Eiffel SDL multimedia library (ESDL)
API extensions
SEMESTER THESIS

Benno Baumgartner

30th March 2004

http://eiffelsdl.sf.net

Student: Benno Baumgartner (benno@student.ethz.ch)

Student-No: 98-727-589

Supervising Assistant: Till G. Bay

Supervising Professor: Bertrand Meyer

Contents

List of Figures

1 Introduction

ESDL is a wrapper for SDL, the Simple Directmedia Layer library [4]. SDL is an open source C library that is very popular among Linux game developers. Besides Linux, SDL is also available for many other platforms such as Windows and MacOS. SDL is a very performing multimedia library composed of various subsystems for graphics, sound, networking, threading, CD-ROM access, window management, joystick handling, event handling and time management. The aim of this project is to provide the Eiffel community with a wrapper of SDL that has an easy to understand and memorable API, that is as performing as SDL is and that runs on different platforms. At the moment Windows and Linux are supported.

ESDL is being developed subsystem by subsystem. This semester thesis report describes the implementation of API extensions of ESDL 0.0.1. Among others I have extended the time subsystem, added extensions to the graphical subsystem that allows a developer to handle RGB bitmap graphics in a very flexible and easy way. I also added collision detection support.

These extensions allow to build complex games in a very easy way as the example application ESDL_Racing in subsection 3.5 on page 29 demonstrates.

The graphical subsystem and the event loop are not described in this report but in the documentation of the first version of ESDL [1].

2 Implementation of the Extensions

2.1 Overview

In subsection 2.2 until 2.9 on page 19 all classes related to ESDL_DRAWABLE are introduced. This classes provide powerful and very flexible functionalities to work with two dimensional RGB bitmaps in ESDL.

In subsections 2.10 on page 19 until 2.10.5 on page 21 all classes related to the time subsystem are covered. The time subsystem provides the programmer with functionalities related to time measurement that is used for all the animation support.

In subsection 2.11 on page 21 all classes related to collision detection are explained.

2.2 ESDL_DRAWABLE

Every class in ESDL describing an object that can be drawn to an ESDL_SURFACE should implement ESDL_DRAWABLE. ESDL_DRAWABLE is a deferred class. It's main features are *draw* and *draw_part*:

deferred class
ESDL_DRAWABLE

```
feature -- commands

    draw (a_surface: ESDL_SURFACE) is
            -- Draws 'current' to 'a_surface'
        require
            a_surface_not_void: a_surface /= void
        deferred
        end

    draw_part (rect: ESDL_RECT; a_surface: ESDL_SURFACE) is
            -- Draws rectangular part of 'current' defined by 'rect' to 'a_surface'
        require
            a_rect_not_void: rect /= void
            a_surface_not_void: a_surface /= void
        deferred
        end

feature -- status

    width: INTEGER is
            -- The 'width' of 'current'
            -- The 'width' is the position of the right most pixel
        deferred
        ensure
            Result >= 0
        end

    height: INTEGER is
            -- The 'height' of 'current'
            -- The 'height' is the position of the bottom most pixel
        deferred
        ensure
            Result >= 0
        end

    x: INTEGER
            -- The distance of 'current' to the coordinate origin
            -- in 'x' (left -> right) direction

    set_x (an_integer: INTEGER) is
            -- Sets 'x'
        require
```

```
        positive: an_integer >= 0
    do
        x := an_integer
    ensure
        set: x = an_integer
    end

y: INTEGER
        -- The distance of 'current' to the coordinate origin
        -- in 'y' (top -> down) direction

set_y (an_integer: INTEGER) is
        -- Sets 'y'
    require
        positive: an_integer >= 0
    do
        y := an_integer
    ensure
        set: y = an_integer
    end

set_x_y (an_x: INTEGER; an_y: INTEGER) is
        -- Sets the 'x' and 'y' value
    require
        positive: an_x >= 0 and an_y >= 0
    do
        x := an_x
        y := an_y
    ensure
        set: an_x = x and an_y = y
    end

invariant

x_positive: x >= 0
y_positive: y >= 0

end
```

The pixel with the coordinate (0,0) is always at the top left of *a_surface* passed to *draw* and *draw_part*. The coordinate (0,0) is called the origin. The *width* is defined as the distance between the origin and the right most pixel of the

drawable. The *height* is defined as the distance between the origin and the bottom most pixel of the drawable. This requires, that the x and y position is always positive. This limitation seams to be stronger then needed at first sight. But only that way the *width* and *height* of the ESDL_DRAWABLE_CONTAINER can be implemented consistent to the definition without extending the interface of ESDL_DRAWABLE.

At the moment 11 effective classes inherit from ESDL_DRAWABLE and can be used as drawables. See Figure 1 on the following page.

