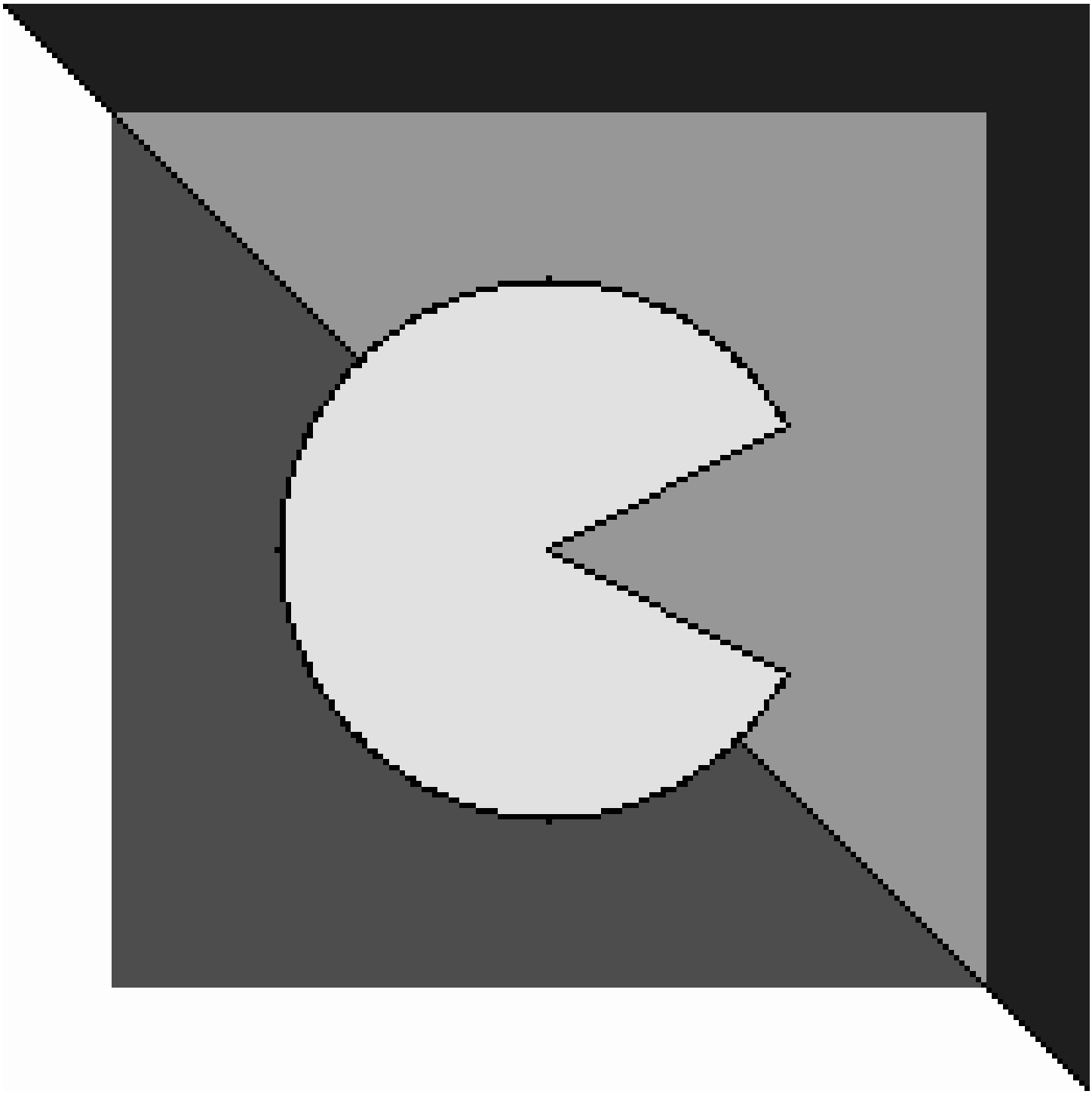

Figure 1.7: test3.gif

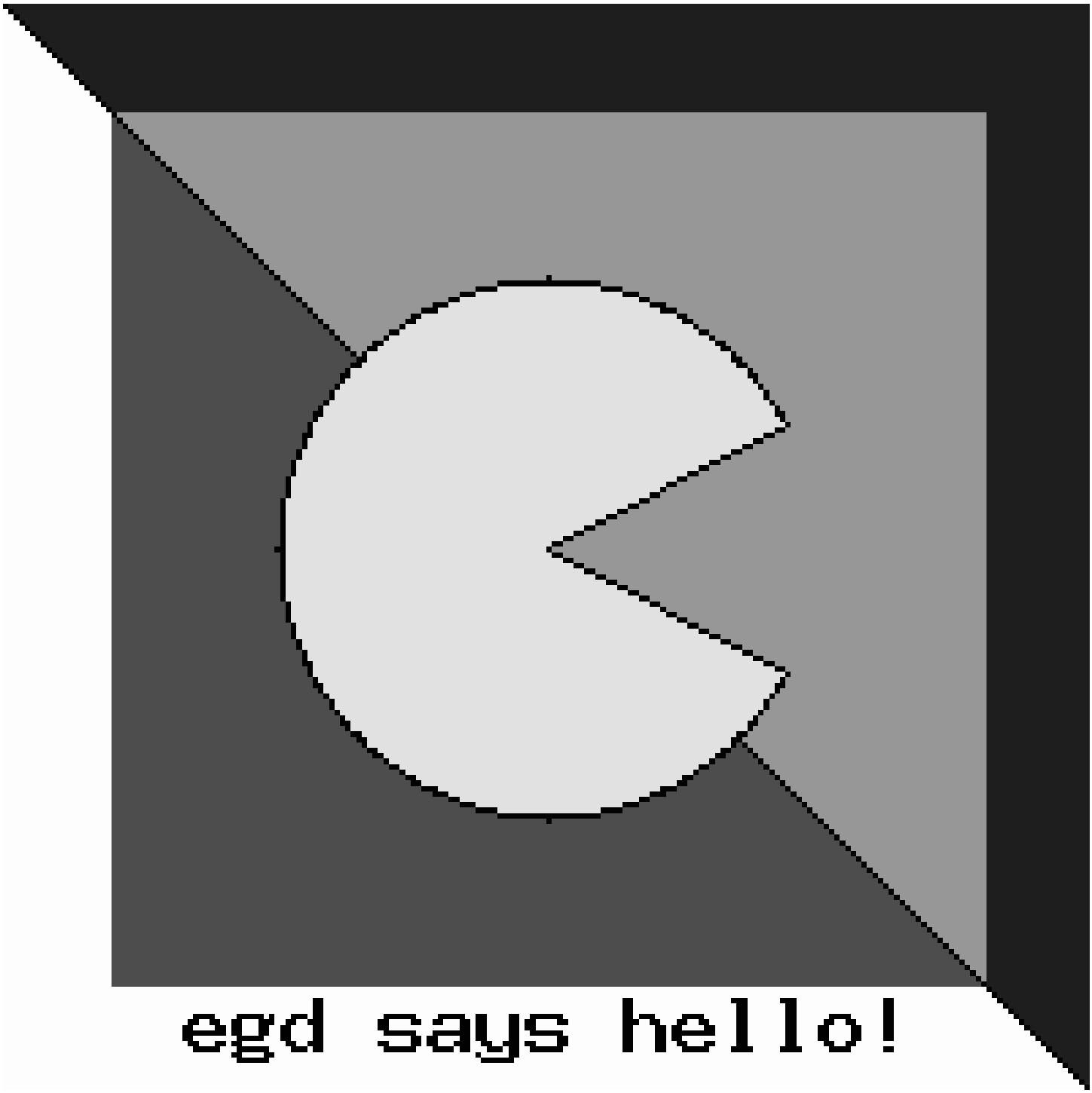


Figure 1.8: test4.gif

### 1.2.3 ESI example

Using egd with inets ESI to generate images on the fly:

```
-module(img_esi).

-export([image/3]).

image(SessionID, _Env, _Input) ->
    mod_esi:deliver(SessionID, header()),
    Binary = my_image(),
    mod_esi:deliver(SessionID, binary_to_list(Binary)).

my_image() ->
    Im = egd:create(300,20),
    White = egd:color(Im, white),
    Black = egd:color(Im, black),
    Red = egd:color(Im, red),
    egd:fill(Im, {20,10}, White),
    egd:filledRectangle(Im, {30,14}, {270,19}, Red),
    egd:rectangle(Im, {30,14}, {270,19}, Black),
    egd:text(Im, medium, {30, 0}, "egd with esi callback", Black),
    Bin = egd:image(Im, png),
    egd:destroy(Im),
    Bin.

header() ->
    "Content-Type: image/png\r
\r
".
```

