Scriptable Mediaplayer Ver. 2
smplayer2 Documentation

Andreas Schiffler, aschiffler@ferzkopp.net, 4 Jan 2009

Introduction

The smplayer2 software was created as a frontend driver for digital signage project and has since been used in some media-art installations.

It is based on a Linux PC with OpenGL graphics acceleration.

Features:

- allows for flat-field calibrated synchronized playback on up to 2 screens
- can handle several SD and HD streams concurrently
- contains many commands for drawing graphics and text to screen
- contains many utility commands for controlling display script (i.e. parallel port IO, http interface)
- many image sources (images, video clips, webcam, VNC)
- based on the LUA scripting language
- runs stable for long times

Limitations (as per Ver 2.0 released 4 Jan 2009):

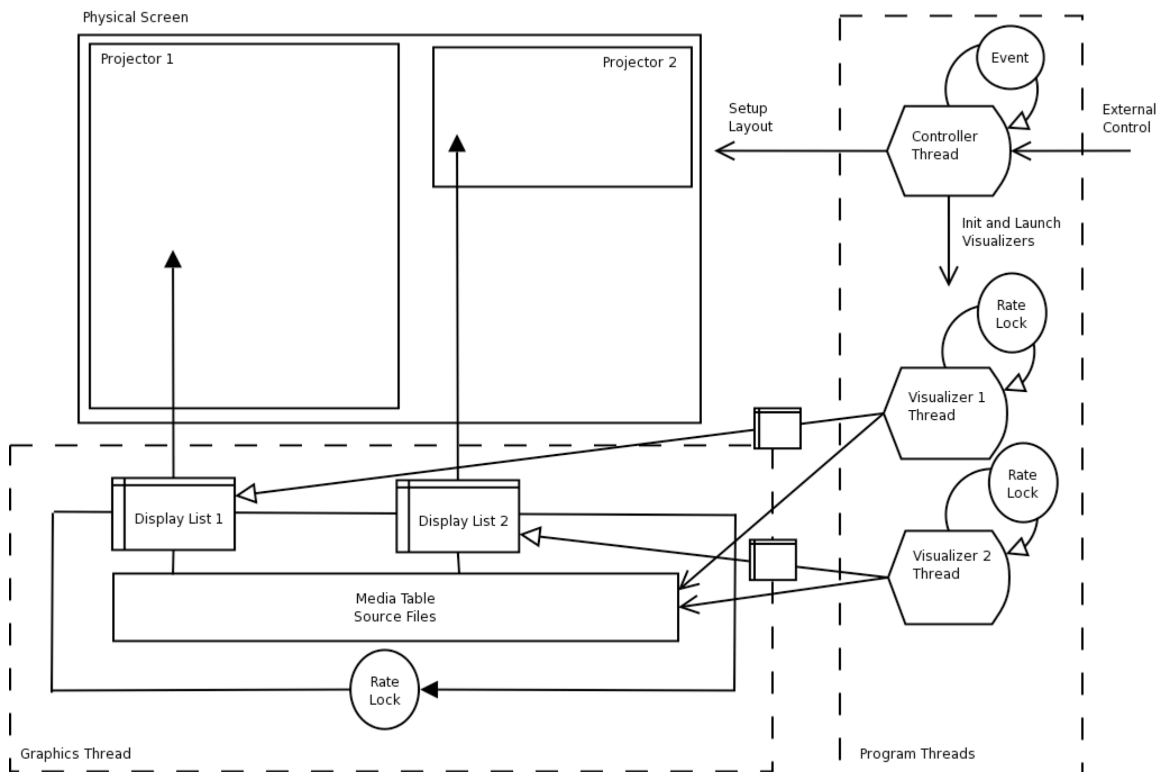
- video must be converted into proprietary file format (stream of compressed textures) before playback. This is done using a custom mplayer binary and the files may consume a lot of disk space.
- Only partially documented LUA API
- Scripts are not easy to program and debug; lack of IDE.
- Currently no (or very limited) audio support.
- … probably more issues ☺ …

Other than that, it is a very unique and capable piece of software that can help solving unique media display problems in a neat way.