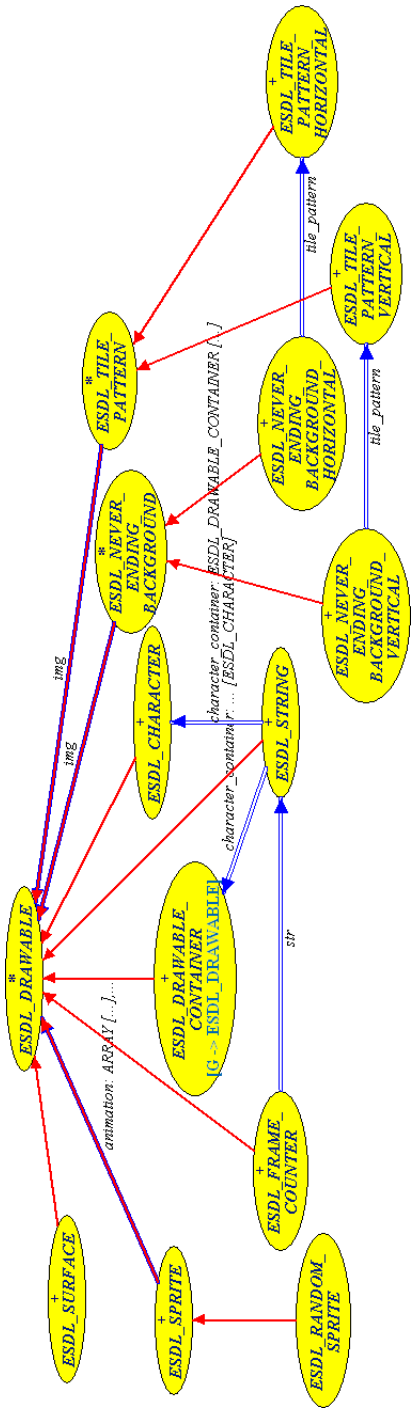

Figure 1: ESDL_DRAWABLE BON Diagram

2.3 ESDL_SURFACE

ESDL_SURFACE is an ESDL_DRAWABLE.

ESDL_SURFACE is a two dimensional RGB bitmap. You can draw any ESDL_DRAWABLE to an ESDL_SURFACE. If you want to program in ESDL you first need a base surface: the screen. You can get a screen with the following sequence of code:

```
local
    esdl: ESDL
    screen: ESDL_SURFACE
do
    create esdl.make
    esdl.set_video_surface_width (width)
    esdl.set_video_surface_height (height)
    esdl.set_video_bpp (resolution)
    esdl.set_fullscreen (True)
    esdl.set_doublebuffered (True)
    esdl.set_hardware_surface (True)
    esdl.set_hwpalette (True)
    esdl.showcursor (False)
    esdl.initialize_video_surface
    screen := esdl.video_surface
end
```

Where *width* is the width of the screen and *height* is the height of the screen. You can use any combination as *width* and *height*, but you should use one that is directly supported by your graphics card, if you want to run your program in full screen mode. A supported resolution is in this case faster than an arbitrary combination. In Windows you can find out which resolutions your card supports by clicking right on the desktop and selecting Properties ▷ Settings. In the Screen area you can see all possible combinations. Very common ones are: 640 by 480 pixels, 800 by 600 pixels and 1024 by 768 pixels. Those should be supported by every modern video card.

The *resolution* is the number of bits used to represent color information. You can use 8, 16, 24 or 32 bits as *resolution*. You should avoid using a *resolution* of 24 bits because such a *resolution* is a lot slower then the others.

Once you have a screen you can draw other ESDL_DRAWABLEs to it. You can also create your own ESDL_SURFACEs. There exist two possibilities for doing so:

1. *make_from_image* (*an_image_file_name*: *STRING*)
   This feature loads an image from a file and creates an ESDL_SURFACE

from this image. The format is the same as the format of the current screen. This feature uses the SDL_image library to load an image [6]. Supported file types are: BMP, PNM, XPM, XCF, PCX, GIF, JPG, TIF, PNG and LBM.

2. *make* (*w*:*INTEGER*; *h*:*INTEGER*)
   This feature creates an ESDL_SURFACE with width *w* and height *h*. The format is the same as the format of the current screen.

You can give an ESDL_SURFACE a so called key color with the command *set_colorkey*. Every pixel with the color defined with *set_colorkey* is not drawn, but it will be transparent.

ESDL_SURFACEs are collected by the Eiffel garbage collector. If you want to force that (at start of a new level for example) inherit from the class MEMORY and call *full_collect*.

2.3.1 Example

```
local
   picture: ESDL_SURFACE
do
   create picture.make_from_image ("lovely.gif")
   picture.set_colorkey (255, 255, 255)
   picture.draw (screen)
   screen.redraw
end
```

2.4 ESDL_SPRITE

An ESDL_SPRITE is an ESDL_DRAWABLE.

ESDL_SPRITE is a player for an ARRAY of ESDL_DRAWABLEs. It works like a movie player for a movie, where the player is the ESDL_SPRITE and the movie is the ARRAY of ESDL_DRAWABLEs. To use an ESDL_SPRITE you first create an ARRAY of ESDL_DRAWABLEs and pass it to an ESDL_SPRITE with *make* (*an_animation*: *ARRAY* [*ESDL_DRAWABLE*]). You can start and stop the playback with the feature *set_stop* (*b*: *BOOLEAN*). You can set the playback speed with the feature *set_frame_rate* (*i*: *INTEGER*). The *frame_rate* indicates how many different pictures of the animation are shown per second. The *frame_rate* can be less than zero, the animation is then played in reverse order. The *frame_rate* **can not** be zero, use *set_stop* (*true*) instead. If *repeat* is true the animation is played over and over again, otherwise the playback ends at the last frame.

