

Percept Application

version 0.8

Typeset in L^AT_EX from SGML source using the DocBuilder-0.9.8.6 Document System.

Contents

1	Percept User's Guide	1
1.1	Percept	1
1.1.1	Introduction	1
1.1.2	Getting started	2
1.2	egd	12
1.2.1	Introduction	12
1.2.2	File example	12
1.2.3	ESI example	18
2	Percept Reference Manual	21
2.1	egd	23
2.2	percept	25
2.3	percept_profile	27
	List of Figures	29

Chapter 1

Percept User's Guide

Percept is an akronym for *Percept - erlang concurrency profiling tool*.

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 27] 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.

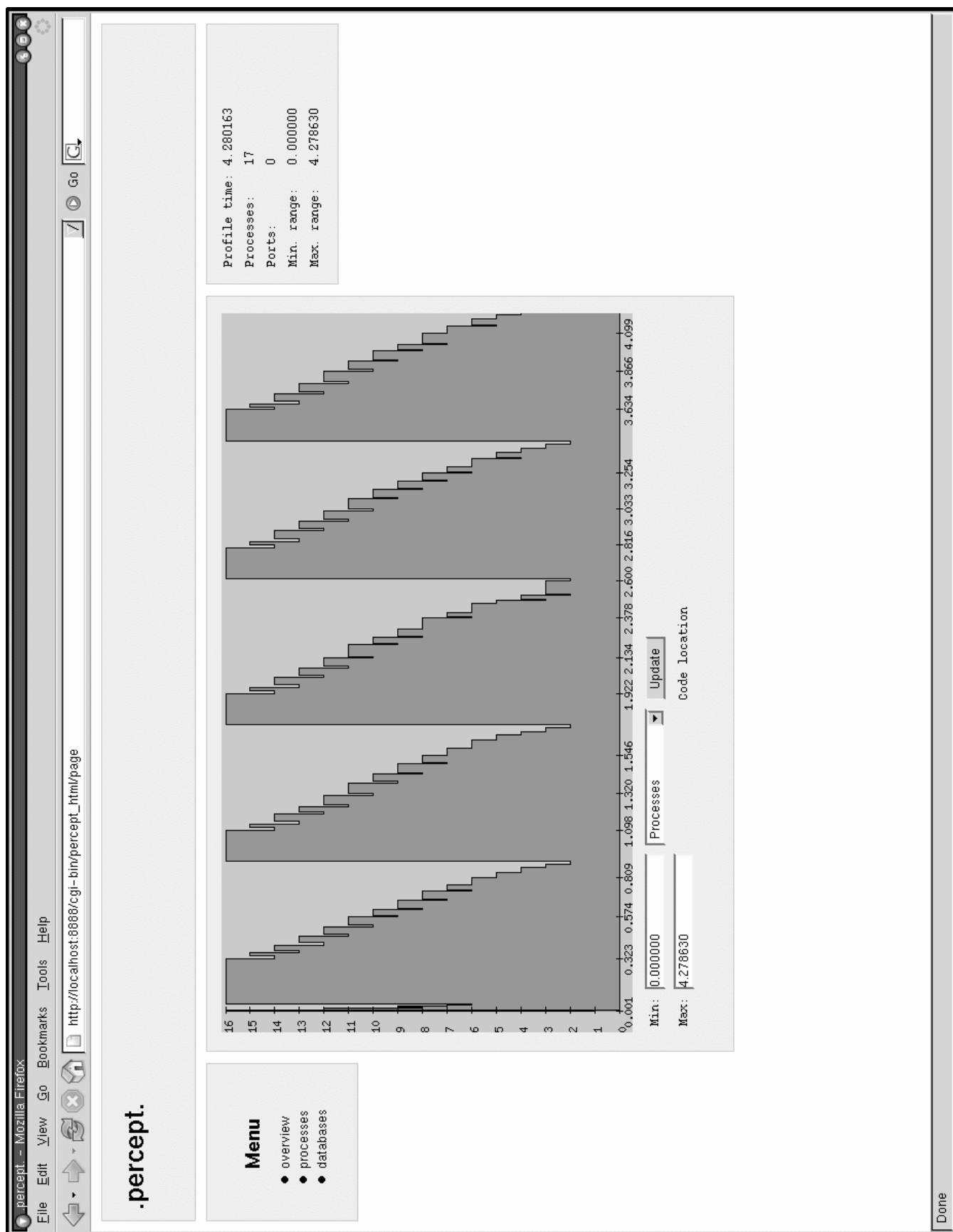


Figure 1.1: Overview selection

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.

