Использование Cairo при программировании графики в Scheme(Guile).

Введение.

Что вы знаете про Cairo? Ну наверное, что то знаете раз обратились к данному документу. Про себя скажу, пару недель назад я про него не то что ничего не знал, но и даже не слышал о нём. Но решивши поупражняться в программировании примеров на CTK+/Gnome встретил интересную отсылку на биндинг для Guile: с помощью пакета guile-cairo, на котором как то можно программировать графику в GTK+. Осмотрев биндинг, я, к сожалению, не обнаружил там примеров, лишь в файле readme разработчики биндинга привели несколько команд показывающих как можно пользоваться их работой, в принципе этих команд достаточно, чтобы начать работу с биндингом и мне конечно их хватило. Но судя по отсутствию многочисленных хвалебных постов в интернете о применении guile-cairo(а хвалить там действительно есть что!) этих команд хватает не всем. Поэтому я и решился проработать один из имеющихся учебников по cairo, написанных на другом языке программирования и переписать представленные там примеры на Scheme с использованием guile-cairo. Я выбрал учебник на английском языке для языка программирования Си, размещен он онлайн по адресу: http://zetcode.com/gfx/cairo/. Но если вы еще не встречались с Cairo, советую почитать немного теории об этой системе, о ней немного написано и на русском, вот ссылки:

http://www.opennet.ru/docs/RUS/tutorial_cairo/
wiki.linuxformat.ru/wiki/LXF71:Cairo
https://www.ibm.com/developerworks/ru/library/l-cairo/
https://progtips.ru/texnologii-programmirovaniya/graficheskaya-biblioteka-cairo.html
и другие, какие найдете.

Итак:

Немного теории Cairo.

Cairo использует следующую модель рисования:

1. Создается или выбирается некоторая графическая поверхность на которую надо выполнить
отрисовку нашего изображения:
(define surf (cairo-image-surface-create 'argb32 250 100))

2. Для этой поверхности создается контекст, в котором будет строиться изображение:
(define cr (cairo-create surf))

3. Выбирается источник, который используется для построения изображения(source). Это может быть цвет, градиент, шаблон или изображение. Шаблон в Cairo может быть сплошным, на основе поверхности или градиента. Сейчас наш источник это цвет:
(cairo-set-source-rgb cr 0.6 0.6 0.6)

4. Выполняется построение векторного изображения, которое становится маской, по которой источник переноситься на поверхность выбранную в указываемом контексте.
(cairo-rectangle cr 0 0 250 100)

5. Выполняется перенос источника(в нашем случае мы установили белый цвет rgb) в соответствии с построенной маской (мы построили прямоугольник охватывающий всю поверхность) на поверхность surf, с помощью операции заполнения(или какой другой, операций переноса источника с использованием определяемой маски много).

(cairo-fill cr)

Далее операции 3, 4, 5 повторяются(при необходимости) и этим самым формируется окончательный вид формируемой поверхности.

еще раз:

```
(cairo-set-source-rgb cr 0 0 0)
(cairo-set-line-width cr 5)
(cairo-move-to cr 10 10)
(cairo-line-to cr 210 10)
(cairo-stroke cr)
(cairo-arc cr 110 50 40 0 (* 2 pi))
(cairo-stroke cr)
```

и еще:

```
(cairo-arc cr 110 50 30 0 (* 2 pi))
(cairo-fill cr)
```

6. Используем построенную поверхность, например выводим ее в файл:

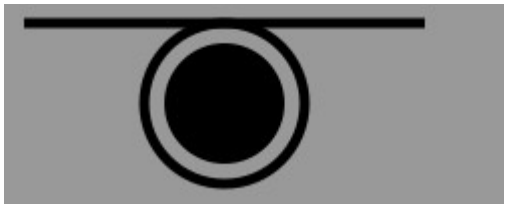
(cairo-surface-write-to-png surf "basedraw0.png")

7. Завершаем работу с Cairo, уничтожая контекст и сформированную поверхность(если она нам больше не нужна).

(**cairo-destroy cr)**
(cairo-surface-destroy surf)

Все, немного ознакомившись с теорией мы построили нашу первую программу по использованию guile-cairo.

Её полный текст приведен в файле: **00_basedraw_png.scm**



Немного изменённая программа рисует «глаз»: **00_draweye_png.scm**

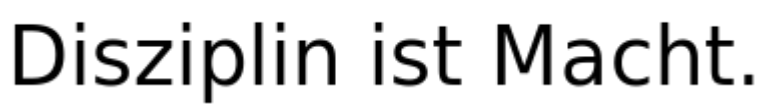
Программа показывает базовые техники работы с Cairo, когда сочетания команд рисования контуров по маске **(cairo-stroke cr)** и заполнения по созданной маске **(cairo-fill cr)** позволяют строить достаточно сложные изображения.

Ну а теперь переходим к описанию примеров из ZetCode.

Cairo Backends.

Бэкэнды Cairo это результаты его работы, это могут быть файлы различных типов(png, pdf, svg), GTK окна, Xlib, OpenGL(Glitz) сцены.

Первое приложение 01_make_png.scm печатает текст и формирует файл png.



В нем мы знакомимся с командой
(cairo-show-text cr "Disziplin ist Macht.")

которая говорит нам, что автор учебника скорее всего имеет немецкие корни, а заодно выполняет одновременное рисование маски(шаг 4) текста и отображение ее на поверхность(шаг 5).

Я проверил, как вывод текста работает с кириллицей см: **01_make_png2.scm**. Работает отлично:

Далее следует приложение выполняющее вывод в pdf файл: **02_make_pdf.scm**
мы определяем поверхность типа pdf
(define surf (cairo-pdf-surface-create 504 648 "cairo2.pdf"))

а затем выполняем «показ страницы»:
(cairo-show-page cr)

Еще один пример это печать в файл svg: **03_make_svg.scm**
Здесь тоже все как обычно:
Создаем поверхность svg

(define surf (cairo-svg-surface-create 390 60 "cairo3.svg"))

и вообщем то все!

Cairo & GTK

Использование следующего бэкэнда GTK вызвало у меня значительные трудности, вы это можете увидеть по множеству за комментированных команд в файле: **04_gtk_widget_explore.scm**

Я исследовал, проверял, пробовал… пока не получилось. Мой вариант отличается от авторского, поскольку я не нашел сигнала «draw» для объекта **gtk-drawing-area**. У меня вместо этого сигнала используется сигнал «event». Так же у автора в обработчик сообщения передается уже построенный контекст cairo, мне же пришлось строить его вручную.

Ну вообщем то за этими небольшими исключениями код идентичный авторскому приведен в файле: **04_gtk_widget.scm**

Создаем окно и формируем его структуру, подсоединяем обработчики событий, отображаем окно и запускаем основной цикл обработки сообщений:

```
(define (main args)
  (let* ([window (make <gtk-window> #:type 'toplevel #:title "Cairo Draw")]
         [da     (gtk-drawing-area-new)])
    (display da) (newline)
    (gtk-container-add window da)

    (connect window 'delete-event  event-delete)
    (connect window 'destroy       event-destroy)
    (connect da     'event         event-draw)

    (gtk-window-set-position      window 'center)
    (gtk-window-set-default-size  window 400 90)

    (show-all window)
    (gtk-main)))
```

Обработчик сообщения event создает контекст Cairo и выполняет процедуру отрисовки содержимого окна:

```
(define (event-draw w event)
  (let ([cr  (gdk-cairo-create (gobject:get-property w 'window))])
    (do-draw cr)
    (CAI:cairo-destroy cr))
  #f)
```

Содержимое окна GTK заполняется точно таким же образом как и любой другой контекст Cairo:

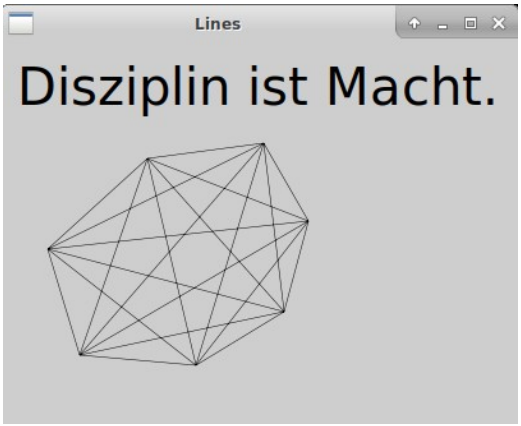
```
(define (do-draw cr)
  (CAI:cairo-set-source-rgb   cr 0 0 0)
  (CAI:cairo-select-font-face cr "Sans" 'normal 'normal)
  (CAI:cairo-set-font-size    cr 40.0)
  (CAI:cairo-move-to          cr 10.0 50.0)
  (CAI:cairo-show-text        cr "Disziplin ist Macht.")
  )
```

Basic drawing in Cairo(основы рисования)

Рисуем линии.

