Использование SDL в программировании на языке Scheme(Guile).

Введение.

С языком Scheme я познакомился совсем недавно, прочитал в интернете статью путешествие питониста в мир схемы, там приводился интересный пример фактически модификации лиспоподобного языка схемы в язык форматирования текста программы пробелами, подобный питону. Конечно это интересная модификация языка, но мне она не особо приглянулась, хотя я знаю не любовь программистов к языку лисп, за его многочисленные скобки. Но я все же решил поупражняться в программировании на этом языке, а заодно изучить книжку SCIP(структура и интерпретация компьютерных программ). В процессе изучения этой весьма занятной книжки я отвлекался изучая другие аспекты этого языка, нашел несколько биндингов позволяющих использовать широко распространенные библиотеки применяемые для графического отображения информации, в основном для создания игр, хотя возможности их конечно гораздо шире. Это биндиги к библиотекам SDL и OpenGL. Однако примеров использования их, за исключением крайне бедных тестов SDL идущих вместе с исходниками биндинга, в интернете я не нашел. Причем тесты они возможно и очень хорошо тестируют возможности библиотеки, но бывают очень сложные для понимания. Поэтому и возникла идея написать несколько примеров демонстрирующих возможности использования библиотеки SDL при написании графических программ на языке Scheme. За темами для примеров я далеко ходить не стал, когда то давно я читал книжку Programming Linux Game, где в 4й главе описаны базовые примеры работы с библиотекой SDL. Переписыванием этих примеров но только на языке scheme я и займусь в этой работе.

Инициализация SDL.

Сейчас наша задача просто проинициализировать библиотеку SDL и проверить, работает там что нибудь или нет, для этого выполним функцию инициализации окна SDL заданного размера, если еще установим опцию fullscreen то окном будет весь экран.

Полный листинг примера в файле: **l04-01_init-sdl.scm**

первоначально у меня библиотека SDL была установлена отдельно от всего пакета guile, поэтому мне пришлось добавить путь к месту установки библиотеки.

```
(setenv "LTDL_LIBRARY_PATH" "/usr/local/lib/guile-sdl")
```

теперь подключаем модуль SDL

```
(use-modules ((sdl sdl) #:prefix SDL:)
        (srfi srfi-1)
        (srfi srfi-2))
```

поскольку в примерах предполагается внезапный выход из программы с помощью си функции atexit устанавливается процедура завершения работы SDL, в guile такой функции нет, поэтому используем ее замену. Смысл действий простой заменим стандартный exit своим, выполним в нем необходимые действия и вызовем стандартный. Эта процедура может создать цепочку завершающих действий.

```
(define (c-atexit proc)
  (let ((old-exit exit))
    (set! exit (lambda args
            (display "atexit\n")
            (proc)
            (apply old-exit args)))))
```

А вот и сам вызов инициализации библиотеки SDL.

```
(if (not (equal? (SDL:init 'video) 0))
    (begin
      (display "Can't initialize SDL!\n")
      (exit 1)))
```

Если инициализация удалась теперь можно устанавливать нашу процедуру завершения, которая просто выполняет SDL:quit

```
(c-atexit (lambda ()
        (display "SDL:quit\n")
        (SDL:quit)))
```

После инициализации можно выполнять различные функции из библиотеки SDL, мы создадим наше базовое окно, являющееся для библиотеки некоей поверхностью, над которой можно выполнять различные действия.

```
(define screen (SDL:set-video-mode 640 480 16 'fullscreen))
;;(define screen (SDL:set-video-mode 1366 768 16 'fullscreen))
(if (not (SDL:surface? screen))
   (begin
     (display "Can't set videomode\n")
     (exit 1)))
```

Все работа завершена.

```
(display "Sucess!\n")
(exit 0)
```

В процессе работы над данным примером, у меня вылетела эта программа, и экран не восстановился, можете себе представить что покажет вам ваш десктоп с разрешением 640x480 и что бы его восстановить я просто ввел необходимые данные разрешения в программу и запустил ее по новому, она вылетела снова, но экран был уже востановлен.

На этом все, пример закончен! Поздравляю вы освоили работу с SDL!!!

Прямое рисование на поверхности.

Библиотека SDL предоставляет возможность непосредственного изменения пикселей поверхности Surface. Для этого надо сначала заблокировать поверхность вызвовом SDL_LockSurface, выполнить изменение пикселей, а затем разблокировать вызовом SDL_UnlockSurface. К сожалению имеющийся биндинг к библиотеке SDL не предоставляет нам таких возможностей. Не смотря на то, что в биндинге есть функция: surface-pixels, с ее помощью мы не сможем отредактировать пикселы у Surface ибо она предоставляет нам доступ лишь к копии данных Surface, эти данные можно менять или анализировать, но вот отобразить обратно их на поверхность мы не сможем. Пример: l04-02_direct-pixel-draw.scm демонстрирует нам тщетность попыток напрямую изменить пикселы на поверхности Surface.