2.4.1 ESDL_ANIMATION

An ESDL_ANIMATION is an ARRAY of ESDL_DRAWABLEs.

An ESDL_ANIMATION can be played by an ESDL_SPRITE.

Besides the normal ARRAY creations: *make* (*min_index*, *max_index*: *INTEGER*) and *make_from_array* (*a*: *ARRAY* [*ESDL_DRAWABLE*]) you can also create an ESDL_ANIMATION with *make_from_file* (*file_name*: *STRING*) where *file_name* is the name of a file describing the animation. The format of the animation file is a list of file names prefixed by the key color. More formal in EBNF (where nl means new line):

```
File      ::= Color_key File_list eof
Color_key ::= color_key_red nl color_key_green nl color_key_blue
File_list ::= [nl file_name_of_picture]*
```

Each *file_name_of_picture* is the file name of a picture like a BMP or GIF file. An example file *f1.anim* from the ESDL Racing example on page 29 looks like the animation script shown below.

```
0
0
0
./pics/f11.gif
./pics/f12.gif
./pics/f11.gif
./pics/f13.gif
```

2.4.2 ESDL_RANDOM_SPRITE

An ESDL_RANDOM_SPRITE is an ESDL_SPRITE where the frames are not shown in order but a randomly chosen frame is shown. If *respect_speed* is *true* then a new frame is chosen each $\frac{1}{\text{frame\_ rate}}$ seconds, otherwise an other frame is chosen each call of *draw* and *draw_part*.

2.4.3 Example

```
local
   anim: ESDL_ANIMATION
   sprite: ESDL_SPRITE
do
   create anim.make_from_file ("f1.anim")
   create sprite.make (anim)
   sprite.draw (screen)
   screen.redraw
end
```

2.5 ESDL_TILE_PATTERN

An ESDL_TILE_PATTERN is an ESDL_DRAWABLE.

ESDL_TILE_PATTERN is deferred. The two children ESDL_TILE_PATTERN_HORIZONTAL and ESDL_TILE_PATTERN_VERTICAL make ESDL_TILE_PATTERN effective.

In an ESDL_TILE_PATTERN an ESDL_DRAWABLE works like a tile that is glued together a number of times in either horizontal or vertical direction to build a pattern. The feature *set_times* (*an_integer*: *INTEGER*) allows to define how many times the ESDL_DRAWABLE is repeated to build the pattern.

2.5.1 ESDL_TILE_PATTERN_HORIZONTAL

An ESDL_TILE_PATTERN_HORIZONTAL is an ESDL_TILE_PATTERN.

In an ESDL_TILE_PATTERN_HORIZONTAL an ESDL_DRAWABLE is glued together in horizontal direction.

2.5.2 ESDL_TILE_PATTERN_VERTICAL

An ESDL_TILE_PATTERN_VERTICAL is an ESDL_TILE_PATTERN.

In an ESDL_TILE_PATTERN_VERTICAL an ESDL_DRAWABLE is glued together in vertical direction.

2.5.3 Example

```
local
   picture: ESDL_SURFACE
   tph: ESDL_TILE_PATTERN_HORIZONTAL
do
   create picture.make_from_image ("lovely.gif")
   create tph.make (picture, 3)
   tph.draw (screen)
   screen.redraw
end
```

2.6 ESDL_NEVER_ENDING_BACKGROUND

An ESDL_NEVER_ENDING_BACKGROUND is an ESDL_DRAWABLE.

ESDL_NEVER_ENDING_BACKGROUND is deferred. The two children ESDL_NEVER_ ENDING_ BACKGROUND _HORIZONTAL and ESDL_NEVER_ ENDING_BACKGROUND _VERTICAL make ESDL_ NEVER _ ENDING_ BACKGROUND effective.

You can think of an ESDL_NEVER_ENDING_BACKGROUND as the kind of background used in old movies if a scene in a car is recorded. It is a large picture that is pulled behind the car. An ESDL_NEVER_ ENDING_BACKGROUND is very similar to that concept too. It expects an ESDL_DRAWABLE as argument. This ESDL_DRAWABLE is glued together over and over again, either in