Эта программа вызвала у меня тоже много трудностей, непонятно было как создавать маску сообщений(event-mask), чтобы добавить ее к объекту gtk-drawing-area, что бы он имел возможность получать сообщения от мыши. Так же надо было расшифровать, что же мы получаем в сообщении event и выделить там координаты положения мыши и номер нажатой клавиши. Справившись с этими трудностями я получил программу: **05_lines.scm**.

Запустив которую мы можем нажимая в области рисования левой кнопкой мыши сформировать сетку из линий, нажатие правой клавиши сбрасывает формируемую сетку.



Код создающий окно, с возможностью получать сообщения от мыши для da — **GtkDrawingArea**:

```
(define (main args)
  (let* ([window (make <gtk-window> #:type 'toplevel #:title "Lines")]
         [da     (gtk-drawing-area-new)])
    (gtk-container-add window da)

    (gtk-widget-add-events da (make <gdk-event-mask> #:value 'button-press-mask))

    (connect window 'delete-event              event-delete)
    (connect window 'destroy                   event-destroy)
    (connect da         'event                 event-draw)
    (connect da         'button-press-event    event-clicked)

    (gtk-window-set-position      window 'center)
    (gtk-window-set-default-size  window 400 300)

    (show-all window)
    (gtk-main)
    ))
```

Обработчик сообщения от мыши:

```
(define (event-clicked window event)
  (let* ([event-vec (gdk-event->vector event)]
         [button  (vector-ref event-vec 7)]
         [x       (vector-ref event-vec 4)]
         [y       (vector-ref event-vec 5)]
         )
    (case button
      [(1) (set! glob (append glob (list (list x y)))) (gtk-widget-queue-draw window)];;left mouse
      [(3) (set! glob '())  (gtk-widget-queue-draw window)]           ;;right mouse - clean
      ))
  #t)
```

Как видно, объект сообщения мы расшифровываем как вектор. В биндинге есть функция для получения координат из сообщения, но вот отдельной функции получения номера нажатия клавиши я не нашел. Впрочем полученного вектора более чем достаточно!

Процедуру отрисовки сетки описывать не буду, она не интересна(хотя на мой взгляд, автор туториала сделал цикл обработки излишним, у него координаты линий обрабатываются по 2 раза.)

Заполнять(fill) и обводить(stroke).

Программа **06_fill_stroke.scm** является примером заполнения поверхностей путем последовательного применения методов отображения (таких как fill и stroke) источника(у нас это цвет) с помощью построенной маски.

```
(define (do-draw cr widget)
  (let ([win (gtk-widget-get-toplevel widget)])
    (call-with-values
       (lambda () (gtk-window-get-size win))  ;; set w and h
      (lambda (w h)
        (CAI:cairo-set-line-width  cr 9)
        (CAI:cairo-set-source-rgb  cr 0.69 0.19 0)
        (CAI:cairo-translate       cr (/ w 2) (/ h 2))
        (CAI:cairo-arc             cr 0 0 50 0 (* 2 pi))
        ;;(CAI:cairo-stroke          cr)
        (CAI:cairo-stroke-preserve cr)

        ;;(CAI:cairo-arc             cr 0 0 50 0 (* 2 pi))
        (CAI:cairo-set-source-rgb  cr 0.3 0.4 0.6)
        (CAI:cairo-fill            cr)
          ))))
```

Здесь мы получаем от объекта gtk его размеры **(gtk-window-get-size win)** и рисуем посредине окна круг **(CAI:cairo-arc cr 0 0 50 0 (* 2 pi))** предварительно переместив центр координат командой **(CAI:cairo-translate cr (/ w 2) (/ h 2))**.

Чтобы использовать одну и туже маску дважды мы применяем метод:

(CAI:cairo-stroke-preserve cr)

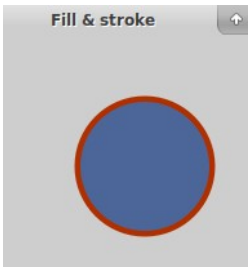
для оформления контура и для заполнения, что для больших масок существенно сократит время работы.

Хотя ничто не мешает нам использовать:

(CAI:cairo-stroke cr)

и затем вновь построить маску с помощью:

(CAI:cairo-arc cr 0 0 50 0 (* 2 pi))



Использование различных типов перьев.

Программа **07_pen_dashes.scm** демонстрирует работу с перьями, которые используются для построения линий в Cairo.

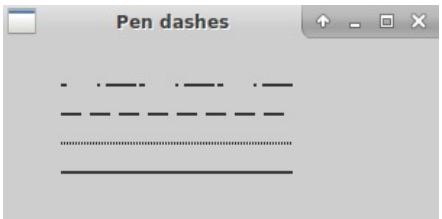
Тип пера устанавливается командой:

(CAI:cairo-set-dash cr dashed1 0)

вот пример её использования:

```
(define (do-draw cr widget)
  (let ([dashed1 '#(4.0 21.0 2.0)]
        [dashed2 '#(14.0 6.0)]
        [dashed3 '#(1.0)]
        [dashed4 '#()])
    (CAI:cairo-set-line-width cr 1.5)

    (CAI:cairo-set-dash cr dashed1 0)
    (CAI:cairo-move-to cr 40 30)
    (CAI:cairo-line-to cr 200 30)
    (CAI:cairo-stroke cr)
```

последний тип **dashed4** устанавливает сплошной тип линии.



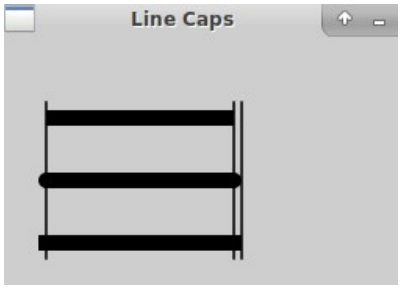
Стили окончания линий.

Программа **08_line_caps.scm** демонстрирует возможность установки стилей окончания линий.

Для этого используется команда:

```
(CAI:cairo-set-line-cap      cr 'butt) ;;см: /guile-cairo/guile-cairo-enum-types.c
(CAI:cairo-move-to           cr 30  50)
(CAI:cairo-line-to           cr 150 50)
(CAI:cairo-stroke            cr)
```

Можно устанавливать типы: **butt, round, square**:



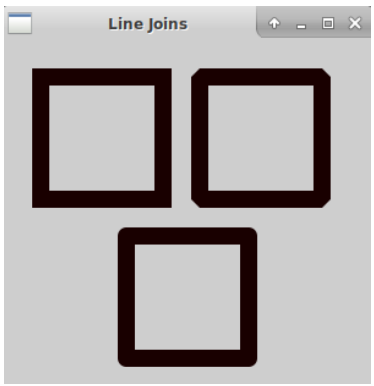
Стили объединения линий.

Стилей объединения линий всего три: bevel — срезанный, round — скругленый, miter — заостренный. Пример работы с ними находиться в файле: **09_line_joins.scm**. Для установки определенного стиля используется функция: **(CAI:cairo-set-line-join cr 'miter)**

работа происходит следующим образом:

```
(CAI:cairo-set-source-rgb  cr 0.1 0 0)

(CAI:cairo-rectangle       cr 30 30 100 100)
(CAI:cairo-set-line-width  cr 14)
(CAI:cairo-set-line-join   cr 'miter)   ;;см: /guile-cairo/guile-cairo-enum-types.c
(CAI:cairo-stroke          cr)
```

В результате получаем 3 квадрата с разными углами:




```
(CAI:cairo-translate       cr 0   100)

(CAI:cairo-rectangle       cr 20  20 120 80)
(CAI:cairo-rectangle       cr 180 20 80  80)
(CAI:cairo-stroke-preserve cr)
```

```
(CAI:cairo-set-source-rgb  cr 0.6 0.6 0.6)
(CAI:cairo-fill          cr)

(CAI:cairo-set-source-rgb  cr 0  0  0)
(CAI:cairo-arc           cr 330 60 40 0 (* 2 pi))
(CAI:cairo-stroke-preserve cr)
(CAI:cairo-set-source-rgb  cr 0.6 0.6 0.6)
(CAI:cairo-fill          cr)
```