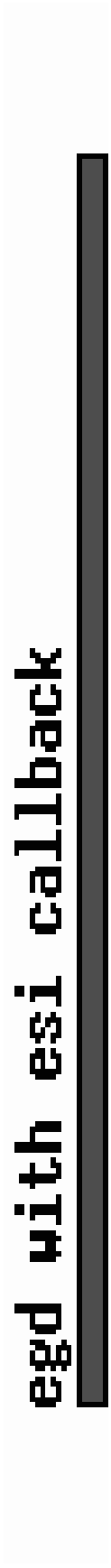


Figure 1.9: Example of result.

For more information regarding ESI, please see inets application [mod_esi].

# Percept Reference Manual

## Short Summaries

- Erlang Module egd [page 24] – Library egd is an interface for the gd library.
- Erlang Module percept [page 27] – Percept - Erlang Concurrency Profiling Tool.
- Erlang Module percept_profile [page 29] – Percept Collector.

### egd

The following functions are exported:

- `arc(Image::image(), Center::point(), Width::integer(), Height::integer(), Start::integer(), End::integer(), Color::color()) -> ok`
  [page 24] Draws an arc with centerpoint in Center, width Width and height Height.
- `color(Image::image(), RGB) -> Color`
  [page 24] Returns a color reference.
- `color(Image::image(), R::byte(), G::byte(), B::byte()) -> color()`
  [page 24] Returns a color reference.
- `create(Width::integer(), Height::integer()) -> image()`
  [page 24] Creates an images with specified dimensions.
- `destroy(Image::image()) -> ok | error`
  [page 24] Destroys specified image.
- `fill(Image::image(), Center::point(), Color::color()) -> ok`
  [page 25] Fills an enclosed area with Center as starting point.
- `filledArc(Image::image(), Center::point(), Width::integer(), Height::integer(), Start::integer(), End::integer(), Color::color(), Options) -> ok`
  [page 25] Draws a filled arc with centerpoint in (Cx, Cy), width Width and height Height.
- `filledEllipse(Image::image(), Center::point(), Width::integer(), Height::integer(), Color::color()) -> ok`
  [page 25] Draws a filled ellipse with color Color and center point at (Cx, Cy) using specified dimensions.
- `filledPolygon(Image::image(), Points::[point()], Color::color()) -> ok`
  [page 25] Draws a filled polygon in the image with color Color.

- `filledRectangle(Image::image(), Start::point(), End::point(), Color::color()) -> ok`
  [page 25] Draws a filled rectangle within the specified coordinates.
- `fontSize(Image::image(), Font::font()) -> {Width, Height}`
  [page 25] Returns the width and height of the font in pixel units.
- `image(Image::image(), Type) -> binary()`
  [page 25] Returns a binary containing the image data.
- `line(Image::image(), Start::point(), End::point(), Color::color()) -> ok`
  [page 25] Draws a line from (X1,Y1) to (X2,Y2) in the image with color Color.
- `pixel(Image::image(), Point::point(), Color::color()) -> ok`
  [page 25] Sets a pixel in the image with color Color.
- `polygon(Image::image(), Points::[points()], Color::color()) -> ok`
  [page 25] Draws a polygon in the image with color Color.
- `rectangle(Image::image(), Start::point(), End::point(), Color::color()) -> ok`
  [page 26] Draws a rectangle within the specified coordinates.
- `resample(Image::image(), Width::integer(), Height::integer()) -> ok`
  [page 26] Resizes the image to the new dimensions specified by Width and Height.
- `rotate(Image::image(), Angle::integer()) -> ok`
  [page 26] Rotates the image Angle degrees (not radians).
- `save(ImageBinary::binary(), Filename::string()) -> {ok, Filename} | {error, Reason}`
  [page 26] Saves the binary of the image to a file.
- `text(Image::image(), Font::font(), Point::point(), String::string(), Color::color()) -> ok`
  [page 26] Draws a horizontal text strip using specified font and color.
- `textUp(Image::image(), Font::font(), Point::point(), String::string(), Color::color()) -> ok`
  [page 26] Draws a vertical text strip using specified font and color.