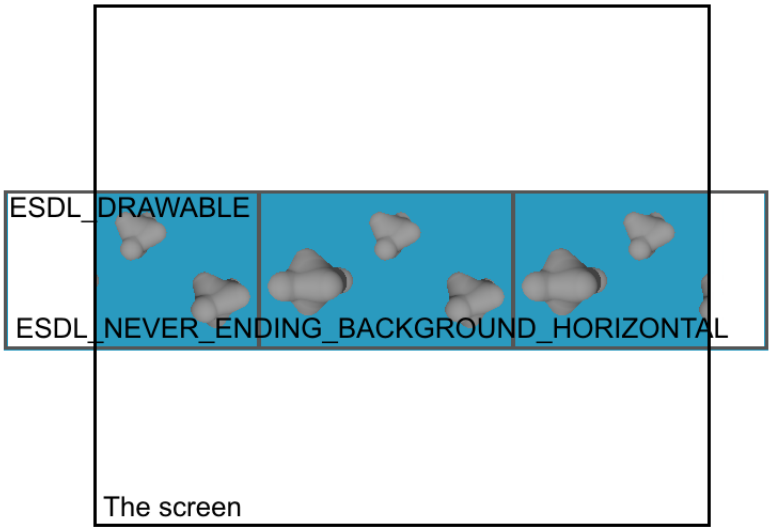

Figure 2: A never ending background

horizontal or vertical direction, to fill the whole size of the screen. Internally ESDL_NEVER_ENDING_BACKGROUND uses a ESDL_TILE_PATTERN.

You can set the speed (in pixels per second) of the background with the command *set_speed*. The background is then moved in either horizontal or vertical direction. See figure 2.

2.6.1 ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL

An ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL is an ESDL_NEVER_ENDING_BACKGROUND constraint to horizontal movement. A positive *speed* value moves the background from right to left. A negative one in the other direction.

2.6.2 ESDL_NEVER_ENDING_BACKGROUND_VERTICAL

An ESDL_NEVER_ENDING_BACKGROUND_VERTICAL is an ESDL_NEVER_ENDING_BACKGROUND constraint to vertical movement. A positive *speed* value moves the background from below to top. A negative one in the other direction.

2.6.3 Example

```
local
   picture: ESDL_SURFACE
   background:
   ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL
do
   create picture.make_from_image ("lovely.gif")
   create background.make (picture)
   background.set_speed (10)
   background.draw (screen)
   screen.redraw
end
```

2.7 ESDL_DRAWABLE_CONTAINER

An ESDL_DRAWABLE_CONTAINER is an ESDL_DRAWABLE and a DS_LINKED_LIST of ESDL_DRAWABLEs.

If an ESDL_DRAWABLE_CONTAINER is drawn to an ESDL_SURFACE all its elements are drawn to this ESDL_SURFACE. First element of the DS_LINKED_LIST is drawn first, last is drawn last.

The ESDL_DRAWABLE_CONTAINER is a very powerful class since an ESDL_DRAWABLE_CONTAINER represents a new coordinate system for all its elements. That means, that the new coordinate origin is the top left pixel of the ESDL_DRAWABLE_CONTAINER. For example: You add an ESDL_SURFACE with position (10,10) to an ESDL_DRAWABLE_CONTAINER at position (20,20). If you then draw the ESDL_DRAWABLE_CONTAINER to the screen, the ESDL_SURFACE is drawn to position (30,30) at the screen. See figure 3 on the next page.

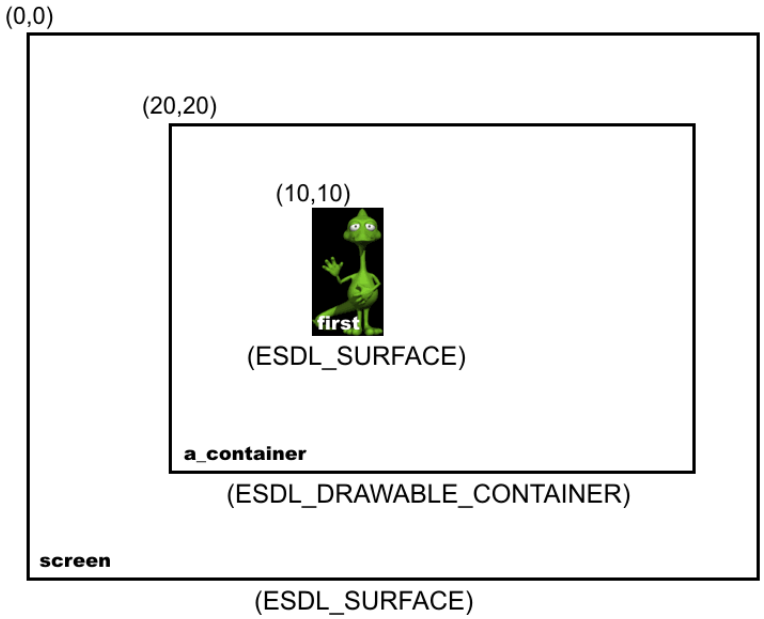

Figure 3: A container

The ability to create new coordinate systems allows to build tree structures of ESDL_DRAWABLEs. You can build ESDL_DRAWABLE_CONTAINERs and add these containers to other ESDL_DRAWABLE_CONTAINERs and so on (Composite Pattern, see [2] and [7]).

If you need to iterate over the elements of the container you have to create your own DS_LINKED_LIST_CURSOR. The reason for that is, if you compile your project in Eiffel Studio, not all your code seams to run in one thread, but in finalized programs all code does. Since we want to be able to debug our code, we have to make sure that the compiled code behaves the same way as the finalized code. Therefore every iteration has to be done with a separate cursor.