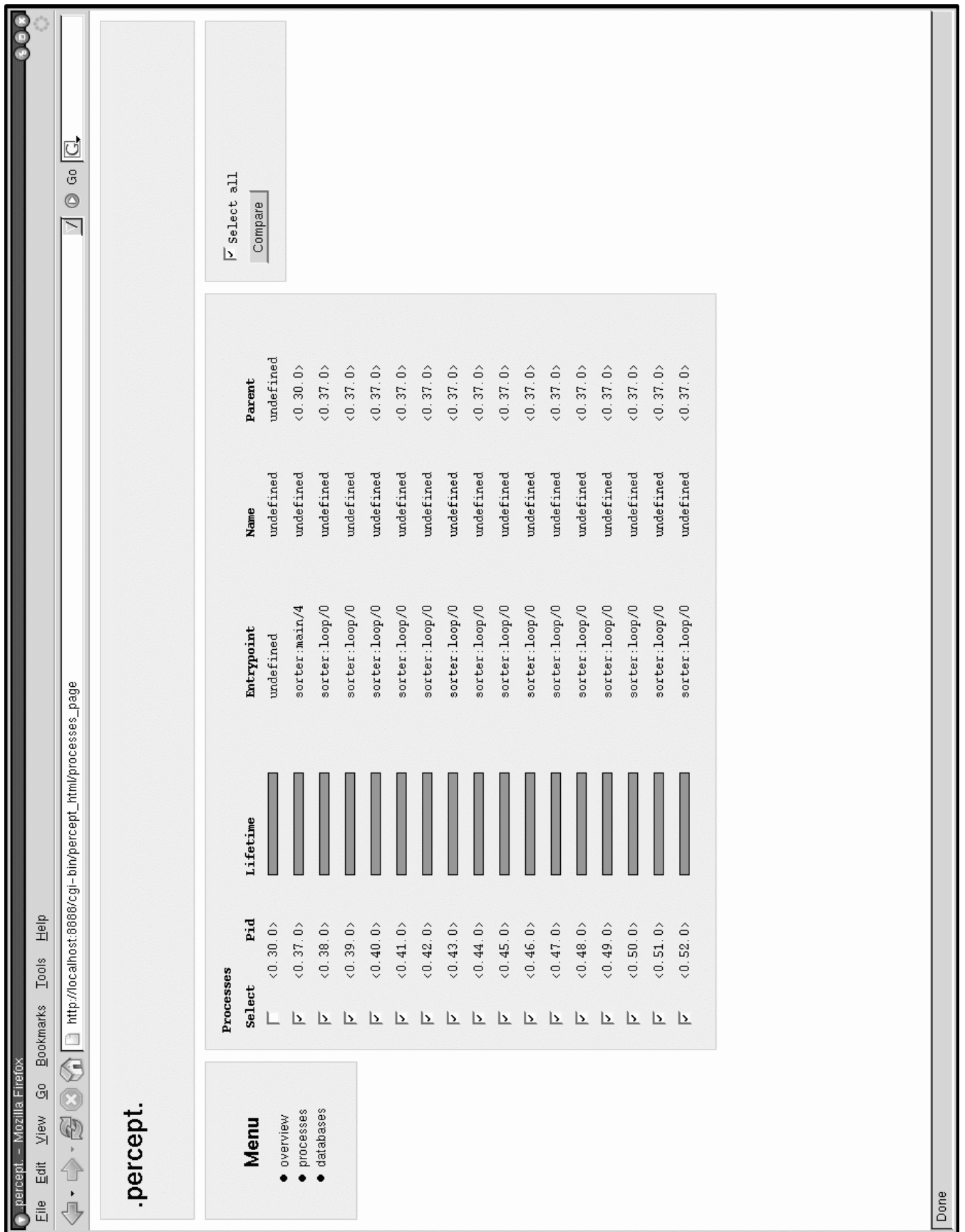


Figure 1.2: Processes selection

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.

