Использование SDL2 в программировании на языке Scheme(Guile).

Введение

В настоящем обзоре я рассмотрю программирование графики и (немного) звука с использованием интерфейсной библиотеки Guile к SDL2, проще говоря биндинга, позволяющему программисту Guile использовать функции SDL2. К сожалению с самим биндингом не поставляются примеры его использования(вернее их всего два), поэтому его использование может вызвать значительные трудности, их преодолению и посвящен данный обзор.

Первое что я обнаружил, что данный биндинг к сожалению не дописан, в нем отсутствуют многие функции, которые необходимы для написания минимально работающих программ. Но есть приятная новость, что он потихоньку дописывается, например в версии 0.3.1 появилась функция set-render-draw-color (хотя надо заметить, что это чуть ли не единственное изменение ;-)). Но это незначительное изменение позволило мне переписать написанное мною для более ранней версии биндинга МИНИМАЛЬНО работающее приложение, использующее биндинг guile-sdl2, без использования моего аддона к данному биндингу. Но аддон(добавление) я все таки буду использовать в дальнейших примерах, хотя бы по причине того чтобы показать, что даже несмотря на отсутствие некоторых функций в основном биндинге, это не причина от него отказываться. Его авторами проделана огромная работа и практически он является полной работоспособной связующей библиотекой для работы с SDL2 из Guile.

Инициализация системы. Первая программа.

В первой программе мы сделаем не много, подключим биндинг и инициализируем подсистему работы с видео, создадим окно, покрасим его и завершим работу. Полный пример можно увидеть в файле: **tut00.scm**

Вначале подключим необходимые библиотеки(предполагая что Guile знает где они установлены):

```
(use-modules ((sdl2) #:prefix SDL:)
             ((sdl2 render) #:prefix SDL:)
             ((sdl2 surface) #:prefix SDL:)
             ((sdl2 video) #:prefix SDL:))
(use-modules ((sdl2 bindings) #:prefix ffi:))
```

Выполняем инициализацию подсистем SDL, в нашем случае это видео система:

```
(SDL:sdl-init '(video))
```

Создаем окно связывая его с переменной win, в котором будем рисовать:

```
(let ((win (SDL:make-window  #:size '(300 350)
                             #:title "Chapter 1"
                             #:position (list ffi:SDL_WINDOWPOS_CENTERED
                                              ffi:SDL_WINDOWPOS_CENTERED)
                             #:show? #t)))…
```

Создаем renderer связанный с нашим окном, который фактически и будет выполнять все операции рисования в окне.

(let ((ren (SDL:make-renderer win)))…

Далее для полученного рендера ren можно выполнять операции рисования, в нашем случае просто очистим его цветом по умолчанию:
для этого установим цвет по умолчанию(слава версии 0.3.1)

(SDL:set-render-draw-color ren 255 0 0 255)

и выполним операцию очистки рендера

(SDL:clear-renderer ren)

Но все эти операции по изменению рендера пока не нашли своего отражения во внутреннем содержании окна, для того чтобы их наконец то увидел внешний наблюдатель надо применить операцию: презентации(отображения)

(SDL:present-renderer ren)

И на этом наше первое приложение окончено, далее идет небольшая задержка и завершение работы.

(SDL:sdl-quit)
(sleep 5)

Все наше действо мы обернули обработкой исключительных ситуаций, которые могут возникнуть в процессе работы

(catch 'sdl-error ...

Написание дополнения к уже существующему биндингу.

Можно конечно включить наши дополнения в уже существующий биндинг и после их проверки отправить запрос на включение в биндинг. Так наверное было бы правильно. Но поскольку целью данной работы было не совершенствование биндинга а попытка понять как он работает и работает ли он вообще, я избрал не очень верный но надежный путь, написания своего собственного модуля, который содержит самую разношерстную коллекцию из функций, которые мне пригодились или могли пригодиться при использовании исследуемого биндинга. Я назвал этот модуль my_addon и заместил в файле **./my_addon/my_addon.scm**. Первой функцией которая там определена как раз и была функция установки цвета рисования:

Определяем функцию-связку с помощью foreigin function interface (FFI), позволяющему связать любую внешнюю функцию из внешней загружаемой библиотеки с именем в Scheme.
Внешняя библиотека здесь скрыта в макросе **sdl-func**, который определен в модуле **sdl2 bindings.**

```
(define sdl-set-render-draw-color
  ( (@@ (sdl2 bindings) sdl-func)
    int "SDL_SetRenderDrawColor" (list '* uint8 uint8 uint8 uint8)))
```

Как видите испольование FFI просто и понятно. Вначале указали тип возвращаемого значения, затем имя функции во внешней загружаемой библиотеки, а затем описали список типов параметров которые она принимает. '* - это указатель, в нашем случае это указатель на структуру рендера, который мы получили ранее вызвав функцию **make-renderer**. В системе Guile он храниться не просто как указатель, а как обернутый указатель, что позволяет выполнять минимально необходимые проверки при вызове данных функций, на соответствие типа передаваемых данных.

Далее описываем функцию использующую определенную через FFI функцию, это необходимо сделать, чтобы во первых производить необходимые проверки, на корректность вызова и во-вторых для удобочитаемости основного кода программы, потому что вызов функции распаковки обернутого указателя **(@@ (sdl2 render) unwrap-renderer) ren)** необходимой для передачи чистого указателя в вызываемую функцию, выглядит немного страшновато.

```
(define (set-render-draw-color ren r g b a)
  (if (renderer? ren)
     (sdl-set-render-draw-color ((@@ (sdl2 render) unwrap-renderer) ren) r g b a)
     (sdl-error "set-render-draw-color" "bad render type")))
```

Далее в секции экспорта модуля мы указываем функцию **set-render-draw-color** и можем пользоваться ей в создаваемых нами программах.

Чтобы Guile смог найти наш модуль мы должны указать к нему путь, это можно сделать разными путями: в частности установкой переменной окружения. Но можно и записать этот путь в файл, но для того что бы вы могли подкорректировать только в одном месте и не изменять для каждого примера я вынес задание этого пути в отдельный файл: **./tutorial/ my_config.scm**

```
(eval-when (load compile)
   (define base-path (string-append (getenv "HOME") "/work/guile/sdl2/")))
(define work-path (string-append base-path "tutorial/"))
(define lib-path (string-append base-path "my_addon/"))
(add-to-load-path lib-path)
```

И теперь подключая этот файл к каждому нашему примеру мы даем Guile информацию где искать дополнительные модули.

Пример использования нашего модуля описан в файле: **./tutorial/tut00a.scm**

Подключаем файл с путями:

```
(eval-when (compile load)
   (load "my_config.scm"))
```

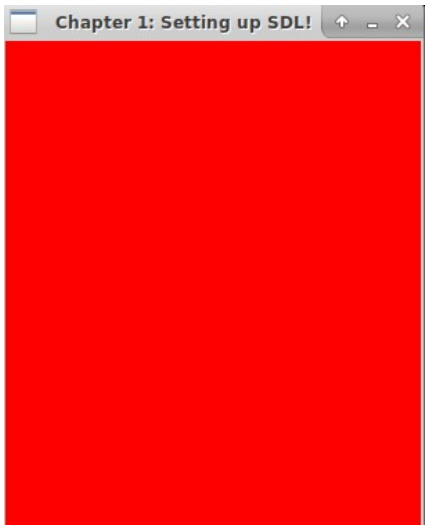
Подгружаем дополнительный модуль:

```
(use-modules ((my_addon) #:prefix mySDL:))
```

И где то в теле программы используем функции определенные в нем:

```
(mySDL:set-render-draw-color ren 255 0 0 255)
….
(mySDL:sdl-delay 5000)
```

Вот такое изображение мы можем видеть, в результате выполнения нашей программы:



Получение и установка значений из(в) библиотеки SDL2.

Вторая наша программа будет не очень полезной, но уж поскольку она есть, приведем и ее. В ней предпринимается попытка устанавливать положение окна и его размер, а также получать некоторую информацию об области отображения. Файл с программой называется: tut01.scm

В этой программе мы также указываем путь к нашей добавочной библиотеке:

```
(eval-when (compile load)
   (load "my_config.scm"))
```

И подключаем ее:

```
(use-modules ((my_addon) #:prefix mySDL:))
```

После создания окна:

```
(win (SDL:make-window #:size (list sizeX sizeY) #:title "Server"
                                #:position (list posX posY)    ))
```

и рендера

(ren (SDL:make-renderer win '(accelerated))) (этот флаг не обязателен и ставиться по умолчанию, приведен, для того чтобы указать как ставятся флаги.)

отображаем в рендере фон по умолчанию(т.е без установки цвета рисования, он черный):

```
(SDL:clear-renderer ren)
(SDL:present-renderer ren)
```

Получаем информацию об окне:

```
(format #t "Window ID: ~d~%"  (SDL:window-id win))
(display "Size window: ")          (display (SDL:window-size win)) (newline)
(display "Logical Size render: ")  (display (mySDL:render-get-logical-size ren))
(newline)
```

Далее просто балуемся изменяя логический размер рендера(не знаю зачем это надо) и положение окна:

```
(mySDL:render-set-logical-size ren sizeXn sizeYn)
(SDL:set-window-position! win  (list posXn posYn))
```

изменяем размер окна:

```
(SDL:set-window-size! win  (list sizeXn sizeYn))
```

Наблюдаем за тем как окно прыгает и как его плющит, на чем программа и завершается.

Получение информации из SDL2(углубляемся в FFI).

В предыдущем разделе приведена функция получающая данные из SDL2: **mySDL:render-get-logical-size ren**. Примечательна она тем, что ее Си-шный прототип получает данные из SDL2, не через возвращаемое значение, а через параметры указатели передаваемые в SDL2.

