RebelSDL 1.0

"I'm Just a Rebel"

Andreas Falkenhahn

Table of Contents

1 General information

1.1 Introduction

RebelSDL is a plugin for Hollywood that allows you to use SDL (Simple DirectMedia Layer) from Hollywood. This makes it possible to write scripts that utilize the host system's graphics hardware to create high-performance, butter-smooth 2D animation that is produced completely in hardware by the GPU of your graphics board. This leads to a huge performance boost over the classic Hollywood graphics API which is mostly implemented in software for maximum portability and compatibility. Especially systems with slower CPUs (like the Raspberry Pi) will benefit greatly from hardware-accelerated drawing, scaling, and transformation offered by SDL.

SDL is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. It is used by video playback software, emulators, and popular games. More information about SDL can be obtained from http://www.libsdl.org. You can find good tutorials about learning SDL all over the web.

RebelSDL transparently replaces Hollywood's inbuilt display handler with its own display handler managed by SDL. Whenever RebelSDL is activated, Hollywood displays are automatically mapped to SDL windows and hardware brushes are mapped directly to SDL textures so that they can be drawn, scaled, and transformed in an extremely quick way on all supported systems. This is especially useful on Windows, Mac OS X, and Linux because Hollywood doesn't support hardware double buffers and hardware brushes on these platforms by default. With RebelSDL, however, hardware double buffers and hardware brushes can be used on these platforms now too. So RebelSDL can also act as a helper plugin here which adds this functionality to Hollywood without having you write a single line of SDL code to utilize it!

On top of that, RebelSDL offers wrapper functions for some useful commands of the SDL API, for example RebelSDL allows you to call SDL's joystick and game controller functions which are much more flexible than Hollywood's inbuilt joystick library. RebelSDL also allows you to access Hollywood hardware brushes as SDL textures and modify their properties via some dedicated SDL calls exposed by RebelSDL.

Finally, RebelSDL also replaces Hollywood's inbuilt audio driver with the audio driver offered by SDL. In contrast to the graphics driver SDL's audio driver probably doesn't have any advantage over Hollywood's inbuilt audio driver but by using it RebelSDL will make your program into a complete SDL application which doesn't only use SDL for graphics output but also for audio output.

RebelSDL utilizes the new display adapter plugin interface introduced with Hollywood 6.0. Thus, the plugin will not work with any older versions of Hollywood. It requires at least Hollywood 6.0. Whenever RebelSDL is activated, all graphics and audio output will automatically be routed through SDL. To benefit from hardware-accelered drawing, however, Hollywood scripts have to follow some rules as described in this manual.

RebelSDL comes with extensive documentation in various formats like PDF, HTML, AmigaGuide, and CHM that contains information about how to use this plugin. On top of

that, many example scripts are included in the distribution archive to get you started really quickly.

All of this makes RebelSDL the ultimate scripting experience for all you SDL rebels by combining the best of both worlds into one powerful plugin: Hollywood's extensive and convenient multimedia function set and SDL's raw graphics power!

1.2 Terms and conditions

1.3 Requirements

- Hollywood 6.0 or better
- Windows: requires at least Windows 2000
- Mac OS X: requires at least 10.5 on PowerPC or 10.6 on Intel Macs
- Raspberry Pi: requires at least a Raspberry Pi 2 and Raspbian Jessie
- MorphOS: requires at least MorphOS 3.8

1.4 Installation

Installing RebelSDL is straightforward and simple: Just copy the file `rebelsdl.hwp` for the platform to Hollywood's plugins directory. On all systems except on AmigaOS and compatibles, plugins must be stored in a directory named `Plugins` that is in the same directory as the main Hollywood program. On AmigaOS and compatible systems, plugins must be installed to `LIBS:Hollywood` instead. On Mac OS X, the `Plugins` directory must be inside the `Resources` directory of the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle itself, namely in `Hollywood.app/Contents/Resources`.

Afterwards merge the contents of the `Examples` folder with the `Examples` folder that is part of your Hollywood installation. All RebelSDL examples will then appear in Hollywood's GUI and you can launch and view them conveniently from the Hollywood GUI or IDE.

On Windows you should also copy the file `RebelSDL.chm` to the `Docs` directory of your Hollywood installation. Then you will be able to get online help by pressing F1 when the cursor is over a RebelSDL function in the Hollywood IDE.

On Linux and Mac OS copy the `RebelSDL` directory that is inside the `Docs` directory of the RebelSDL distribution archive to the `Docs` directory of your Hollywood installation. Note that on Mac OS the `Docs` directory is within the `Hollywood.app` application bundle, i.e. in `Hollywood.app/Contents/Resources/Docs`.

2 About RebelSDL

2.1 Credits

RebelSDL was written by Andreas Falkenhahn. Work on this project was started in Summer 2014 as a proof-of-concept demonstration of Hollywood 6.0's powerful new display adapter and audio adapter API which allows plugins to take over Hollywood's entire display and audio handler and replace it with a custom driver. Writing RebelSDL also helped to conceptualize Hollywood's plugin interfaces to make them as flexible as possible for new backends. That is why RebelSDL is also a test case for most of the features offered by the Hollywood SDK. RebelSDL has many features that can be optionally enabled to test certain functionalities of the Hollywood SDK. See Section 3.1 [Activating RebelSDL], page 7, for details. Later RebelSDL was expanded to allow access to other SDL functionality like joystick and game controller support. Finally, some wrapper functions were added to allow scripts to access Hollywood hardware brushes as SDL textures.

If you need to contact me, you can either send an e-mail to `andreas@airsoftsoftwair.de` or use the contact form on `http://www.hollywood-mal.com`.

2.2 Frequently asked questions

This section covers some frequently asked questions. Please read them first before asking on the mailing list or forum because your problem might have been covered here.

Q: Why is RebelSDL so slow on my Raspberry Pi?

A: First make sure that you're using a 2017 or better version of Raspbian Jessie or Stretch. Then make sure to enable the experimental vc4 KMS/DRM OpenGL driver for X11 in `raspi-config`. See Section 3.10 [Raspberry Pi peculiarities], page 13, for details. Note that if you're still on a Raspberry Pi 1 there is nothing you can do to get hardware acceleration from RebelSDL. The Raspberry Pi 1 is currently unsupported by RebelSDL. You need at least a Raspberry Pi 2.

Q: By default, RebelSDL uses Direct3D on Windows. How can I force it to use OpenGL instead?

A: Just set the `RenderDriver` tag to `opengl` when `@REQUIRE'ing` RebelSDL. See Section 3.1 [Activating RebelSDL], page 7, for details.

Q: Why doesn't RebelSDL report non-ASCII keys through OnKeyDown and OnKeyUp?

A: That's a limitation of SDL. It only supports ASCII and control keys through the `OnKeyDown` and `OnKeyUp` event handlers. If you have Hollywood 7.0 or better, you can just listen to the `VanillaKey` event handler to get the real keyboard events with full Unicode support.

Q: Is there a Hollywood forum where I can get in touch with other users?

A: Yes, please check out the "Community" section of the official Hollywood Portal online at `http://www.hollywood-mal.com`.

Q: How do I quit scripts that run in fullscreen mode?

A: Just press CTRL+C. This will always work except when CTRL+C has been explicitly disabled using Hollywood's `CtrlCQuit()` function.

Q: Where can I ask for help?