Возможность вызова функции surface-pixels пока можно использовать лишь для анализа изображения и не более. Поэтому выполнять данный пример мы будем с помощью функции draw-point из библиотеки SDL_gfx.

Полный листинг примера в файле: **l04-02_nodirect-pixel-draw.scm**

При подключении модулей добавляется библиотека GFX

```
(use-modules ((sdl sdl) #:prefix SDL:)
        ((sdl gfx) #:prefix GFX:)
        (srfi srfi-1)
        (srfi srfi-2))
```

К сожалению формирование цвета для отображаемого пиксела с помощью формата получнного от поверхности стандартной функцией работало не корретно, я сильно заморачиваться этим вопросом не стал а написал свое формирование цвета:

```
(define (make-color format r g b alpha)
  (logior alpha (ash (logior b (ash (logior g (ash r 8)) 8)) 8)))
```

Далее идет стандартная инициализация SDL.
Стандартное формирование окна для отрисовки изображения.

```
(define s_width 256)
(define s_height 256)
(define screen (SDL:set-video-mode s_width s_height 16))
(if (not (SDL:surface? screen))
   (begin
     (display "Can't set videomode\n")
     (exit 1)))
```

Получаем формат пикселей изображения применяемый для нашей поверхности(мне он не помог, но в других местах работает)

```
(define screen-format (SDL:surface-get-format screen))
```

И запускаем цикл отрисовки пикселей на нашей поверхности с помощью функции draw-point

```
(let ((x 0) (y 0) (offset 0) (pixel_color 0) (alpha 255) (r 0) (g 0) (b 0))
  (do ((x 0 (+ x 1)))
     ((>= x s_width))
    (begin
     (do ((y 0 (+ y 1)))
        ((>= y s_height))
       (begin
         (set! r x)
         (set! b y)
         (set! pixel_color (make-color screen-format r g b alpha))
         (GFX:draw-point screen x y pixel_color)
         ))
     )))
```

Запускаем функцию обновления окна:

```
(SDL:update-rect screen 0 0 0 0)
```

Вуаля! Получили раскрашенное окно.

Рисование с помощью Blits

(блиттинг, а проще говоря быстрое копирование одной поверхности на другую).

Рисовать с помощью draw-point в окне не очень хорошая идея, т. к. эта процедура будет выполнять чрезвычайно медленно. И даже прямое рисование пикселей не лучший подход для динамичного отображения на экране(или в окне). Общий смысл ускорения состоит в том чтобы какими то функциями подготовить изображение в обычной памяти, а затем с помощью быстрой функции блиттинга вывести подготовленное изображение на экран. Освоением этого приема мы и займемся в нашем текущем примере.

Полный листинг примера в файле: **l04-03_blitting-surfaces-sdl.scm**

Вначале идет обычное подключение модулей(GFX здесь не нужен, но я оставил, мне он не мешает). Инициализация SDL и окна вывода screen.
Далее загружаем картинку которую собираемся нарисовать в окне:

```
(define name-image "test-image.bmp")

(define image (SDL:load-bmp name-image))
(if (not (SDL:surface? image))
```

(begin
(display "Unable load bitmap. \n")
(exit 1)))

Определим области на поверхностях предназначенные для копирования(в нашем случае это границы загруженного изображения)
(define src (SDL:make-rect 0 0 (SDL:surface:w image) (SDL:surface:h image)))
(define dst (SDL:make-rect 0 0 (SDL:surface:w image) (SDL:surface:h image)))

И вот знаменитый блитинг, вывод подготовленной картинки в окно.
(SDL:blit-surface image src screen dst)

Но само отображение картинки произойдет только после вызова update-rect
(SDL:update-rect screen 0 0 0 0)

Ну вот и все, мы освоили важнейшее умение в отображении графической информации на экране с помощью библиотеки SDL- блиттинг!

Ключевой цвет и прозрачность.

Ключевой цвет позволяет нам рисовать не просто прямоугольники изображений, а выделять из этих прямоугольников полезное изображение, а остальную часть, закрашенную ключевым цветом отбрасывать. Рассмотрим пример отображения картинки с использованием ключевого цвета и без него.

Полный листинг примера в файле: **l04-04_colorkeys-sdl.scm**

Инициализация проходит аналогично предыдущим примерам.
Заглужаем два изображения, background — подстилающая картинка и image, то что будет рисоваться поверх бэкграунда.
(define background (load-check-bmp "bg.bmp"))
(define image (load-check-bmp "tux.bmp"))

Сначала копируем на экран background
;создадим границы прямоугольников копирования
(define src (SDL:make-rect 0 0 (SDL:surface:w background) (SDL:surface:h background)))
(define dst (SDL:make-rect 0 0 (SDL:surface:w background) (SDL:surface:h background)))
;;рисуем фон
(SDL:blit-surface background src screen dst)