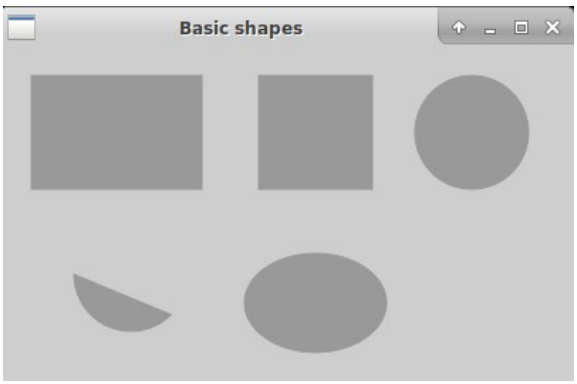
Shapes and fills(Фигуры и заполнения)

Basic Shapes (Основные фигуры)

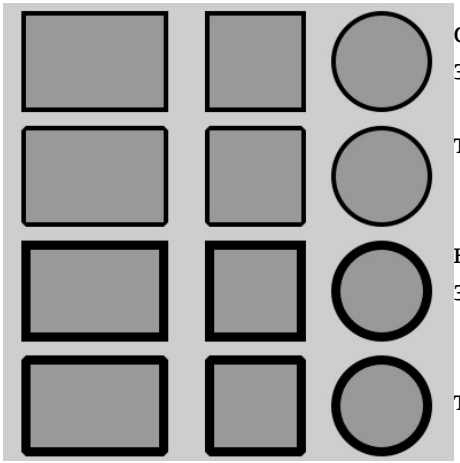
Пример построения базовых фигур можно найти в файле: **10_basic_shapes.scm**
Фигуры строятся с помощью операций:

```
(CAI:cairo-rectangle      cr 20  20 120 80) ;; строим прямоугольник
(CAI:cairo-rectangle      cr 180 20 80  80) ;; строим квадрат
(CAI:cairo-stroke-preserve cr)              ;; очерчиваем контур фигур сохраняя маску
(CAI:cairo-fill          cr)                ;; заполняем внутренне содержание по маске

(CAI:cairo-arc           cr 330 60 40 0 (* 2 pi))  ;; строим круг
(CAI:cairo-stroke-preserve cr)
(CAI:cairo-fill          cr)

(CAI:cairo-arc           cr 90 160 40 (/ pi 4)  pi)   ;; строим полукруг
(CAI:cairo-close-path     cr)
(CAI:cairo-stroke-preserve cr)
(CAI:cairo-fill          cr)

(CAI:cairo-translate      cr 220 180)       ;; строим овал
(CAI:cairo-scale          cr 1 0.7)
(CAI:cairo-arc           cr 0 0 50 0 (* 2 pi))
```

В результате получаем следующую поверхность:



Здесь не очень четко видно какая разница заполнять сначала и потом рисовать контур, или наоборот, сначала рисовать контур, а потом заполнять фигуру. Поэтому я немного переделал пример, что бы более четко подчеркнуть разницу в результатах при изменении последовательности операций. См файл: **10a_basic_shapes.scm**



сначала cairo-stroke-preserve
затем cairo-fill

тоже но с cairo-set-line-join cr 'bevel

начала cairo-fill-preserve
затем cairo-stroke

тоже но с cairo-set-line-join cr 'bevel

Другие фигуры можно рисовать комбинацией использования базовых примитивов, команд линий и команды замыкания пути **(cairo-close-path cr)**, пример приведен в файле: **11_other_shapes.scm**

например строим сложный контур, по списку точек:

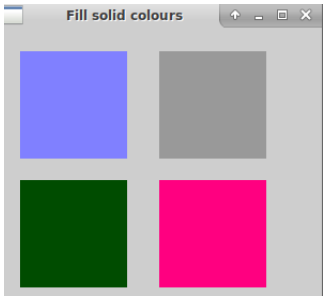
```
(define points '((0 85)   (75 75)  (100 10)  (125 75)  (200 85)  (150 125) (160 190)
                 (100 150)  (40  190) (50  125)  (0   85)))

  (let ([t (car points)])
      (CAI:cairo-move-to      cr   (car t) (cadr t)) )
 (fold (lambda (coord unused)
        (CAI:cairo-line-to  cr (car coord) (cadr coord)) 0)
        0 (cdr points))
  (CAI:cairo-close-path     cr)
  (CAI:cairo-stroke-preserve cr)
  (CAI:cairo-fill           cr)
```

Fills(Заполнение)

Заполнять внутренность фигур можно: сплошным цветом, шаблоном и градиентом.

Заполнение сплошным цветом.
Смотрим файл: **12_fills_solid_colours.scm**
В нем мы рисуем квадраты:

```
(CAI:cairo-set-source-rgb  cr 0.5 0.5 1)
(CAI:cairo-rectangle       cr 20 20 100 100)
(CAI:cairo-fill            cr)
```



Заполнение по шаблону.

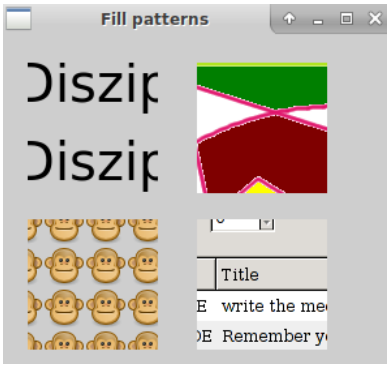
Пример приведён в файле: **13_fills_patterns.scm**
Первым делом загружаем поверхности шаблонов из файлов рисунков:

```
(define (create-surfaces)
  (set! surface1 (CAI:cairo-image-surface-create-from-png "cairo1.png"))
  (set! surface2 (CAI:cairo-image-surface-create-from-png "test1.png"))
  (set! surface3 (CAI:cairo-image-surface-create-from-png "face-monkey.png"))
  (set! surface4 (CAI:cairo-image-surface-create-from-png "orage_eventlist.png"))
  )
```

После этого можно использовать их при рисовании:

```
(define (do-draw cr widget)

 ;; создаем шаблоны на основе поверхностей
  (let ([pattern1 (CAI:cairo-pattern-create-for-surface surface1)]
        [pattern2 (CAI:cairo-pattern-create-for-surface surface2)]
        [pattern3 (CAI:cairo-pattern-create-for-surface surface3)]
        [pattern4 (CAI:cairo-pattern-create-for-surface surface4)]
        )

;; устанавливаем шаблон в качестве источника.
    (CAI:cairo-set-source        cr pattern1)

;; при необходимости говорим cairo, что шаблон надо повторять
    (CAI:cairo-pattern-set-extend (CAI:cairo-get-source cr) 'repeat)
    (CAI:cairo-rectangle       cr 20 20 100 100)
    (CAI:cairo-fill            cr)
```

типы расширения шаблонов можно найти в файле: **guile-cairo-enum-types.c**



Gradients(Заполнение по градиенту)

Линейный градиент.

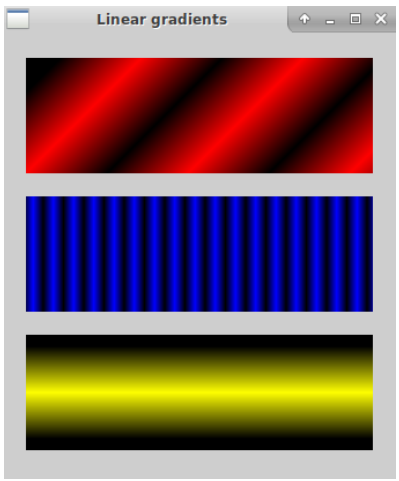
Пример создания и использования линейных градиентов приведен в файле: **14_linear_gradients.scm**

Создаем паттерн:

```
(define (create-pattern)
  (set! pat1 (CAI:cairo-pattern-create-linear 0.0    0.0 350.0 350.0))
  (do ((j 0.1 (+ j 0.1)) (count 1 (1+ count)))
      ((>= j 1))
    (if (odd? count)
        (CAI:cairo-pattern-add-color-stop-rgb pat1 j 0 0 0)
        (CAI:cairo-pattern-add-color-stop-rgb pat1 j 1 0 0)
    ))
```

Устанавливаем его в качестве источника, формируем маску(rectangle), производим заполнение(fill) формируемой поврехности(в ранее созданном контексте cr):

```
  (CAI:cairo-rectangle       cr 20 20 300 100)
  (CAI:cairo-set-source      cr pat1)
  (CAI:cairo-fill            cr)
```



Радиальный градиент.