A: There's a lively forum at `http://forums.hollywood-mal.com` and we also have a mailing list which you can access at `airsoft_hollywood@yahoogroups.com`. Visit `http://www.hollywood-mal.com` for information on how to join the mailing list.

Q: I have found a bug.

A: Please post about it in the dedicated sections of the forum or the mailing list.

2.3 Known issues

Here is a list of things that RebelSDL doesn't support yet or that may be confusing in some way:

- the MorphOS version is very unstable, e.g. fullscreen mode crashes, iconifying a window crashes, the mouse pointer is always hidden, etc.; this isn't RebelSDL's fault but MorphOS' SDL2 port which seems to be in alpha state; I've tried to get in touch with SDL2's MorphOS maintainer but didn't get any reply so you'll have to live with these issues for the time being
- menus are unsupported
- the mouse wheel is unsupported
- the keyboard listener that is mapped to Hollywood's `OnKeyDown` and `OnKeyUp` event handlers currently only supports ASCII and control keys; this is because of a limitation in SDL which doesn't support fine-tuned listening (i.e. key down, key repeat, key up) using international keyboards; people who are on Hollywood 7.0 or better can just use the `VanillaKey` event handler instead; this event handler will deliver real keyboard events including full Unicode support
- not all display styles are supported

2.4 Future

Here are some things that are on my to do list:

- add support for more SDL calls

Don't hesitate to contact me if RebelSDL lacks a certain feature that is important for your project.

2.5 History

Please see the file `history.txt` for a complete change log of RebelSDL.

3 Usage

3.1 Activating RebelSDL

All you have to do to make your script use SDL instead of Hollywood's inbuilt graphics driver is adding the following line to the top of your script:

```
@REQUIRE "rebelsdl"
```

Alternatively, if you are using Hollywood from a console, you can also start your script like this:

```
Hollywood test.hws -requireplugins rebelsdl
```

Once the RebelSDL plugin has been activated for your script, it will reroute all of Hollywood's graphics output through SDL. Note that this will usually be slower than Hollywood's inbuilt graphics driver for scripts that aren't optimized for RebelSDL. To get an optimal performance with SDL, your script needs to use a hardware-accelerated double buffer. See Section 3.2 [Using a hardware double buffer], page 8, for details.

RebelSDL accepts the following arguments in its `@REQUIRE` call:

`EnableVSync:`
: By default, RebelSDL's hardware double buffer is sync'ed with the monitor's vertical refresh. This means that `Flip()` will always block until the next vertical refresh and then flip the buffers. This will generate perfectly smooth graphics but of course it also means that you can't draw faster than the monitor's vertical refresh, typically around 60 times per second. If you want to ignore the monitor's vertical refresh, set this tag to `False` and RebelSDL won't throttle double buffer flipping. It will then flip buffers as fast as the hardware allows. Defaults to `True`.

`ForceFullRefresh:`
: If this tag is set to `False`, RebelSDL will only refresh the parts of the display that have actually changed. This is quicker but it can lead to some refresh problems depending on the way your script draws its graphics. That is why this tag defaults to `True`, which means that RebelSDL will always refresh the full display whenever something is drawn. This is slower but guarantees that there will be visual artefacts because front and back buffers will always be completely in sync.

`RenderDriver:`
: This tag allows you to select a different render driver than the default one. This is mostly useful for testing purposes. For example, it is possible to force RebelSDL to use OpenGL on Windows instead of the default Direct3D driver with this tag. Possible values for this tag are `direct3d`, `opengl`, `opengles`, `opengles2`, `rpi`, and `software`. See Section 3.10 [Raspberry Pi peculiarities], page 13, for more information on the rpi driver.

`UseAudioAdapter:`
: By default, RebelSDL will replace Hollywood's inbuilt audio driver with a custom audio driver that uses SDL to play audio. If you don't want that, set this

tag to `False`. Then Hollywood's inbuilt audio driver will be used even when RebelSDL is active. Normally, however, it's not necessary to set this tag unless you experience problems with RebelSDL's audio driver. Defaults to `True`.

`UseBitmapAdapter:`
: If this is set to `True`, RebelSDL will override Hollywood's inbuilt handler for software bitmaps. This doesn't have any practical advantages and was only implemented to test the corresponding Hollywood SDK functionality. Defaults to `False`.

`UseDesktopFullScreen:`
: SDL offers a special display mode that automatically scales windows opened by SDL to the dimensions of the desktop. The window will then occupy all screen space without changing the monitor's resolution. You can activate this mode by setting this tag to `True`. Setting this tag to `True` will also automatically activate autoscaling for your display. Note that a similar effect can be achieved by using Hollywood's `FullScreenScale` display mode but it's preferrable to use `UseDesktopFullScreen` because it is directly tied to SDL. Defaults to `False`.

`UseDoubleBufferAdapter:`
: If this is set to `False`, RebelSDL won't support hardware double buffers. Since hardware double buffers are one of the most important features of RebelSDL, there's probably no case where you'd want to disable this feature. It's mostly here for debugging purposes. Defaults to `True`.

`UseSoftwareRenderer:`
: By default, SDL will try to use the GPU to draw graphics whenever and wherever possible. If you don't want this, you can set this tag to `True` to put SDL into pure software drawing mode. This is probably only of use for testing and debugging purposes because normally you'd want to use the hardware renderer for the best performance. Defaults to `False`.

`UseVideoBitmapAdapter:`
: If this is set to `False`, RebelSDL won't support hardware brushes. Since hardware brushes are one of the most important features of RebelSDL, there's probably no case where you'd want to disable this feature. It's mostly here for debugging purposes. Defaults to `True`.

Here is an example of how to pass arguments to the `@REQUIRE` preprocessor command:

```
@REQUIRE "rebelsdl", {UseDesktopFullScreen = True}
```

Alternatively, you can also use the `-requiretags` console argument to pass these arguments. See the Hollywood manual for more information.

3.2 Using a hardware double buffer

If you want your script to benefit from RebelSDL's hardware-accelerated drawing functions, you need to use a hardware double buffer and do all your drawing within that double buffer. Using a hardware double buffer will also ensure that graphics output is synchronized with your monitor's refresh rate to prevent any flickering. To get an optimal performance with RebelSDL, your main loop should always look like this:

```
@REQUIRE "rebelsdl"
```

```
BeginDoubleBuffer(True)  ; set up a hardware double buffer

Repeat
    ....           ; draw the next frame here
    Flip()         ; wait for vertical refresh, then flip buffers
    CheckEvent()   ; run event callbacks
Forever
```

The call to `CheckEvent()` is only necessary if your script needs to listen to event handlers that have been installed using `InstallEventHandler()`. Note that you should not draw the next frame in an interval callback that runs at a constant frame rate (say 50fps) because such a setup won't guarantee that drawing is synchronized with the vertical refresh as different monitors use different refresh rates so you might get flickery graphics. If you do your drawing like above, you can be sure that front and back buffers will be flipped in perfect synchronization with the monitor's vertical refresh.

Additionally, you need to take care of how you actually draw your graphics because most of Hollywood's drawing commands operate entirely in software mode and thus do not benefit from hardware acceleration. See Section 3.3 [Drawing graphics], page 9, for details.

When drawing brushes in a hardware double buffer, make sure that you use only hardware brushes because only those can be drawn directly using hardware acceleration. Drawing normal brushes to hardware double buffers is possible too, but it will be very slow. See Section 3.4 [Hardware brushes], page 10, for details.

