# Eiffel SDL multimedia library (ESDL)
# 3D API
# SEMESTER THESIS

Patrick Ruckstuhl

23rd March 2005

http://eiffelsdl.sf.net

**Student:** Patrick Ruckstuhl (`rupatric@student.ethz.ch`)

**Student-No:** 01-917-772

**Supervising Assistant:** Till G. Bay

**Supervising Professor:** Bertrand Meyer

# Contents

# 1 Introduction

ESDL is a wrapper for SDL, the Simple Directmedia Layer library [3]. 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 understandable and memorable API, that is as performing as SDL and runs on different platforms. At the moment Windows, Linux and Mac OSX are supported.

ESDL is being developed subsystem by subsystem. At the moment the library allows only the drawing of 2 dimensional objects on the screen. The goal of this semester thesis is to extend the library with a binding to openGL that allows to create 3 dimensional objects.

To show the extended functionality an example application will be created. This example application is a computer game that starts in a 2 dimensional view and then switches to a 3 dimensional perspective.

# 2 Implementation of the 3D API

## 2.1 Overview

The SDL library does support a binding to openGL and so openGL[2] was used for the 3D API in ESDL.

For the 3D part to work an openGL compatible graphic card and drivers need to be installed.

The openGL was wrapped using the eiffel wrapper generator [1] and a sed [4] script.

## 2.2 Basic setup

To use the 3D mode in esdl the procedure is almost identically to setting up a normal 2D screen. The only thing that is needed is to call `set_opengl( true )` on the `video_subsystem` before the `video_subsystem` is enabled. If some more specific details about the openGL screen wants to be set this can be done with several methods (`gl_set_*`). This methods have also to be called befor the subsystem is enabled.

See Hello 3D world (2.8.1) for a simple example.

## 2.3 2D mode

### 2.3.1 Overview

If the screen surface is in openGL mode normal blitting operations to screen don't work anymore. To still be able to use things like fonts the blitting operations were reimplemented in openGL mode.

First `gl_enter_2d` needs to be called on the screen surface. This sets up the projection and other settings that things can be easily drawn with 2D coordinates. After this, the normal blitting and drawing commands can be called. In this methods there is a distinction between normal and openGL case and the according implementation will be called.

### 2.3.2 Details

The blitting in openGL is implemented with textures. OpenGL only allows textures that have dimensions that are a power of two. The pixels for the textures also need to be in a certain format. To get compatible pixel data, a new surface is created with `make_gl` which sets the correct bitmask for the pixels, so that this is compatible with openGL. The surface will be created with power of two dimensions that are equal or greater than the size of the current surface. After this, the current surface will be blitted to the newly create surface. This blitting is only software blitting and hasn't anything to do with openGL so the normal blitting operations are used. From this newly created surface an openGL texture will be created and automatically downloaded into the graphics card memory. This whole procedure will only be called again if the surface changed (after blitting something on it or creating a new surface).

At the end a rectangle will be created on the correct position and the texture will be assigned to it.

### 2.3.3 Important performance informations

In some situations the new 2D mode can be a lot faster than the normal 2D mode. To create fast applications some things need to be taken into considerations. As on every change of a surface a new texture has to be created and loaded into the video memory changing a surface is expensive whereas blitting an unchanged surface to a position on the screen is very cheap. The texture is already on the graphics hardware and only very little information about the position have to be sent to the graphics card. With this in mind it is better to minimise the number of changes to a surface. For example if there is an animation. It makes sense to create a surface for each picture of the animation instead of changing one surface.

## 2.4 Mipmap textures

### 2.4.1 Overview

Textures are used to make objects more realistic. Mipmaping is a technique that results in better looking results if an object with a texture is on different distance of the camera. To ease the use of textures a `gl_texture_mipmap` method was added to the surface. This method allows easy creation of such mipmaps. For example a surface can be created from an image and then from this a texture mipmap will be created.

### 2.4.2 Details

This is implemented very similar to the creation of normal textures (2.3.2) but some things are simpler because there are helper functions in the GL utility library that allow to create a mipmap texture out of pixel data (that does not need to have dimensions that are a power of two).

## 2.5 3D object

An ESDL_3D_OBJECT is the basic building block for 3D scenes. This objects can be plased on a position, resized and rotated around the three axis through their origin. They also have a width, height and depth property that specifies their surounding object bounding box. The bounding box will automatically be updated through the transformation operations.

### 2.5.1 3D object container

An ESDL_3D_OBJECT_CONTAINER is a container for several ESDL_3D_OBJECTs and is itself also such an object. The container can set up a new coordinate system and all the coordinates of its elements are relative to this coordinate system. At the moment the computation of the bounding box for the container out of the elements isn't supported and the bounding box has to be set manually.

### 2.5.2 3D vector

The ESDL_3D_VECT class implements some common used vector functionality.