General Software Design

The software implements a multithreaded scriptable “game loop”. The graphics thread controls the drawing of OpenGL textures at a fixed rate to the physical screen. The drawing algorithm allows for flatfield calibration (keystone). The program thread executes the media script and performs all

“work” in controller- and visualizer-coroutines (not real threads). These make callbacks to scripts wich control visualizers.



LUA Scripting Engine

smplayer2 uses the LUA programming engine for screen and media setup and to control the display while the player is running.

Lua is a very compact, relatively fast and easily extensible functional programming language. Smplayer2 extends it with the following APIs:

- smplayer2 specific objects and functions
- utility APIs (byte buffer, bit manupulation, compression, etc.)
- curl API for web/internet access
- various utility functions written in LUA code

Compiling smplayer2

The smplayer2 code is designed to compile and run on a 32bit Linux installation (i.e. last used version was Mandriva Linux 2008.1). Since the player requires hardware accelerated OpenGL support, the proprietary drivers (i.e. from Nvidia) are typically required.

The core mediaplayer depends on many secondary libraries common to a Linux distribution. These should be installed as development packages prior to compiling smplayer2:

- SDL : simple direct media lib, core for graphics
- SDL_gfx : graphics primitives
- SDL_image : image loading
- SDL_net : network primitives
- SDL_ttf : font support and text drawing
- curl : http access, used by lcurl
- expat : XMl parser
- ffmpeg : video codecs, used by custom mplayer
- freetype : TTF rendering, used by SDL_ttf
- jpeg : JPG codec, used by SDL_image
- libbgrab : v4l interface, used by SDL_bgrab
- liblzf : fast compression, used by lbuffer and custom mplayer
- png : PNG codec, used by SDL_image
- openssl : SSL support, used by curl
- zlib : ZIP codec, used by SDL_image

The following locally provided custom libraries need to be also compiled and installed:

- SDL_bgrab : libbgrab interface for SDL, multithreaded v4l-based framegrabber interface
- SDL_vnc : simple multithreaded VNC client for SDL

The LUA scripting engine is provided and extended using these libraries which need to be compiled and installed:

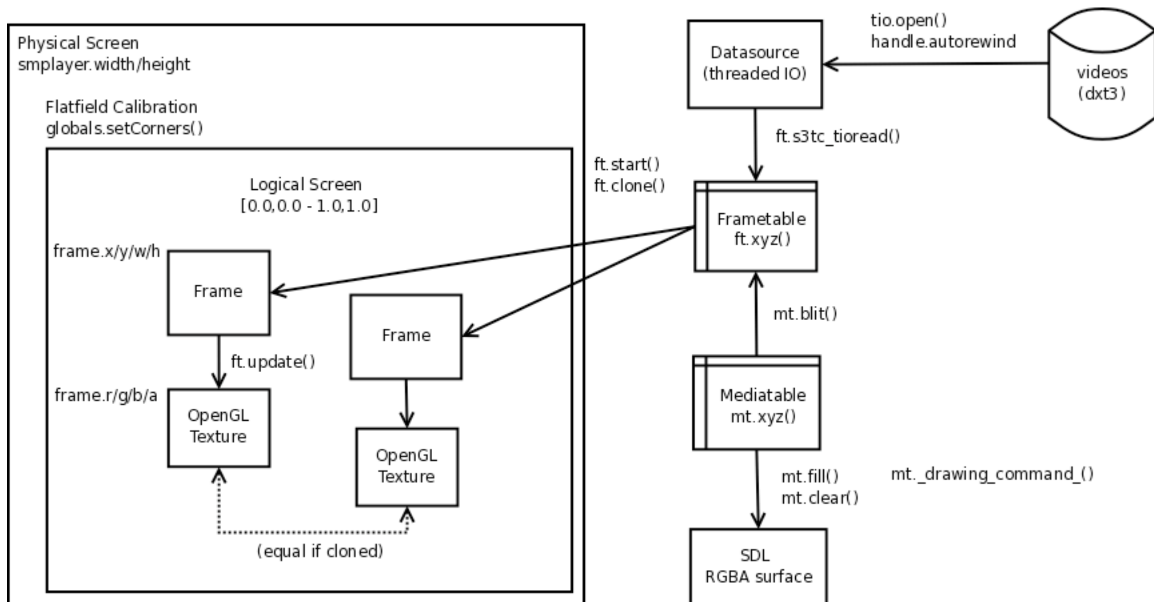
- lua-5.0.2 : custom lua interpreter
- liblzf : fast compression, used by lbuffer
- lbuffer : LUA memory buffers
- lcurl : LUA interface to libcurl
- lthread : LUA threading interface
- lutils : collection of LUA utilities