Пример создания и работы с радиальными градиентами дан в файле: **15_radial_gradients.scm**.

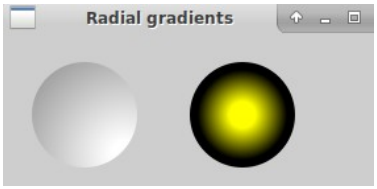
Действие происходит аналогично созданию и использованию линейного градиента:

Формируем паттерн:

```
(define (create-pattern)
  (set! pat1 (CAI:cairo-pattern-create-radial 30  30  10  30 30 90))
  (CAI:cairo-pattern-add-color-stop-rgba pat1  0   1   1   1  1)
  (CAI:cairo-pattern-add-color-stop-rgba pat1  1 0.6 0.6 0.6  1)
```

Устанавливаем его в качестве источника и используем:

```
  (CAI:cairo-set-source      cr pat1)
  (CAI:cairo-arc             cr 0  0 40  0 (* 2 pi))
  (CAI:cairo-fill            cr)
```



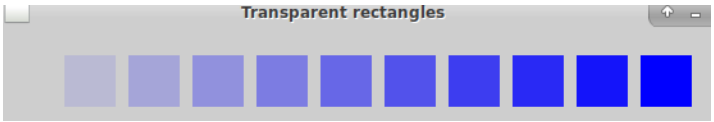
Transparency(Прозрачность)

Прозрачные прямоугольники.

Пример установки прозрачности получаемых рисунков можно найти в файле: **16_transparent_rectangles.scm**.

В котором рисуется 10 прямоугольников различной прозрачности, изменение прозрачности достигается путем изменения альфа канала устанавливаемом источнике цвета.

```
  (do ((i 1 (1+ i)))
     ((> i 10))
    (CAI:cairo-set-source-rgba cr 0  0  1  (* i 0.1))
    (CAI:cairo-rectangle       cr (* 50 i) 20 40 40)
    (CAI:cairo-fill            cr)
```



Puff Effect

В файле: **17_transparent_puff_effect.scm** находиться пример программы с так называемым пуфф эффектом: когда по мере увеличения размера отображаемого предмета(текста) увеличивается его прозрачность и в конечном счете не смотря на свои размеры предмет исчезает. Получить скрин каст такого эффекта тяжело, поэтому я слегка

модифицировал эту программу и добавил остановку и запуск этого эффекта по нажатию левой клавиши мыши, а правая клавиша возобновляет эффект сначала.

Кстати говоря в этих примерах можно найти как работать с таймером в gtk из guile.
В (main) запускаем таймер и устанавливаем ему задержку:

(g-timeout-add 14 (lambda () (time-handler window)))

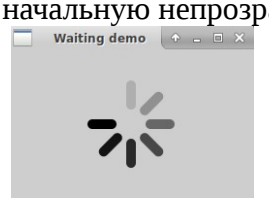
и далее возобновляем работу таймера в обработчике, попутно выполняя периодически требуемую работу:

```
(define (time-handler w)
  (if (glob? glob)
     (if (glob-timer glob)
        (begin
          (gtk-widget-queue-draw w)
          #t)
        #f)
     #f) )
```

Waiting demo

Еще один пример использования прозрачности, в виде цикла бесконечного ожидания можно найти в файле: **18_transparent_waiting_demo.scm**

Программа отрисовывает 8 линий с различной степенью прозрачности, меняя начальную непрозрачную линию в цикле.



Compositing(Композиция, объединение)

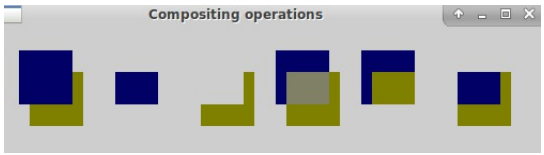
Композиция это комбинирование визуальных элементов из различных изображений. Для проведения композиции доступны операции композиции. Все их можно найти в файле: **guile-cairo-enum-types.c**
Примеры использования операций композиции приведены в файле: **19_compositing_operation.scm**

В файле сформирован список операций для демонстрации работы композиции:

```
 (let ([opers '(dest-over dest-in out add atop dest-atop)]
```

далее в цикле выбирается одна из операций и выполняется процедура композиции для демонстрации работы данной операции (op):

```
(define  (one-draw cr x y w h op)

;; вначале формируем промежуточные поверхности и их контексты
  (let* ([first     (CAI:cairo-surface-create-similar (CAI:cairo-get-target cr) 'color-alpha w h)]
         [second    (CAI:cairo-surface-create-similar (CAI:cairo-get-target cr) 'color-alpha w h)]
         [first-cr  (CAI:cairo-create first)]
         [second-cr (CAI:cairo-create second)])

;; формируем изображение на первой промежуточной поверхности
    (CAI:cairo-set-source-rgb first-cr  0  0  0.4)
    (CAI:cairo-rectangle      first-cr  x  y  50 50)
    (CAI:cairo-fill           first-cr)

;; формируем изображение на второй промежуточной поверхности
    (CAI:cairo-set-source-rgb second-cr  0.5  0.5  0)
    (CAI:cairo-rectangle      second-cr  (+ x 10)  (+ y 20)  50 50)
    (CAI:cairo-fill           second-cr)

;; устанавливаем операцию композиции для ПЕРВОЙ промежуточной поверхности
    (CAI:cairo-set-operator       first-cr op)
;; устанавливаем в качестве источника вторую промежуточную поверхность
    (CAI:cairo-set-source-surface  first-cr second  0 0)
;; производим композицию в соответствии с выбранной операцией
    (CAI:cairo-paint              first-cr)

;; полученный результат из первой промежуточной поверхности копируем в наш контекст
    (CAI:cairo-set-source-surface  cr   first  0 0)
    (CAI:cairo-paint               cr)

;; удаляем промежуточные контексты и поверхности
    (CAI:cairo-surface-destroy     first)
    (CAI:cairo-surface-destroy     second)

    (CAI:cairo-destroy             first-cr)
    (CAI:cairo-destroy             second-cr)
    ))
```



Clipping and masking

Clipping(Обрезка изображения)

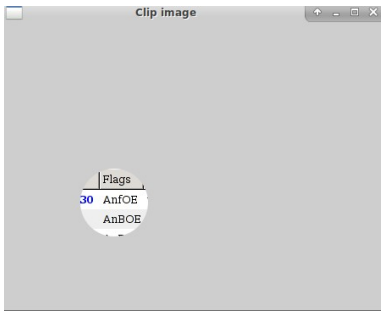
Clipping это ограничение области рисования. Пример использования этого эффекта приведен в файле: **20_clipping.scm**

Вначале загружаем файл и формируем поверхность которая в дальнейшем будет использована в качестве источника:

```
(let ([image (CAI:cairo-image-surface-create-from-png "orage_eventlist.png")])
  (set! glob (make-glob image)))
```

Далее в программе рисования:

```
;;Устанавливаем эту поверхность в качестве источника
   (CAI:cairo-set-source-surface  cr (glob-image glob) 1 1)
;;формируем маску( у нас это круг)
    (CAI:cairo-arc              cr pos-x pos-y radius 0 (* pi 2))
;; говорим cairo что использовать маску надо как область отсечения
    (CAI:cairo-clip             cr)
;; выполняем рисование источника на поверхность в соответствии с установленной областью отсечения
    (CAI:cairo-paint            cr)
```



Mask(Установка и использование маски)

Прежде чем источник будет перенесен на поверхность, он может быть подвергнут фильтрации. Построенная маска может использоваться как фильтр. Маска определяет где источник будет применен, а где нет. Непрозрачные части маски позволяют применить источник, а прозрачные не позволяют копировать источник на поверхность.

Пример использования изображения в качестве маски можно найти в файле: **21_mask.scm**.