Important: SDL is designed to be used with double buffers. Thus, you can only benefit from hardware acceleration when drawing within a double buffer. Drawing outside a double buffer is possible but it will be much slower.

3.3 Drawing graphics

For an optimal performance you need to be very careful concerning the way you draw your graphics. Most of Hollywood's drawing commands are implemented in software only, i.e. they draw using the CPU instead of the GPU. This can become quite a bottleneck especially on slower CPUs. Thus, you should know which drawing functions are hardware-accelerated and which aren't and then write your scripts accordingly.

The following Hollywood commands are hardware-accelerated when RebelSDL is active and they are used within a hardware double buffer:

```
Box()
Cls()
Line()
Plot()
DisplayBrush()
```

`DisplayBrush()` will only use hardware acceleration when called with a hardware brush. See Section 3.4 [Using hardware brushes], page 10, for details. When used with a software brush, i.e. a brush that doesn't reside in video memory, `DisplayBrush()` will draw the brush using the CPU which is much slower.

`Box()`, `Line()`, and `WritePixel()` will only use hardware acceleration in case the fill style is either `#FILLNONE` or `#FILLCOLOR` and no other form styles like `#EDGE` or `#SHADOW` are

active. As soon as you want to draw with other fill or form styles, these commands will fall back to their software counterparts and thus will be very slow. You can work around this problem by simply drawing the graphics into a hardware brush first and then just drawing this hardware brush. This is a good strategy because then Hollywood's software renderer will be used only once, i.e. when drawing the graphics into a hardware brush, and after that you'll profit from hardware acceleration all the time because hardware brushes can be drawn really quickly.

Finally, don't forget that you should do all your drawing inside a hardware double buffer loop. See Section 3.2 [Using a hardware double buffer], page 8, for details.

3.4 Using hardware brushes

RebelSDL supports the creation of hardware brushes. Hardware brushes reside in GPU memory and thus can be drawn in no time. On most graphics boards, they can also be scaled and transformed by the GPU in an extremely efficient way. To make Hollywood create a hardware brush, all you have to do is set the optional `Hardware` tag to `True`. This tag is supported by most of the Hollywood commands which create brushes.

Here is an example:

```
@REQUIRE "rebelsdl"  ; make sure this line is first
@BRUSH 1, "sprites.png", {Hardware = True}
```

In the code above, RebelSDL will create brush 1 in video memory. It can then be drawn using the GPU at almost no cost. Keep in mind, though, that hardware brushes can only be drawn to hardware double buffers. See Section 3.2 [Using a hardware double buffer], page 8, for details.

To transform a hardware brush, you can use the `ScaleBrush()`, `RotateBrush()`, and `TransformBrush()` commands. Transformations of hardware brushes are usually also GPU-accelerated and thus many times faster than transformations done by the CPU.

Note that hardware brushes can only be drawn to the display that was specified when allocating them. Thus, if your script uses multiple displays, you need to tell Hollywood the identifier of the display you want to use this hardware brush with. This can be done by specifying the "Display" tag along the "Hardware" tag. Here is an example:

```
@REQUIRE "rebelsdl"  ; make sure this line is first
@DISPLAY 1, {Title = "First display"}
@DISPLAY 2, {Title = "Second display"}
@BRUSH 1, "sprites.png", {Hardware = True, Display = 1}
@BRUSH 2, "sprites.png", {Hardware = True, Display = 2}
```

The code above will allocate brush 1 in a way that it can be drawn to display 1 and it will allocate brush 2 in a way that it can be drawn to display 2. It won't be possible, however, to draw brush 2 to display 1 or brush 1 to display 2! RebelSDL hardware brushes are always display-dependent and can only be drawn to the display they were allocated for.

Please see the Hollywood manual for more information on hardware brushes and hardware double buffers.

You can also use RebelSDL as a helper plugin to add hardware brush support to Hollywood on Windows, Mac OS X, and Linux. By default, Hollywood doesn't support hardware

brushes on these systems but RebelSDL can add this feature to Hollywood. See Section 3.9 [RebelSDL as a helper plugin], page 12, for details.

The `SmoothScroll.hws` example script that comes with RebelSDL demonstrates how to use hardware brushes and a hardware double buffer to achieve butter-smooth scrolling that is fully synchronized with the monitor's vertical refresh.

3.5 Offscreen rendering

If you pass `True` to the `sdl.EnableOffscreenRender()` command, all hardware brushes that are created after the call to `sdl.EnableOffscreenRender()` can be drawn to using Hollywood's `SelectBrush()` command. These drawing operations can also be hardware-accelerated which gives them an advantage over Hollywood's default offscreen drawing routines which can only draw to software brushes.

There is, however, a major limitation that you have to keep in mind: Hardware brushes which can be drawn to using `SelectBrush()` will lose their contents on Windows whenever the window size is changed or when the window's display mode changes, e.g. from full screen mode to window or the other way round. Thus, using hardware brushes which can be drawn to really only makes sense if you update them every frame or if you are able to re-initialize them whenever the window's size or display mode changes. If your script isn't able to handle that, you will be in trouble. Because of this limitation, it is also highly advised to call `sdl.EnableOffscreenRender()` with its argument set to `False` immediately after you have created the hardware brushes that you want to use for offscreen drawing. Otherwise, all hardware brushes created subsequently will also be prepared for offscreen drawing and will thus suffer from the limitations described above.

Note that in order to utilize hardware acceleration on offscreen drawing you need to follow the same rules as when drawing to a hardware double buffer. Only a few graphics operations are hardware-accelerated. See Section 3.3 [Drawing graphics], page 9, for details.

It is also important to note that the different combo modes supported by `SelectBrush()` won't work with RebelSDL. Instead, graphics are always drawn to the color and the alpha channel of the hardware brush, no matter which mode you specify. Also, you cannot use `SelectMask()` or `SelectAlphaChannel()` with hardware brushes. It's only possible to use `SelectBrush()` to draw into color and alpha channels at the same time.

3.6 Using the SDL renderer

For advanced users RebelSDL allows you to call SDL renderer functions directly for the ultimate flexibility. Take note, though, that if you do this then you'll be operating at a very low level and some Hollywood features like the autoscaling engine won't work any more because all your drawing calls are routed directly through SDL without Hollywood having a chance to intervene. This has very little overhead but comes at the expense of certain Hollywood features not working any longer, like the autoscaling engine.

If you do make SDL renderer calls directly, you should also not mix them with Hollywood drawing calls. You should either draw entirely with Hollywood functions or with SDL renderer functions. Mixing both is possible but might yield some unexpected results because Hollywood drawing functions will of course change state information of the SDL renderer themselves, i.e. if you call `sdl.SetRenderDrawColor()` to set the drawing color to red

and then use Hollywood's `Box()` command to draw a blue box, then the renderer color will suddenly be blue because the call to `Box()` has changed the renderer color to blue. These are side effects you have to be able to deal with when mixing SDL renderer calls and Hollywood drawing calls.