The video compression format is based on a custom mplayer output mode “glstream”. The source archive contains a custom mplayer version “MPlayer-1.0rc1-glstream” which should be compiled separately.

Data Flow

The following diagram shows the typical data flow and relevant commands between the main API objects:

- globals : physical screen calibration
- smplayer : logical screen information
- ft : frame table, maps of logical frames to the logical screen, corresponds to OpenGL textures
- mt : media table, store of in-memory display surfaces



When smplayer is initially started, the frametable and mediatable are initially empty. The script's Init() callback is executed which generally sets up the physical screen, the calibration and ALL frames and media used during the execution of the program. The the display is opened and the "game loop" entered. Repeated callbacks are made for each frame in the frametable to see if their textures need to be updated. On Mouse or Keyboard events, optional callbacks are executed. If the game loop is terminated the Done() callback is executed.

Script Callback Interface

The smplayer2 executable is started with a LUA script as input. The script needs to provide several callback functions. The following script is the default that needs to be extended by the user:

```
-- Default smplayer2 script
```

```
dofile("utility50.lua")
function Config()
  -- Setup display configuration
end

function Init()
  -- Script initialization
end

function Done()
  -- Script cleanup
end

function Redraw(frame)
  -- Redraw of frame 'frame' requested
end

function Key(c,s)
  -- Key c (string), sym s (int) pressed
end

function Mouse(s,x,y,t)
  -- Mouse clicked (t=1) or
  -- dragged (t=2) on screen s at coordinate x,y
end
```

Example Program (Code)

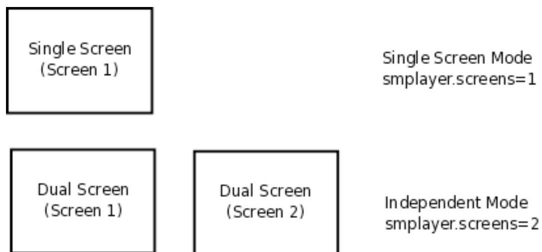
```
-- Single frame showing single static image from file
function Init()
 -- create a frame which will display at 1Hz
 myframe=ft.start(1, 1024, 768, 1.00);
 -- create an empty output pane used to assemble graphics
 mypane=mt.create(1024,768);
 -- load an image into (also creates a pane)
 myimg=mt.image("Media/column.jpg");
 -- draw the image into the output pane
 mt.blit(mypane, myimg, 0.25*1024,0,0);
 -- create a text pane
 mytext = mt.text(“Hello”, “arial.ttf”,18,0, "white");
 -- draw the text into the output pane
 mt.blit(mypane, mytext, 10, 10, 1);
 -- update frame with pane
 ft.blit(myframe, mypane, 0, 0, 0);
end
```

```
function Redraw(frame)
 -- update screen with frame
 ft.update(myframe);
end

function Done()
 -- cleanup all objects
 myframe=nil
 mypane=nil
 myimg=nil
 mytext=nil
 collectgarbage();
end
```

Screen Setup

The system supports one or two physical screens.