Сначала мы загружаем изображение и формируем поверхность которая в дальнейшем будет использована как маска:

```
(let ([image (CAI:cairo-image-surface-create-from-png "cairo2.png")])
  (set! glob (make-glob image)))
```

;;В качестве источника указываем цвет(желтый)
(CAI:cairo-set-source-rgb cr 1 1 0)
;; а вот загруженный рисунок устанавливаем в качестве маски
(CAI:cairo-mask-surface cr (glob-image glob) 0 0)
;; и далее просто копируем источник в контексте cr, при этом копирование будет осуществленно с применением установленной маски.
(CAI:cairo-fill cr)

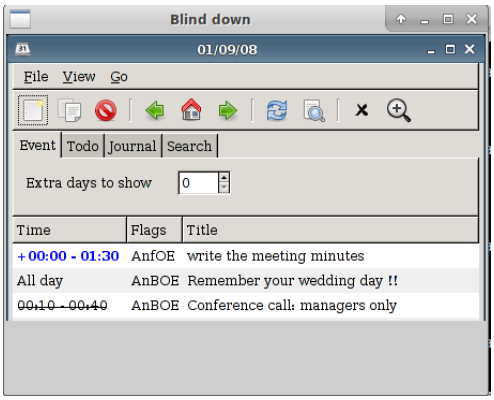
А ну да, в биндинге обнаружился небольшой баг, исправить его крайне легко:

```
;; in file guile-cairo.c ver 1.10
;;SCM_DEFINE_PUBLIC (scm_cairo_mask_surface, "cairo-mask-surface", 2, 0, 0,.
;;                 (SCM ctx, SCM surf, SCM x, SCM y),                  change to:
;;SCM_DEFINE_PUBLIC (scm_cairo_mask_surface, "cairo-mask-surface", 4, 0, 0,
;;                 (SCM ctx, SCM surf, SCM x, SCM y),
;; rebuil and reinstall!!!
```

неправильно указано количество параметров для guile функции.

Blind down effect. (эффект опускающихся жалюзи)

Пример использования маски приведен в файле: **22_blind_down_effect.scm**
Заключается он в постепенном появлении изображения на экране.



Transformation(Преобразования)

Афинные преобразования могут объединтья в ноль или больше преобразований(вращение, масштабирование или сдвиг) и перемещение. Несколько линейных преобразований могут быть объединены в одну матрицу.

Translation

Пример переноса начала координат приведен в файле: **23_translation_simple.scm**
Начало координат переноситься с помощью команды:

(CAI:cairo-translate cr 20 20)

;; далее следуют обычные команды cairo

(CAI:cairo-set-source-rgb cr 0.8 0.3 0.2)
(CAI:cairo-rectangle cr 0 0 30 30)
(CAI:cairo-fill cr)

Shear(Сдвиг)

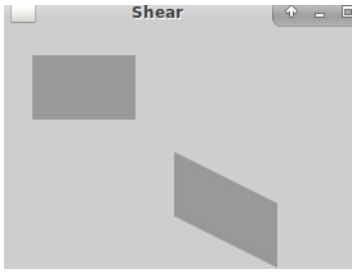
Пример создания эффекта сдвига приведен в файле: **24_translation_shear.scm**

Сдвиг задается применением операции **(CAI:cairo-transform cr matrix)**, как показано ниже:

;;матрица 3x2 задается по столбцам!!!!

(let ([matrix '#2f64((1.0 0.0 0.0) (0.5 1.0 0.0))])
….

(CAI:cairo-transform cr matrix)
(CAI:cairo-rectangle cr 130 30 80 50)
(CAI:cairo-fill cr)



Scaling(Маштабирование)

Пример маштабирования в Cairo приведен в файле: **25_translation_scaling.scm**

Эффект масштабирования достигается применением операции:

(CAI:cairo-scale cr 0.8 0.8)
(CAI:cairo-set-source-rgb cr 0.8 0.8 0.2)
(CAI:cairo-rectangle cr 50 50 90 90)
(CAI:cairo-fill cr)

Isolating transformations(Изоляция преобразований)

Результат операций преобразования имеет свойство накапливаться, для избежания этого эффекта можно сохранять текущую(исходную) трансформацию и восстанавливать ее при необходимости. Сделать это можно применив операции: **(CAI:cairo-save cr)** и **(CAI:cairo-restore cr)**

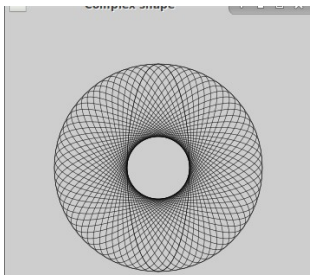
Пример работы с изолированными преобразованиями дан в файле: **26_translation_isolate.scm**

```
(CAI:cairo-save cr)
(CAI:cairo-scale cr 0.6 0.6)
(CAI:cairo-set-source-rgb cr 0.8 0.3 0.2)
(CAI:cairo-rectangle cr 30 30 90 90)
(CAI:cairo-fill cr)
(CAI:cairo-restore cr)
```

Donut (Бублик)

Комплексный пример работы с преобразованиями можно найти в файле: **27_translation_donut.scm**

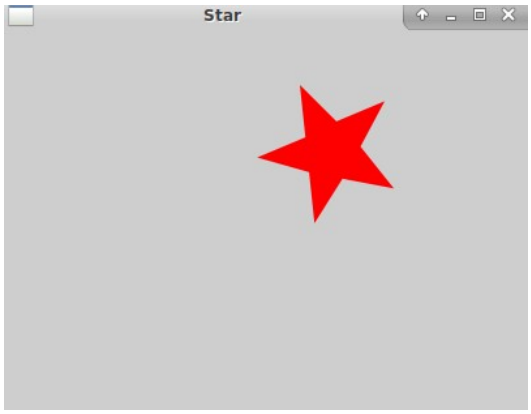
Где выполняется построение бублика, путем получения множества эллипсов, повернутых друг относительно друга.



Star(Звезда)

Еще один комплексный пример работы с преобразованиями дан в файле: **28_translation_star.scm**

В нем по таймеру выполняется вывод звезды в различных положениях в окне, небольшой пример построения анимации с помощью **gtk/cairo**. Вот один кадр из этой анимации.



Естественно я поправил авторский код(у автора координаты звезды выставляются неправильно), впрочем как и цвет звезды. Звезда должна быть красной, это ясно каждому мальчишке родившемуся в СССР.

Текст в Cairo.

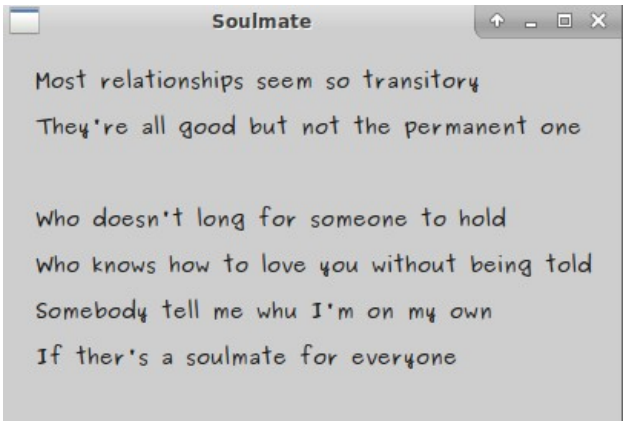
Базовый вывод текста.(Soulmate)

Пример вывода текста был продемонстрирован в одной из первых уроков данного учебника. Более расширенный пример вывода текста находится в файле: **29_text_soulmate.scm**

Последовательность операций при простом выводе такая:

```
;;Устанавливаем источник(у нас это цвет, близкий к черному)
(CAI:cairo-set-source-rgb      cr  0.1 0.1 0.1)
;;Выбираем и загружаем шрифт
 (CAI:cairo-select-font-face    cr "Purisa" 'normal 'bold)
;;Устанавливаем высоту шрифта
 (CAI:cairo-set-font-size       cr  13)

;;Устанавливаем начальную позицию, с которой будет выводиться текст
 (CAI:cairo-move-to             cr  20 30)
;;выводим текст
 (CAI:cairo-show-text           cr "Most relationships seem so transitory")
;; повторяем установку позиции и вывод текста(при необходимости).
 (CAI:cairo-move-to             cr  20 60)
 (CAI:cairo-show-text           cr "They're all good but not the permanent one")
```

Результат будет примерно следующим:



Определение размеров занимаемого выводимым текстом прямоугольника.

На самом деле вывод текста без учета размеров выводимой области, имеет очень ограниченную область применения, для автоматизированного форматирования программа должна знать какую область займет выводимый текст. В Cairo есть функция определения размера которую займет выводимый текст. Это функция **cairo-text-extents**. Пример ее использования дан в файле: **30_text_centered.scm**.

В этой программе выводиться всего одно слово, но перед выводом программа вычисляет размер выводимого текста и на основе этой информации центрирует текст в выводимой области.

```
;; получаем размер занимаемой области
(let* ([extents (CAI:cairo-text-extents cr txt)]
       [txt-w (f64vector-ref extents 2)]
       [txt-h (f64vector-ref extents 3)])
;; на основе полученных данных перемещаем позицию вывода текста
(CAI:cairo-move-to          cr
                            (- x  (/ txt-w  2))
                            (+ y  (/ txt-h  2)))
;;выводим текст
(CAI:cairo-show-text        cr txt)
```

Shaded(Текст с тенью)

Небольшой трюк по выводу текста с тенью дан в примере: **31_text_shaded.scm**

Трюк с тенью заключается в двойном выводе текста, с небольшим смещением, создающим эффект тени.