3.7 Joysticks and game controllers

RebelSDL allows you to access SDL's comprehensive joystick and game controller functionalities. For the best compatibility, it is recommended to place a file named `gamecontrollerdb.txt` in the same directory as your script. This file has to contain calibration information for all the different joysticks and game controllers out there. You can find a community-maintained version of this file here: https://github.com/gabomdq/SDL_GameControllerDB

3.8 Increasing execution speed

To increase the raw execution speed of your script, you can disable Hollywood's line hook using the `DisableLineHook()` and `EnableLineHook()` commands. This will improve your script's execution speed significantly in case lots of Hollywood code needs to be run to draw the next frame. Keep in mind, though, that you have to enable the line hook for every frame you draw or your window will become unresponsive. Here's what a speed-optimized implemention of the main loop could look like:

```
@REQUIRE "rebelsdl"

BeginDoubleBuffer(True)  ; set up a hardware double buffer

Repeat
    DisableLineHook() ; disable line hook while drawing the next frame
    p_DrawFrame()     ; draw the next frame here
    EnableLineHook()  ; enable line hook again
    Flip()            ; wait for vertical refresh, then flip buffers
    CheckEvent()      ; run event callbacks
Forever
```

Note that you'll only notice a speed difference here if `p_DrawFrame()` executes many lines of Hollywood code. If `p_DrawFrame()` only consists of 20 lines of code, you won't notice any difference. It's only noticeable with hundreds of code lines or long loops.

See the documentation of `DisableLineHook()` and `EnableLineHook()` in the Hollywood manual for more information.

3.9 RebelSDL as a helper plugin

RebelSDL can also be used as a helper plugin to work around the problem that Hollywood only supports hardware-accelerated double buffers and brushes on AmigaOS and compatibles. They aren't supported on Windows, Mac OS X, or Linux. If you install and `@REQUIRE` RebelSDL, however, hardware double buffer and hardware brush support will also be available on Windows, Mac OS X, and Linux because RebelSDL supports this.

Thus, you can also use RebelSDL as a helper plugin just to get hardware-accelerated double buffer support on Windows, Mac OS X, and Linux. You don't even have to use any of the SDL commands directly. You can just `@REQUIRE` RebelSDL, set up a hardware double buffer and then draw to it using hardware brushes. This allows you to utilize hardware acceleration without having to write a single line of SDL code!

On AmigaOS and compatibles this isn't necessary since Hollywood already supports hardware accelerated double buffers and brushes by default. Still, using RebelSDL on AmigaOS as a hardware double buffer driver can be of benefit in full screen mode because RebelSDL uses drawing which is perfectly synchronized with the monitor's vertical refresh so it usually looks better than double buffers managed by Hollywood directly.

See Section 3.2 [Using a hardware double buffer], page 8, for details.

See Section 3.4 [Using hardware brushes], page 10, for details.

The `SmoothScroll.hws` example script that comes with RebelSDL demonstrates how to use hardware brushes and a hardware double buffer to achieve butter-smooth scrolling that is fully synchronized with the monitor's vertical refresh.

3.10 Raspberry Pi peculiarities

RebelSDL can be very useful on the Raspberry Pi because of its comparatively poor CPU which makes Hollywood's inbuilt CPU renderer very slow. Using RebelSDL will give you hardware-accelerated drawing and scaling which can boost your script's performance dramatically. However, there are some things that you have to keep in mind. First of all, there are two entirely different drivers available for the Raspberry Pi:

1. VideoCore 4 OpenGL driver (KMS/DRM): Since 2017 Raspbian Jessie ships with an experimental vc4 KMS/DRM OpenGL driver. To enable this driver, you need to run `sudo raspi-config` go to the `Advanced Options` settings and then select `GL (Full KMS)` in the `GL Driver` menu. Once you have rebooted your system, X11 will then use this new vc4 driver and RebelSDL will be able to use it too. Note that you have to make sure that you're using a recent version of Raspbian Jessie (2017 or newer) or Raspbian Stretch. Older versions of Jessie don't have this driver yet. Also note that as of September 2017, activating the new vc4 driver breaks HDMI audio output on Jessie (not on Stretch). So as of September 2017, you'll only be able to get audio through the headphone output on Jessie when the vc4 driver is enabled.
2. Raspberry Pi native driver: This is an alternative driver which directly accesses the graphics hardware of the Pi bypassing OpenGL and X11 completely. This means that you can run your scripts from outside X11 as well. This driver will also work with older versions of Raspbian Jessie but only if the experimental OpenGL driver (see above) is disabled. To activate the native Raspberry Pi renderer, you need to pass `rpi` to the `RenderDriver` tag when `@REQUIRE'ing` RebelSDL, like so:

   ```
   @REQUIRE "rebelsdl", {RenderDriver = "rpi"}
   ```

   RebelSDL will then use the native Raspberry Pi renderer. Note that since the native Pi renderer operates outside X11 it will always take over the whole screen. It is not possible to run the native Pi renderer in windowed mode. It will always fill the entire screen. All Hollywood features that require a window won't work with the native Pi renderer either.

By default, RebelSDL will use the vc4 OpenGL driver and if it is not there, it will fall back to software OpenGL mode, which is of course very slow. The native Pi renderer will only be activated if you explicitly request it like shown above.

4 Examples

4.1 Examples

RebelSDL comes with a number of examples that demonstrates how to use the plugin and should allow you to get started really quickly. Here's a list of examples that are distributed with RebelSDL:

Aladdin A RebelSDL port of the Prodigy cracktro for Aladdin.

BeastScroll
: A remake of the famous Shadow of the Beast scroller in RebelSDL.

CannonFodder
: A RebelSDL port of the Cannon Fodder cracktro by Crystal.

Creatures A RebelSDL port of the Creatures cracktro by Crystal.

Dynablaster
: A RebelSDL port of the Dynablaster cracktro by Vision Factory.

GPUScale Demonstrates how to use GPU-accelerated scaling and rotation with RebelSDL.

Lemmings A RebelSDL port of the Lemmings cracktro by Skid Row.

Moonstone
: A RebelSDL port of the Moonstone cracktro by Crystal.

MultiDisplays
: Demonstrates how to use multiple displays with RebelSDL.

Pang A RebelSDL port of the Pang cracktro by Horizon.

PPHammer
: A RebelSDL port of the PP Hammer cracktro by Crystal.

SmoothScroll
: Demonstrates how to achieve perfectly smooth scrolling with hardware brushes.

SteelEmpire
: A RebelSDL port of the Steel Empire cracktro by Crystal.

Superfrog A RebelSDL port of the Superfrog cracktro by Crystal.

SuperStardust
: A RebelSDL port of the Super Stardust cracktro by Prestige.

Turrican2 A RebelSDL port of the Turrican 2 cracktro by Defjam.

Turrican3 A RebelSDL port of the Turrican 3 cracktro by Hoodlum.

Zool A RebelSDL port of the Zool cracktro by Crystal.

5 Joystick reference

5.1 sdl.ForceJoystickMode

NAME

sdl.ForceJoystickMode – force game controller into joystick mode

SYNOPSIS

```
sdl.ForceJoystickMode(port)
```

FUNCTION

Use this function to force the game controller at the specified port into joystick mode. You will then be able to query the game controller's state as if it was a joystick.

INPUTS

`port`	game port to use

5.2 sdl.GetAxis

NAME

sdl.GetAxis – query state of specified axis

SYNOPSIS

```
state = sdl.GetAxis(port, axis)
```

FUNCTION

If the device at the specified port is a joystick, `axis` must be the number of the axis to query. 0 is typically used for the x-axis and 1 for the y-axis. Some joysticks use axes 2 or 3 for extra buttons. You can use `sdl.GetNumAxes()` to find out the number of axes.