Её определение через FFI указывает, что в SDL передаются три указателя, и ничего, в качестве возврата не ожидается.

```
(define sdl-render-get-logical-size
   ( (@@ (sdl2 bindings) sdl-func)
     void "SDL_RenderGetLogicalSize" (list '* '* '*)))
```

на самом деле мы ожидаем что SDL2 заполнит два буфера второго и третьего параметра, размер которых соответствует си определению функции:

```
;void SDL_RenderGetLogicalSize(SDL_Renderer * renderer, int *w, int *h)
```

Буфера под эти параметры выделяются и обрабатываются в отдельной функции:

```
(define (%get-coords-ren ren proc)
  (let ((bv (make-bytevector (* 2 (sizeof int)) 0)))
    (proc ((@@ (sdl2 render) unwrap-renderer) ren)
        (bytevector->pointer bv)
        (bytevector->pointer bv (sizeof int)))
    (bytevector->sint-list bv (native-endianness) (sizeof int))))
```

Эта функция принимая в качестве параметра функцию **sdl-render-get-logical-size**(или ей подобную) выделяет память вызовом **make-bytevector**, соответствующий размеру ожидаемых

данных. Выполняет вызов переданной функции, при этом создавая указатели выделенные буфера и передавая их в качестве параметров функции:

(bytevector->pointer bv)
(bytevector->pointer bv (sizeof int))

а также «развертывая» обернутый указатель на рендер:

(@@ (sdl2 render) unwrap-renderer) ren)

который является первым параметром-указателем в вызываемой функции.

После возврата функции, пользуясь тем что мы знаем что возвращаемые значения имеют один тип пользуемся функцией создающей список из вектора значений типа **sint**:

(bytevector->sint-list bv (native-endianness) (sizeof int))

Этим приемом выделения буфера часто придется пользоваться если вы будете писать свои функции для взаимодействия с Си-шными функциями, т. к. многие из них ожидают, что вы передадите им готовый буфер, определенного размера(в нашем случае два буфера, размером **int**), который они с удовольствием заполнят своими данными. Некоторые же из этих Си функций, в порыве особого усердия и рвения, могут и заполнить буфер и сверх выделенной им меры, будьте к этому готовы.

И наконец сама функция, которую мы видим и применяем для получения данных из SDL2:

```
(define (render-get-logical-size ren)
  (if (renderer? ren)
     (%get-coords-ren ren sdl-render-get-logical-size)
     (sdl-error "render-get-logical-size" "bad render type")))
```

Обработка сообщений в SDL2.

Следующая наша программа: **tut02.scm** представляет собой пример обработки сообщений(в нашем случае клавиатурных) в SDL2. В некотором смысле её можно представить как макет простейшей 2D игры. Программа должна организовать обработку клавиатурных сообщений, переводя их в перемещения синего квадратика в окне зеленого цвета.

В данной программе уже мы пытаемся как то структурировать программу, выделяя в ней фазу инициализации:

(ren (InitEverything posX posY sizeX sizeY))

создающую рендер и фазу взаимодействия с пользователем, обрабатывающую сообщения от пользователя:

(RunGame ren rect)

А также функцию отображения текущего состояния всех спрайтов и других видимых объектов в рендере(в нашем случае это всего один объект :-)):

(Render ren rect)

В ней мы еще больше используем функций из модуля my_addon.

Но в начале я расскажу о структуре rect, которая определяется в my_addon. В самом биндинге к SDL2 есть свое определение структуры rect:

```
(define-public sdl-rect
  (list int int int int))
```

используемой для работы с ней через FFI. Я предпочел создать свою структуру и вот почему.

Во первых определяемый **sdl-rect** мало выразителен, не понятно где ширина, длина, или положение. Во вторых, сами функции FFI создающие(**make-c-struct**) и читающие(**parse-c-struct**) явно не достаточны и медленны. Ладно структура с 4мя значениями типа **int**, но встречаются структуры с гораздо большим количеством значений, и все их FFI предлагает пересоздавать при необходимости изменить одного или двух значений. Тоже происходит при попытке чтения необходимых значений. Функций чтения или записи отдельных полей НЕТ!

Их можно создать, но уж больно коряво это получиться, гораздо проще реализовать маленький модуль на Си: у меня это файл: ./**my_addon/my_addon_c.c**

где и определить необходимые функции(их можно расширить, к примеру для наиболее часто, возникающих операций изменения положения

```
void  rect_set_xy(SDL_Rect * r, int t1, int t2) {
  r->x = t1;
  r->y = t2;
  return; }
```

или размера

```
void
rect_set_xy(SDL_Rect * r, int newW, int newH) {...
```

скомпилировав разделяемую библиотеку **my_addon_c.so**

и загрузив ее, например в модуле **my_addon**:

создаем замыкание, позволяющее определять на Scheme функции интерфейсы к функциям из библиотеки **my_addon_c.so**, в котором есть переменная lib связанная с загруженной библиотекой

```
(define my-addon-func
   (let ((lib (dynamic-link (string-append lib-path "my_addon_c.so"))))
      (lambda (return-type function-name arg-types)
          (pointer->procedure return-type
                  (dynamic-func function-name lib)
                  arg-types))))
```

и используя это замыкание-функцию можно, теперь используя FFI определить все функции для работы со структурой rect, например для получения данных о координате x:

```
(define rect-x
  (let ([f  (my-addon-func
          int "rect_get_x" (list '*))])
    (lambda (r)
```

```
(if (rect? r)
   (f (unwrap-rect r))
   (sdl-error "rect-x" "Error: is not pointer!" r)))))
```

для внутреннего же представления в Scheme указателя на внешнюю структуру rect, мы создаем описание, которое оборачивает указатель на внешнюю структуру специальной оберткой, позволяющей проверять тип хранимого указателя.

```
(define-wrapped-pointer-type <rect>
 rect?
 wrap-rect unwrap-rect
 (lambda (r port)
  (format port "#<rect ~x>: x:~d, y:~d, w:~d, h:~d"
        (pointer-address (unwrap-rect r))
        (rect-x r) (rect-y r)
        (rect-w r) (rect-h r))))
```

Значение этого типа(<**rect**>) и возвращает функция **make-rect**

```
(define make-rect
 (let ([f  (my-addon-func
          '* "make_rect" (list int int int int))])
  (lambda (x y w h)
   ;;здесь можно проверить параметры
   (wrap-rect (f x y w h)))))
```

Именно с возвращаемым данной функцией обернутым указателем и работает наша программа.

Обработка сообщений осуществляется в функции RunGame, где организован цикл обработки сообщений, путем чтения сообщений функцией (**event (SDL:poll-event))**

Если сообщение прочитано, то происходит его обработка:

```
(cond
 [(SDL:quit-event? event)
  (display "bye!\n")]
 [(SDL:keyboard-down-event? event)
  (display "Key: ")     (display (SDL:keyboard-event-key event))
  (display ", Scan: ")  (display (SDL:keyboard-event-scancode event)) (newline)
  (let ([k (SDL:keyboard-event-scancode event)])
   (cond
    [(eq?  k 'up)
     (display "move UP\n")    (mySDL:rect-y-set! rect (- (mySDL:rect-y rect) 5))]
    [(eq?  k 'down)
     (display "move DOWN\n")  (mySDL:rect-y-set! rect (+ 5 (mySDL:rect-y rect)))]
    [(eq?  k 'left)
     (display "move LEFT\n")  (mySDL:rect-x-set! rect (- (mySDL:rect-x rect) 5))]
    [(eq?  k 'right)
     (display "move RIGHT\n") (mySDL:rect-x-set! rect (+ 5 (mySDL:rect-x rect)))]
    ))

  (Render ren rect)
```

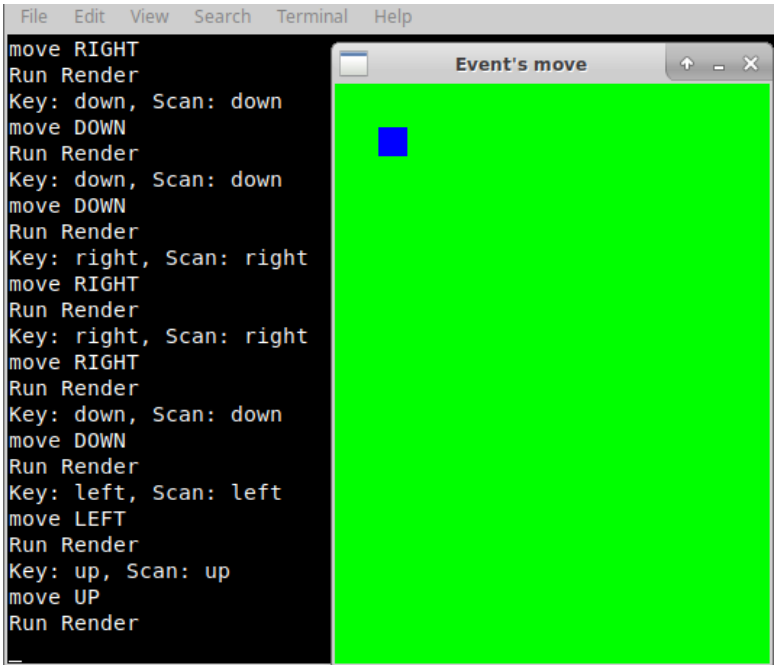
И отрисовка рендера, в соответствии с новым положением **rect**.

В нашем случае мы обрабатываем только клавиатурные сообщения.

Само отображение рендера представляет очистку рендера зеленым цветом и рисование синего квадрата задаваемого структурой **rect**

```
(define (Render ren rect)
 (SDL:set-render-draw-color ren 0 255   0   255)   ;;установка зеленого цвета
 (SDL:clear-renderer ren)
 (SDL:set-render-draw-color ren 0   0  255  255)   ;;установка синего цвета
 (mySDL:render-fill-rect ren rect)
 (SDL:present-renderer ren)
 )
```

Скриншот работы программы приведен ниже:



Загрузка и отображение изображений.