Вывод текста раскрашенного по шаблону линейного градиента.

Вывод текста с использованием градиентной заливки приведен в файле: **32_text_filled_gradient.scm**

Он не лишь незначительно отличается от предыдущих примеров, тем что мы вместо цвета, выбираем в качестве источника паттерн, созданный на основе градиента:

```
;; создаем линейный шаблон
(let...
   [pat   (CAI:cairo-pattern-create-linear  0  15 0 (* h 0.8))])
 (CAI:cairo-pattern-set-extend          pat   'repeat)
 (CAI:cairo-pattern-add-color-stop-rgb    pat   0.0  1  0.6  0)
 (CAI:cairo-pattern-add-color-stop-rgb    pat   0.5  1  0.3  0)
  ;; устанавливаем позицию для вывода текста
 (CAI:cairo-move-to         cr
                    (- x  (/ txt-w  2))
                    (+ y  (/ txt-h  2)))
  ;; создаем маску для источника на основе выводимого текста
 (CAI:cairo-text-path       cr txt)
  ;;устанавливаем источник
 (CAI:cairo-set-source      cr pat)
  ;;переносим источник на создаваемую поверхность в соответствии с заданной маской.
 (CAI:cairo-fill            cr)
```

Glyphs (Глифы)

Вывод текста с помощью **(CAI:cairo-show-text cr txt)** это самый простейший способ формирования текста, для сложных вариантов(с различными промежутками между буквами) лучше выполнять вывод с помощью функции: **(CAI:cairo-show-glyphs cr vec-glyphs)**

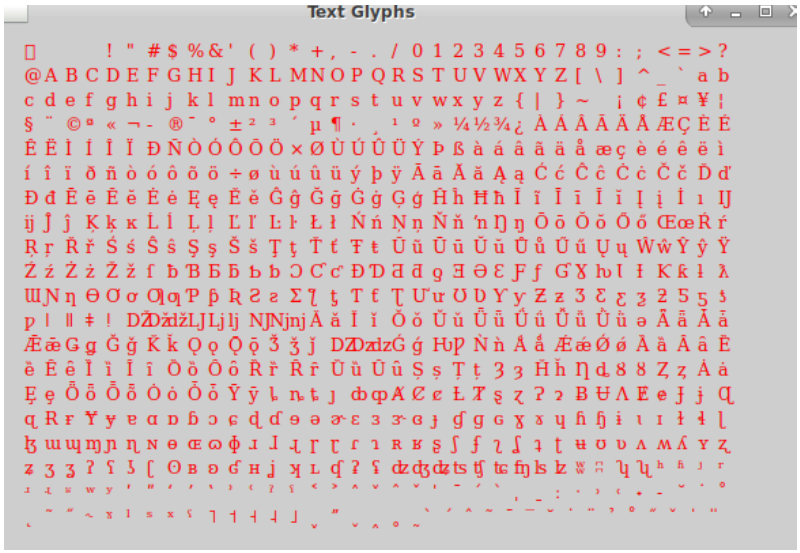
Пример работы с этой функцией приведен в файле: **33_text_show_glyphs.scm**

В ней формируется вектор содержащий положения 20 строк и 35 колонок для различных глифов и затем выводиться с помощью вышеуказанной функции.

```
;; список из кода глифа и положения переводим в вектор
  (let ([vec-glyphs (list->vector (map list->vector lst-glyphs))])
       ;; устанавливаем источник
   (CAI:cairo-set-source-rgb     cr  1   0 0)
```

;;выполняем вывод вектора глифов.

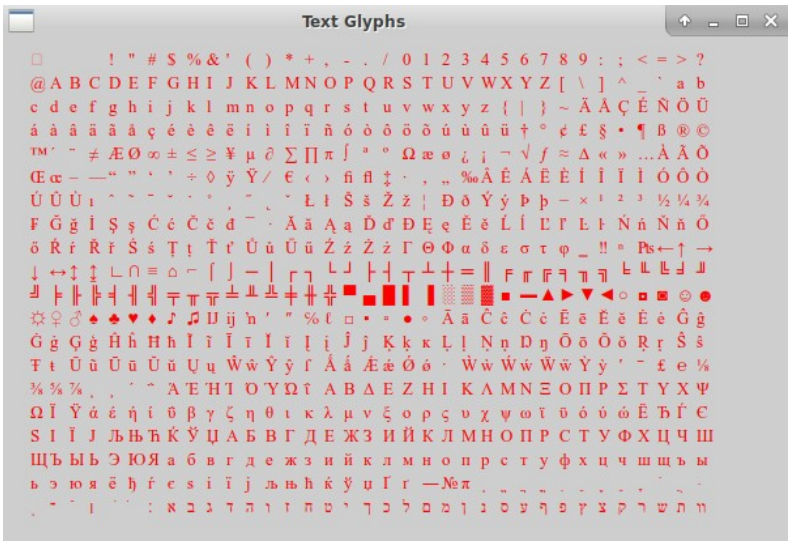
(CAI:cairo-show-glyphs cr vec-glyphs)



Как видно русских букв в этом фонте не видно, вот изменив фонт, на

(CAI:cairo-select-font-face cr "Times New Roman" 'normal 'normal)

в программе: **33a_text_show_glyphs.scm** мы можем видеть глифы русских букв.



Images in Cairo. Изображения.

Отображение изображений.

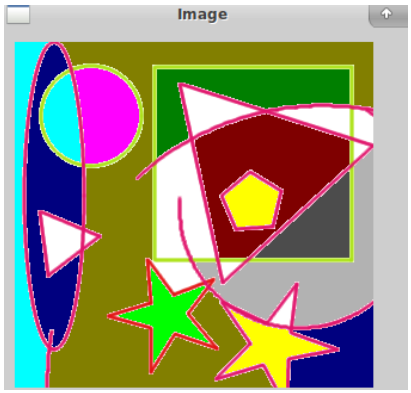
Пример работы с изображениями приведен в файле: **34_image.scm**

Чтобы отобразить изображение в окне или на какой другой поверхности, необходимо сначала его загрузить:

```
(let ([image (CAI:cairo-image-surface-create-from-png "test1.png")])
  (set! glob (make-glob image)))
```

А затем при необходимости отобразить на требуемую поверхность, установить поверхность с изображением в качестве источника и дать команду paint без установки маски.

```
(CAI:cairo-set-source-surface   cr (glob-image glob) 10 10)
(CAI:cairo-paint                cr)
```

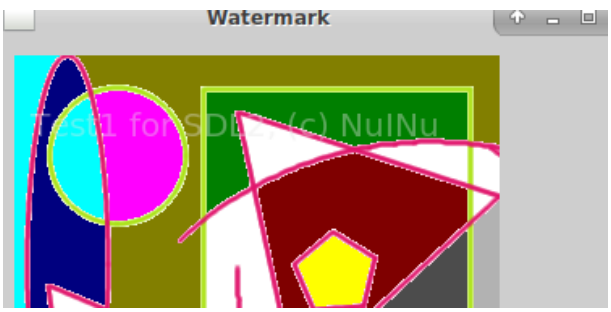


WaterMark(водяные знаки)

Пример рисования на загруженном изображении можно найти в файле: **35_image_watermark.scm.**

В нем мы загружаем изображение, и затем изменяем саму поверхность изображения, отображая на ней некоторый текст. Затем уже эту, преобразованную, поверхность мы можем наблюдать в окне gtk.