If the device at the specified port is a game controller, `axis` must be one of the following predefined constants:

```
#SDL_AXIS_LEFTX
#SDL_AXIS_LEFTY
#SDL_AXIS_RIGHTX
#SDL_AXIS_RIGHTY
#SDL_AXIS_TRIGGERLEFT
#SDL_AXIS_TRIGGERRIGHT
```

The return value is a value ranging from -32768 to 32767. It may be necessary to impose certain tolerances on these values to account for jitter. Note that game controller triggers, however, range from 0 to 32767. They never return a negative value.

INPUTS

`port`	game port to query
`axis`	axis to query

RESULTS

`state`	state of the specified axis (typically -32768 to 32767)

5.3 sdl.GetBall

NAME

sdl.GetBall – query state of specified ball

SYNOPSIS

```
dx, dy = sdl.GetBall(port, ball)
```

FUNCTION

Use this function to get the ball axis change since the last poll. This is only possible for joysticks, not for game controllers.

You have to pass the ball index to query in `ball`. Ball indices start at index 0. `sdl.GetBall()` will return the difference in the x and y axis position since the last poll. Note that since trackballs can only return relative motion these return values are delta values.

INPUTS

`port`	game port to query
`ball`	ball index to query

RESULTS

`dx`	the difference in the x axis position since the last poll
`dy`	the difference in the y axis position since the last poll

5.4 sdl.GetButton

NAME

sdl.GetButton – query state of specified button

SYNOPSIS

```
state = sdl.GetButton(port, button)
```

FUNCTION

If the device at the specified port is a joystick, `button` must be the index of the desired button (starting from 0). The number of joystick buttons can be found out by calling `sdl.GetNumButtons()`.

If the device at the specified port is a game controller, `button` must be one of the following predefined constants:

```
#SDL_BUTTON_A
#SDL_BUTTON_B
#SDL_BUTTON_X
#SDL_BUTTON_Y
#SDL_BUTTON_BACK
#SDL_BUTTON_GUIDE
#SDL_BUTTON_START
#SDL_BUTTON_LEFTSTICK
#SDL_BUTTON_RIGHTSTICK
```

```
#SDL_BUTTON_LEFTSHOULDER
#SDL_BUTTON_RIGHTSHOULDER
#SDL_BUTTON_DPAD_UP
#SDL_BUTTON_DPAD_DOWN
#SDL_BUTTON_DPAD_LEFT
#SDL_BUTTON_DPAD_RIGHT
```

INPUTS

`port` game port to query

`button` button to query

RESULTS

`state` state of the specified button (1 for pressed state, 0 non-pressed state)

5.5 sdl.GetHat

NAME

sdl.GetHat – query state of specified hat

SYNOPSIS

```
state = sdl.GetHat(port, hat)
```

FUNCTION

Use this function to get the current state of a POV hat on a joystick. This is only possible for joysticks, not for game controllers. You have to pass the hat index to get the state from. Hat indices start at index 0.

The return value will be one of the following predefined constants:

```
#SDL_HAT_CENTERED
#SDL_HAT_UP
#SDL_HAT_RIGHT
#SDL_HAT_DOWN
#SDL_HAT_LEFT
#SDL_HAT_RIGHTUP
#SDL_HAT_RIGHTDOWN
#SDL_HAT_LEFTUP
#SDL_HAT_LEFTDOWN
```

INPUTS

`port` game port to query

`hat` hat index whose state to get

RESULTS

`state` state of specified hat

5.6 sdl.GetJoysticks

NAME
sdl.GetJoysticks – get number of available joysticks

SYNOPSIS
```
n = sdl.GetJoysticks()
```

FUNCTION
Use this function to count the number of joysticks attached to the system.

INPUTS
none

RESULTS

`n`	number of attached joysticks

5.7 sdl.GetNumAxes

NAME
sdl.GetNumAxes – get number of joystick axes

SYNOPSIS
```
num = sdl.GetNumAxes(port)
```

FUNCTION
Use this function to get the number of general axis controls on a joystick. This is only possible for joysticks, not for game controllers.

INPUTS

`port`	game port to query

RESULTS

`num`	number of joystick axes

5.8 sdl.GetNumBalls

NAME
sdl.GetNumBalls – get number of joystick balls

SYNOPSIS
```
num = sdl.GetNumBalls(port)
```

FUNCTION
Use this function to get the number of trackballs on a joystick. This is only possible for joysticks, not for game controllers.

INPUTS

`port`	game port to query

RESULTS

`num`	number of joystick trackballs

5.9 sdl.GetNumButtons

NAME
sdl.GetNumButtons – get number of joystick buttons

SYNOPSIS
```
num = sdl.GetNumButtons(port)
```

FUNCTION
Use this function to get the number of buttons on a joystick. This is only possible for joysticks, not for game controllers.

INPUTS

`port`	game port to query

RESULTS

`num`	number of joystick buttons

5.10 sdl.GetNumHats

NAME
sdl.GetNumHats – get number of joystick hats

SYNOPSIS
```
num = sdl.GetNumHats(port)
```

FUNCTION
Use this function to get the number of POV hats on a joystick. This is only possible for joysticks, not for game controllers.

INPUTS

`port`	game port to query

RESULTS

`num`	number of joystick hats

5.11 sdl.IsGameController

NAME
sdl.IsGameController – check if input device is a game controller

SYNOPSIS
```
res = sdl.IsGameController(port)
```

FUNCTION
Use this function to find out whether the input device at the specified port is a game controller or a joystick. The function will return `True` for game controllers and `False` for joysticks.

INPUTS

`port`	game port to query

RESULTS

`res`	boolean value

6 Keyboard reference

6.1 sdl.SetTextInputRect

NAME
sdl.SetTextInputRect – set the rectangle used to type Unicode text inputs

SYNOPSIS
`sdl.SetTextInputRect(x, y, width, height)`

FUNCTION
Use this function to set the rectangle used to type Unicode text inputs.

INPUTS

`x`	x position
`y`	y position
`width`	rectangle width
`height`	rectangle height

6.2 sdl.StartTextInput

NAME
sdl.StartTextInput – start accepting Unicode text events

SYNOPSIS
`sdl.StartTextInput()`

FUNCTION
This function will start accepting Unicode text input events in the focused RebelSDL window, and start emitting `VanillaKey`. Please use this function in pair with `sdl.StopTextInput()`.

On some platforms using this function activates the screen keyboard.

INPUTS
none

6.3 sdl.StopTextInput

NAME
sdl.StopTextInput – stop receiving Unicode text events

SYNOPSIS
`sdl.StopTextInput()`

FUNCTION
Use this function to stop receiving any Unicode text input events. See Section 6.2 [sdl.StartTextInput], page 23, for details.

INPUTS

none

7 Renderer reference

7.1 sdl.EnableOffscreenRender

NAME

sdl.EnableOffscreenRender – enable hardware brush offscreen rendering

SYNOPSIS

```
sdl.EnableOffscreenRender(on)
```

FUNCTION

This function can be used to enable or disable offscreen rendering to hardware brushes, depending on what you pass in the `on` argument.

Please note that the new setting will only affect hardware brushes created after you have made this call. Hardware brushes created before your call of this function will use the old setting.