The application assumes that the X-server has been correctly configured beforehand (i.e. Nvidia driver with OpenGL and TWINVIEW enabled). Smplayer2 will first parse the Init() section of the input script and then open a window of size [width * screens, height].

Physical to Logical Mapping

Once the physical window has been created, the media script always uses the logical screen coordinates of [0,0 – 1,0] for all frame positioning and drawing commands. This means:

- screen size changes do not affect the playback
- the script must manage all aspect ratios and changes

Debugging of Setup

Additional settings can be specified to assist in the debugging of the application:

smplayer.cursor=1 – enables the mouse cursor
smplayer.outline=1 – enables the display of texture outlines

Flatfield Calibration

To allow for matching the full-screen output of the display device to the actual screen, the system supports a full flatfield calibration (4-point keystone correction). This can be used to:

- pixel match the output of two projectors displaying the same image
- make the images of two adjacent projectors to appear seamless
- fit a screen area to an object

To do that the setCorners command receives an array 8 numbers corresponding to the 4 corners of the screen in the unit coordinate system [0,0 – 1,1].

```
-- default corners
data_screen_one =
     {[1]=0,[2]=0,[3]=1,[4]=0,
     [5]=1,[6]=1,[7]=0,[8]=1,}
globals.setCorners(1,data_screen_one)

-- half size centered corners
data_screen_two =
     {[1]=0.25,[2]=0.25,[3]=0.75,[4]=0.25,
     [5]=0.75,[6]=0.75,[7]=0.25,[8]=0.75,}
globals.setCorners(2,data_screen_two)
```

Interactive Screen Setup (Code)

The following Mouse callback function fragment can be used to interactively change the screen calibration on the fly.

```
-- Assumes the variable edit_screen was setup
-- and is 1 or 2 if in edit mode or 0 otherwise
-- TODO: toggle edit_screen from the keyboard
function Mouse(screen,x,y,t)
 -- warp screen
 if (edit_screen>0) then
  if (edit_screen==screen) then
   corners = smplayer.getCorners(edit_screen)
```

```
   if (edit_corner==1) then
    corners[1]=x
    corners[2]=y
   elseif (edit_corner==2) then
    corners[3]=x
    corners[4]=y
   elseif (edit_corner==3) then
    corners[5]=x
    corners[6]=y
   elseif (edit_corner==4) then
    corners[7]=x
    corners[8]=y
   elseif (edit_corner==5) then
    -- no shift
   end
   smplayer.setCorners(edit_screen, corners)
   -- TODO: save the current configuration stored
   -- in the 'corners' variable
  end
 end
end
```

Video Compression

The smplayer2 video playback uses a stream of compressed textures which are generated by the "glstream" output plugin of a custom mplayer version. The glstream writer supports several GL pixel modes, S3TC compressed textures, alpha channel creation and LZF compression of frames.

The source to the driver can be found in the libvo\vo_glstream.c file.

Mplayer Usage

Example:
mplayer -vo glstream:verbose:dxt5:file=output.dat input.avi

<u>Compression Options:</u>

none	(no compression, no output - for benchmarking)
rgba	(uncompressed texture, 32bpp)
lum	(uncompressed texture, luminance, 8bpp)
luma	(uncompressed texture, luminance-alpha, 16bpp)
dxt1	(6bpp)
dxt3	(default, 8bpp)

dxt5 (8bpp)