2.7.1 Example

```
local
    a_container:
    ESDL_DRAWABLE_CONTAINER [ESDL_SURFACE]
    alien:  ESDL_SURFACE
    cursor:  DS_LINKED_LIST_CURSOR [ESDL_SURFACE]
do
    create alien.make_from_image ("alien.gif")
```

```
        alien.set_x_y (10, 10)
        create a_container.make
        a_container.set_x_y (20, 20)
        a_container.extend (alien)
        a_container.draw (screen)
        screen.redraw
        create cursor.make (a_container)
        from
            cursor.start
        until
            cursor.off
        loop
            cursor.item.set_x (cursor.item.x + 10)
            cursor.forth
        end
    end
```

2.8 ESDL_STRING

An ESDL_STRING is an ESDL_DRAWABLE.

An ESDL_STRING expects a STRING and an ESDL_FONT at creation. ESDL_STRING uses an ESDL_DRAWABLE_CONTAINER containing a list of ESDL_CHARACTERs. The ESDL_FONT stored in *font* is used to draw every character of the STRING stored in *value*. ESDL_STRING is mutable, that means you can change the *value* with *set_value* (*a_value*: *STRING*). This is a slow operation. You can also change the used ESDL_FONT with *set_font* (*a_font*: *ESDL_FONT*).

2.8.1 ESDL_CHARACTER

An ESDL_CHARACTER is an ESDL_DRAWABLE.

An ESDL_CHARACTER expects a CHARACTER and an ESDL_FONT at creation. The ESDL_FONT stored in *font* is used to draw the CHARACTER stored in *character*.

2.8.2 ESDL_FONT

The deferred class ESDL_FONT should be implemented by every kind of font. An ESDL_FONT can draw a CHARACTER or part of it to an ESDL_SURFACE to a given position. An ESDL_FONT can also be asked for the *width* and the *height* of a CHARACTER.

```
deferred class
   ESDL_FONT

feature -- commands

   draw (a_character: CHARACTER;
         a_surface: ESDL_SURFACE;
         x: INTEGER; y: INTEGER) is
         -- Draws 'a_character' to 'a_surface'
         -- to position 'x' 'y'
      require
         a_surface /= void
         a_character /= void
      deferred
      end

   draw_part (rect: ESDL_RECT; a_character: CHARACTER;
         a_surface: ESDL_SURFACE;
         x: INTEGER; y: INTEGER) is
         -- Draws 'rect' part of 'a_character' to
         --'a_surface' to position 'x' 'y'
      require
         a_surface /= void
         a_character /= void
         rect /= void
      deferred
      end

feature -- queries

   width (a_character: CHARACTER): INTEGER is
         -- The width of 'a_character'
      deferred
      end

   height (a_character: CHARACTER): INTEGER is
         -- The height of 'a_character'
      deferred
      end

end
```

Figure 5: Bitmap font Arial_256

At the moment only one class is effecting ESDL_FONT: The ESDL_BITMAPFONT. In the future an ESDL_VECTORFONT can be implemented without changing any class. See figure 4.

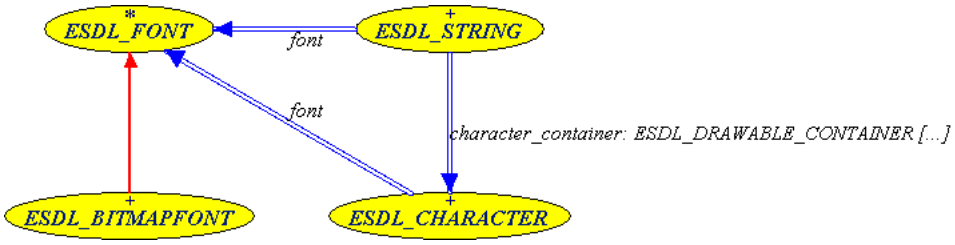

Figure 4: ESDL_FONT BON Diagram

2.8.3 ESDL_BITMAPFONT

ESDL_BITMAPFONT is an ESDL_FONT.

An ESDL_BITMAPFONT expects an ESDL_DRAWABLE that represents a matrix. Every row and every column contains 16 ASCII characters. That are 16*16=256 characters. The character with the ASCII number 0 is at the top left, the character with the ASCII number 255 is at the bottom right. The program Bitmap font Builder (www.lmnopc.com/bitmapfontbuilder/) can create such images (see figure 5)

The *height* of a CHARACTER is the *height* of *font* divided by 16, the *width* is the *width* of *font* divided by 16.

Since you can pass an ESDL_DRAWABLE as matrix you can, for example, build animated fonts. You should not pass an ESDL_NEVER_ENDING_BACKGROUND because of the resulting *width* or *height* of a CHARACTER.

2.8.4 Example

```
local
   str: ESDL_STRING
   font: ESDL_BITMAPFONT
   matrix: ESDL_SURFACE
do
   create matrix.make_from_image ("arial256.gif")
   create font.make (matrix)
   create str.make ("Hello world", font)
   str.draw (screen)
   screen.redraw
end
```

2.9 ESDL_FRAME_COUNTER

An ESDL_FRAME_COUNTER is an ESDL_DRAWABLE.