Передвигать квадратик на зеленом фоне это конечно интересно, при развитой фантазии в нем можно увидеть: танк, зайчика, цветок и т. п. Но когда вам надо отобразить различные объекты и направить фантазию пользователя в нужное русло, желательно вместо условного квадрата отображать конкретное изображение. В файле **tut03.scm** рассматривается пример загрузки изображения и отображения его на экране, его также можно передвигать стрелками и масштабировать клавишами + и -.

Но давайте разберем пример по порядку:

Вначале выполняем подключение необходимых модулей:

```
(use-modules ((sdl2) #:prefix SDL:)…
```

Далее выполняем подключение конфигурационного файла и вспомогательного модуля дополняющего биндинг SDL

```
(eval-when (compile load)
   (load "my_config.scm")
   (setenv "LTDL_LIBRARY_PATH" lib-path))  ;;для поиска разделяемой библиотеки
(use-modules ((my_addon) #:prefix mySDL:))
```

Вызов загрузки изображения **(SDL:load-bmp f-name)** выполняется в функции оболочке формирующей текстуру:

```
(define (load-image f-name ren)
  (let* ([temp-surface (SDL:load-bmp f-name)]
         [sprite-w (SDL:surface-width  temp-surface)]
         [sprite-h (SDL:surface-height temp-surface)])
    (let ([tex (SDL:surface->texture ren temp-surface)])
      (SDL:delete-surface! temp-surface)
      (list  tex sprite-w sprite-h))
    ))
```

вызов этой функции выполняем в функции main

```
[image-list  (load-image  (string-append work-path "tux.bmp") ren)])
```

и отображение загруженного изображения производим в функции Render

```
 (mySDL:render-copy ren img #:srcrect s-rect #:dstrect d-rect)
```

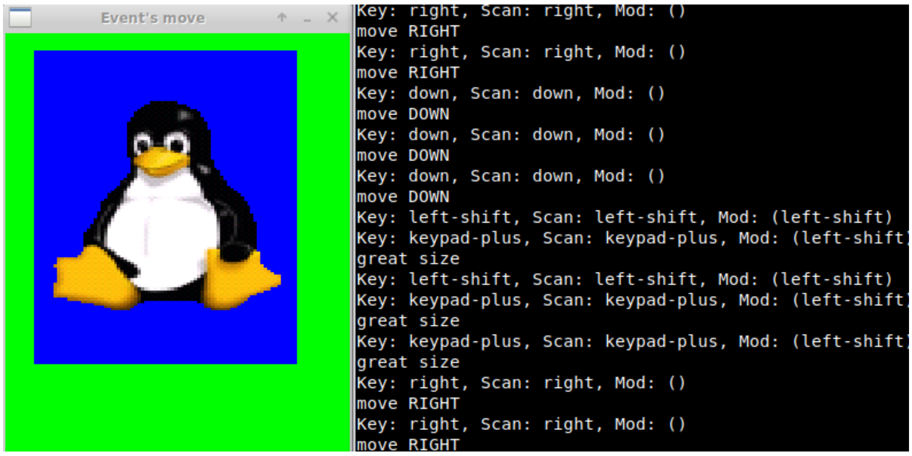
мы вызываем ее вместо рисования синего квадрата.
далее отличий от предыдущей программы очень мало, и они чисто косметические:
в **main** это разбор возвращенных параметров функцией **load-image** и определение структур rect для вывода изображения:

```
(set! img   (car   image-list))
(mySDL:rect-w-set! s-rect (cadr  image-list))
(mySDL:rect-h-set! s-rect (caddr image-list))
(mySDL:rect-w-set! d-rect (cadr  image-list))
(mySDL:rect-h-set! d-rect (caddr image-list))
```

в RunGame это обработка нажатий клавиш масштабирования изображения:

```
[(or (eq?  k 'keypad-minus)
     (eq?  k 'minus))
 (when (and (> (mySDL:rect-w d-rect) 5) (> (mySDL:rect-h d-rect) 5))
       (display "less size\n")
       (mySDL:rect-w-set! d-rect (- (mySDL:rect-w d-rect) 5))
       (mySDL:rect-h-set! d-rect (- (mySDL:rect-h d-rect) 5)))]
[(or (eq?  k 'keypad-plus)
     (and (eq?  k 'equals) (member 'left-shift m)))
 (begin
   (display "great size\n")
   (mySDL:rect-w-set! d-rect (+ (mySDL:rect-w d-rect) 5))
   (mySDL:rect-h-set! d-rect (+ (mySDL:rect-h d-rect) 5)))]
```

Скриншот демонстрирующий работу программы:

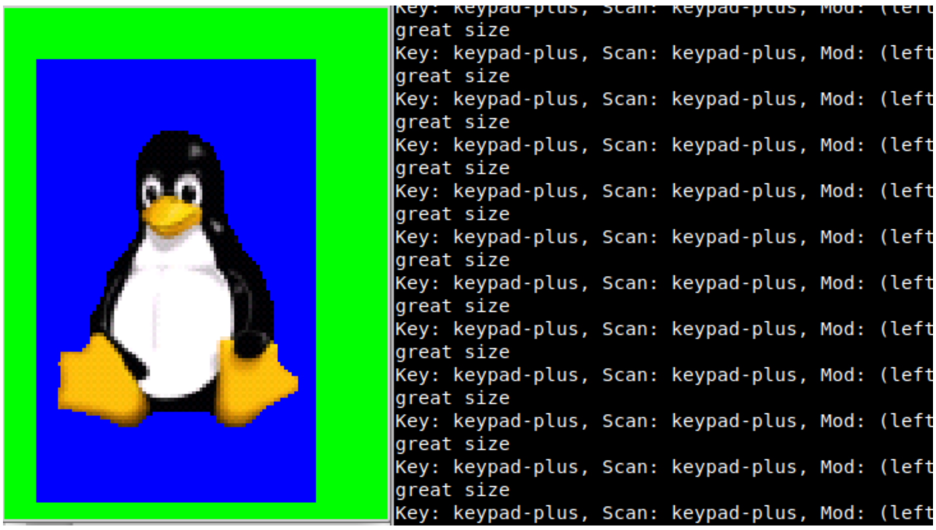


Возможно из скриншота не видно, но пингвин после увеличения получился слегка раздутым, это произошло из-за не сбалансированного по осям масштабирования. Этот недостаток я пытаюсь исправить в программе: tut03_1.scm создав отдельную процедуру масштабирования:

```
(define (change-rect-by-delta-width! rect delta)
  (let ([w (mySDL:rect-w rect)]
        [h (mySDL:rect-h rect)])
    (when (or (> delta 0)
              (> w (abs delta)))
        (let* ([new-w (+ w delta)]
               [new-h (round (/ (* h new-w) w))])
          (mySDL:rect-w-set! rect new-w)
          (mySDL:rect-h-set! rect new-h)))) )
```

вызывая ее в процедуре RunGame при обработке клавиш масштабирования изображения:

```
[(or (eq?  k 'keypad-minus)
     (eq?  k 'minus))
 (when (and (> (mySDL:rect-w d-rect) 5) (> (mySDL:rect-h d-rect) 5))
      (display "less size\n")
      (change-rect-by-delta-width! d-rect -5))]
```



еНу вот, на этот раз, даже при значительном увеличении наш пингвин сохранил стройность.

В программе **tut03_2.scm** я произвожу еще менее значительные изменения, связанные с удобочитаемостью, переводя цикл обработки сообщений на цикл **while**, что позволяет легко ввести обработку выхода по нажатию клавиши q и на мой взгляд улучшает читабельность программы.

Загрузка и отображение Спрайтов!

Программа **tut04.scm** показывает как загружать спрайт и перемещать его с помощью клавиатуры, данный спрайт можно также масштабировать, как изображение в предыдущей программе.

Работа со спрайтом, а точнее с текстурой(объектом texture получаемой из объекта surface) ничем не отличается от работы с обычным изображением. Небольшое отличие состоит в том, что при формировании текстуры мы указываем для surface ключевой цвет вызывая функцию:

(**mySDL:set-color-key temp-surface color)**

тот цвет который мы укажем и будет отсечен, а заполненное им место в изображении станет прозрачным, при формировании текстуры функцией:

(SDL:surface->texture ren temp-surface)