Other options:

file=filename	Save to file <filename> (default: texture.dat)
chromakey	Chromakey blue for alpha (default: off)
lzf	LZF compress frame (default: off)
verbose	Turn on verbose output (default: off)

glstream Fileformat

```
 Fileformat is:
  [ file ] = [ fragment 0 ] [ fragment 1 ] . . .
  [ fragment X ] = [ header X ] [ frame X ]

 where

  [ header (22 bytes) ] =
    [ magic (4 char) ]
    [ flags (uint16) ]
    [ width (uint16) ]
    [ height (uint16) ]
    [ OpenGL format (uint32) ]
    [ x uncompressed bytes per frame (uint32) ]
    [ y compressed bytes per frame (uint32) ]

  [ frame ] =
    [ data (x or y bytes per frame depending on flags) ]

  [ magic ] =
    GLST

  [ OpenGL format ] =
    GL_RGBA |
    GL_LUMINANCE |
    GL_LUMINANCE_ALPHA |
    GL_COMPRESSED_RGBA_S3TC_DXT1_EXT |
    GL_COMPRESSED_RGBA_S3TC_DXT3_EXT |
    GL_COMPRESSED_RGBA_S3TC_DXT5_EXT

  [ flags ] =
   0x0001 - LZF compressed frame

Note: Headers are always repeated before each frame to make reading
streams easier.
```

Sample Conversion Script

The following bash script will convert all input AVI files into “square pixel” SD quality output texture streams cropping the center 480x480 square from the source video frame.

```
#!/bin/sh
MPLAYER="/usr/local/bin/mplayer"
PARAMS="-nosound -noaspect -vop scale=256:256,crop=480:480:80:0 -vo
glstream:dxt3:lzf:verbose:file="
for i in *.avi; do
 INPUT=$i
 OUTPUT=${i%%.avi}_box.dxt3_lzf
 $MPLAYER $PARAMS$OUTPUT $INPUT
done
```

Note: The target frame size (set with ft.start() in the LUA script) and the texture size (set by the scale option of mplayer) should match for best performance during playback.

Smplayer2 LUA API

Global API

<u>Namespace</u>

globals.xyz
smplayer.xyz

<u>Properties</u>

width (Get, Set)
height (Get, Set)
screens (Get, Set)
cursor (Get, Set)
outline (Get, Set)
draw (Get, Set)

<u>Functions</u>

create() : create object and make it global (automatically called by smplayer2)
list() : displays current smplayer configuration for debugging purposes

float[8] = getCorners(screen) : returns current display positions of screens

setCorners(screen, float[8]) : set display positions of screens for flatfield calibration

quit() : quit game loop and exit smplayer

Frametable API

<u>Namespace</u>

ft.xyz

<u>Properties</u>

screens (get, set) : the screen to place the frame onto
tw (get) : the texture width; power of two
th (get) : the texture height; power of two
r (get, set) : red filter, range=[0.0 – 1.0], default=1.0
g (get, set) : green filter, range=[0.0 – 1.0], default=1.0
b (get, set) : blue filter, range=[0.0 – 1.0], default=1.0
a (get, set) : transparency of frame, range=[0.0 – 1.0], default=1.0
rate (get, set) : update framerate in Hz
x (get, set) : top right X position on logical screen, range=[0.0 – 1.0]
y (get, set) : top right Y position on logical screen, range=[0.0 – 1.0]
w (get, set) : width on logical screen, range=[0.0 – 1.0]
h (get, set) : height on logical screen, range=[0.0 – 1.0]

<u>Functions</u>

start(frame_id, texture_width, texture_height, framerate_hz) : creates new frame object in frametable

list() : lists all frames in frametable for debugging purposes
stop(frame_id) : deletes frame from frametable

clone(source_frame_id, target_frame_id) : sets the content to be cloned from existing frame

unclone(frame_id) : disables cloning for a frame

delay(delay_milliseconds) : do a millisecond delay

color(frame_id, r, g, b, a) : sets the color filters and transparency

rate(frame_id, rate) : sets the frame rate for updates

float[8] = getCorners(frame_id) gets the current corner coordinates in the logical coordinate system [0,0 – 1,1]

setCorners(frame_id, float[8]) : sets the current corner coordinates coordinates in the logical coordinate system [0,0 – 1,1]