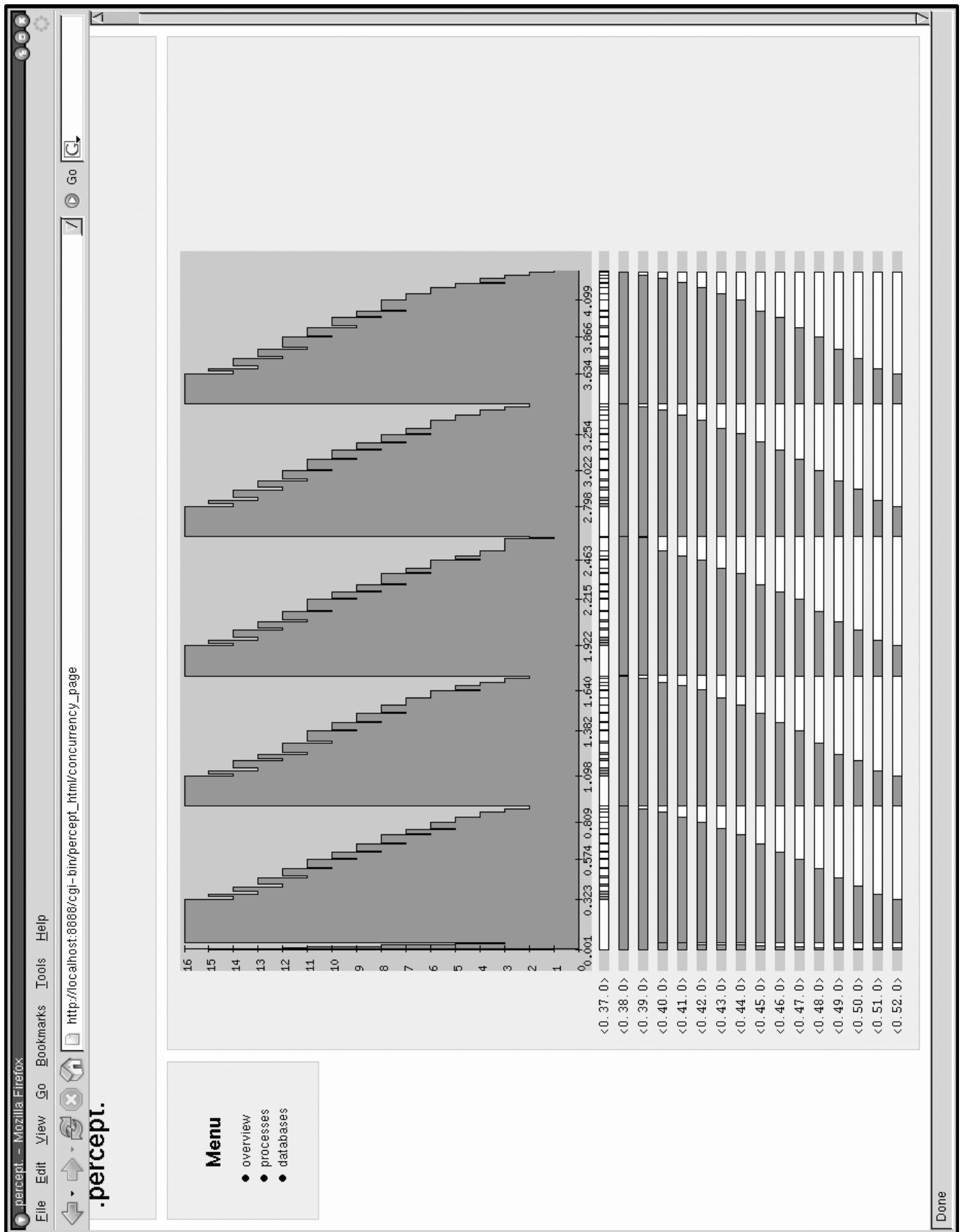


Figure 1.3: Processes compare selection

Process information selection Here we can see 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.