Also note that hardware brushes that can be drawn to face some limitations. See Section 3.5 [Offscreen rendering], page 11, for details. That's why you should only use this function if your script is able to deal with these limitations. The advantage is that `sdl.EnableOffscreenRender()` allows your script to draw to offscreen hardware brushes with hardware acceleration. But you have to keep some things in mind when doing so. See Section 3.5 [Offscreen rendering], page 11, for details.

INPUTS

`on`	boolean indicating whether to enable or disable hardware brush offscreen rendering

EXAMPLE

```
sdl.EnableOffscreenRender(True)
CreateBrush(1, 640, 480, #BLACK, {Hardware = True})
sdl.EnableOffscreenRender(False)
```

The code above creates brush 1 as a hardware brush that can be drawn to using `SelectBrush()`. All other hardware brushes cannot be drawn to because we immediately set the enable offscreen render flag to `False` again.

7.2 sdl.GetCurrentRenderDriver

NAME

sdl.GetCurrentRenderDriver – get current render driver

SYNOPSIS

```
d$ = sdl.GetCurrentRenderDriver(display)
```

FUNCTION

This function returns the name of the current render driver, e.g. `opengl`, `direct3d`, `opengles`, etc.

INPUTS

`display`	identifier of display whose renderer should be retrieved

RESULTS

`d$` name of the current render driver

7.3 sdl.GetRenderDrawBlendMode

NAME

sdl.GetRenderDrawBlendMode – get draw blend mode

SYNOPSIS

```
blendmode = sdl.GetRenderDrawBlendMode(display)
```

FUNCTION

Use this function to get the blend mode used for drawing operations. See Section 7.24 [sdl.SetRenderDrawBlendMode], page 36, for a list of blend modes.

INPUTS

`display` identifier of display whose renderer should be used

RESULTS

`blendmode`
current blend mode

7.4 sdl.GetRenderDrawColor

NAME

sdl.GetRenderDrawColor – get draw color

SYNOPSIS

```
r, g, b, a = sdl.GetRenderDrawColor(display)
```

FUNCTION

Use this function to get the color used for drawing operations (rect, line and clear).

INPUTS

`display` identifier of display whose renderer should be used

RESULTS

`r` the red value used to draw on the rendering target (ranging from 0 to 255)

`g` the green value used to draw on the rendering target (ranging from 0 to 255)

`b` the blue value used to draw on the rendering target (ranging from 0 to 255)

`a` the alpha value used to draw on the rendering target (ranging from 0 to 255)

7.5 sdl.GetRendererOutputSize

NAME
sdl.GetRendererOutputSize – get renderer output size

SYNOPSIS
```
w, h = sdl.GetRendererOutputSize(display)
```

FUNCTION
Use this function to get the output size in pixels of a rendering context. If an error occurs, -1 is returned in both return values.

INPUTS

`display`	identifier of display whose renderer should be used

RESULTS

`w`	output width
`h`	output height

7.6 sdl.GetTextureAlphaMod

NAME
sdl.GetTextureAlphaMod – get texture alpha modulation

SYNOPSIS
```
alpha = sdl.GetTextureAlphaMod(tex)
```

FUNCTION
Use this function to get the additional alpha value multiplied into render copy operations. The `tex` argument must simply be the identifier of a hardware brush.

INPUTS

`tex`	identifier of hardware brush

RESULTS

`alpha`	alpha modulation or -1 on error

7.7 sdl.GetTextureBlendMode

NAME
sdl.GetTextureBlendMode – get texture blend mode

SYNOPSIS
```
mode = sdl.GetTextureBlendMode(tex)
```

FUNCTION
Use this function to get the blend mode used for texture copy operations. The `tex` argument must simply be the identifier of a hardware brush.

See Section 7.24 [sdl.SetRenderDrawBlendMode], page 36, for a list of blend modes.

INPUTS

`tex` identifier of hardware brush

RESULTS

`mode` the blend mode or -1 on error

7.8 sdl.GetTextureColorMod

NAME

sdl.GetTextureColorMod – get texture color modulation

SYNOPSIS

```
r, g, b = sdl.GetTextureColorMod(tex)
```

FUNCTION

Use this function to get the additional color value multiplied into render copy operations. The `tex` argument must simply be the identifier of a hardware brush.

On error, this function returns -1 in all return values.

INPUTS

`tex` identifier of hardware brush

RESULTS

`r` the red color value multiplied into copy operations (ranging from 0 to 255)

`g` the green color value multiplied into copy operations (ranging from 0 to 255)

`b` the blue color value multiplied into copy operations (ranging from 0 to 255)

7.9 sdl.RenderClear

NAME

sdl.RenderClear – clear target

SYNOPSIS

```
sdl.RenderClear(display)
```

FUNCTION

Use this function to clear the current rendering target with the drawing color. This function clears the entire rendering target, ignoring the viewport and the clip rectangle.

INPUTS

`display` identifier of display whose renderer should be used

7.10 sdl.RenderCopy

NAME

sdl.RenderCopy – draw texture

SYNOPSIS

```
sdl.RenderCopy(display, tex[, src, dst, angle, center, flip])
```

FUNCTION

Use this function to copy a portion of the texture specified by `tex` to the display specified by `display`, optionally rotating it by `angle` around the given center point and also flipping it top-bottom and/or left-right. The `tex` argument must simply be the identifier of a hardware brush.

If specified, `src` and `dst` must be tables containing the following fields:

`x` Left position of rectangle.

`y` Top position of rectangle.

`w` Rectangle width.

`h` Rectangle height.

Alternatively, you can also set `src` and/or `dst` to `Nil`. Passing `Nil` in `src` and/or `dst` means to use the entire size of the source or destination, respectively. If source and destination sizes do not match, `sdl.RenderCopy()` will automatically stretch the texture to fit to the destination rectangle.

If specified, `center` must be a table containing the following fields:

`x` Left position of center point.

`y` Top position of center point.

Alternatively, you can also set `center` to `Nil`. In that case, the center point will be set to the center of the destination rectangle.

If specified, `flip` must be a combination of the following predefined constants:

`#SDL_FLIP_NONE`
Do not flip

`#SDL_FLIP_HORIZONTAL`
Flip horizontally

`#SDL_FLIP_VERTICAL`
Flip vertically

The texture is blended with the destination based on its blend mode set with `sdl.SetTextureBlendMode()`.

The texture color is affected based on its color modulation set by `sdl.SetTextureColorMod()`.

The texture alpha is affected based on its alpha modulation set by `sdl.SetTextureAlphaMod()`.

INPUTS

`display` identifier of display whose renderer should be used

`tex`	identifier of hardware brush
`src`	optional: rectangle in the texture to use (defaults to `Nil` which means use the whole texture)
`dst`	optional: rectangle to draw to the display (defaults to `Nil` which means fill the entire display)
`angle`	optional: angle in degrees to apply to the destination rectangle in clockwise direction (defaults to 0)
`center`	optional: point around which to rotate the destination rectangle (defaults to `Nil` which means destination rectangle center)
`flip`	optional: flipping actions that should be performed (defaults to `#SDL_FLIP_NONE`)

7.11 sdl.RenderDrawLine

NAME

sdl.RenderDrawLine – draw line

SYNOPSIS

```
sdl.RenderDrawLine(display, x1, y1, x2, y2)
```

FUNCTION

Use this function to draw a line on the current rendering target.

The current drawing color is set by `sdl.SetRenderDrawColor()`, and the color's alpha value is ignored unless blending is enabled with the appropriate call to `sdl.SetRenderDrawBlendMode()`.