теперь нарисуем изображение image два раза, без использования ключевого цвета и с его использованием.
(SDL:rect:set-x! src 0)
(SDL:rect:set-y! src 0)
(SDL:rect:set-w! src (SDL:surface:w image))
(SDL:rect:set-h! src (SDL:surface:h image))
(SDL:rect:set-x! dst 30)
(SDL:rect:set-y! dst 90)
(SDL:rect:set-w! dst (SDL:surface:w image))
(SDL:rect:set-h! dst (SDL:surface:h image))

(SDL:blit-surface image src screen dst)

Теперь зададим ключевой цвет и установим его на поверхность image.
(define format-image (SDL:surface-get-format image))
(define colorkey (SDL:map-rgb format-image 0 0 255))
(SDL:surface-color-key! image colorkey)

теперь рисуем второе изображение, использующее ключевой цвет.
```
(SDL:rect:set-x! dst (- (SDL:surface:w screen) (SDL:surface:w image) 30))
(SDL:rect:set-y! dst 90)
(SDL:rect:set-w! dst (SDL:surface:w image))
(SDL:rect:set-h! dst (SDL:surface:h image))

(SDL:blit-surface image src screen dst)
```

И обновление экрана:
```
(SDL:update-rect screen 0 0 0 0)
```

Как видно из результата на второй картинке сидит «чистый» пингвин, без фонового цвета. На этом пример закончен.

Альфа смешивание(Alpha blending)

Альфа представляет собой четвертый канал цвета определяющий прозрачность всего пиксела, это так называемый RGBA пиксел, где A это и есть альфа. Если смотреть выше я его уже использовал в примере попиксельного рисвоания:
```
(define (make-color format r g b alpha)
  (logior alpha (ash (logior b (ash (logior g (ash r 8)) 8)) 8)))
```
Альфа там не менялась и была полностью не прозрачна: (alpha 255)
Разберем текущий пример по подробнее.

Полный листинг примера в файле: **l04-05_alpha-sdl.scm**
Инициализация проходит аналогично предыдущим примерам.

Загружаем три картинки, в одной из них в файле определен алфа канал (with-alpha)
```
(define (load-check-img name-image)
 (let ((image (SDL:load-image name-image)))
  (if (not (SDL:surface? image))
     (begin
       (display (string-append "Unable load image:" name-image "\n"))
       (exit 1)))
  image))

(define image-with-alpha (load-check-img "with-alpha.png"))
(define image-without-alpha (load-check-img "without-alpha.png"))
(define background (load-check-img "bg.png"))
```

Выводим на экран фон:
```
(define src (SDL:make-rect 0 0 (SDL:surface:w background) (SDL:surface:h background)))
(define dst (SDL:make-rect 0 0 (SDL:surface:w background) (SDL:surface:h background)))

(SDL:blit-surface background src screen dst)
```

Выводим изображение имеющее альфа канал:
```
(SDL:surface-alpha! image-with-alpha 255)
(SDL:rect:set-x! src 0)
(SDL:rect:set-y! src 0)
(SDL:rect:set-w! src (SDL:surface:w image-with-alpha))
(SDL:rect:set-h! src (SDL:surface:h image-with-alpha))

(SDL:rect:set-x! dst 40)
(SDL:rect:set-y! dst 50)
(SDL:rect:set-w! dst (SDL:rect:w src))
(SDL:rect:set-h! dst (SDL:rect:h src))
```

Аналогично предыдущем изображению выводим изображение не имеющее альфа канала:
(SDL:surface-alpha! image-without-alpha 128)
(SDL:rect:set-x! src 0)
(SDL:rect:set-y! src 0)
(SDL:rect:set-w! src (SDL:surface:w image-without-alpha))
(SDL:rect:set-h! src (SDL:surface:h image-without-alpha))

(SDL:rect:set-x! dst 180)
(SDL:rect:set-y! dst 50)
(SDL:rect:set-w! dst (SDL:rect:w src))
(SDL:rect:set-h! dst (SDL:rect:h src))

(SDL:blit-surface image-without-alpha src screen dst)

По результатам работы программы видно что для первого изображения вызов функции surface-alpha! значения не имеет(его можно даже закомментировать)! Рисунок получается не прозрачный, а области которые в файле отмечены как альфа, не копируются. С другой стороны для изображения не имеющего альфа канала, значение альфа канала очень важно, оно определяет степень прозрачности полученного изображения.

Установку альфы можно использовать с ключевым цветом, я чуть изменил предыдущий пример и добавил туда установку альфа цвета, для пингвина с ключевым цветом, он стал полупрозрачным. см файл: **l04-05b_alpha-sdl.scm**
(SDL:surface-alpha! image 70)

Ну и на этом о ключевом цвете и альфе все, переходим к анимации.

Анимация, первая попытка.

Наша анимация будет заключаться в движении ста статичных картинок по экрану, отскакивающих от бортов. Двигаться естественно будут пингвины.