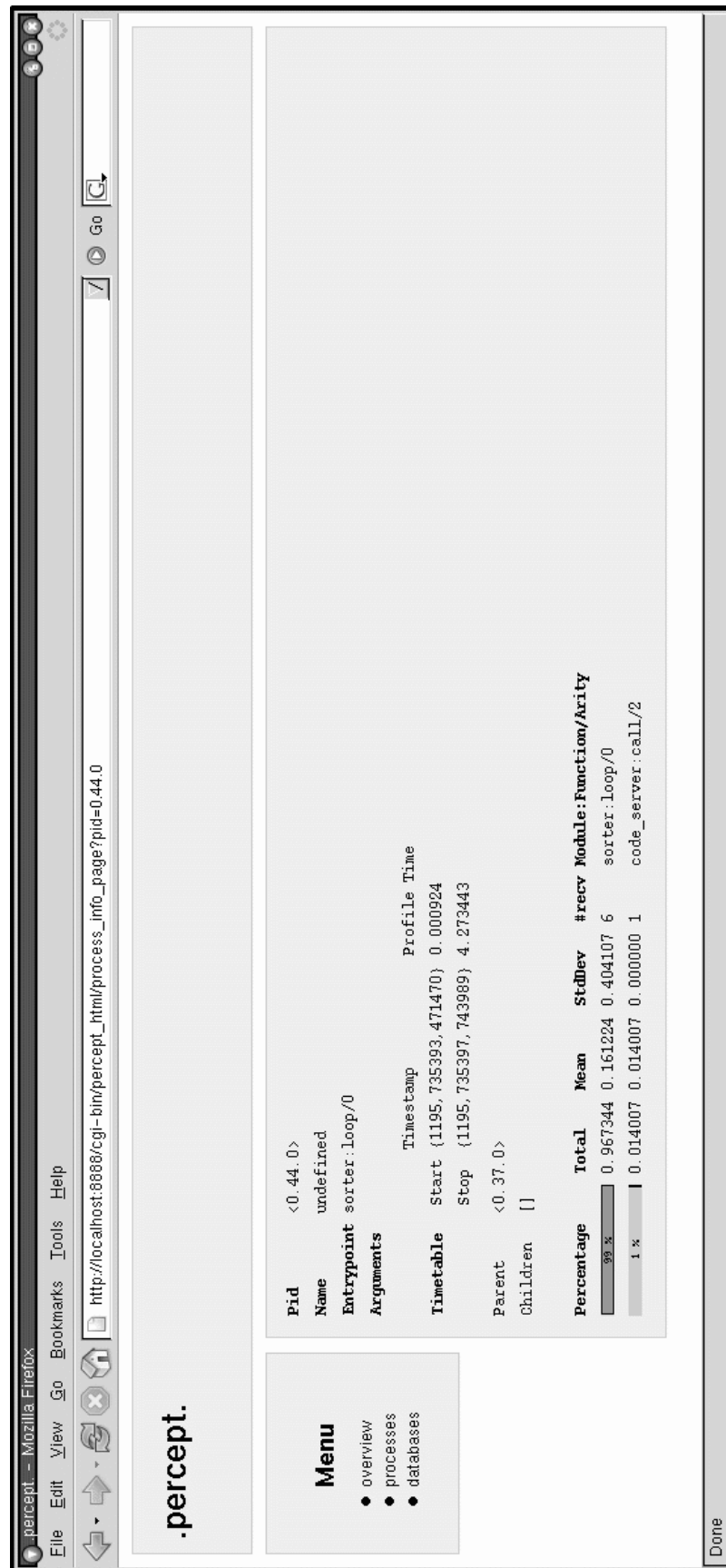


Figure 1.4: Process information selection

1.2 egd

1.2.1 Introduction

The egd module is an interface for 2d-image rendering and is used by Percept to generate dynamic graphs to its web pages. All code is pure erlang, no drivers needed.

The library is intended for small to medium image sizes with low complexity for optimal performance. The library handles horizontal lines better than vertical lines.

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),  
    Red = egd:color({255,0,0}),  
    Green = egd:color({0,255,0}),  
    Blue = egd:color({0,0,255}),  
    Black = egd:color({0,0,0}),  
    Yellow = egd:color({255,255,0}),  
  
    % Line and fillRectangle  
  
    egd:filledRectangle(Im, {20,20}, {180,180}, Red),  
    egd:line(Im, {0,0}, {200,200}, Black),  
  
    egd:save(egd:render(Im, png), "/home/egil/test1.png"),  
  
    egd:filledEllipse(Im, {45, 60}, {55, 70}, Yellow),  
    egd:filledEllipse(Im, {145, 60}, {155, 70}, Blue),  
  
    egd:save(egd:render(Im, png), "/home/egil/test2.png"),  
  
    R = 80,  
    X0 = 99,  
    Y0 = 99,  
  
    Pts = [ { X0 + trunc(R*math:cos(A*math:pi()*2/360)),  
            Y0 + trunc(R*math:sin(A*math:pi()*2/360))  
            } || A <- lists:seq(0,359,5)],  
    lists:map(  
        fun({X,Y}) ->  
            egd:rectangle(Im, {X-5, Y-5}, {X+5,Y+5}, Green)  
        end, Pts),
```



```
egd:save(egd:render(Im, png), "/home/egil/test3.png"),

% Text
Filename = filename:join([code:priv_dir(percept), "fonts", "6x11_latin1.wingsfont"]),
Font = egd_font:load(Filename),
{W,H} = egd_font:size(Font),
String = "egd says hello",
Length = length(String),