INPUTS

`display`	identifier of display whose renderer should be used
`x1`	the x coordinate of the start point
`y1`	the y coordinate of the start point
`x2`	the x coordinate of the end point
`y2`	the y coordinate of the end point

7.12 sdl.RenderDrawPoint

NAME

sdl.RenderDrawPoint – draw point

SYNOPSIS

```
sdl.RenderDrawPoint(display, x, y)
```

FUNCTION

Use this function to draw a point on the current rendering target.

The current drawing color is set by `sdl.SetRenderDrawColor()`, and the color's alpha value is ignored unless blending is enabled with the appropriate call to `sdl.SetRenderDrawBlendMode()`.

INPUTS

`display`	identifier of display whose renderer should be used
`x`	the x coordinate of the point
`y`	the y coordinate of the point

7.13 sdl.RenderDrawRect

NAME

sdl.RenderDrawRect – draw rectangle outline

SYNOPSIS

```
sdl.RenderDrawRect(display[, x, y, w, h])
```

FUNCTION

Use this function to draw a rectangle on the current rendering target. If you leave out the optional arguments, the whole rendering target will be outlined.

The current drawing color is set by `sdl.SetRenderDrawColor()`, and the color's alpha value is ignored unless blending is enabled with the appropriate call to `sdl.SetRenderDrawBlendMode()`.

INPUTS

`display`	identifier of display whose renderer should be used
`x`	optional: the x coordinate of the upper left corner
`y`	optional: the y coordinate of the upper left corner
`w`	optional: the rectangle width
`h`	optional: the rectangle height

7.14 sdl.RenderFillRect

NAME

sdl.RenderFillRect – draw filled rectangle

SYNOPSIS

```
sdl.RenderFillRect(display[, x, y, w, h])
```

FUNCTION

Use this function to fill a rectangle on the current rendering target. If you leave out the optional arguments, the whole rendering target will be filled.

The current drawing color is set by `sdl.SetRenderDrawColor()`, and the color's alpha value is ignored unless blending is enabled with the appropriate call to `sdl.SetRenderDrawBlendMode()`.

INPUTS

`display`	identifier of display whose renderer should be used
`x`	optional: the x coordinate of the upper left corner
`y`	optional: the y coordinate of the upper left corner
`w`	optional: the rectangle width
`h`	optional: the rectangle height

7.15 sdl.RenderGetClipRect

NAME

sdl.RenderGetClipRect – get clip rectangle

SYNOPSIS

```
x, y, w, h = sdl.RenderGetClipRect(display)
```

FUNCTION

Use this function to get the clip rectangle for the current target.

INPUTS

`display`	identifier of display whose renderer should be used

RESULTS

`x`	the x coordinate of the upper left corner
`y`	the y coordinate of the upper left corner
`w`	the rectangle width
`h`	the rectangle height

7.16 sdl.RenderGetLogicalSize

NAME

sdl.RenderGetLogicalSize – get logical size

SYNOPSIS

```
w, h = sdl.RenderGetLogicalSize(display)
```

FUNCTION

Use this function to get device independent resolution for rendering. If this function is called on a renderer which never had its logical size set by `sdl.RenderSetLogicalSize()`, this function returns 0 in both `w` and `h`.

INPUTS

`display`	identifier of display whose renderer should be used

RESULTS

`w`	the width of the logical resolution
`h`	the height of the logical resolution

7.17 sdl.RenderGetScale

NAME
sdl.RenderGetScale – get scaling factors

SYNOPSIS
```
scalex, scaley = sdl.RenderGetScale(display)
```

FUNCTION
Use this function to get the drawing scale for the current target.

INPUTS

`display`	identifier of display whose renderer should be used

RESULTS

`scalex`	the horizontal scaling factor
`scaley`	the vertical scaling factor

7.18 sdl.RenderGetViewport

NAME
sdl.RenderGetViewport – get viewport

SYNOPSIS
```
x, y, w, h = sdl.RenderGetViewport(display)
```

FUNCTION
Use this function to get the drawing area for the current target.

INPUTS

`display`	identifier of display whose renderer should be used

RESULTS

`x`	the x coordinate of the upper left corner
`y`	the y coordinate of the upper left corner
`w`	the rectangle width
`h`	the rectangle height

7.19 sdl.RenderPresent

NAME
sdl.RenderPresent – flip buffers

SYNOPSIS
```
sdl.RenderPresent(display)
```

FUNCTION

Use this function to update the screen with any rendering performed since the previous call.

SDL's rendering functions operate on a backbuffer; that is, calling a rendering function such as `sdl.RenderDrawLine()` does not directly put a line on the screen, but rather updates the backbuffer. As such, you compose your entire scene and present the composed backbuffer to the screen as a complete picture.

Therefore, when using SDL's rendering API, one does all drawing intended for the frame, and then calls this function once per frame to present the final drawing to the user.

The backbuffer should be considered invalidated after each present; do not assume that previous contents will exist between frames. You are strongly encouraged to call `sdl.RenderClear()` to initialize the backbuffer before starting each new frame's drawing, even if you plan to overwrite every pixel.

INPUTS

`display`	identifier of display whose renderer should be used

7.20 sdl.RenderSetClipRect

NAME

sdl.RenderSetClipRect – set clip rectangle

SYNOPSIS

```
sdl.RenderSetClipRect(display[, x, y, w, h])
```

FUNCTION

Use this function to set the clip rectangle for rendering on the specified target. If you leave out the optional arguments, clipping will be disabled.

INPUTS

`display`	identifier of display whose renderer should be used
`x`	optional: the x coordinate of the upper left corner
`y`	optional: the y coordinate of the upper left corner
`w`	optional: the rectangle width
`h`	optional: the rectangle height

7.21 sdl.RenderSetLogicalSize

NAME

sdl.RenderSetLogicalSize – set logical size

SYNOPSIS

```
sdl.RenderSetLogicalSize(display, w, h)
```

FUNCTION

Use this function to set a device independent resolution for rendering.

INPUTS

`display`	identifier of display whose renderer should be used
`w`	the width of the logical resolution
`h`	the height of the logical resolution

7.22 sdl.RenderSetScale

NAME

sdl.RenderSetScale – set scaling factors

SYNOPSIS

```
sdl.RenderSetScale(display, scalex, scaley)
```

FUNCTION

Use this function to set the drawing scale for rendering on the current target. The drawing coordinates are scaled by the x/y scaling factors before they are used by the renderer. This allows resolution independent drawing with a single coordinate system.

If this results in scaling or subpixel drawing by the rendering backend, it will be handled using the appropriate quality hints. For best results use integer scaling factors.

INPUTS

`display`	identifier of display whose renderer should be used
`scalex`	the horizontal scaling factor
`scaley`	the vertical scaling factor

7.23 sdl.RenderSetViewport

NAME

sdl.RenderSetViewport – set viewport

SYNOPSIS

```
sdl.RenderSetViewport(display[, x, y, w, h])
```

FUNCTION

Use this function to set the drawing area for rendering on the current target. If you leave out the optional arguments, the viewport is set to the entire target. When the window is resized, the current viewport is automatically centered within the new window size.