An ESDL_FRAME_COUNTER is a very useful utility. You can *draw* a frame counter every frame to the screen and get information about the performance of your program. The frame counter is used in many of the examples. It calculates the frame rate averaged over the last 10 frames. The frame rate is a measure for how many times per second the computer executes *draw* or *draw_part* feature of ESDL_FRAME_COUNTER. ESDL_FRAME_COUNTER uses an ESDL_BITMAPFONT to print the rate to *a_surface*. If you use an ESDL_FRAME_COUNTER you have to provide a copy of the file *arial256.gif* in the same directory where the ACE file of your program is. You find the file *arial256.gif* in most of the example directories (see figure 5 on the preceding page).

2.9.1 Example

```
local
   fc: ESDL_FRAME_COUNTER
do
   create fc.make
   fc.draw (screen)
   screen.redraw
end
```

2.10 ESDL_TIME

The class ESDL_TIME provides the ESDL programmer with functionalities related to time measurement. The two most important features are *delay* (*mil-*

liseconds: *INTEGER*) and *get_ticks*: *INTEGER*. A call to the command *delay* needs at least *milliseconds* time. This gives you the possibility to hold the execution of your program for a given amount of time. The SDL documentation states that:

> Note: Count on a delay granularity of at least 10 ms. Some platforms have shorter clock ticks but this is the most common.

The feature *get_ticks* returns the number of milliseconds since ESDL initialization. Do not count on getting precise results from *get_ticks* either. If you need very precise information you have to calculate an average somehow. I did that in the feature *update_cur_position* of the class ESDL_NEVER_ENDING_ BACKGROUND, to allow very smooth motion of a never ending background. Since many of the classes in ESDL require that the time subsystem is initialized, the time subsystem is initialized when you create an ESDL object. You can not instantiate the class ESDL_TIME afterwards, if you need its functionality inherit from ESDL_TIME_SINGLETON and call feature *get_time*: *ESDL_TIME*.

2.10.1 Example

```
inherit
    ESDL_TIME_SINGLETON

feature

    test is
        local
            i: INTEGER
        do
            i := get_time.get_ticks
            get_time.delay (1000)
            i := get_time.get_ticks - i
            print (i.out + " >= 1000 may be true")
        end
```

2.10.2 ESDL_TIME_SINGLETON

ESDL_TIME_SINGLETON ensures that only one instance of ESDL_TIME exists in the system. To get that instance inherit from ESDL_TIME_SINGLETON and call *get_time*: *ESDL_TIME*. Since Eiffel doesn't support explicit static called features the pattern described in [3] and implemented by ESDL_TIME_ SINGLETON is a possibility to ensure that only one instance of a class exists in the system, or at least, make it hard to build a second one. The reason why ESDL_TIME must be a singleton is its ability to store and call a list of callbacks.

2.10.3 Callbacks

An other feature of ESDL_TIME is the feature *add_timed_callback* (*interval*: *INTEGER*; *an_action*: *FUNCTION* [*ANY*, *TUPLE* [*INTEGER*], *INTEGER*])

The feature allows to pass a pointer to a function (agent) which has one parameter of type INTEGER and an INTEGER as result. This function is called after *interval* milliseconds. The value of the parameter is either the number of milliseconds since the function was called last time or interval if it is the first call. The resulting value can be less or equal to 0 if you don't want ESDL_TIME to call the function again or any number greater 0. If the number is greater 0 the function is called again after *result* number of milliseconds. The agent *an_action* should always be executed faster than *interval*. The agent *an_action* runs in a separate thread. You can use ESDL_MUTEX to synchronize between two threads.

2.10.4 Example

```
make is
   do
      get_time.add_timed_callback
      (100, agent action (?))
   end

action (interval: INTEGER): INTEGER is
      -- action is called every 100 milliseconds
   do
      Result := interval
   end
```

2.10.5 ESDL_MUTEX

You can use an ESDL_MUTEX to synchronize threads. Its two features are *lock* and *unlock*. The feature *lock* waits until the caller gets a lock on current. The feature *unlock* releases the lock.

2.11 ESDL_COLLIDABLE

Classes describing objects that can collide with other objects should be of type ESDL_COLLIDABLE. If an object collides with an other object *on_collide* (*other*: *ESDL_COLLIDABLE*) is called by the ESDL_COLLISION_DETECTOR. The *type_id* query returns a number that is the same for all instances of a class. This allows in *on_collide* to determine of what kind the other object is.

```
deferred class
   ESDL_COLLIDABLE

feature -- Status report

   does_collide (other: like Current): BOOLEAN is
         -- Does 'current' collide with 'other'?
      require
         other_not_void: other /= void
      deferred
      end

   type_id: INTEGER is
         -- The id given to all instances of 'current' generating
         -- class
      deferred
      end

feature {ESDL_COLLISION_DETECTOR} -- Implementation

   on_collide (other: ESDL_COLLIDABLE) is
         -- 'current' collided with 'other'
      require
         other_not_void: other /= void
      deferred
      end

end
```