egd:text(Im, {round(100 - W*Length/2), 200 - H - 5}, Font, String, Black),

egd:save(egd:render(Im, png), "/home/egil/test4.png"),

egd:destroy(Im).
```

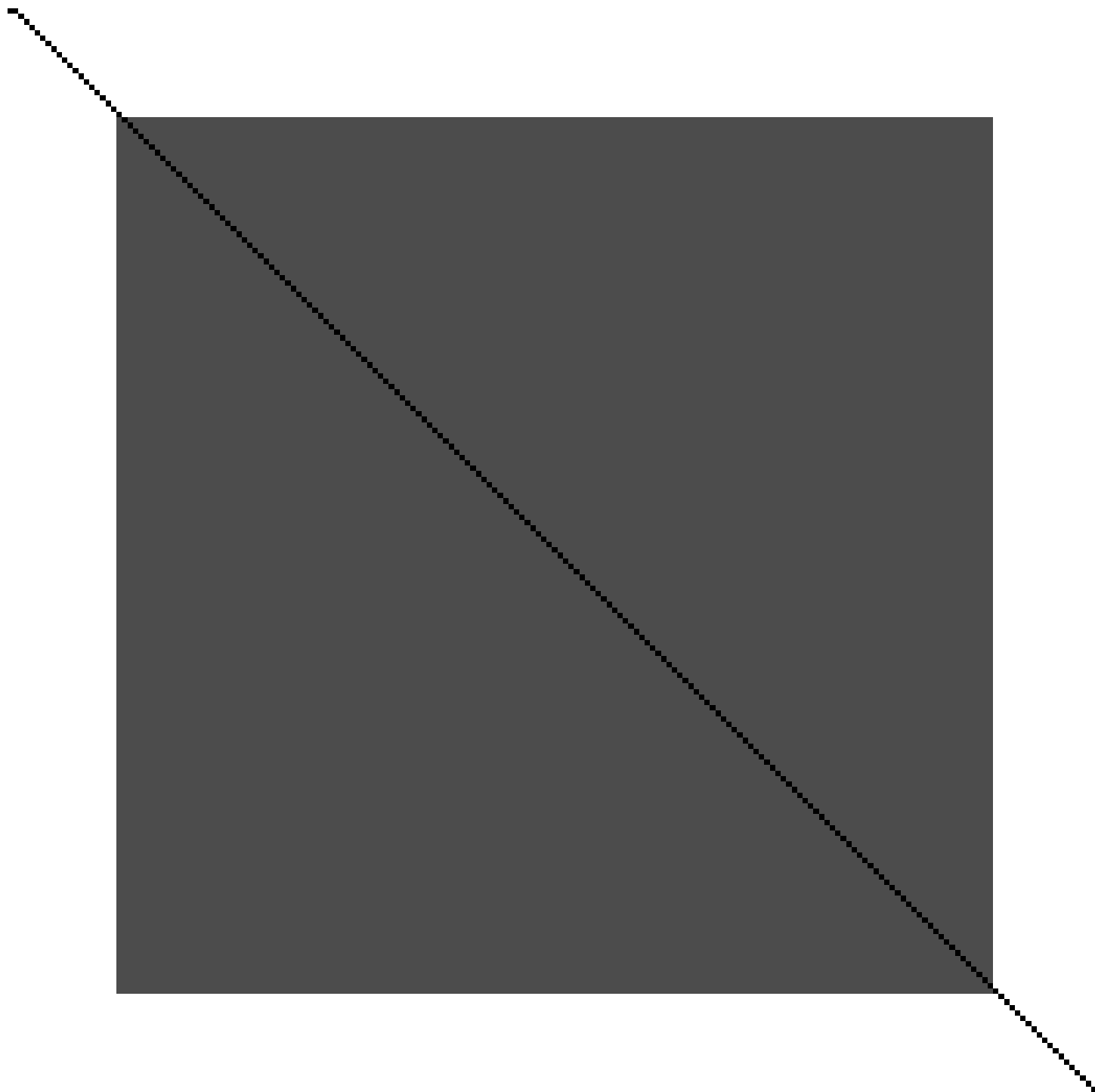


Figure 1.5: test1.gif

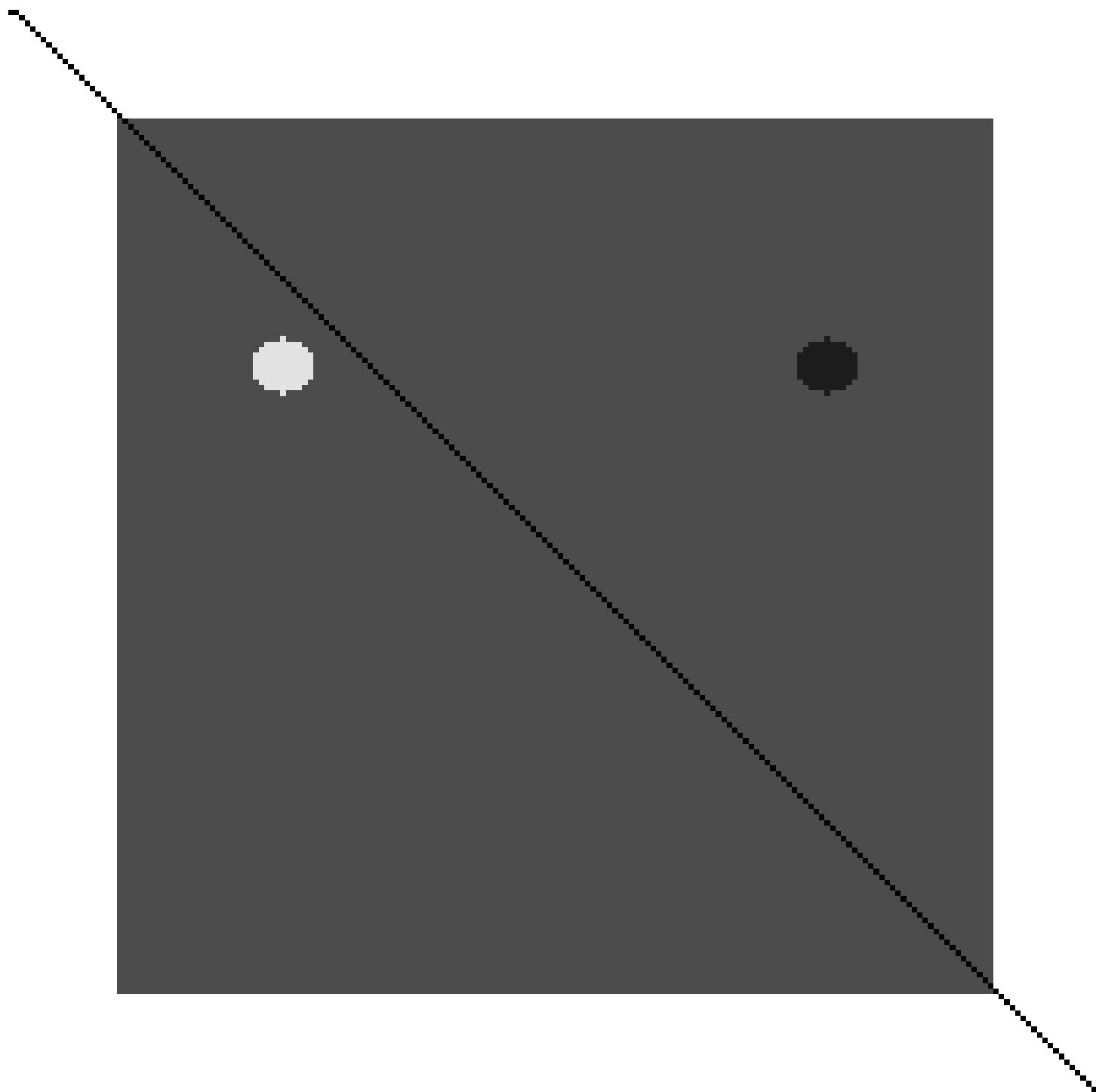


Figure 1.6: test2.gif

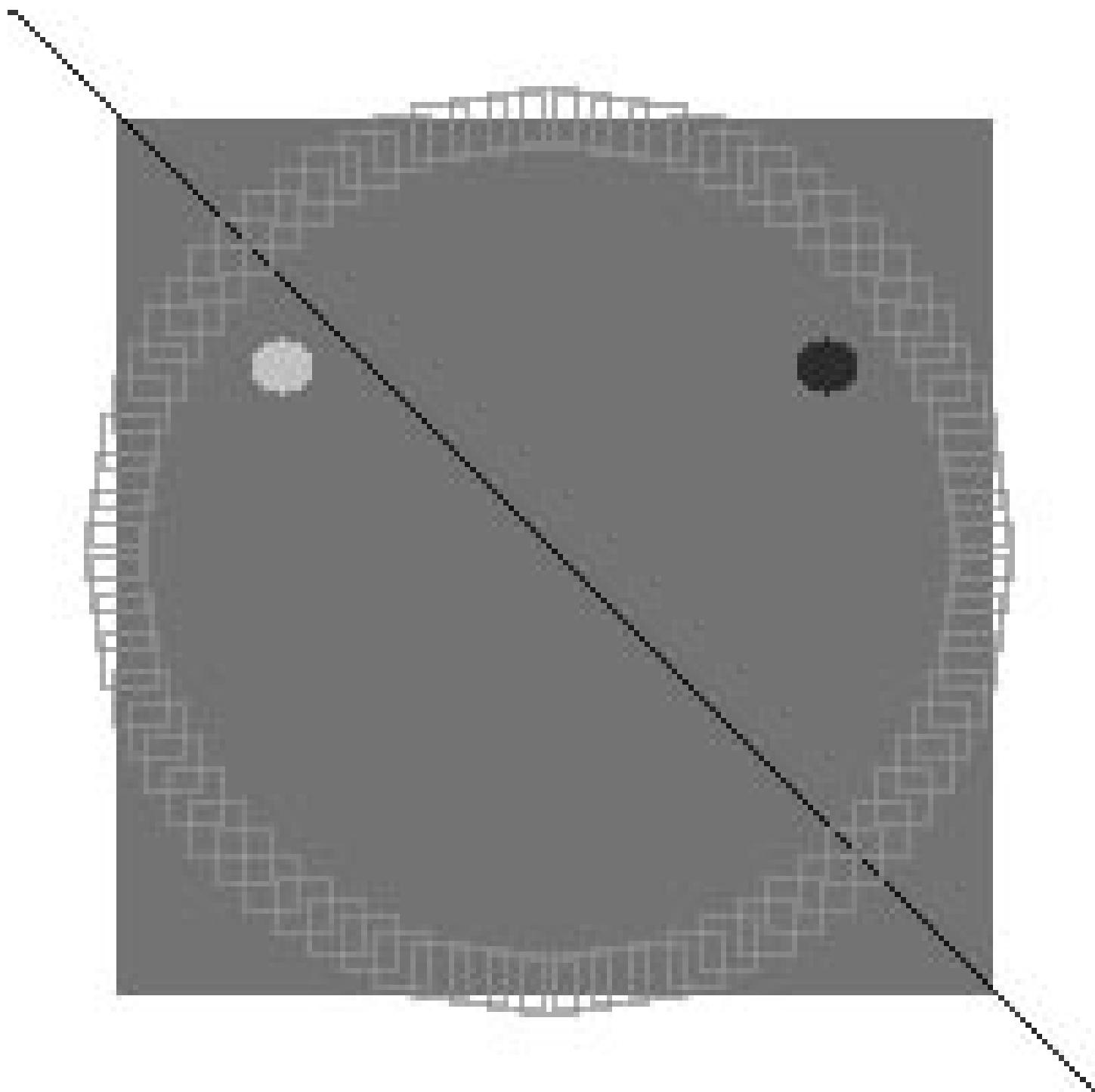


Figure 1.7: test3.gif

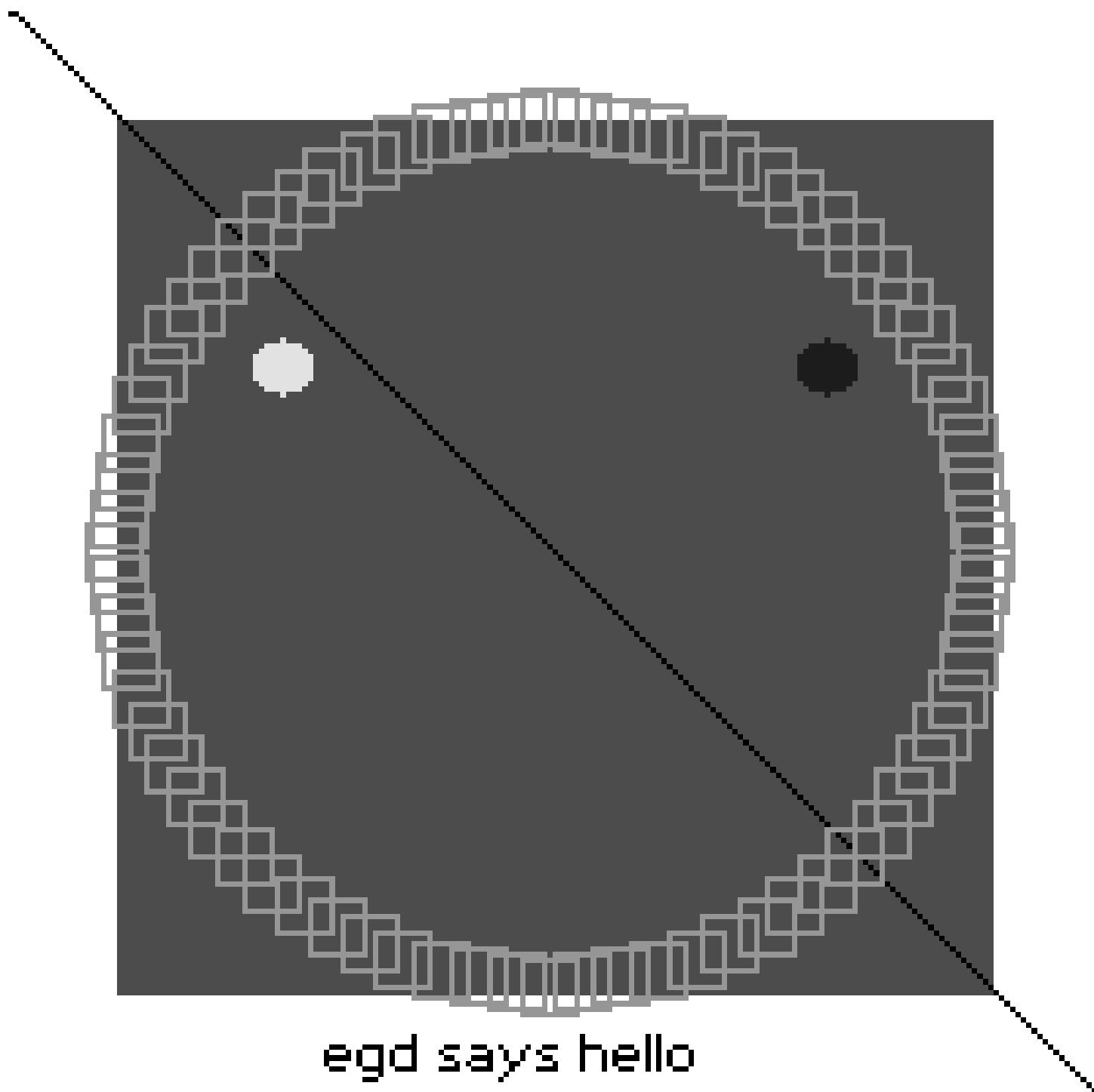


Figure 1.8: test4.gif

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),  
    Black = egd:color({0,0,0}),  
    Red = egd:color({255,0,0}),  
    egd:filledRectangle(Im, {30,14}, {270,19}, Red),  
    egd:rectangle(Im, {30,14}, {270,19}, Black),  
  
    Filename = filename:join([code:priv_dir(percept), "fonts", "6x11_latin1.wingsfont"]),  
    Font = egd_font:load(Filename),  
    egd:text(Im, {30, 0}, Font, "egd with esi callback", Black),  
    Bin = egd:render(Im, png),  
    egd:destroy(Im),  
    Bin.  
  
header() ->  
    "Content-Type: image/png\r  
\r  
".
```