## 2.6 3D object factory

### 2.6.1 Overview

Very often an object will not be used only once but multiple times and maybe even several different instances of it. To make this easier a object factory is provided. From which one can inherit to very easily create an own object factory. To do this inherit from ESDL_3D_OBJECT_FACTORY, specify the with, height and depth of the bounding box and implement the `specify_object` method in this method you need to draw your object (using normal openGL commands) with the lower, left, front point in (0/0/0). You then can get your objects with the create_object method.

### 2.6.2 Details

The object factory is implemented with openGL displaylists. This lists are the compiled informations of an object on the video card. The compile method creates such a list and then calls the `specify_object` method where the openGL commands for drawing the object are and this commands will be inserted into the displaylist by openGL. For the programmer a displaylist is represented with an integer value. With this integer value a new object of ESDL_3D_OBJECT_DISPLAYLIST will be called which implements the draw method as a simple call to the displaylist with the specified integer value.

### 2.6.3 Important performance informations

The loading of textures into video memory is a very performance intensive task and if done in the specify_object method will be executed on each redraw of the scene. It is important to load the textures into the video memory before the drawing. In the most cases this is done in the constructor and the textures are only assigned in the specify_object method. See the examples Objects 3D (2.8.2) and Racing 3D (2.8.3).

### 2.6.4 Wavefront obj loader

The ESDL_3D_OBJ_LOADER is an implementation of an ESDL_3D_OBJECT_FACTORY that allows to import geometric data directly from a file. The Wavefront obj format is a format that most 3D creation programms provide an export functionality to. The format is an ASCII file format where each line provides some informations. There are several different informations that can be in a line. There can be informations about position, normal and texture coordinates of a vertex and there are lines that describe which vertex informations make up a face. In addition there is also the possibility to describe material and texture informations about such faces. At the moment only a subset of the format is understand by the loader. Only position for vertices is supported and only faces of three vertices are supported. For the full description of the file format see [5]. A supported file looks like this

```
# comment
v 2 4 3
v 1 1 1
v 3 2 1
v 1 1 1
v 3 3 3
```

```
v 2 2 2
f 1 2 3
f 4 5 6
```

When the face data is loaded the face normals are computed by the loader (simple crossproduct). This allows so called flat shadding in openGL. The loader could be extended to also compute the vertex normals and with that also allow smoth shadding.

Additionally to loading the data and generating the face normals, the object bounding box for the object is computed to allow collision detection.

## 2.7 Collision detection

For most uses of 3D graphics also a system for collision detection is needed. As there already was a basic setup for collision handling in 2D this was extended to handle the 3D case. At the moment the implementation is based on bounding spheres, which is good enough for most cases. In addition there exists also an implementation based on infinite axis aligned planes to handle collision with a skybox. For an example see Racing 3D (2.8.3).

### 2.7.1 Sphere collision

For an object to be able to be checked against sphere collisions it has to inherit from ESDL_3D_SPHERE_COLLIDABLE and implement the deferred features. These are the bounding_sphere_center, the bounding_sphere_radius, on_collide (which gets executed if the object collides) and type_id.

### 2.7.2 Skybox collision

If a skybox should be collidable it has to inherit from ESDL_3D_SKYBOX_COLLIDABLE and implement the deferred features. These are mainly the start and end coordinates of the skybox.

### 2.7.3 Further possibilities

For an even better handling of collisions an algorithm based on bounding boxes could be used. The needed information for the objects is already in the 3D_OBJECT. A possible algorithm could be found at [6].

## 2.8 Examples

### 2.8.1 Hello 3D world

This is a simple example that shows the basic setup of a 3D scene. A spinning pyramid is shown.

### 2.8.2 Objects 3D

This shows the use of ESDL_3D objects and of ESDL_3D_OBJECT_FACTORY. Three pyramids are shown. Two of the pyramids have a texture on their bottom.

### 2.8.3 Racing 3D

This is a complex example that shows all the different parts that were done.

First comes the menu that takes use of the 2D mode (2.3). After this comes the scene which consists of a skybox for the background graphics, several buildings and a car that can be driven around. After some seconds the scene morphes from the 2d perspective to a 3d perspective. This can also be triggered by pressing the space key. Around the whole scene is a skybox which is collidable with the car. Also the buildings are collidable. Over the car is an arrow which points to a target that should be reached. The target makes use of an openGL technique called environment map to show the effect of a reflection. When the target is reached, the game has ended and the time to the target is displayed.

# A References

## References

[1] Eiffel wrapper generator. URL `http://ewg.sf.net`.

[2] *openGL Documentation*. URL `http://www.opengl.org`.

[3] *Simple Directmedia Layer Documentation*. URL `http://www.libsdl.org`.

[4] Stream editor. URL `http://www.student.northpark.edu/pemente/sed/`.

[5] Wavefront obj. URL `http://netghost.narod.ru/gff/graphics/summary/waveobj.htm`.

[6] D. Manocha S. Gottschalk, M.C. Lin. URL `ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/sig96.pdf`.