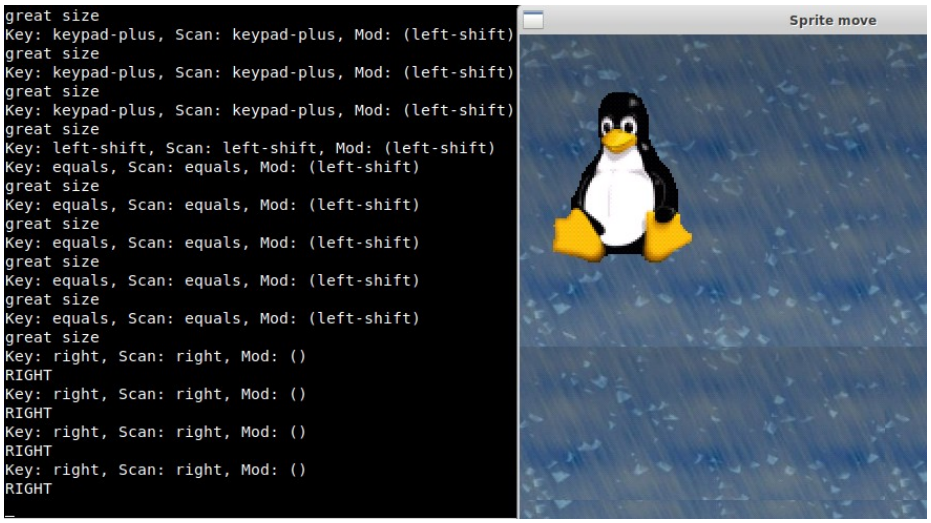
Этот процесс у нас происходит на этапе загрузки спрайта, в функции:

```
(define (load-sprite f-name ren)
  (let* ([temp-surface (SDL:load-bmp f-name)]
         [sprite-w (SDL:surface-width  temp-surface)]
         [sprite-h (SDL:surface-height temp-surface)]
         [t-pixels (SDL:surface-pixels temp-surface)])
    (if (SDL:pixel-format? temp-surface)
        (mySDL:set-color-key temp-surface
                (ffi:boolean->sdl-bool #t)
                (bytevector-u8-ref t-pixels 0))
        (let* ([pf  (SDL:surface-pixel-format temp-surface)]
               [bpp (SDL:pixel-format-bits-per-pixel pf)])
          (case bpp ;;bits-per-pixel
            [(15) (mySDL:set-color-key temp-surface
                                (ffi:boolean->sdl-bool #t)
                                (logand (bytevector-u16-native-ref t-pixels 0)
                                      #x000007ff))]
            [(16) (mySDL:set-color-key temp-surface
                                (ffi:boolean->sdl-bool #t)
                                (bytevector-u16-native-ref t-pixels 0))]
            [(24) (mySDL:set-color-key temp-surface
                                (ffi:boolean->sdl-bool #t)
                                (logand (bytevector-u32-native-ref t-pixels 0)
                                      #x00ffffff))]
            [(32) (mySDL:set-color-key temp-surface
                                (ffi:boolean->sdl-bool #t)
                                (bytevector-u32-native-ref t-pixels 0))]
```

```
      )))
  (let ([tex (SDL:surface->texture ren temp-surface)])
    (SDL:delete-surface! temp-surface)
    (list  tex sprite-w sprite-h))
  ))
```

После того как спрайт(текстура) загружен, работа программы ничем не отличается от предыдущих версий, за исключением того, что в рендере вместо цвета фона мы отображаем изображение заставку(background).

```
(define (Render ren s-rect d-rect bg-rect)
  (mySDL:render-copy ren bg   #:srcrect bg-rect #:dstrect bg-rect)
  (mySDL:render-copy ren img #:srcrect s-rect   #:dstrect d-rect)
  (SDL:present-renderer ren) )
```



Вращение и отражение изображений.

В SDL2 передусмотрена операция расширенного копирования текстуры, она позволяет в частности задавать угол поворота, с которым будет выполняться копирование(параметр angle) и возможность отражения по вертикали и/или горизонтали копируемого изображения (параметр flip).

Пример использования этой операции приведен в программе: **tut05.scm**.

По сравнению с программой **tut04** меняется не много вместо операции копирования: **mySDL:render-copy** используется ее более продвинутый вариант:

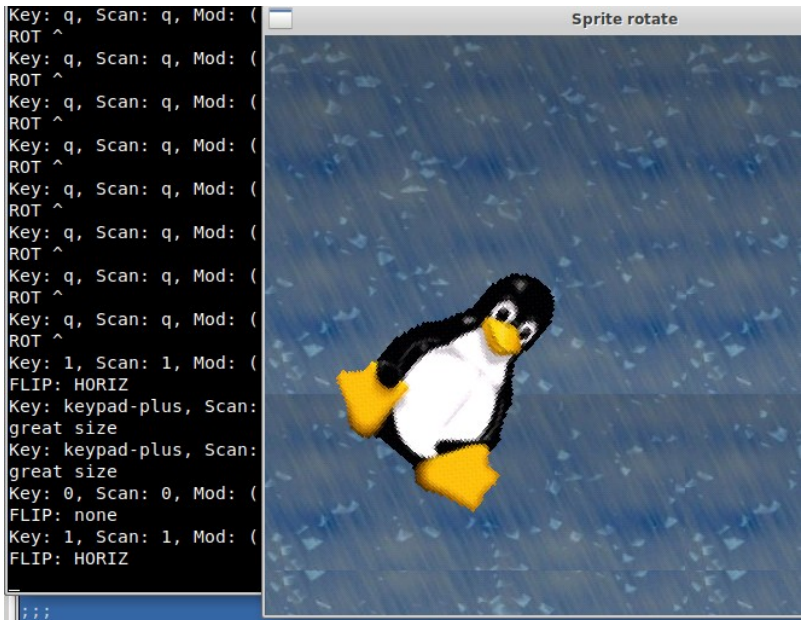
(mySDL:render-copy-ex ren img #:srcrect s-rect #:dstrect d-rect #:angle angle #:flip flip)

позволяющий вращать изображение и отражать его(есть еще параметр center типа point, но мне он был не нужен).

Остальные изменения косметические позволяющие изменять указанные выше новые параметры. Клавиши 0, 1, 2, 3 меняют параметр flip. А клавиши q и w задают поворот изображения.

```
        [(eq?  k '0)(display "FLIP: none\n")  (set! flip  mySDL:SDL_FLIP_NONE)]
        [(eq?  k '1)(display "FLIP: HORIZ\n")  (set! flip
mySDL:SDL_FLIP_HORIZONTAL)]
```

[(eq? k '2)(display "FLIP: VERT\n") (set! flip mySDL:SDL_FLIP_VERTICAL)]
[(eq? k '3)(display "FLIP: HORIZ and VERT\n") (set! flip (logior mySDL:SDL_FLIP_HORIZONTAL mySDL:SDL_FLIP_VERTICAL))]
[(eq? k 'q)(display "ROT ^\n") (set! angle (floor-remainder (- angle 5) 360.0))]
[(eq? k 'w)(display "ROT v\n") (set! angle (floor-remainder (+ angle 5) 360.0))]



На данном скриншоте пингвин не просто увеличен и повернут, но и отражен по горизонтали.

Скорость вывода спрайтов в SDL2

Для определения скорости с которой работает SDL2, напишем программу аналогичную тестовой из поставки SDL2 **testspriteminimal.c**. Её я слегка изменил, чтобы она засекала время выполнения и работала вместо смайликов с пингвинами. Аналогом этой программы является программа на схеме: **testspriteminimal.scm**, которая также определяет скорость выполнения программы.

Особо интересного по сравнению с предыдущими программами там ничего нет, просто вместо одного спрайта выводятся одновременно 100 спрайтов, в случайно заданные позиции, и передвигаются по случайным направлениям с выбранными скоростями, по программе идеального газа(т.е не сталкиваясь, но отскакивая от стенок).

Единственное что хотелось бы сказать, что в этой программе определяется своя структура **<rect>**. Как бы странно это не выглядело, но необходимость ее использования зависит от того, как и где вы обрабатываете данные о спрайтах. Если вас устраивают си-функции для работы с rect или необходимые вы можете легко дописать на си, то лучше конечно работать со структурой из си. Если же большинство ваших функций работают в Scheme, то эта структура имеет право на существование, и перенос данных из нее можно осуществлять в си структуру только на этапе вывода информации на экран(в рендере).

В функции **main** мы инициализируем заданный вектор спрайтов:

```
        [spr-vect (make-vector max-spr)])
        (vector-map! (lambda (ind el)
                (init-sprite-data bound spr-w spr-h))
                 spr-vect)
```

Функция инициализации данных о спрайте задает начальное положение спрайта и проекции по осям скорости его движения:

```
(define init-sprite-data
  (let ([MAX-VX  MAX_SPEED]
        [MAX-VY  MAX_SPEED]
        [MIN-VX (- MAX_SPEED)]
        [MIN-VY (- MAX_SPEED)])
    (lambda (bound s-width s-height)
      (if (rect? bound)
          (let ([max-x  (- (rect-w bound) s-width)]
                [max-y  (- (rect-h bound) s-height)])
            (make-sprite-data (random max-x)
                              (random max-y)
                              (+ MIN-VX (random (- MAX-VX MIN-VX)))
                              (+ MIN-VY (random (- MAX-VY MIN-VY)))
                              ))
          (SDL:sdl-error "init-sprite-data" "invalid type bound")
          ))))
```

Далее инициализированный вектор спрайтов, мы передаем в функцию **run-game**

```
        (run-game scr-rect ren spr-vect sprite spr-rect bg)
```

которая засекает время и вычисляет время вывода одного кадра:

```
(define  (run-game scr-rect ren spr-vect sprite spr-rect bg)
  (let ([start-time (gettimeofday)] [MAX_CYCLES 1300]
        [end-time -1]  [delta-mks 0]  [mks-per-cadr 0] [cadr-in-sec  0])
    (move-vect-spr ren (car sprite)  MAX_CYCLES spr-vect scr-rect spr-rect bg)
    (set! end-time (gettimeofday))
    (set! delta-mks (+ (* 1000000 (- (car end-time) (car start-time)))
                       (- (cdr end-time) (cdr start-time))))
    (set! mks-per-cadr (quotient delta-mks MAX_CYCLES))
    (set! cadr-in-sec  (quotient 1000000 mks-per-cadr))
    (display (string-append "All time execute: " (number->string delta-mks)
                            "mks, mks per cadr: " (number->string mks-per-cadr)
                            "mks, Cadr in sec: " (number->string cadr-in-sec) "\n"))
    ))
```