Полный листинг примера в файле: **l04-06_anim1.scm**

Вначале мы определяем структуру которая будет отслеживать положение и скорость каждого спрайта, вся работа с ней будет происходить через функции — доступа(ацессоры).

```
(define NUM_PENGUINS 100)
(define MAX_SPEED    6)

(define (make-penguin x y dx dy)
  (cons (cons x y)
        (cons dx dy)))

(define (penguin:x p)
  (caar p))
(define (penguin:y p)
  (cdar p))
(define (penguin:dx p)
  (cadr p))
(define (penguin:dy p)
  (cddr p))
```

```
(define (penguin-print p)
  (display (string-append
          "x: "    (number->string (penguin:x p))
          ", y: "  (number->string (penguin:y p))
          ", dx: " (number->string (penguin:dx p))
          ", dy: " (number->string (penguin:dy p)) "\n")))
```

Далее как обычно инициализируем библиотеку SDL и окно для отображения нашей сцены.

Определяем функции работающие со всем массивом пингвиньих спрайтов.Функция инициализации списка пингвинов, проста создает список сослучайными значениями позиций и скоростей для каждого спрайта.

```
(define (init_penguins)
  (let ((i 0) (penguins '()))
    (do ((i 0 (+ i 1)))
       ((>= i NUM_PENGUINS))
      (set! penguins
           (cons
            (make-penguin
             (random s_width (random-state-from-platform))
             (random s_height (random-state-from-platform))
             (- (random (* MAX_SPEED 2) (random-state-from-platform)) MAX_SPEED)
             (- (random (* MAX_SPEED 2) (random-state-from-platform)) MAX_SPEED))
            penguins)))
    penguins))
```

Функции перемещения пингвинов

Перемещение одного пингвина

```
(define (move_penguin p surface)
  (let ((x 0) (y 0) (dx (penguin:dx p)) (dy (penguin:dy p)))
    (set! x (+ (penguin:x p) dx))
    (set! y (+ (penguin:y p) dy))
    (if (or (< x 0)
            (> x (- (SDL:surface:w surface) 1)))
        (set! dx (- dx)))
    (if (or (< y 0)
            (> y (- (SDL:surface:h surface) 1)))
        (set! dy (- dy)))
    (make-penguin x y dx dy)))
```

Перемещение всего списка, возвращает список перемещенных пингвинов.

```
(define (move_penguins penguins surface)
  (map (lambda (p) (move_penguin p surface)) penguins))
```

Функция рисования пингвинов на экране

```
(define (draw_penguins penguins image surface)
  (let ((src (SDL:make-rect 0 0 (SDL:surface:w image) (SDL:surface:h image)))
        (dst (SDL:make-rect 0 0 (SDL:surface:w image) (SDL:surface:h image)))
        (half_w (quotient (SDL:surface:w image) 2))
        (half_h (quotient (SDL:surface:h image) 2)))
    (for-each (lambda (p)
              (SDL:rect:set-x! dst (- (penguin:x p) half_w))
              (SDL:rect:set-y! dst (- (penguin:y p) half_h))
              (SDL:blit-surface  image  src surface dst))
            penguins)))
```

Далее описываем функции загрузки изображений:
;; загрузка картинки
```
(define (load-check-img name-image)
  (let ((image (SDL:load-image name-image)))
    (if (not (SDL:surface? image))
        (begin
          (display (string-append "Unable load image:" name-image "\n"))
          (exit 1)))
    image))

(define (load-check-bmp name-image)
  (let ((image (SDL:load-bmp name-image)))
    (if (not (SDL:surface? image))
        (begin
          (display (string-append "Unable load image:" name-image "\n"))
          (exit 1)))
    image))
```

Грузим изображения,их всего два бекграунд и изображение пингвина.
```
(define background (load-check-bmp "bg.bmp"))
(define penguin-image (load-check-bmp "smallpenguin.bmp"))
```

Для изображения пингвина задаем ключевой цвет:
;;зададим colorkey для изображения пингвина
```
(define format-image (SDL:surface-get-format penguin-image))
(define colorkey (SDL:map-rgb format-image 0 0 255))
(SDL:surface-color-key! penguin-image colorkey)
```

Сгененируем пингвинов с помощью определенной ранее функции
```
(define penguins (init_penguins))
```

Ну а теперь можно приступать к самой анимации, т.е выполнению перемещения множества спрайтов в цикле, у нас цикл не бесконечный, мы ограничим его конкретным числом шагов:
```
(define MAX_CYCLES 1300)
```

;;создадим границы прямоугольников копирования
```
(define src (SDL:make-rect 0 0 (SDL:surface:w background) (SDL:surface:h background)))
(define dst (SDL:make-rect 0 0 (SDL:surface:w background) (SDL:surface:h background)))

(let ((i 0))
  (do ((i 0 (+ i 1)))
      ((>= i MAX_CYCLES))
    (begin
      (SDL:blit-surface background src screen dst)            ;;рисуем фон
      (draw_penguins  penguins  penguin-image screen)    ;;рисуем пингвинов
      (SDL:update-rect screen 0 0 0 0)                        ;; отображаем все на экране
      (set! penguins (move_penguins penguins screen)))))  ;; перемещаем пингвинов.
```

Вот такой вот не сложный пример анимации.