INPUTS

`display`	identifier of display whose renderer should be used
`x`	optional: the x coordinate of the upper left corner
`y`	optional: the y coordinate of the upper left corner
`w`	optional: the rectangle width
`h`	optional: the rectangle height

7.24 sdl.SetRenderDrawBlendMode

NAME

sdl.SetRenderDrawBlendMode – set draw blend mode

SYNOPSIS

```
sdl.SetRenderDrawBlendMode(display, blendmode)
```

FUNCTION

Use this function to set the blend mode used for drawing operations (fill and line).

`#SDL_BLENDMODE_NONE`
no blending

```
dstRGBA = srcRGBA
```

`#SDL_BLENDMODE_BLEND`
alpha blending

```
dstRGB = (srcRGB * srcA) + (dstRGB * (1-srcA))
dstA = srcA + (dstA * (1-srcA))
```

`#SDL_BLENDMODE_ADD`
additive blending

```
dstRGB = (srcRGB * srcA) + dstRGB
dstA = dstA
```

`#SDL_BLENDMODE_MOD`
color modulate

```
dstRGB = srcRGB * dstRGB
dstA = dstA
```

INPUTS

`display` identifier of display whose renderer should be used

`blendmode`
one of the blend modes from above

7.25 sdl.SetRenderDrawColor

NAME

sdl.SetRenderDrawColor – set draw color

SYNOPSIS

```
sdl.SetRenderDrawColor(display, r, g, b[, a])
```

FUNCTION

Use this function to set the color for drawing or filling rectangles, lines, and points, and for `sdl.RenderClear()`.

If you want to have alpha blending, use `sdl.SetRenderDrawBlendMode()` to specify how the alpha channel is used.

INPUTS

`display` identifier of display whose renderer should be used

`r`	the red value used to draw on the rendering target (ranging from 0 to 255)
`g`	the green value used to draw on the rendering target (ranging from 0 to 255)
`b`	the blue value used to draw on the rendering target (ranging from 0 to 255)
`a`	optional: the alpha value used to draw on the rendering target (defaults to `#SDL_ALPHA_OPAQUE`)

7.26 sdl.SetRenderTarget

NAME

sdl.SetRenderTarget – set render target

SYNOPSIS

```
sdl.SetRenderTarget(display[, tex])
```

FUNCTION

Use this function to set a texture as the current rendering target. The `tex` argument must simply be the identifier of a hardware brush. If you leave out the `tex` argument, the default render target is used.

INPUTS

`display`	identifier of display whose renderer should be used
`tex`	optional: hardware brush to use as target

7.27 sdl.SetTextureAlphaMod

NAME

sdl.SetTextureAlphaMod – set texture alpha modulation

SYNOPSIS

```
r = sdl.SetTextureAlphaMod(tex, alpha)
```

FUNCTION

Use this function to set an additional alpha value multiplied into render copy operations. When this texture is rendered, during the copy operation the source alpha value is modulated by this alpha value according to the following formula:

```
srcA = srcA * (alpha / 255)
```

Note that alpha modulation is not always supported by the renderer; it will return -1 if alpha modulation is not supported.

The `tex` argument must simply be the identifier of a hardware brush.

INPUTS

`tex`	identifier of hardware brush
`alpha`	the source alpha value multiplied into copy operations (ranging from 0 to 255)

RESULTS

`r` 0 on success or a negative error code on failure

7.28 sdl.SetTextureBlendMode

NAME

sdl.SetTextureBlendMode – set texture blend mode

SYNOPSIS

```
r = sdl.SetTextureBlendMode(tex, blendmode)
```

FUNCTION

Use this function to set the blend mode for a texture, used by `sdl.RenderCopy()`.

`blendmode` may be one of the following:

`#SDL_BLENDMODE_NONE`
no blending

```
dstRGBA = srcRGBA
```

`#SDL_BLENDMODE_BLEND`
alpha blending

```
dstRGB = (srcRGB * srcA) + (dstRGB * (1-srcA))
dstA = srcA + (dstA * (1-srcA))
```

`#SDL_BLENDMODE_ADD`
additive blending

```
dstRGB = (srcRGB * srcA) + dstRGB
dstA = dstA
```

`#SDL_BLENDMODE_MOD`
color modulate

```
dstRGB = srcRGB * dstRGB
dstA = dstA
```

If the blend mode is not supported, the closest supported mode is chosen and this function returns -1. The `tex` argument must simply be the identifier of a hardware brush.

INPUTS

`tex` identifier of hardware brush

`blendmode`
the blend mode to use for texture blending (see above)

RESULTS

`r` 0 on success or a negative error code on failure

7.29 sdl.SetTextureColorMod

NAME

sdl.SetTextureColorMod – set texture color modulation

SYNOPSIS

```
r = sdl.SetTextureColorMod(tex, r, g, b)
```

FUNCTION

Use this function to set an additional color value multiplied into render copy operations. When this texture is rendered, during the copy operation each source color channel is modulated by the appropriate color value according to the following formula:

```
srcC = srcC * (color / 255)
```

Color modulation is not always supported by the renderer; it will return -1 if color modulation is not supported.

The `tex` argument must simply be the identifier of a hardware brush.

INPUTS

`tex`	identifier of hardware brush
`r`	the red color value multiplied into copy operations (ranging from 0 to 255)
`g`	the green color value multiplied into copy operations (ranging from 0 to 255)
`b`	the blue color value multiplied into copy operations (ranging from 0 to 255)

RESULTS

`r`	0 on success or a negative error code on failure

8 System reference

8.1 sdl.ClearError

NAME
sdl.ClearError – clear last error

SYNOPSIS
`sdl.ClearError()`

FUNCTION
Use this function to clear any previous error message.

INPUTS
none

8.2 sdl.GetCurrentVideoDriver

NAME
sdl.GetCurrentVideoDriver – get current video driver

SYNOPSIS
`d$ = sdl.GetCurrentVideoDriver()`

FUNCTION
This function returns the name of the current video driver, e.g. `windows`, `cocoa`, `x11`, etc.

INPUTS
none

RESULTS

`d$`	name of the current video driver

8.3 sdl.GetError

NAME
sdl.GetError – get last error

SYNOPSIS
`e$ = sdl.GetError()`

FUNCTION
Returns a message with information about the specific error that occurred, or an empty string if there hasn't been an error message set since the last call to `sdl.ClearError()`. The message is only applicable when an SDL function has signaled an error. You must check the return values of SDL function calls to determine when to appropriately call `sdl.GetError()`.

INPUTS
none

RESULTS

`e$`	last error message or empty string

8.4 sdl.GetVersion

NAME
sdl.GetVersion – get SDL version

SYNOPSIS
```
ver, rev, patch = sdl.GetVersion()
```

FUNCTION
Use this function to get the version of SDL that is linked against your program.

INPUTS
none

RESULTS

`ver`	major version
`rev`	minor version
`patch`	update version (patch level)

9 Window reference

9.1 sdl.SetWindowFullscreen

NAME

sdl.SetWindowFullscreen – switch display mode

SYNOPSIS

```
sdl.SetWindowFullscreen(display, mode)
```

FUNCTION

Use this function to set a window's fullscreen state.

`mode` must be one of the following predefined constants:

`#SDL_WINDOW_FULLSCREEN`
: Real fullscreen with a videomode change

`#SDL_WINDOW_FULLSCREEN_DESKTOP`
: Fake fullscreen that takes the size of the desktop

`#SDL_WINDOW_WINDOW`
: Windowed mode

INPUTS

`display`	identifier of display whose renderer should be used
`mode`	new display mode

Appendix A Licenses

A.1 SDL license

Index