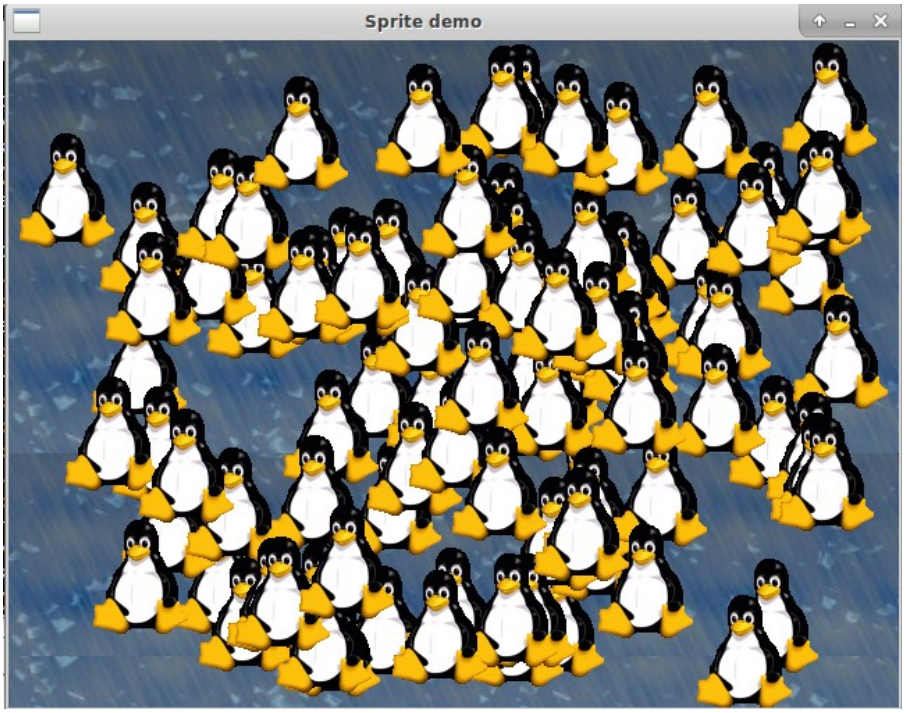
Основная же работа по перемещению спрайтов, проверке пересечения границ проводиться в функции **move-vect-spr**:

```
(define (move-vect-spr ren spr-tex num-cycles vect-spr bound spr-rect bg)
  (let ([num-sprite (vector-length vect-spr)]
        [s-rect     (mySDL:make-rect (rect-x spr-rect)
                                     (rect-y spr-rect)
                                     (rect-w spr-rect)
                                     (rect-h spr-rect))]
        [d-rect     (mySDL:make-rect 0 0
                                     (rect-w spr-rect)
                                     (rect-h spr-rect))]
        [bg-tex     (car bg)]
        [bg-rect    (mySDL:make-rect 0 0
                                     (rect-w bound)
                                     (rect-h bound))]
        )
    (do ([i 0 (1+ i)])
        ((>= i num-cycles))
      (mySDL:render-copy ren bg-tex #:srcrect bg-rect #:dstrect bg-rect)
      (do ([i-spr 0 (1+ i-spr)]) ;;move  and draw all sprite
          ((>= i-spr num-sprite))

        (let ([spr-dat (vector-ref vect-spr i-spr)])
          (move-sprite-and-reflect-bound2 spr-dat  bound spr-rect)  ;;move
          (mySDL:rect-x-set! d-rect (sprite-data-x spr-dat))  ;;draw
          (mySDL:rect-y-set! d-rect (sprite-data-y spr-dat))
          (mySDL:render-copy ren spr-tex #:srcrect s-rect #:dstrect d-rect)))
      (SDL:present-renderer ren)
      ;;(usleep 5000)
      )  ))
```

Результатом запуска аналогичных тестов для старого SDL было 102 кадра в секунду для scm программы и 112 кадров в секунду для си программы. Причем это стабильно достигаемый результат, не зависящий от текущей загрузки процессора. Результаты же запуска программ для SDL2 меня очень обрадовали. Scheme программа выдала 167 кадров в секунду, что в полтора раза было быстрее Си программы для SDL! Но ничто не стоит на месте и в один прекрасный день мой Линукс обновился, после этого началось какое то сумашествие. Си программа SDL2 стала выдавать 650-700 кадров в секунду, Scheme программа выдает от 560 до 670 кадров в секунду. Сильно изменяясь от запуска к запуску, вероятно в связанной с загрузкой системы. Скорость программ SDL1 тоже изменилась, для Си она стала от прежних 112 до 155, для Scheme от прежних 102 до 147. Но в любом случае, SDL2 в 5-6 раз быстрее чем SDL1. Вариативность в скорости связана скорее всего с улучшенной диспетчеризацией задач в Линукс для многоядерных система(у меня два ядра). И если диспетчер Линкус бросает задачу на не загруженное ядро, то там получается (для SDL1 более 150 кадров в секунду), если на загруженное то прежние 102-111, аналогичное справедливо и для SDL2, только там скорость работы в несколько раз выше.

Чтобы получить скриншот работы программы scm пришлось вставить значительную задержку: (usleep 15000) и в этом случае программа тратила на вывод кадра 18255 из которых 15000(приблизительно) приходилось на usleep.



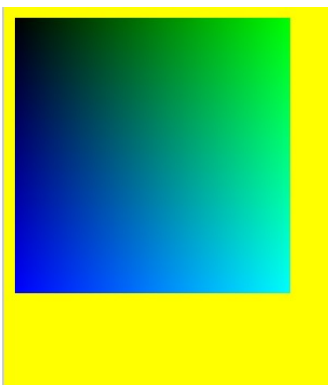
Рисование в SDL2 с помощью библиотеки расширения SDL2_gfx

Базовая библиотека SDL2 предоставляет возможности рисования точек и линий, все эти функции описаны в **SDL_render.h**. Но существует библиотека графических расширений gfx которая реализует множество подобных функций рисования с возможностью задания цвета рисования как в виде отдельной переменной типа color, так и в виде составного цвета RGB.

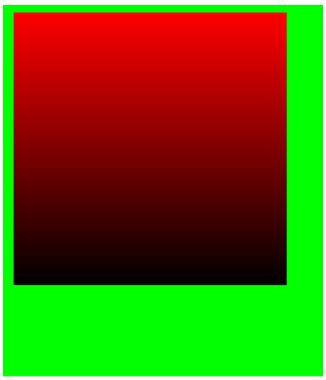
Эти функции реализуют рисование пикселов, линий(вертикальных, горизонтальных и произвольных), улучшенных линий, прямоугольников, закрашенных прямоугольников, кругов, эллипсов, секторов, треугольников, многоугольников, текстурированных многоугольников, линий Безье и смешной вывод моноширинного текста, а также функции масштабирования и поворота поверхностей. Биндинга к этой библиотеки нет, но кое какие функции я описал в отдельном модуле, который называется: **my_gfx.scm** — в нем описаны функции графических примитивов, реализующих указанные выше функции.

Пример использования всех этих функция я свел в один файл: **tut07.scm** все примеры там сведены в один список и вызываются после нажатия пробела по очереди.