Анимация улучшенная версия.

Наш предыдущий пример хорошо работает, но не быстро и существуют два варианта что бы увеличить скорость его работы. Первый вариант это установить на поверхность screen флаг двойного буфера. (doublebuffer). При этом все отображения спрайтов инкрементально добавляемые к поверхности screen будут копироваться в предварительный

буфер, находящийся в общей оперативной памяти, а затем функцией (SDL:flip screen) этот буфер будет отображаться на экране. И второй вариант ускорения это приведение всех копируемых изображений к формату экрана, это приведение позволит в дальнейшем обойтись без преобразования форматов при копировании изображений, хоть эта работа и скрыта от пользователя. она тем не менее может существенно замедлить формирование конечного изображения.

Полный листинг примера в файле: **l04-07_anim2.scm**

Поскольку данный пример от предыдущего отличается лишь парой мелких изменений опишу лишь их:
Первое изменение при инициализации окна, здесь как раз и устанавливается двойной буфер:
(define screen (SDL:set-video-mode s_width s_height 16 'doublebuf))

Второе изменение при загрузке изображений, вначале мы формируем временную поверхность, а затем на ее основе формируем поверхность приведенную к формату экранного изображения display-format

```
(define temp (load-check-bmp "bg.bmp"))
(define background (SDL:display-format temp))


(set! temp (load-check-bmp "smallpenguin.bmp"))
```

зададим colorkey для изображения пингвина

```
(define format-image (SDL:surface-get-format temp))
(define colorkey (SDL:map-rgb format-image 0 0 255))
(SDL:surface-color-key! temp colorkey)

(define penguin-image (SDL:display-format temp))
(if (not (SDL:surface? penguin-image))
   (begin
     (display (string-append "Unable to conver bitmap penguin image \n"))
     (exit 1)))
```

И далее в цикле отображения вместо update используется flip, переключение буфера.

```
(let ((i 0))
  (do ((i 0 (+ i 1)))
     ((>= i MAX_CYCLES))
    (begin
     (SDL:blit-surface background src screen dst)          ;;рисуем фон
     (draw_penguins  penguins  penguin-image screen)   ;;рисуем пингвинов
     (SDL:flip screen)                                     ;; отображаем все на экране
     (set! penguins (move_penguins penguins screen)))))  ;; перемещаем пингвинов.
```

В принципе это все, единственное что хотелось бы добавить, что в документации написано что функция SDL:display-format (SDL DisplayFormat) может уничтожить альфа канал в изображениях с альфа каналом.

Заканчивая с анимацией мне хотелось бы упомянуть о двух файлах: l04-06_anim1tm.scm и l04-07_anim2tm.scm, которые являются точной копией предыдущего и текущего примеров, но в которых производиться замер быстродействия вывода кадров анимации на экран.

У меня для первого варианта оказалось 50 кадров в секунду, а для второго 103. А если сравнить с чистым кодом на Си из PLG, то там производительность, оптимизированного кода 112 кадров в секунду. Согласитесь, разница между двумя оптимизированными версиями не большая.

Получение и обработка сообщений.

Взаимодействие SDL с пользователями происходит посредством обработки программой SDL цикла сообщений. Базовый шаблон такой обрабоки, в котором ничего не происходит, кроме приема сообщений и вывода их на экран, приведен в примере: **l04-08_base-events.scm**

Инициализация окна происходит как и в предыдущих примерах, только в место ожидания в конце запускаем цикл обработки сообщений, я выбрал для себя цикл с формой do

```
(let ((stop #f) (ret #t) (event (SDL:make-event)) (type #f) (cont #t))
  (do ((cont #t (identity #t)))
      ((or stop
           (not (SDL:wait-event event))))
    (begin
      (display "Event:")
      (display event) (newline)
      (set! type (SDL:event:type event))
      (display (string-append "Get event type:"))
      (display type)
      (display "\n")
      (if (equal? type 'quit)
          (begin
            (display "Get quit event\n")
            (set! stop #t)))
      )))
```

Цикл ждет сообщения в вызове SDL:wait-event и далее обрабатывает входящее сообщение. В нашем случае вся обработка сводится к определению типа сообщения и выводу сообщения на экран. Еще обрабатывается тип сообщения quit, завершающий цикл обработки сообщений.

Обработка сообщений от мыши.