A vertical rectangular bar with a light gray background and a dark gray border. The text "egd with esi callback" is written vertically inside the bar in a bold, black, monospace font.

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 23] – egd - erlang graphical drawer.
- Erlang Module **percept** [page 25] – Percept - Erlang Concurrency Profiling Tool.
- Erlang Module **percept_profile** [page 27] – Percept Collector.

egd

The following functions are exported:

- `color(Color::Value | Name) -> color()`
[page 23] Creates a color reference.
- `create(Width::integer(), Height::integer()) -> egd_image()`
[page 23] Creates an image area and returns its reference.
- `destroy(Image::egd_image()) -> ok`
[page 23] Destroys the image.
- `filledEllipse(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok`
[page 23] Creates a filled ellipse object.
- `filledRectangle(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok`
[page 23] Creates a filled rectangle object.
- `line(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok`
[page 23] Creates a line object from P1 to P2 in the image.
- `rectangle(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok`
[page 24] Creates a rectangle object.
- `render(Image::egd_image()) -> binary()`
[page 24] Equivalent to `render(Image, png, [{render_engine, opaque}])`.
- `render(Image::egd_image(), Type::png | raw_bitmap) -> binary()`
[page 24] Equivalent to `render(Image, Type, [{render_engine, opaque}])`.
- `render(Image::egd_image(), Type::png | raw_bitmap, Options::[render_option()]) -> binary()`
[page 24] Renders a binary from the primitives specified by `egd_image()`.
- `save(Binary::binary(), Filename::string()) -> ok`
[page 24] Saves the binary to file.