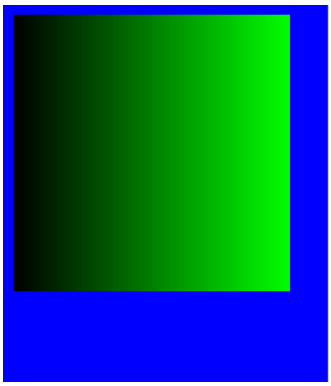
тест вывода пикселей



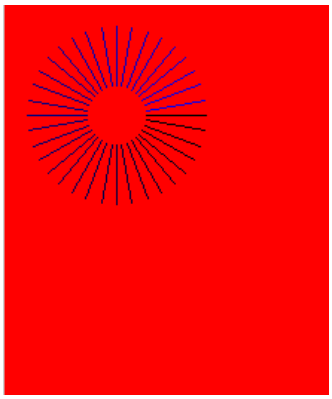
тест горизонтальных линий

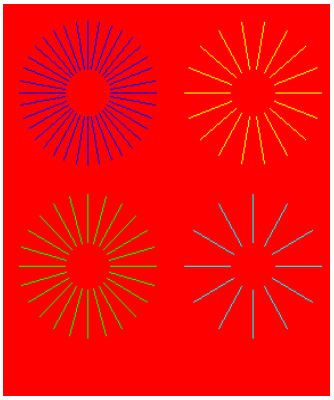


тест вертикальных линий

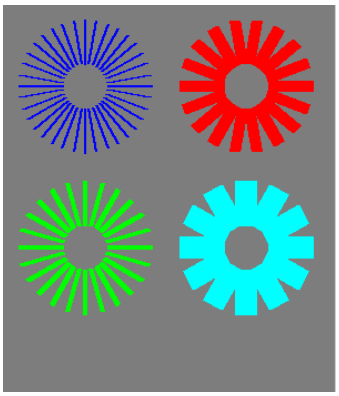


тест произвольных линий

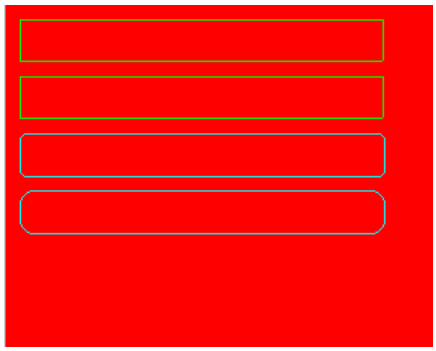


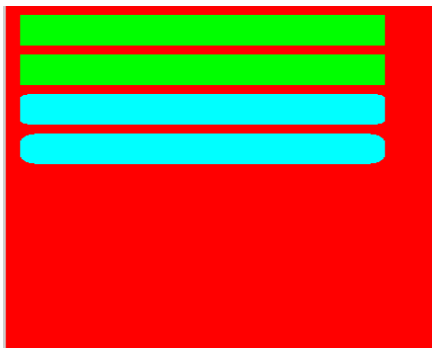


тест широких линий

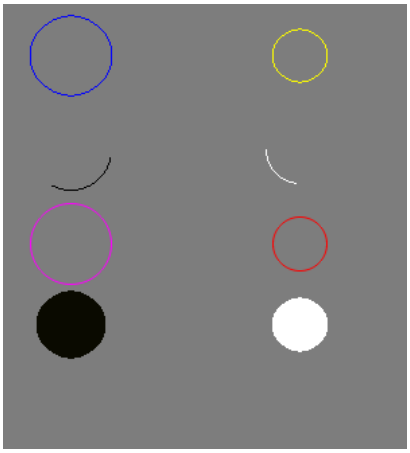


тест прямоугольников

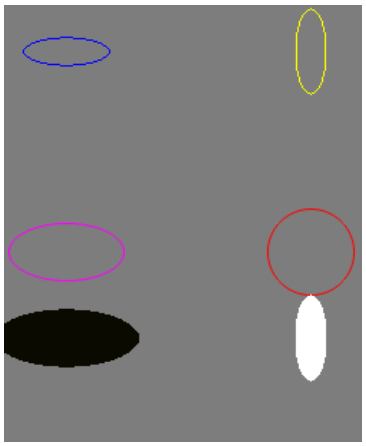




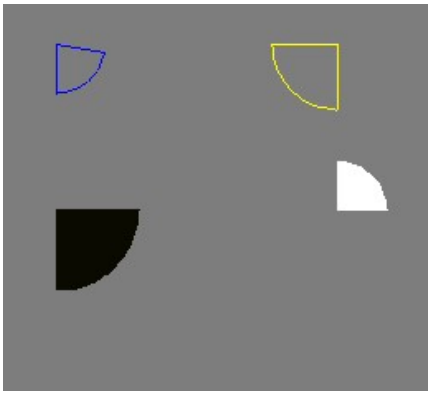
тест окружностей



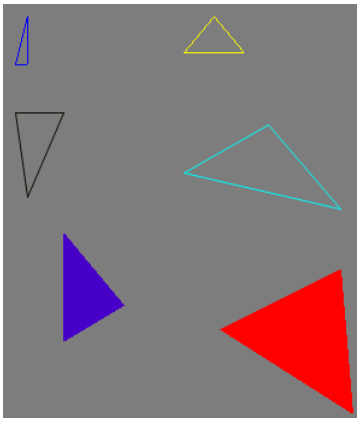
тест эллипсов



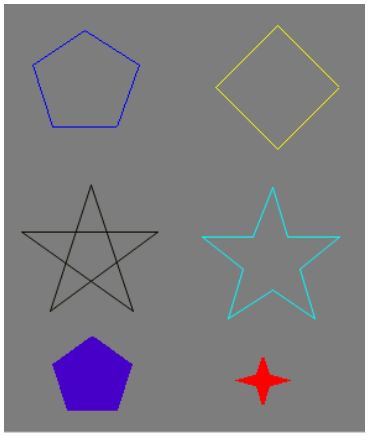
тест секторов



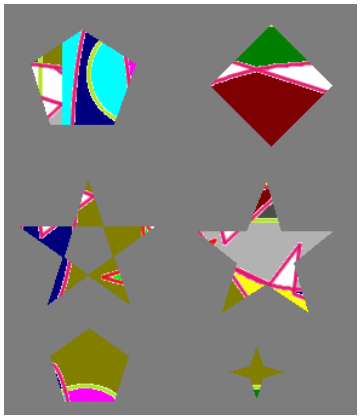
тест треугольников



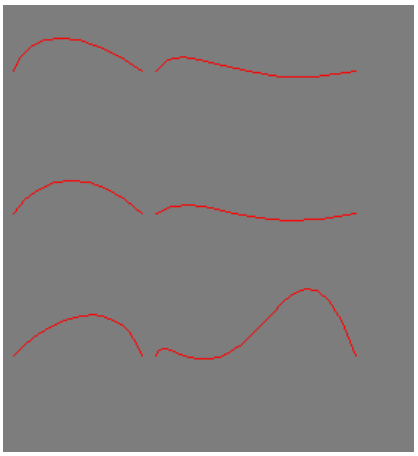
тест многоугольников



тест текстурированных многоугольников



кривая Безье



Все эти картинки вы можете наблюдать запустив программу. Единственное что хотелось бы отметить, что отображение текстурированных многоугольников таинственным образом не требует обновления рендера, что говорит о том что данную функцию желательно использовать не при непосредственном рисовании на экране, а в фазе подготовки изображений на временных поверхностях(surface).

Функцию печати моноширинного текста не описываю, она работает со встроенным по умолчанию шрифтом, в котором нет русских букв, свой шрифт добавить можно, но он представляет собой побайтово задаваемый вектор, где байт или несколько байтов(в зависимости от ширины символа) представляют собой одну горизонтальную линию символа, последовательность таких байтов определяет символ. А весь шрифт задается последовательностью таких вот последовательностей определений символов. Подробнее можно ознакомиться в исходниках библиотеки gfx.

А мы рассмотрим:

Вывод сообщений с помощью фонта TTF.

Пример вывода текстовых сообщений в SDL2 приведен в файле: **tut08.scm**
Для работы с фонтами TTF мы должны вначале загрузить модуль TTF

```
(use-modules ...
        ((sdl2 ttf) #:prefix SDL:)
        ((sdl2 bindings) #:prefix ffi:))
```

затем систему ttf надо инициализировать:

```
(SDL:ttf-init)
```

после этого можно загрузить интересующий нас фонт:

```
(define font1 (SDL:load-font "schou___.ttf" 20))
```

и выводить сообщения, при этом формируются промежуточные surface и texture, которые желательно удалять:

```
(define s1 (SDL:render-font-solid font1 "Hello World!" (SDL:make-color 255 0 0 255)))
(define tex1 (SDL:surface->texture ren s1))
```

отображаем полученную текстуру на экран:

```
(SDL:render-copy ren tex1
                 #:srcrect (list 0 0 (SDL:surface-width s1) (SDL:surface-height s1))
                 #:dstrect (list 10 30 (SDL:surface-width s1) (SDL:surface-height s1)))
```

удаляем промежуточные поверхности и текстуры:

```
(SDL:delete-surface! s1)
(mySDL:destroy-texture tex1)
```

это необходимо делать иначе возникают «утечки» памяти. В данном примере я хотел обойтись без своего my_addon. Но в биндинге не определена функция уничтожения ненужных более текстур. Пришлось написать свой биндинг к этой функции и разместить его в my_addon.

Когда все выведено можно обновить рендер

```
(SDL:present-renderer ren)
```

и наблюдать в течении 10 секунд окно с построенным сообщением:

Работа с OpenGL в SDL2.

Работа с OpenGL в SDL2 практически ничем не отличается от аналогичной работы в SDL1. Продемонстрируем ее на примере крутящегося разноцветного треугольника, код приведен в файле: **tut09_1.scm**.

Чтобы можно было работать с функциями OpenGL загружаем модули биндинга к OpenGL.

```
(use-modules ((gl)   #:prefix GL:))
(use-modules ((glu)   #:prefix GLU:))
(use-modules ((gl enums) #:prefix GL-ENUM:))
```

После создания окна устанавливаем через функции SDL атрибуты OpenGL

```
;;OpenGL attribute
(SDL:set-gl-attribute! 'double-buffer 1)
```

```
(SDL:set-gl-attribute! 'red-size 5)
(SDL:set-gl-attribute! 'green-size 6)
(SDL:set-gl-attribute! 'blue-size 5)
(define context (SDL:make-gl-context win))
```

и используя функции OpenGL инициализируем сцену

```
;;init OpenGL
(GL:gl-viewport 0 0 win_w win_h)
(GL:gl-frustum -1.0 1.0 -1.0 1.0 1.0 100.0)
(GL:set-gl-clear-color 0.0 0.0 0.0 1.0)
(GL:set-gl-matrix-mode (GL-ENUM:matrix-mode  modelview))
```

Поскольку нам необходимо добиться вращения, запускаем цикл, в котором просто увеличиваем счетчик от 0 до 360. в процедуре: (**move-cycle2 win**)

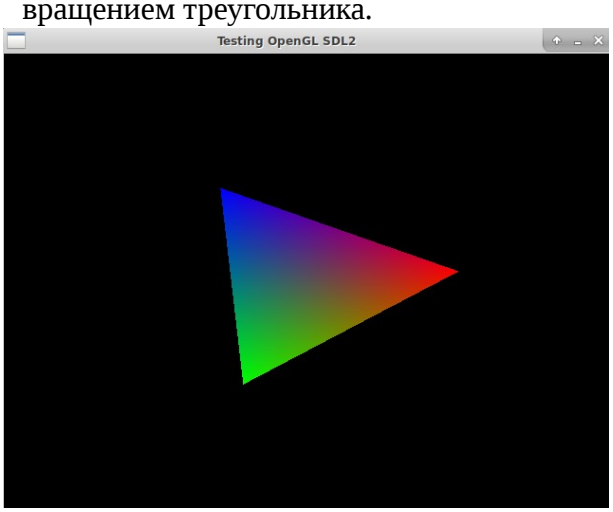
```
…
   (set! zrf (floor-remainder (+ zrf 5.0) 360.0))
    (draw-triangle zrf)
    (SDL:swap-gl-window win)
```

Функция **(draw-triangle zrf)** рисует разноцветный треугольник с заданным углом поворота, а фукнция (**SDL:swap-gl-window win)** отображает буфер в заданном окне.