Пример в файле: **l04-08_mouse-events.scm**

Этот пример слегка расширяет обработку сообщений из предыдущего примера. В основном обработку попадают сообщения связанные с мышью: mouse-motion, mouse-button-down, mouse-button-up. Обработка состоит в выводе на терминал информационных сообщений.

```
(let ((stop #f) (ret #t) (event (SDL:make-event)) (type #f) (cont #t))
  (do ((cont #t (identity #t)))
      ((or stop
           (not (SDL:wait-event event))))
    (begin
      (set! type (SDL:event:type event))
      (cond
       ((equal? type 'mouse-motion)
        (begin
          (display "Mouse motion: ")
          (display (string-append "New pos: x: " (number->string (SDL:event:motion:x event))
                                  ", y: " (number->string (SDL:event:motion:y event)) "\n"))))
       ((equal? type 'mouse-button-down)
          (display "Mouse button down: ")
          (display (string-append "btn: " (symbol->string (SDL:event:button:button event))
                                  ", x: " (number->string (SDL:event:button:x event))
                                  ", y: " (number->string (SDL:event:button:y event)) "\n")))
```

```
    ((equal? type 'mouse-button-up)
      (display "Mouse button up: ")
      (display (string-append "btn: " (symbol->string (SDL:event:button:button event))
                              ", x: " (number->string (SDL:event:button:x event))
                              ", y: " (number->string (SDL:event:button:y event)) "\n")))
    ((equal? type 'quit)
     (begin
      (display "Get quit event\n")
      (set! stop #t)))
   ))))
```

Обработка клавиатурных сообщений.

Работа с клавиатурой ничем не отличается от обработки сообщений от мыши, только типы сообщений другие, остальное все аналогично.

```
(let ((stop #f) (ret #t) (event (SDL:make-event)) (type #f) (cont #t))
  (do ((cont #t (identity #t)))
      ((or stop
           (not (SDL:wait-event event))))
    (begin
      (set! type (SDL:event:type event))
      (cond
       ((equal? type 'key-down)
        (begin
          (display "Key down: ")
          (display (string-append "Keysym: '" (symbol->string (SDL:event:key:keysym:sym
event))))
          (display "', Mod: ") (display (SDL:event:key:keysym:mod event)) (newline)
          (if (equal? (SDL:event:key:keysym:sym event) 'q)
              (begin
                (display "'Q' pressed, exiting!\n")
                (set! stop #t)))))
       ((equal? type 'key-up)
        (begin
          (display "Key up: ")
          (display (string-append "Keysym: '" (symbol->string (SDL:event:key:keysym:sym
event))))
          (display "', Mod: ") (display (SDL:event:key:keysym:mod event)) (newline)))
       ((equal? type 'quit)
        (begin
          (display "Get quit event\n")
          (set! stop #t)))
       ))))
```

Примера для работы с джойстиком у меня нет, за отсутствием такового. Думаю там ничего сложного нет.

Многопоточная работа с SDL.

Работа с нитями из библиотеки SDL в данном биндинге не предусмотрена. При необходимости работать с потоками разработчик предлагает воспользоваться внутренними средствами guile, что собственно и демонстрируется в нашем примере. Задача простая есть общая переменная и три потока, в каждом из потоков выполняется инкремент общей переменной и сон продолжительностью в некую случайную величину. Надо досчитать до 20(или чуть больше).

Полный листинг примера в файле: **l04-11_threadings.scm**

Инициализация проходит аналогично предыдущим примерам, единственное отличие это подключение библиотеки для работы с нитями.

```
(use-modules (ice-9 threads))
```

Создаем набор переменных с которыми будем работать, во первых это счетчик который будут увеличивать процессы работающие в нитях, counter_mutex, специальная переменная регулирующая доступ к счетчику из различных нитей и флаг завершения нитей, который проверяется каждым процессом в нити и служит сигналом завершения работы нити(а у нас всех нитей).

```
(define counter 0)
(define counter_mutex (make-mutex))
(if (not (mutex? counter_mutex))
   (begin
     (display "Can't create mutex\n")
     (exit 1)))
(define exit_flag #f)
```

Определяем функцию которая будет выполняться в каждой нити, ее задача получить разрешение у мьютекса на доступ к счетчику, заблокировав его, увеличить счетчик и освободить мьютекс, вся работа выполняется до тех пор пока кто либо не установит флаг завершения работы exit_flag:

```
(define (ThreadEntryPoint data)
  (let ((name data) (i 0))
    (do ((i 0 (identity i)))
       ((identity exit_flag))
      (begin
        (display (string-append "This is: " name "\n"))
        (lock-mutex counter_mutex)
        (display (string-append "The counter is curently: " (number->string counter) "\n"))
        (set! counter (+ counter 1))
        (unlock-mutex counter_mutex)
        (SDL:delay (random 3000 (random-state-from-platform)))))
    (display "Exit from:") (display name) (newline)))
```

Далее идет основной цикл главной процедуры, в начале мы создаем три нити, которые начинают работать, а основной цикл ждет пока не счетчик меньше 20, как только счетчик перевалит за эту магическую цифру, основная процедура устанавливает флаг завершения и немного ждет, пока все нити не завершат свою работу. Хотя факт завершения работы нитей и не проверяется.

```
(let ((thread1 (make-thread ThreadEntryPoint "Thread 1"))
      (thread2 (make-thread ThreadEntryPoint "Thread 2"))
      (thread3 (make-thread ThreadEntryPoint "Thread 3"))
      (i 0))
  (do ((i 0 (identity i)))
      ((> counter 20))
    (SDL:delay 1000))
  (set! exit_flag #t)
  (display "exit_flag has been set by main()\n")
  (SDL:delay 3500))
```

Ну вот и все о нитях в guile. Для начала изучения работы с нитями вполне годный пример.