warp(frame_id, x1,y1,x2,y2,x3,y3,x4,y4) : sets the current corner coordinates coordinates in the logical coordinate system [0,0 – 1,1]; equivalent to setCorners but different call interface

crop(frame_id, float[8]) : : sets new source texture coordinates; used for cropping or enlarging textures

blit(frame_id, media_id, x, y, blendFlag) : transfer mediatable content into frame

s3t_read(frame_id, fileHandle) : read texture into frame; sync IO

s3tc_tioread(frame_id, tioFileHandle) : read texture into frame; multithreaded async IO

save(frame_id, filename) : save current frame to a file for debugging

update(frame_id) : mark frame for texture update when rate interval expires

vncopen(frame_id, serverHost, serverPort, serverEncoding, serverPassword) : open connection to VNC server

vncupdate(frame_id) : update frame from vnc connection

vncclose(frame_id) : close vnc connection

grabopen(frame_id, device, width, height, channel, mode, frequencyRegion, frequencyIndex) : open framegrabber device
grabupdate(frame_id, deinterlaceFlag) : update frame from framegrabber device

grabclose(frame_id) : close framegrabber connection

Mediatable API

Namespace

mt.xyz

Properties

w (Get) : width of media surface
h (Get) : heigh of media surface
cx (Get, Set) : top right X position of clipping rectangle of media surface
cy (Get, Set) : top right Y position of clipping rectangle of media surface
cw (Get, Set) : width of clipping rectangle of media surface
ch (Get, Set) : height of clipping rectangle of media surface

Functions

image(filename) : create new media surface by load image from file

clone(media_id) : create new media surface from existing media

create(width, height) : create new blank media surface

text(text, fontfile, fontsize, fontstyle, color) : create new media surface from text

rotozoom(media_id, angle, zoomFactor) : create new media surface by rotating and zooming existing media surface

list() : display all media surfaces

clear(media_id, alpha) : clear media surface to particular alpha value

fill(media_id, color) : fill media surface with color

width(media_id, w) : set width of media surface

height(media_id, h) : set height of media surface

blit(source_media_id, target_media_id, x, y, useAlphaFlag) : blit source media surface to target media surface at specified position optionally using alpha blending

clip(media_id, x,y,w,h) : set clipping rectangle for drawing operations on media surface

unclip(media_id) : clear clipping rectangle of media surface

<u>SDL_gfx Mapped Functions</u>
pixel(media_id, x, y, color) : draw pixel
hline(media_id, x1, x2, y, color) : draw horizontal line
vline(media_id, x, y1, y2, color) : draw vertical line
rectangle(media_id, x1,y1,x2,y2, color) : draw rectangle
box(media_id, x1,y1,x2,y2, color) : draw filled rectangle
line(media_id, x1,y1,x2,y2, color) : draw line
aaline(media_id, x1,y1,x2,y2, color) : draw antialiased line
circle(media_id, x,y,r, color) : draw circle
aacircle(media_id, x,y,r, color) : draw antialiased circle
filledcircle(media_id, x,y,r, color) : draw filled circle
ellipse(media_id, x,y,rx,ry, color) : draw ellipse
aaellipse(media_id, x,y,rx,ry, color) : draw antialiased ellipse
filledellipse(media_id, x,y,rx,ry, color) : draw filled ellipse
pie(media_id, x,y,rad,start,end, color) : draw pie
filledpie(media_id, x,y,rad,start,end, color) : draw filled pie
trigon(media_id, x1,y1,x2,y2,x3,y3, color) : draw triangle
aatrigon(media_id, x1,y1,x2,y2,x3,y3, color) : draw antialiased triangle
filledtrigon(media_id, x1,y1,x2,y2,x3,y3, color) : draw filled triangle
polygon() : not implemented
aapolygon() : not implemented
filledpolygon() : not implemented
bezier() : not implemented
string(media_id, x,y, text, color) : draw text in 8x8 fixed width font to media surface