Вот описание команд OpenGL, для вывода нашего треугольника:

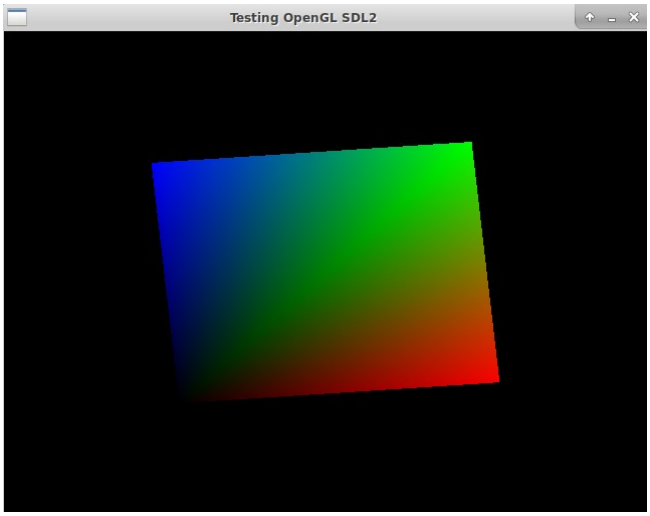
```
(define (draw-triangle rf)
  (GL:gl-clear (GL-ENUM:clear-buffer-mask color-buffer))
  (GL:with-gl-push-matrix
   (GL:gl-translate 0.0 0.0 -2.0)
   (GL:gl-rotate rf 0.0 0.0 1.0)
   (GL:gl-begin (GL-ENUM:begin-mode  triangles)
              (GL:gl-color 1.0 0 0)
              (GL:gl-vertex 0.0 1.0 0.0)
              (GL:gl-color 0 1.0 0)
              (GL:gl-vertex 0.866 -0.5 0.0)
              (GL:gl-color 0  0  1.0)
              (GL:gl-vertex  -0.866  -0.5 0.0))
  ))
```

В принципе это все, можно запускать программу и наблюдать за «гипнотизирующим» вращением треугольника.



В файле **tut09_2.scm** приведена программа аналогичная предыдущей но выводящая квадрат поворачиваемый на заданный угол:

```
(define (draw-quads rf)
  (GL:gl-clear (GL-ENUM:clear-buffer-mask color-buffer))
  (GL:with-gl-push-matrix
   (GL:gl-translate 0.0 0.0 -2.0)
   (GL:gl-rotate rf 0.0 0.0 1.0)
   (GL:gl-begin (GL-ENUM:begin-mode  quads)
                (GL:gl-color 1.0 0 0)
                (GL:gl-vertex 1.0 1.0 0.0)
                (GL:gl-color 0 1.0 0)
                (GL:gl-vertex -1.0 1.0 0.0)
                (GL:gl-color 0  0  1.0)
                (GL:gl-vertex  -1.0  -1.0 0.0)
                (GL:gl-color 0  0  0.0 0.0)
                (GL:gl-vertex  1.0  -1.0 0.0))
   ))
```



Все бы ничего, но вот вид у, нашего равностороннего треугольника, немного не равносторонний, а квадрата — не квадратный!

Попробуем исправить это недоразумение в программе: **tut09_3.scm** и **tut09_4.scm**, соответственно для треугольника и квадрата, а заодно добавив в программу обработчик сообщения изменения размера.

Для этого создаем окно изменяемого размера:

```
(define win (SDL:make-window #:size (list win_w win_h)
                             #:title "Testing OpenGL SDL2"
                             #:position (list 10 10)
                             #:resizable? #t
                             #:opengl? #t
                             #:show? #t))
```

Добавляем функцию обработки сообщения изменения размера:

```
(define (change-size base-xy base-z  w h)
  (let ([loc-h h])
    (when (eq? 0 loc-h)  ;;don't divide 0!
        (set! loc-h 1))
    (GL:gl-viewport 0 0 w loc-h)
    (GL:set-gl-matrix-mode (GL-ENUM:matrix-mode  projection))
    (GL:gl-load-identity)
    (let ([aspect-ratio (/ w loc-h)])
      (if (< w loc-h)
        (GL:gl-ortho (- base-xy) base-xy
                  (/ (- base-xy) aspect-ratio) (/ base-xy aspect-ratio)
                  base-z (- base-z))
        (GL:gl-ortho (* (- base-xy) aspect-ratio) (* base-xy aspect-ratio)
                  (- base-xy) base-xy
                  base-z (- base-z))
      ))
    (GL:set-gl-matrix-mode (GL-ENUM:matrix-mode  modelview))
    (GL:gl-load-identity)))
```

которая пересчитывает размеры проецируемого объема нашей модели в текущие координаты окна, позволяя сохранить пропорции модели независимыми от производимых с окном трансформаций.

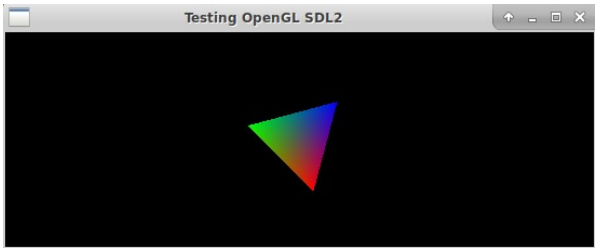
Вставляем ее вызов сразу после формирования окна:

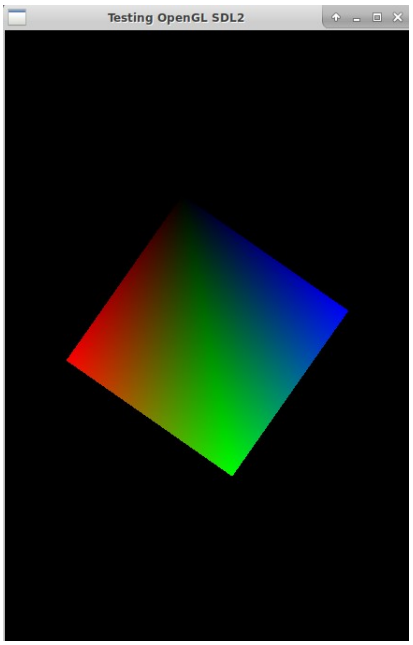
```
  (change-size base-xy base-z win_w win_h)
```

И в обработчик сообщения изменения размеров окна в функции (move-cycle2 win):

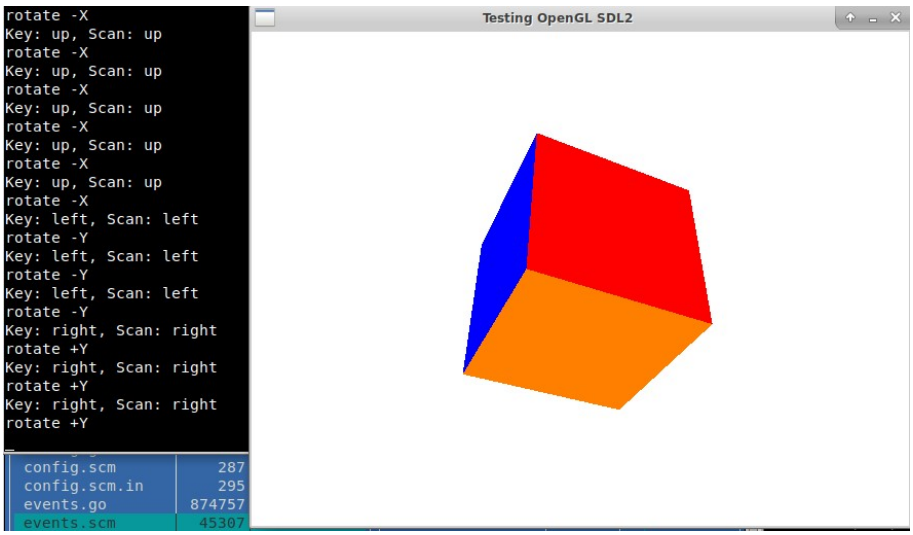
```
          [(SDL:window-resized-event? event)
           (let ([size (SDL:window-event-vector event)])
             (set! win_w (car size))
             (set! win_h (cadr size))
             (change-size base-xy base-z win_w win_h)  )]
```

Все!!! Теперь наш равносторонний треугольник действительно равносторонний, а квадрат квадратный, вне зависимости от пропорций окна(но соответственно изменяется его размер).





Также в качестве примера работы с OpenGL привожу программу: **opengl_demo01.scm**, выводящую на экран окно с разноцветным кубиком, который можно вращать, нажимая на клавиши стрелки и q, a, вокруг различных осей.



Внимательно изучив код данной программы, вы возможно заметили закомментированный вызов функции:

;;(mySDL:gl-make-current win context)

который судя по названию должен устанавливать текущим полученный нами контекст для рисования OpenGL. Закомментирован этот вызов по очень простой причине — он ненужен!!!

Полученный контекст OpenGL функцией:

(define context (SDL:make-gl-context win))

и так становиться текущим по умолчанию.

Так зачем же нужна функция без которой и так все работает? Немного поразмыслив над этим вопросом я все-таки решил написать пример многооконной работы с OpenGL, в котором эта функция как раз и проявляет свою работоспособность.

Многооконная работа в SDL2 с OpenGL.

Пример работы с многооконным интерфейсом SDL2 OpenGL приведен в файле: **tut09_5.scm**, этот пример как бы объединяет две программы **tut09_3.scm** и **tut09_4.scm**, выводя два окна с вращающимися фигурами. Единственное сообщение обрабатываемое отдельно окнами это изменение их размера.