Программирование Звука.

В книжке PLG в этом месте разобран достаточно сложны пример работы со звуком в котором микшируются несколько звуков, я постараюсь разобрать здесь пример попроще, с загрузкой звука и его проигрыванием/остановкой/возобновлением, а работу с микшированием просто добавлю во втором примере к этой теме.

Полный листинг примера в файле: **l04-12_audio-sdl.scm**

Основное отличие от предыдущих примеров при инициализации, надо добавить модуль обеспечивающий работу со звуком (sdl mixer) и затем указать необходимость инициализации аудио системы при вызове SDL:init

```
(use-modules ((sdl sdl) #:prefix SDL:)
             ((sdl mixer) #:prefix MIX:)
             (srfi srfi-1)
             (srfi srfi-2))
….
(if (not (equal? (SDL:init '(video audio)) 0))
    (begin
      (display "Can't initialize SDL!\n")
      (exit 1)))
```

Да и еще надо указать модуль позволяющий принимать несколько значений из возвращаемой функции, такая функция используется в дальнейшем MIX:device-ffc — функция возвращающая текущие настройки аудио системы, частоту, формат и используемые каналы:

```
(use-modules (ice-9 receive))
```

После этого можно начинать работать со звуком.Первым делом открываем миксер с необходимыми нам параметрами, вызовом функции open-audio [freq [format [stereo [chunksize]]]], но я устанавливать никаких параметров не буду, так что вызовем ее без параметров.

```
(MIX:open-audio)
```

Загружаем музыку. На самом деле тут есть небольшая тонкость, мы можем загрузить один и тот же файл как музыку(load-music) и как звук(load-wave). Отличие этих действий состоит в том, что в дальнейшем их проигрывание должно выполняться соответствующими функциями play-music и play-channel. Разница между которыми, в свою очередь, состоит в том что проигрывание музыки возможно только одной, и если вы вызовете повторно эту функцию то текущая проигрываемая музыка замениться новой, а при проигрывании звука, вы во первых не отключаете проигрывание музыки, во вторых каждый вызов проигрывания звука добавляет еще один микширующийся с другими воспроизводимыми звуками, заданными ранее. Таким вот простым образом можно достичь весьма сложных звуковых эффектов. Ну и уж коль речь зашла о звуковых эффектах, при проигрывании звука есть возможность установить виртуальное положение звука, т.е угол откуда он звучит и дистанцию, функцией: MIX:set-position. Но поскольку я все делал на ноутбуке, как все это работает я не проверял. Итак загружаем музыку:

```
(define (load-music)
  (MIX:load-music "background.ogg"))
;; load the files
(define background (load-music))
```

Итак файл с музыкой загружен но еще не звучит. Его воспроизведение мы будем выполнять по нажатии клавиши p, что и укажем в цикле обработки сообщений.

```
(let ((ch 0) (freq #f) (format #f) (channels #f) (event (SDL:make-event)) (type #f) (cont #t))
  (let loop ((i 1))
    (and (and cont
              (SDL:wait-event event))
         (begin
           (set! type (SDL:event:type event))
           (cond
            ((equal? type 'key-down)
             (begin
               (display "Key down: ")
               (display (string-append "Keysym: '" (symbol->string
                                                    (SDL:event:key:keysym:symevent))))
               (display "', Mod: ") (display (SDL:event:key:keysym:mod event)) (newline)
               (cond
                  ((equal? (SDL:event:key:keysym:sym event) 'q)
                   (begin
                    (display "'Q' pressed, exiting!\n")
                    (set! cont #f)))
                  ((equal? (SDL:event:key:keysym:sym event) 'g)
                   (begin
                     (display "device-ffc is: ")
                     (receive (l_freq l_format l_channels) (MIX:device-ffc)
                              (set! freq l_freq)
                              (set! format l_format)
                              (set! channels l_channels))
                     (display "Freq: ") (display freq)
                     (display ", format: ") (display format )
                     (display ", Channels: ") (display channels)
                     (newline)))
                          ;;играть музыку
                  ((equal? (SDL:event:key:keysym:sym event) 'p)
                   (begin
                     (display "Play music command\n")
                     (set! ch (MIX:play-music background))
                     (display "Channel play: ")
                     (display ch)
                     (newline)))
                           ;;остановить проигрывание музыки
                  ((equal? (SDL:event:key:keysym:sym event) 's)
                   (begin
                     (display "Stop music command\n")
                     (MIX:halt-music)
                     ))
                          ;;пауза в проигрывании музыки
                  ((equal? (SDL:event:key:keysym:sym event) 'h)
                   (begin
                     (display "Pause music command\n")
                     (MIX:pause-music)
                     ))
                          ;;возобновить проигрывание музыки
                  ((equal? (SDL:event:key:keysym:sym event) 'r)
                   (begin
                     (display "Resume music command\n")
                     (MIX:resume-music)
                     ))
```

```
    )))
  ((equal? type 'quit)
   (begin
    (display "Get quit event\n")
    (set! cont #f)))
  )
 (loop (+ i 0))))))
```

В общем из кода видно, что каждому действию сопоставляется соответствующая функция.