2.11.1 ESDL_RECT_COLLIDABLE

ESDL_RECT_COLLIDABLE is an ESDL_COLLIDABLE. An ESDL_RECT rectangle is used to describe the size of an object. This rectangle is called *bounding_box*. If the intersection of the *bounding_box* of one ESDL_RECT_COLLIDABLE object and an other is not empty, the two objects collide. Finding out if such an intersection is empty or not, can be done very fast.

```
deferred class
    ESDL_RECT_COLLIDABLE

inherit
    ESDL_COLLIDABLE

feature -- Status

    does_collide (other: like Current): BOOLEAN is
            -- Does 'current' collide with 'other'?
        do
            Result :=
            Current.bounding_box.intersects
            (other.bounding_box)
        end

    bounding_box: ESDL_RECT is
            -- The active elements of Current
        deferred
        end

end
```

2.11.2 ESDL_COLLISION_DETECTOR

An ESDL_COLLISION_DETECTOR can be used to check if any of its containing ESDL_COLLIDABLE objects collide with any other. You can add an ESDL_COLLIDABLE to an ESDL_COLLISION_DETECTOR with the command *add* (*a_collidable*: *ESDL_COLLIDABLE*). Every time you call *check_for_collision* every object added to ESDL_COLLISION_DETECTOR is tested with every other object added to ESDL_COLLISION_DETECTOR for collision ($O(n^2)$). If a collision is detected either the *on_collide* feature of one of the two colliding objects is called if *call_both* is equals *false* or the *on_collide* features of both objects are called otherwise.

2.11.3 Example

See the collidable_string example on page 28.

3 Examples

3.1 Bitmap fonts

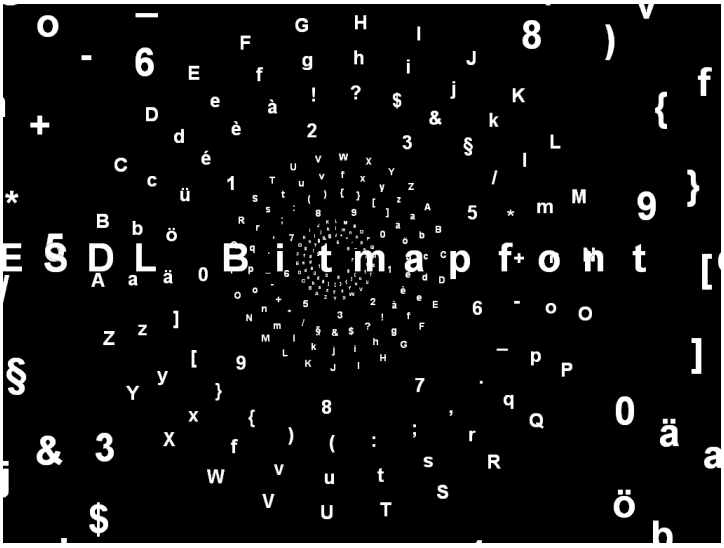

Figure 6: Bitmap font example

The bitmap font example demonstrates how to use the class ESDL_BITMAPFONT to print characters on the screen.

3.2 Drawable demo

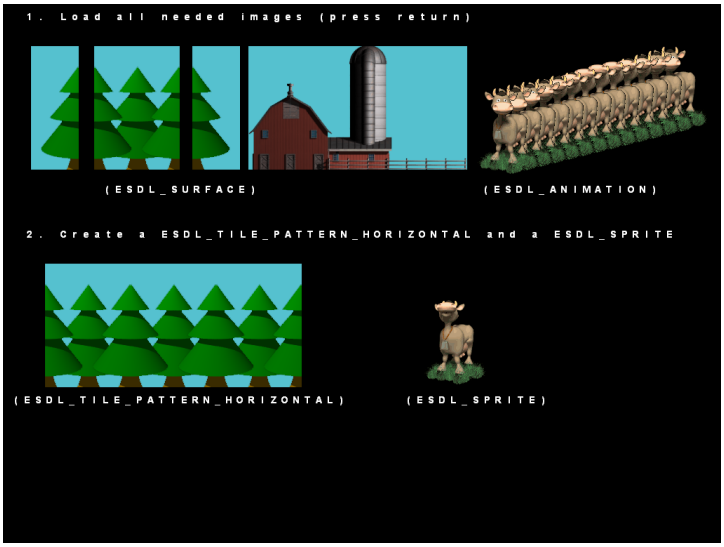

Figure 7: A drawable demo part 1

Figure 8: A drawable demo part 2

The drawable_demo example demonstrates how you can combine some of the ESDL_DRAWABLE classes to build more and more complex objects.