Для разработки многооконной программы нам потребуется структура в которой мы будем хранить данные об окне: сам объект окно, идентификатор окна, необязательные размеры окна и контекст OpenGL этого окна, мы создадим ее в виде записи **<win-data>**.

```
(define-record-type <win-data>
  (make-win-data win id w h ctx)
  win-data?
  (win win-data-win win-data-win-set!)
```

```
 (id  win-data-id  win-data-id-set!)
 (w   win-data-w   win-data-w-set!)
 (h   win-data-h   win-data-h-set!)
 (ctx win-data-ctx win-data-ctx-set!)
 )
```

Данные об окнах, структуры <**win-data**> мы будем записывать в список **wins**. И для удобства поиска необходимых окон по идентификатору окна , создадим функцию-предикат возвращающую истину(сам объект **win-d** типа <**win-data**>) при сравнении с эквивалентным идентификатором окна **id**

```
(define (eq-by-win-id? id win-d)
  (if (eq? id (win-data-id win-d))
     win-d
     #f))
```

Теперь можем создавать окна и заполнять наш список **wins**
определяем переменную **wins** как пустой список

```
(define wins '())
```

устанавливаем общие OpenGL атрибуты

```
 ;;OpenGL attribute
 (SDL:set-gl-attribute! 'double-buffer 1)
 (SDL:set-gl-attribute! 'red-size 5)
 (SDL:set-gl-attribute! 'green-size 6)
 (SDL:set-gl-attribute! 'blue-size 5)
```

создам окно, создаем контекст OpenGL для данного окна, получаем идентификатор окна и помещаем данные о нем в структуру <**win-data**>, которую, в свою очередь, помещаем в список **wins**:

```
 (let* ([win (SDL:make-window #:size (list win_w win_h)
                              #:title "Win1 OpenGL SDL2"
                              #:position (list 10 10)
                              #:resizable? #t
                              #:opengl? #t
                              #:show? #t)]
        [context (SDL:make-gl-context win)]
        [id      (SDL:window-id win)])
   (mySDL:gl-make-current win context)
   (GL:set-gl-clear-color 0.0 0.0 0.0 1.0)
   (change-size base-xy base-z win_w win_h)
   (set! wins (append wins  (list (make-win-data win id win_w win_h context)))))
```

Это действие можно выполнять для произвольного количества окон, в нашем случае их два.

Процедуры прорисовки окон **(draw-triangle rf)** и **(draw-quads rf)** оставляем неизменными. И затем меняем процедуру **(move-cycle2 wins)** так чтобы она могла

обрабатывать сообщения от каждого окна индивидуально, в нашей программе мы обрабатываем только сообщение об изменении размеров окна — **window-resized-event**:

```
[(SDL:window-resized-event? event)
 (let ([size    (SDL:window-event-vector    event)]
       [win-id  (SDL:window-event-window-id event)])
   (let ([new_w (car size) ]
         [new_h (cadr size)]
         [win-d (find (lambda (t) (eq-by-win-id? win-id t))  wins)])
     (if (win-data? win-d)
         (begin
           (win-data-w-set! win-d new_w)
           (win-data-h-set! win-d new_h)
           (mySDL:gl-make-current (win-data-win win-d) (win-data-ctx win-d))
           (change-size base-xy base-z new_w new_h)
           )
         (format #t "Unknown message from: ~d~%" win-id)))  ) ]
```

Для этого получаем идентификатор окна от которого пришло сообщение:

[win-id (SDL:window-event-window-id event)]

по данному идентификатору находим структуру **<win-data>** в списке **wins** используя предикат **eq-by-win-id?**:

[win-d (find (lambda (t) (eq-by-win-id? win-id t)) wins)]

и по полученным данным устанавливаем текущий контекст для выполнения команд OpenGL:

(**mySDL:gl-make-current (win-data-win win-d) (win-data-ctx win-d))**

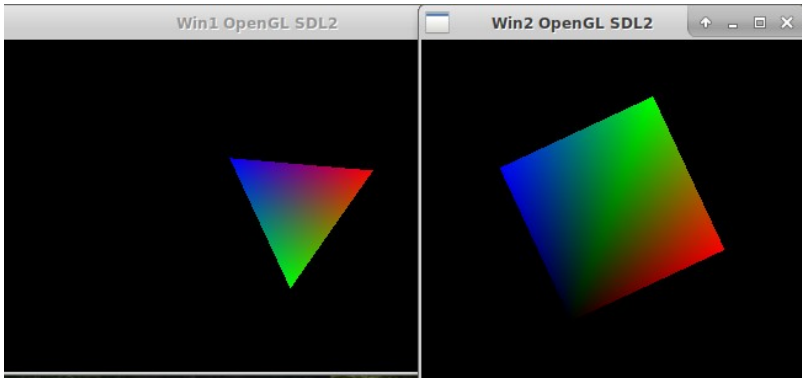
и уже в этом контексте выполняем процедуру изменения размера окна:

(change-size base-xy base-z new_w new_h)

После этого нам остается только вывести изображение на экран, с помощью функций **(draw-triangle zrf)** и **(draw-quads zrf**) выполнив их каждую в своем контексте OpenGL и обновить окна с помощью процедуры **(SDL:swap-gl-window (win-data-win w1))**, которая тоже контекстно зависима:

```
(define (draw-wins wins zrf)
  (let ([w1 (car  wins)]
        [w2 (cadr wins)])
    (mySDL:gl-make-current (win-data-win w1) (win-data-ctx w1))
    (draw-triangle zrf)
    (SDL:swap-gl-window (win-data-win w1))
    (mySDL:gl-make-current (win-data-win w2) (win-data-ctx w2))
    (draw-quads zrf)
    (SDL:swap-gl-window (win-data-win w2))))
```

Подобную функцию отображения можно значительно упростить, если разместить указатель на функцию обновления окна, типа: **(draw-triangle zrf)** в структуру <**win-data**>. Тогда она будет работать универсально, для произвольного количества окон. А мы запустив приложение, можем произвольно поменять размеры окон и увидеть картинку подобную этой:



Работа со звуком в SDL2

В качестве примера работы со звуком я приведу программу в файлах: **tut10.scm** и **tut10_1.scm**, они практически одинаковы, разница в том что вращение квадрата(спецэффект) в версии 10_1 управляемо и зависит от проигрывания музыки.

Чтобы звук работал надо загрузить модуль **mixer** из биндинга SDL2:

```
(use-modules ((sdl2) #:prefix SDL:)
             ...
            ((sdl2 mixer) #:prefix MIX:))
```

При инициализации SDL инициализировать еще и **audio**

```
(SDL:sdl-init '(video audio))
```

Затем загружаем файл, который будем проигрывать(можно загрузить свою музыку указав полный путь к файлу в командной строке):

```
(MIX:open-audio)
(define music #f)
(if (> (length (command-line)) 1)
   (set! music (MIX:load-music (cadr (command-line))))
   (set! music (MIX:load-music "background.ogg")))
```

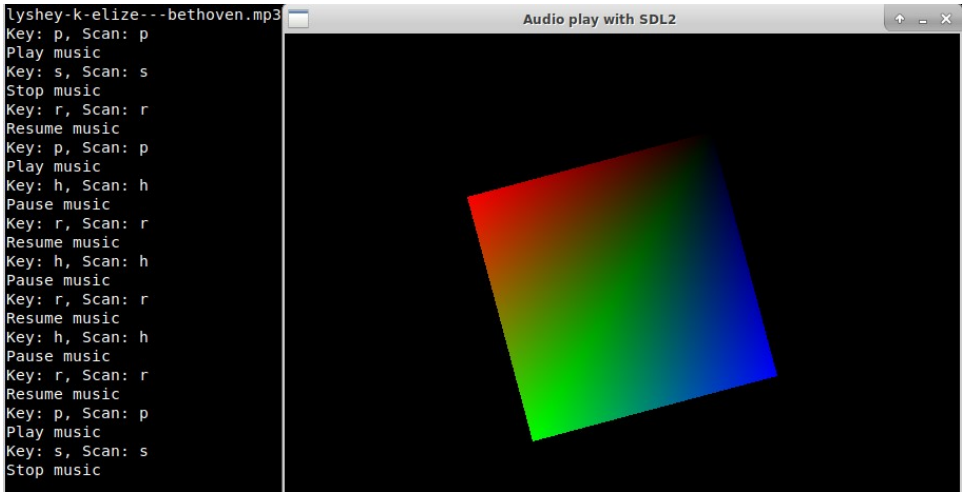
запуск на проигрывание, а также останов, паузу и возобновление проигрывание выполняются по нажатии клавиш: p, s, h, r их обработка происходит в процедуре: **(move-cycle2 win)**

```
                [(eq?  k 'p)     (display "Play music\n")    (set! zrf 0.0) (MIX:play-music! music)
]
                [(eq?  k 's)     (display "Stop music\n")    (MIX:stop-music!)  ]
                [(eq?  k 'h)     (display "Pause music\n")   (MIX:pause-music!) ]
                [(eq?  k 'r)     (display "Resume music\n")  (MIX:resume-music!)]
```

В версии 10_1, происходит управление вращением квадрата с помощью условия:

```
(when (and (MIX:music-playing?) (not (MIX:music-paused?)))
   (begin
    (set! zrf (floor-remainder (+ zrf 5.0) 360.0))
    (draw-quads zrf)
    (SDL:swap-gl-window win)
   ))
```

Вообщем-то это и ВСЁ! Музыка играет, квадрат вращается — красота!



Небольшое отвлечение/развлечение!
Совместное использование Cairo и SDL2.

Примеры **tut03_3.scm - tut03_5.scm** содержат код показывающий как можно построить взаимодействие между Cairo и SDL2. Использованию Cairo в guile посвящено отдельное руководство: GuileCairoGtkTutorial.pdf, где описывается, что такое Cairo и для чего он нужен.

Заключение.

Ну вот и все, дорогие друзья. Про SDL2 можно много еще чего написать, но приходит время завершать любую работу и я ее завершаю. Надеюсь мой скромный труд будет Вам полезен. Удачи!

С Вами был Гагин Михаил aka NuINu.