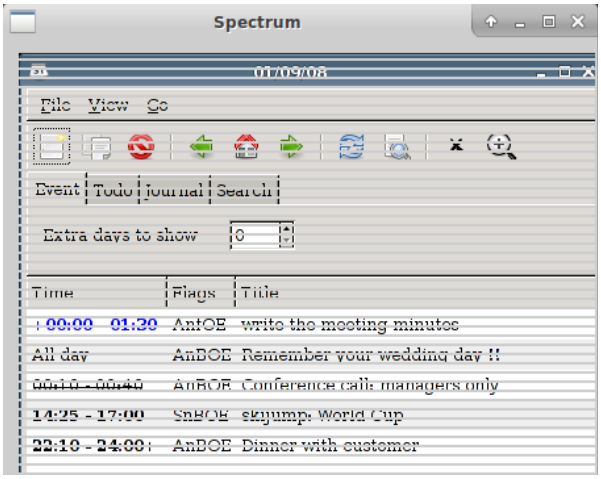
```
;;загружаем изображением
  (let ([image (CAI:cairo-image-surface-create-from-png "test1.png")])
    (set! glob (make-glob image)))

;; рисуем на поверхности изображения «водяной знак»
(define (draw-mark txt)
  (let ([ic (CAI:cairo-create       (glob-image glob))])
    (CAI:cairo-set-font-size        ic  20)
    (CAI:cairo-set-source-rgba      ic  0.9 0.9 0.9 0.4)
    (CAI:cairo-move-to              ic  10  50)
    (CAI:cairo-show-text            ic  txt)
    (CAI:cairo-stroke               ic)
    (CAI:cairo-destroy              ic)
    ))

;; в процедуре отображения все выглядит обычно:
  (CAI:cairo-set-source-surface   cr (glob-image glob) 10 10)
  (CAI:cairo-paint                cr)
```



Спектрум эффект.

Может кто помнит, были такие компьютеры — Спектрумы, на процессоре z80. Вот автор приводит пример видео эффекта из того давнего времени, его реализацию в guile можно найти в файле: **36_image_spectrum_effect.scm**

Пояснять там нечего, эффект он эффект и есть.



Root Window(Корневое окно)

Прозрачное окно.

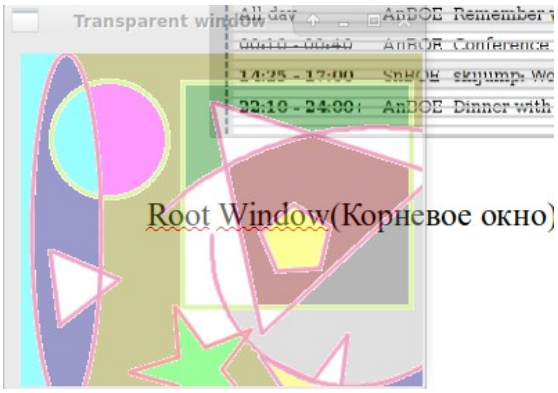
Пример создания прозрачного окна приведен в программе: **37_root_win_transparent.scm**

Код автора использовал функции cairo и gtk, чтобы сделать окно прозрачным, к сожалению это код уже устарел, и функция **gtk_widget_set_visual(win, visual)**; больше не используется.

Поэтому этот пример нельзя уже отнести к полноценной работе Cairo, он больше показывает работу GTK. И чтобы сделать окно прозрачным в нем надо установить свойство opacity после отображения всех окон:

```
(show-all window)
(set  window 'opacity 0.4)
```

всё, окно становиться прозрачным.

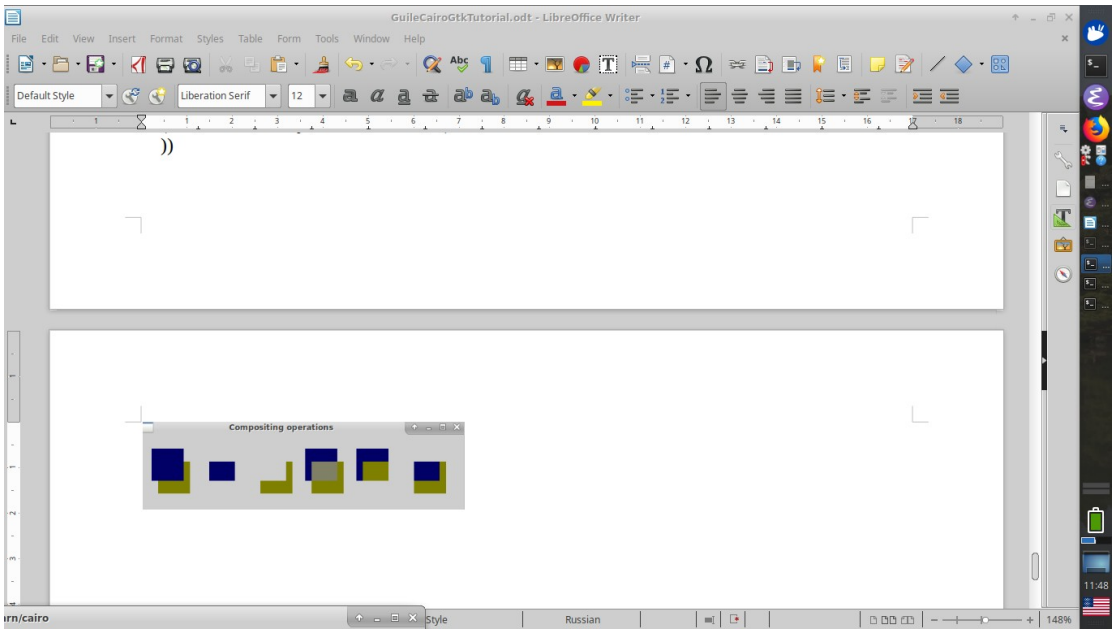


Получение скриншота.

Эту программу можно найти в файле: **38_root_taking_screenshot.scm**
При запуске она делает скриншот экрана и сохраняет его в файл: screenshot.png

```
;; при запуске получаем указатель на корневое окно и получаем его размеры
 (let ([root-win (gdk-get-default-root-window)])
   (receive (x y width height depth) (gdk-window-get-geometry   root-win)
;;создаем Cairo поверхность с аналогичными размерами
          (let* ([surface (CAI:cairo-image-surface-create 'argb32 width height)]
                [cmap    (gdk-drawable-get-colormap    root-win)]
;; получаем данные из главного окна в созданный pixbuf gdk
                [pb      (gdk-pixbuf-get-from-drawable
                         (gdk-pixbuf-new   'rgb #t  8 width height) ;;space for pixbuf
                         root-win
                         (gdk-drawable-get-colormap    root-win)   ;; ооопс!! у нас уже есть cmap
                         0 0
                         0 0
                         width height)]
                [cr      (CAI:cairo-create           surface)])
;; устанавливаем в качестве источника созданный pixbuf gdk с данными из корневого окна
          (gdk-cairo-set-source-pixbuf      cr pb 0 0)
;; выводим их контекст cairo и сохраняем поверхность данного контекста в файл.
          (CAI:cairo-paint                  cr)
          (CAI:cairo-surface-write-to-png   surface "screenshot.png")
```

Вот такой скриншот я получил на своем компьютере:



Окно с терминалом откуда я запускал программу снимающую скриншот, спрятано слева внизу.

Показ сообщения

Файл с данным примером завершает наш краткий учебник, его имя: **39_root_show_message.scm**

Здесь я откровенно схалявил, дело в том что автор опять использует устаревшую функцию set_visual и чего он хочет добиться используя операции cairo я не понял. Поэтому программа работает но функций cairo не использует, используются лишь свойства установленные для окна.

```
(gtk-widget-set-app-paintable    win #t)
(gtk-window-set-type-hint        win 'dock)
(gtk-window-set-keep-below       win #t)
```

Выводиться окно с меткой, без заголовка, как бы сразу прямо поверх экранных обоев. Но насколько нужно окно, которое находиться внизу всех остальных не знаю.

И на этом я завершаю перечисление примеров от ZetCode переведенных мною с Си на Guile. С вами был Гагин Михаил aka NuINu, используйте Cairo и Gtk в Guile, все отлично работает.

Послесловие

После завершения описания всех примеров, Заводной Кот упомянул о том, что неплохо было бы посмотреть как Cairo работает с SDL2. Поскольку у меня было немного времени и желания я написал пару примеров работы в guile с SDL2 и Cairo в качестве backenda.

Cairo & SDL2

Начальный пример показывающий как можно работать с Cairo в качестве Backenda в SDL2 показан в файле: **40_sdl2_base.scm**.

Но чтобы он заработал вам надо иметь биндинг к SDL2 и мой учебник(а точнее исходники из него) по SDL2, т.к пример опирается на расширение биндинга SDL2. Что бы my_addon был виден из примера надо настроить файл **my_config.scm** и указать в нем правильные пути к аддону. Но в принципе, использование функций из аддона не обязательно, в примере используются функции относящиеся к структуре rect, а ее можно использовать и без аддона используя функции биндинга SDL2.