- `text(Image::egd_image(), P::point(), Font::font(), Text::string(), Color::color()) -> ok`
[page 24] Creates a text object.

percept

The following functions are exported:

- `analyze(Filename::string()) -> ok | {error, Reason}`
[page 25] Analyze file.
- `profile(Filename::string()) -> {ok, Port} | {already_started, Port}`
[page 25]
- `profile(Filename::string(), Options::[percept_option()]) -> {ok, Port} | {already_started, Port}`
[page 25]
- `profile(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok | {already_started, Port} | {error, not_started}`
[page 25]
- `start_webserver() -> {started, Hostname, Port} | {error, Reason}`
[page 25] Starts webserver.
- `start_webserver(Port::integer()) -> {started, Hostname, AssignedPort} | {error, Reason}`
[page 25] Starts webserver.
- `stop_profile() -> ok | {error, not_started}`
[page 26]
- `stop_webserver() -> ok | {error, not_started}`
[page 26] Stops webserver.

percept_profile

The following functions are exported:

- `start(Filename::string()) -> {ok, Port} | {already_started, Port}`
[page 27] Equivalent to `start(Filename, [procs])`.
- `start(Filename::string(), Options::[percept_option()]) -> {ok, Port} | {already_started, Port}`
[page 27] Starts profiling with supplied options.
- `start(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok | {already_started, Port} | {error, not_started}`
[page 27] Starts profiling at the entrypoint specified by the MFA.
- `stop() -> ok | {error, not_started}`
[page 27] Stops profiling.

egd

Erlang Module

egd - erlang graphical drawer

DATA TYPES

`color()`

`egd_image()`

`point() = {integer(), integer()}`

`render_option() = {render_engine, opaque} | {render_engine, alpha}`

Exports

`color(Color::Value | Name) -> color()`

Types:

- Value = {byte(), byte(), byte()} | {byte(), byte(), byte(), byte()}
- Name = black | silver | gray | white | maroon | red | purple | fuchsia | green | lime | olive | yellow | navy | blue | teal | aqua

Creates a color reference.

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

Creates an image area and returns its reference.

`destroy(Image::egd_image()) -> ok`

Destroys the image.

`filledEllipse(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok`

Creates a filled ellipse object.

`filledRectangle(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok`

Creates a filled rectangle object.

`line(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok`

Creates a line object from P1 to P2 in the image.

```
rectangle(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok
```

Creates a rectangle object.

```
render(Image::egd_image()) -> binary()
```

Equivalent to `render(Image, png, [{render_engine, opaque}])` [page 24].

```
render(Image::egd_image(), Type::png | raw_bitmap) -> binary()
```

Equivalent to `render(Image, Type, [{render_engine, opaque}])` [page 24].

```
render(Image::egd_image(), Type::png | raw_bitmap, Options::[render_option()]) ->  
binary()
```

Renders a binary from the primitives specified by `egd_image()`. The binary can either be a raw bitmap with rgb tripplets or a binary in png format.

```
save(Binary::binary(), Filename::string()) -> ok
```

Saves the binary to file.

```
text(Image::egd_image(), P::point(), Font::font(), Text::string(), Color::color()) ->  
ok
```

Creates a text object.

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}`

See also: `percept_profile` [page 27].

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

See also: `percept_profile` [page 27].

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

See also: `percept_profile` [page 27].

`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}

See also: `percept_profile` [page 27].

`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 27].

```
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

1.1	Overview selection	5
1.2	Processes selection	7
1.3	Processes compare selection	9
1.4	Process information selection	11
1.5	test1.gif	14
1.6	test2.gif	15
1.7	test3.gif	16
1.8	test4.gif	17
1.9	Example of result.	19

Index of Modules and Functions

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

analyze/1
 percept , 25

color/1
 egd , 23

create/1
 egd , 23

destroy/1
 egd , 23

egd
 color/1, 23
 create/1, 23
 destroy/1, 23
 filledEllipse/1, 23
 filledRectangle/1, 23
 line/1, 23
 rectangle/1, 24
 render/1, 24
 save/1, 24
 text/1, 24

filledEllipse/1
 egd , 23

filledRectangle/1
 egd , 23

line/1
 egd , 23

percept
 analyze/1, 25
 profile/1, 25
 start_webserver/0, 25
 start_webserver/1, 25
 stop_profile/0, 26
 stop_webserver/0, 26

percept_profile
 start/1, 27
 stop/0, 27

profile/1
 percept , 25

rectangle/1
 egd , 24

render/1
 egd , 24

save/1
 egd , 24

start/1
 percept_profile , 27

start_webserver/0
 percept , 25

start_webserver/1
 percept , 25

stop/0
 percept_profile , 27

stop_profile/0
 percept , 26

stop_webserver/0
 percept , 26

text/1
 egd , 24