## percept

The following functions are exported:

- `analyze(Filename::string()) -> ok | {error, Reason}`
  [page 27] Analyze file.
- `profile(Filename::string()) -> {ok, Port} | {already_started, Port}`
  [page 27] Equivalent to percept_profile:start(Filename, [procs]).
- `profile(Filename::string(), Options::[percept_option()]) -> {ok, Port} | {already_started, Port}`
  [page 27] Equivalent to percept_profile:start(Filename, Options).
- `profile(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok | {already_started, Port} | {error, not_started}`
  [page 27] Equivalent to percept_profile:start(Filename, MFA, Options).
- `start_webserver() -> {started, Hostname, Port} | {error, Reason}`
  [page 27] Starts webserver.

- `start_webserver(Port::integer()) -> {started, Hostname, AssignedPort} | {error, Reason}`
  [page 27] Starts webserver.
- `stop_profile() -> ok | {error, not_started}`
  [page 28] Equivalent to percept_profile:stop().
- `stop_webserver() -> ok | {error, not_started}`
  [page 28] Stops webserver.

## percept_profile

The following functions are exported:

- `start(Filename::string()) -> {ok, Port} | {already_started, Port}`
  [page 29] Equivalent to start(Filename, [procs]).
- `start(Filename::string(), Options::[percept_option()]) -> {ok, Port} | {already_started, Port}`
  [page 29] Starts profiling with supplied options.
- `start(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok | {already_started, Port} | {error, not_started}`
  [page 29] Starts profiling at the entrypoint specified by the MFA.
- `stop() -> ok | {error, not_started}`
  [page 29] Stops profiling.

# egd

Erlang Module

Library egd is an interface for the gd library. Currently only a subset of gd's functions are implemented. The egd module should be considered experimental and behaviour may change in future releases.

## DATA TYPES

`color()` Color reference. Do not assume anything about the refrence data structure. It may change in future releases.

`font() = tiny | small | medium | large | giant` Font atoms.

`image()` Image reference. Do not assume anything about the refrence data structure. It may change in future releases.

`point() = {integer(), integer()}` Point tuple with X and Y coordinates.

## Exports

`arc(Image::image(), Center::point(), Width::integer(), Height::integer(), Start::integer(), End::integer(), Color::color()) -> ok`

Draws an arc with centerpoint in Center, width Width and height Height. Start and End are degrees (not radians) of the arc.

`color(Image::image(), RGB) -> Color`

Types:

- RGB = white | black | red | blue | green | {byte(), byte(), byte()}
- Color = color() | {error, {invalid_color, RGB}}

Returns a color reference.

`color(Image::image(), R::byte(), G::byte(), B::byte()) -> color()`

Returns a color reference.

`create(Width::integer(), Height::integer()) -> image()`

Creates an images with specified dimensions. The color palette is in truecolor.

`destroy(Image::image()) -> ok | error`

Destroys specified image.

`fill(Image::image(), Center::point(), Color::color()) -> ok`

Fills an enclosed area with Center as starting point.

`filledArc(Image::image(), Center::point(), Width::integer(), Height::integer(), Start::integer(), End::integer(), Color::color(), Options) -> ok`

Types:

- Options = [Option]
- Option = arc | chord | no_fill | edged

Draws a filled arc with centerpoint in (Cx, Cy), width Width and height Height. Start and End are degrees (not radians) of the arc.

`filledEllipse(Image::image(), Center::point(), Width::integer(), Height::integer(), Color::color()) -> ok`

Draws a filled ellipse with color Color and center point at (Cx, Cy) using specified dimensions.

`filledPolygon(Image::image(), Points::[point()], Color::color()) -> ok`

Draws a filled polygon in the image with color Color. The last point and first points forms the polygon closure.