Пример разработан на основе примера **tut03_2.scm** из учебника по SDL2. Но вместо загружаемого изображения пингвина мы формируем отображаемое изображение с помощью Cairo.

```
(set! img (cairo-fill-img ren))
```

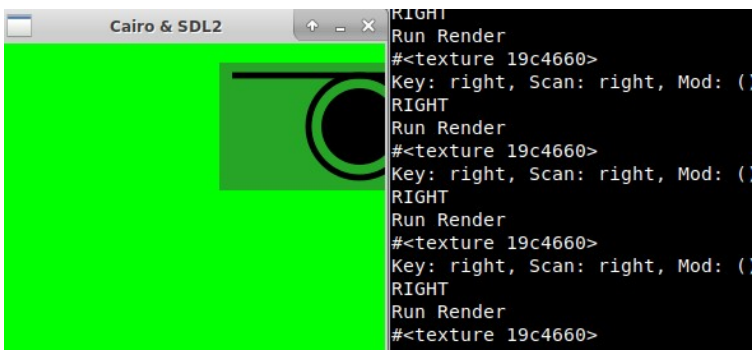
в функции main, заранее установив высоту и ширину изображения:

```
(define img-w 250)
(define img-h 100)
```

Функцию cairo-fill-img определяем следующим образом:

```
(define  (cairo-fill-img ren)

 ;; создаем временную поверхность SDL2
  (let* ([i       (SDL:make-rgb-surface img-w img-h 32)]

;; используя буффер пикселей от поверхности i из SDL2 создаем поверхность Cairo
         [surface (CAI:cairo-image-surface-create-for-data (SDL:surface-pixels i)
                                                           'argb32
                                                           (SDL:surface-width  i)
                                                           (SDL:surface-height i)
                                                           (SDL:surface-pitch  i))]
;; создаем Cairo контекст в котором будем выполнять построение изображения
        [cr (CAI:cairo-create surface)])

;; рисуем изображение в созданном контексте cr
    (CAI:cairo-set-source-rgba cr 0.6 0.6 0.6 0.5)
    (CAI:cairo-rectangle       cr 0 0 250 100)
    (CAI:cairo-fill            cr)

    (CAI:cairo-set-source-rgb  cr 0 0 0)
    (CAI:cairo-set-line-width  cr 5)
    (CAI:cairo-move-to         cr 10  10)
    (CAI:cairo-line-to         cr 210 10)
    (CAI:cairo-stroke          cr)
    (CAI:cairo-arc             cr 110  50 40 0 (* 2 pi))
    (CAI:cairo-stroke          cr)

    (CAI:cairo-arc             cr 110 50 30 0 (* 2 pi))
    (CAI:cairo-fill            cr)

;;всё, буфер пикслей заполнен, строим текстуру из поверхности SDL2
    (let ([tex (SDL:surface->texture ren i)])
;; и уничтожаем промежуточные структуры
      (CAI:cairo-destroy        cr)
      (CAI:cairo-surface-destroy surface)
      (SDL:delete-surface!      i)
      tex)
  ))
```

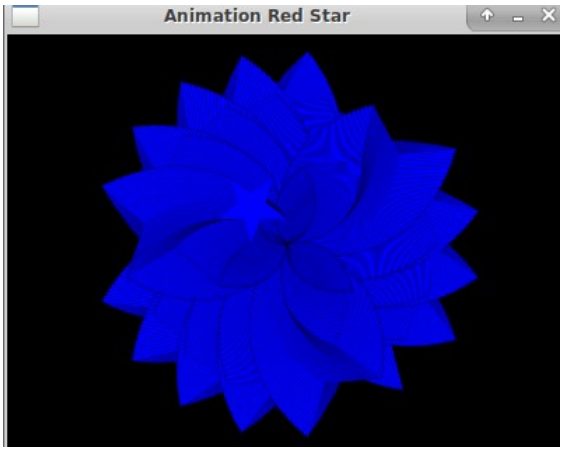
Построив наше простенькое изображение мы можем перемещать его и масштабировать в SDL2.



Cairo & SDL2 расширенный пример.

Сделав предыдущий пример, мне стало понятно, что Cairo и SDL2 отлично совместимы друг с другом. И я задумался над небольшим расширенным примером анимации, демонстрирующей совместную работу Cairo и SDL2. В качестве базы построения анимации я взял пример из Cairo **28_translation_star.scm.**

Неправильная версия этой анимации приведена в файле: **41_sdl2_anim_star.scm**



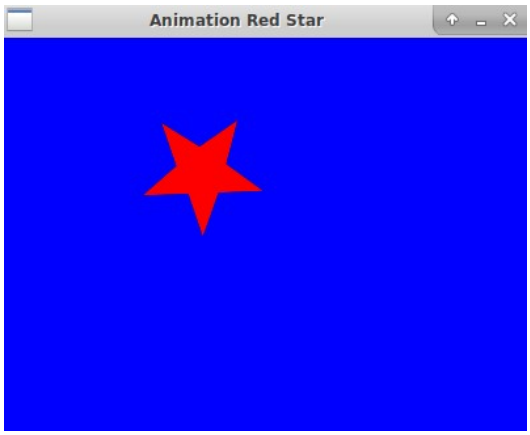
Как видно из примера буфер в котором мы рисуем не очищается. И во вторых, звезда которую мы получаем синяя!!! Исследовав этот вопрос я обнаружил, что биндинги Cairo и SDL2 создают несовместимые типы буферов. В cairo это argb32, а в SDL2 это abgr8888. После длительных попыток сменить тип буфера в SDL2(для Cairo установить другой тип буфера просто невозможно, их там слишком мало), мне удалось это сделать, только внимательно изучив исходный код и биндинга и си код SDL2. Это не очень хорошее решение, но оно работает. Правильным решением создавать сразу правильный тип буфера, но для этого надо будет переписать функцию биндинга: **make-rgb-surface**, она слишком много на себя берет и сильно ограничивая пользователя устанавливая единственный тип формата пикселей буфера. Например для себя я это сделал, добавив подобную функцию к файлу: **my_addon.scm**

```
(define (make-rgb-surface width height depth)
  "Create a new SDL surface with the dimensions WIDTH and HEIGHT and
DEPTH bits per pixel."
  ((@@ (sdl2 surface) wrap-surface)
   (if (eq? (native-endianness) 'big)
      (ffi:sdl-create-rgb-surface 0 width height depth
                                  #x0000ff00
                                  #x00ff0000
                                  #xff000000
                                  #x000000ff)
      (ffi:sdl-create-rgb-surface 0 width height depth
                                  #x00ff0000
                                  #x0000ff00
                                  #x000000ff
                                  #xff000000))))
```

Этот вариант сразу устанавливает совместимый с Cairo тип буфера argb32, хотя наверное более правильным вариантом было бы предоставление права пользователю самому устанавливать маски битов цвета.

Итак правильный, работающий пример рисующий перемещающуюся и вращающуюся красную звезду на синем фоне можно найти в файле: **42_sdl2_anim_star.scm**

```
(define (Render ren s-rect d-rect)
  (let* ([w (mySDL:rect-w s-rect)]
         [h (mySDL:rect-h s-rect)]
    ;; создаем временную поверхность SDL2
         [i      (SDL:make-rgb-surface w h 32)])
   ;; модифицируем формат пикселей для нашей временной поверхности, чтобы он был
   ;; совместим с Cairo
   (set! i (SDL:convert-surface-format i 'argb8888))
   ;; создаем поверхность Cairo  на основе буфера SDL2 и Cairo контекст для рисования
   (let* ([surface (CAI:cairo-image-surface-create-for-data (SDL:surface-pixels i)
                                                'argb32
                                                (SDL:surface-width  i)
                                                (SDL:surface-height i)
                                                (SDL:surface-pitch  i))]
         [cr (CAI:cairo-create surface)])
     ;;рисуем на поверхности с помощью функций SDL2
     ;;( в нашем случае это просто очистка экрана
     (SDL:set-render-draw-color ren 0 0 255 255)
     (SDL:clear-renderer ren)
     ;; рисуем в буфере с помощью Cairo
     (do-draw cr w h)
     ;; создаем текстуру для рисования на экране и удаляем промежуточные структуры.
     (let ([tex (SDL:surface->texture ren i)])
       (CAI:cairo-destroy        cr)
       (CAI:cairo-surface-destroy surface)
       (SDL:delete-surface!      i)
        ;; выполняем вывод на экран
       (mySDL:render-copy ren  tex #:srcrect s-rect #:dstrect d-rect)
       (mySDL:destroy-texture  tex)
       )))
 ;;отобразим изменения
 (SDL:present-renderer ren)
 )
```



Ну вот, теперь то уж точно. ВСЁ!!!