Теперь опишем как добавить микширование.
Полный листинг примера можно найти в файле: **l04-12_audio2-sdl.scm**
Как я уже сказал для загрузки звуковых эффектов проигрываемых с помощью функции play-channel мы используем функцию load-wave

```
(define fx (MIX:load-wave "fx.ogg"))
(display fx) (newline)
```

Загруженный звук мы будем воспроизводить по нажатию клавиши b. вот код ее обработки из цикла обработки сообщений:

```
((equal? (SDL:event:key:keysym:sym event) 'b) ;;проигрывание звука fx
 (begin
  (display "Bang fx!\n")
  (set! ch (MIX:play-channel fx))
  ;;(MIX:set-position 1 90 0)
  (display "Playing fx with channel: ")
  (display ch)
  (newline)
  ))
```

В приведенном мной примере канал для микширования выделяется автоматически — первый свободный. Теперь можно запустив прогрывание основного фона — музыки, клавишей p, быстро быстро нажимать клавишу b, добавлять к фоновой музыке звуковой эффект fx, накладывающийся на воспроизведение основной музыки.

Совместное использование OpenGL и SDL.

Для демонстрации возможности использования OpenGL нам необходимо что бы в guile был инсталлирован биндинг OpenGL. Рисовать будем пример из PGL, только на scheme.

Полный листинг примера находиться в файле: **l04-13a_opengl-sdl.scm**
Обратите внимание что здесь я разбираю файл 13a а не просто 13, их отличие заключается в том что библиотека gl enums в 13a подгружается отдельно и имеет отдельный префикс ENUM. Я посчитал сделать это необходимым, т.к простое использование GL, приводит к тому что функции из enums загружаются и используются без префикса, что не очень хорошо в больших проектах.

Подключаем необходимые для работы модули:

```
(use-modules ((gl)   #:prefix GL:))
(use-modules ((gl enums) #:prefix ENUM:))
(use-modules ((sdl sdl) #:prefix SDL:)
        (srfi srfi-1)
        (srfi srfi-2))
```

Инициализация происходит аналогично обычному SDL

```
(if (not (equal? (SDL:init 'video) 0))
   (begin
     (display "Can't initialize SDL!\n")
     (exit 1)))
```

После инициализации выполняем установку атрибутов для настройки OpenGL

```
(SDL:gl-set-attribute (ENUM:get-p-name  doublebuffer) 1)
(SDL:gl-set-attribute (ENUM:get-p-name  red-bits) 5)
(SDL:gl-set-attribute (ENUM:get-p-name  green-bits) 6)
(SDL:gl-set-attribute (ENUM:get-p-name  blue-bits) 5)
```

Хотя это и не обязательно.

Создаем окно, указывая в параметрах что мы будем работать с OpenGL

```
(define screen (SDL:set-video-mode s_width s_height 16 '(doublebuf opengl)))
(if (not (SDL:surface? screen))
   (begin
     (display "Can't set videomode\n")
     (exit 1)))
```

Все, теперь в созданном окне можно использовать команды OpenGL, для построение необходимого нам изображения.

```
(GL:gl-viewport 80 0 480 480)

(GL:set-gl-matrix-mode (ENUM:matrix-mode  projection))
(GL:gl-load-identity)

(GL:gl-frustum -1.0 1.0 -1.0 1.0 1.0 100.0)
(GL:set-gl-clear-color 0 0 0 0)
(GL:set-gl-matrix-mode (ENUM:matrix-mode modelview))
(GL:gl-load-identity)
(GL:gl-clear (ENUM:clear-buffer-mask color-buffer))
(GL:gl-begin (ENUM:begin-mode  triangles)
 (GL:gl-color 1.0 0 0)
          (GL:gl-vertex 0.0 1.0 -2.0)
          (GL:gl-color 0 1.0 0)
          (GL:gl-vertex 1.0 -1.0 -2.0)
          (GL:gl-color 0  0  1.0)
          (GL:gl-vertex -1.0  -1.0 -2.0))
```

И отображение на экране:

```
(SDL:gl-swap-buffers)
```

Вот на этом и все! т.е вобще все! примеры закончены, удачного и счастливого вам программирования.

Примеры PGL перевел на Scheme Гагин Михаил aka NuINu