`filledRectangle(Image::image(), Start::point(), End::point(), Color::color()) -> ok`

Draws a filled rectangle within the specified coordinates.

`fontSize(Image::image(), Font::font()) -> {Width, Height}`

Types:

- Widht = integer()
- Height = integer()

Returns the width and height of the font in pixel units.

`image(Image::image(), Type) -> binary()`

Types:

- Type = gif | png | jpeg | {jpeg, Quality}
- Quality = integer()

Returns a binary containing the image data.

`line(Image::image(), Start::point(), End::point(), Color::color()) -> ok`

Draws a line from (X1,Y1) to (X2,Y2) in the image with color Color.

`pixel(Image::image(), Point::point(), Color::color()) -> ok`

Sets a pixel in the image with color Color.

`polygon(Image::image(), Points::[points()], Color::color()) -> ok`

Draws a polygon in the image with color Color. The last point and first points forms the polygon closure.

```
rectangle(Image::image(), Start::point(), End::point(), Color::color()) -> ok
```

Draws a rectangle within the specified coordinates.

```
resample(Image::image(), Width::integer(), Height::integer()) -> ok
```

Resizes the image to the new dimensions specified by Width and Height.

```
rotate(Image::image(), Angle::integer()) -> ok
```

Rotates the image Angle degrees (not radians).

```
save(ImageBinary::binary(), Filename::string()) -> {ok, Filename} | {error, Reason}
```

Saves the binary of the image to a file.

```
text(Image::image(), Font::font(), Point::point(), String::string(), Color::color())
    -> ok
```

Draws a horizontal text strip using specified font and color.

```
textUp(Image::image(), Font::font(), Point::point(), String::string(),
    Color::color()) -> ok
```

Draws a vertical text strip using specified font and color.

# percept

Erlang Module

Percept - Erlang Concurrency Profiling Tool

This module provides the user interface for the application.

## DATA TYPES

```
percept_option() = procs | ports | exclusive
```

## Exports

```
analyze(Filename::string()) -> ok | {error, Reason}
```

Analyze file.

```
profile(Filename::string()) -> {ok, Port} | {already_started, Port}
```

Equivalent to percept_profile:start(Filename, [procs]) [page ??].

```
profile(Filename::string(), Options::[percept_option()]) -> {ok, Port} |
        {already_started, Port}
```

Equivalent to percept_profile:start(Filename, Options) [page ??].

```
profile(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok |
        {already_started, Port} | {error, not_started}
```

Equivalent to percept_profile:start(Filename, MFA, Options) [page ??].

```
start_webserver() -> {started, Hostname, Port} | {error, Reason}
```

Types:

- Hostname = string()
- Port = integer()
- Reason = term()

Starts webserver.

```
start_webserver(Port::integer()) -> {started, Hostname, AssignedPort} | {error,
        Reason}
```

Types:

- Hostname = string()

- AssignedPort = integer()
- Reason = term()

Starts webserver. If port number is 0, an available port number will be assigned by inets.

`stop_profile() -> ok | {error, not_started}`

Equivalent to percept_profile:stop() [page ??].

`stop_webserver() -> ok | {error, not_started}`

Stops webserver.

# percept_profile

Erlang Module

Percept Collector

This module provides the user interface for the percept data collection (profiling).

## DATA TYPES

```
percept_option() = procs | ports | exclusive
```

## Exports

`start(Filename::string()) -> {ok, Port} | {already_started, Port}`

Equivalent to start(Filename, [procs]) [page 29].

`start(Filename::string(), Options::[percept_option()]) -> {ok, Port} | {already_started, Port}`

Types:

- Port = port()

Starts profiling with supplied options. All events are stored in the file given by Filename. An explicit call to stop/0 is needed to stop profiling.

`start(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok | {already_started, Port} | {error, not_started}`

Types:

- Port = port()

Starts profiling at the entrypoint specified by the MFA. All events are collected, this means that processes outside the scope of the entry-point are also profiled. No explicit call to stop/0 is needed, the profiling stops when the entry function returns.

`stop() -> ok | {error, not_started}`

Stops profiling.

# List of Figures

# Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.