3.3 Never ending background

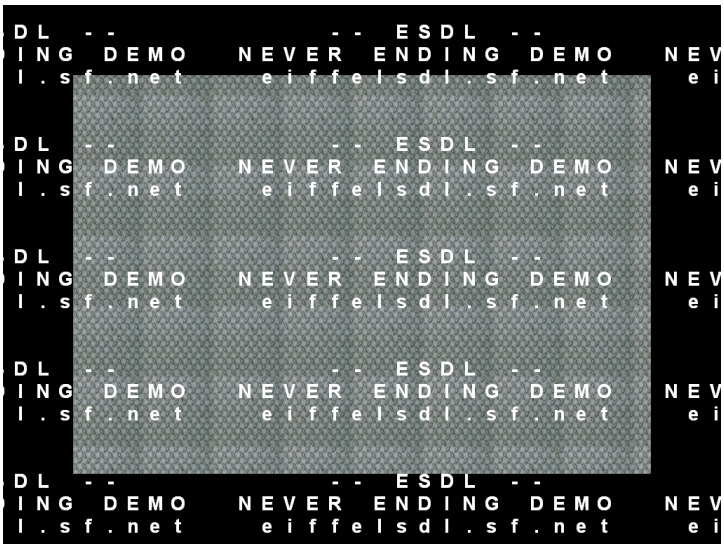

Figure 9: A demo of a never ending background

The never_ending_background example demonstrates how you can combine an ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL with an ESDL_NEVER_ENDING_BACKGROUND_VERTICAL to build a never ending background in any direction.

3.4 Collidable string

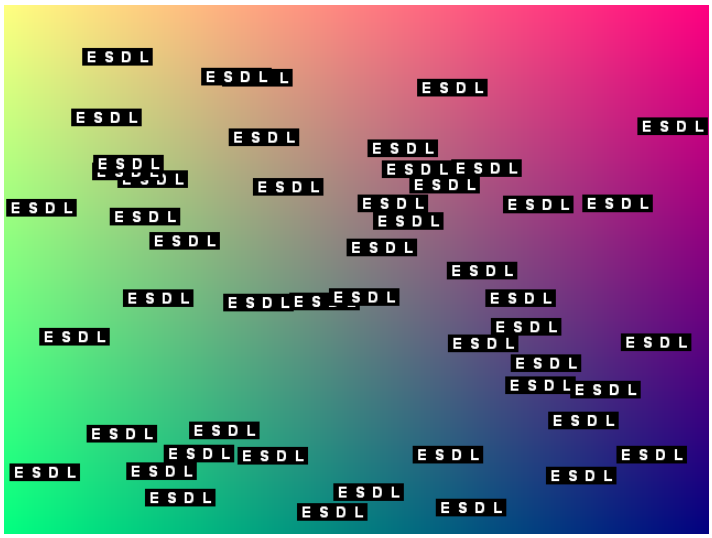

Figure 10: A demo of collidable strings

The collidable_string example demonstrates how you can use the ESDL_RECT_COLLIDABLE class in combination with the ESDL_COLLISION_DETECTOR class to detect collisions of objects.

3.5 ESDL Racing

Figure 11: ESDL Racing level 1

Figure 12: ESDL Racing level 2

Figure 13: ESDL Racing level 3

The esdl_racing example is the largest example. It is a fully playable racing game. All of the abilities of the classes described in this semester thesis report are used. The example also demonstrates how you can implement menus connecting multiple scenes. Every scene should inherit from the deferred class ESDL_SCENE. The class MENU and the class LEVEL both extend ESDL_SCENE. An ESDL_SCENE contains an *event_loop* and the very important feature *next_scene*: *ESDL_SCENE*. After current scene ends the *next_scene* is initialized by *initialize_scene* and then executed by calling *run* from the main loop if *next_scene* is not *void*:

```
-- Main loop
from
   scene := create {START}
until
   scene = void
loop
   scene.initialize_scene
   scene.run (screen)
   scene := scene.next_scene
end
```

START is the first scene, the main menu you see when you execute esdl_racing.

If you want to create your own level inherit from LEVEL. See one of the classes LEVEL01, LEVEL02 or LEVEL03 as an example. To make sure your

level is executed after LEVEL03 set the *next_scene* of LEVEL03 accordingly in the feature *on_win*. To make testing easier there exist two cheat codes: press c to turn off collision detection, press e to jump directly to the end of a level.

4 Future Work

- Implement scaling, rotation support and panning for surfaces
  Using a filtering approach (A deferred FILTER class, together with classes effecting it - e.g. SCALE_FILTER that takes a surface and returns one).
- Extend event handling support for mouse events
- Extend font support
  ESDL_VECTORFONT - eventually with the metafont project that would have to be wrapped separately.
- Implement audio subsystem
  A format independent approach like the one taken for image formats is desirable.
- Implement EiffelVision2 widgets (Linux and Windows)
- Extend video subsystem API with primitives (circles, lines, rectangles etc.), anti aliasing, alpha channels, YUV

References

[1] Till G. Bay, 2003, Eiffel SDL multimedia library (ESDL) Version 0.0.1
http://n.ethz.ch/student/bayt/publications

[2] GOF patterns for GUI Design, 1998, James Noble, side 6
citeseer.nj.nec.com/noble98gof.html

[3] How to get a Singleton in Eiffel?, 2004, Karine Arnout, Éric Bezault
http://se.inf.ethz.ch/people/arnout/arnout_bezault_singleton.pdf

[4] Simple Directmedia Layer
http://www.libsdl.org

[5] Eiffel Wrapper Generator
http://ewg.sourceforge.net/

[6] SDL image
http://www.libsdl.org/projects/SDL_image/

[7] Design Patterns, 1995, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides
http://www.amazon.com/exec/obidos/ASIN/0201633612/javaworld