/\*

u8g\_polygon.c

Implementation of a polygon draw algorithm for "convex" polygons.

Universal 8bit Graphics Library

Copyright (c) 2013, olikraus@gmail.com

All rights reserved.

Redistribution and use in source and binary forms, with or without modification,

are permitted provided that the following conditions are met:

\* Redistributions of source code must retain the above copyright notice, this list

of conditions and the following disclaimer.

\* Redistributions in binary form must reproduce the above copyright notice, this

list of conditions and the following disclaimer in the documentation and/or other

materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND

CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,

INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF

MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE

DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR

CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,

SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT

NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER

CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,

STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)

ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF

ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

See also:

http://www.angelfire.com/linux/myp/ConvexPolRas/ConvexPolRas.html

Computer Graphics, Principles and Practice, Foley, van Dam, Feiner, Hughes (pp 92)

Michael Abrash's Graphics Programming Black Book, Special Edition (Chapter 38 and 39)

Optimized for embedded systems

- static memory usage only

- consistent data types

- low flash ROM consumption

\*/

#include "u8g.h"

/\*===========================================\*/

/\* procedures, which should not be inlined (save as much flash ROM as possible \*/

static uint8\_t pge\_Next(struct pg\_edge\_struct \*pge) PG\_NOINLINE;

static uint8\_t pg\_inc(pg\_struct \*pg, uint8\_t i) PG\_NOINLINE;

static uint8\_t pg\_dec(pg\_struct \*pg, uint8\_t i) PG\_NOINLINE;

static void pg\_expand\_min\_y(pg\_struct \*pg, pg\_word\_t min\_y, uint8\_t pge\_idx) PG\_NOINLINE;

static void pg\_line\_init(pg\_struct \* const pg, uint8\_t pge\_index) PG\_NOINLINE;

/\*===========================================\*/

/\* line draw algorithm \*/

static uint8\_t pge\_Next(struct pg\_edge\_struct \*pge)

{

if ( pge->current\_y >= pge->max\_y )

return 0;

pge->current\_x += pge->current\_x\_offset;

pge->error += pge->error\_offset;

if ( pge->error > 0 )

{

pge->current\_x += pge->x\_direction;

pge->error -= pge->height;

}

pge->current\_y++;

return 1;

}

/\* assumes y2 > y1 \*/

static void pge\_Init(struct pg\_edge\_struct \*pge, pg\_word\_t x1, pg\_word\_t y1, pg\_word\_t x2, pg\_word\_t y2)

{

pg\_word\_t dx = x2 - x1;

pg\_word\_t width;

pge->height = y2 - y1;

pge->max\_y = y2;

pge->current\_y = y1;

pge->current\_x = x1;

if ( dx >= 0 )

{

pge->x\_direction = 1;

width = dx;

pge->error = 0;

}

else

{

pge->x\_direction = -1;

width = -dx;

pge->error = 1 - pge->height;

}

pge->current\_x\_offset = dx / pge->height;

pge->error\_offset = width % pge->height;

}

/\*===========================================\*/

/\* convex polygon algorithm \*/

static uint8\_t pg\_inc(pg\_struct \*pg, uint8\_t i)

{

i++;

if ( i >= pg->cnt )

i = 0;

return i;

}

static uint8\_t pg\_dec(pg\_struct \*pg, uint8\_t i)

{

i--;

if ( i >= pg->cnt )

i = pg->cnt-1;

return i;

}

static void pg\_expand\_min\_y(pg\_struct \*pg, pg\_word\_t min\_y, uint8\_t pge\_idx)

{

uint8\_t i = pg->pge[pge\_idx].curr\_idx;

for(;;)

{

i = pg->pge[pge\_idx].next\_idx\_fn(pg, i);

if ( pg->list[i].y != min\_y )

break;

pg->pge[pge\_idx].curr\_idx = i;

}

}

static uint8\_t pg\_prepare(pg\_struct \*pg)

{

pg\_word\_t max\_y;

pg\_word\_t min\_y;

uint8\_t i;

/\* setup the next index procedures \*/

pg->pge[PG\_RIGHT].next\_idx\_fn = pg\_inc;

pg->pge[PG\_LEFT].next\_idx\_fn = pg\_dec;

/\* search for highest and lowest point \*/

max\_y = pg->list[0].y;

min\_y = pg->list[0].y;

pg->pge[PG\_LEFT].curr\_idx = 0;

for( i = 1; i < pg->cnt; i++ )

{

if ( max\_y < pg->list[i].y )

{

max\_y = pg->list[i].y;

}

if ( min\_y > pg->list[i].y )

{

pg->pge[PG\_LEFT].curr\_idx = i;

min\_y = pg->list[i].y;

}

}

/\* calculate total number of scan lines \*/

pg->total\_scan\_line\_cnt = max\_y;

pg->total\_scan\_line\_cnt -= min\_y;

/\* exit if polygon height is zero \*/

if ( pg->total\_scan\_line\_cnt == 0 )

return 0;

/\* if the minimum y side is flat, try to find the lowest and highest x points \*/

pg->pge[PG\_RIGHT].curr\_idx = pg->pge[PG\_LEFT].curr\_idx;

pg\_expand\_min\_y(pg, min\_y, PG\_RIGHT);

pg\_expand\_min\_y(pg, min\_y, PG\_LEFT);

/\* check if the min side is really flat (depends on the x values) \*/

pg->is\_min\_y\_not\_flat = 1;

if ( pg->list[pg->pge[PG\_LEFT].curr\_idx].x != pg->list[pg->pge[PG\_RIGHT].curr\_idx].x )

{

pg->is\_min\_y\_not\_flat = 0;

}

else

{

pg->total\_scan\_line\_cnt--;

if ( pg->total\_scan\_line\_cnt == 0 )

return 0;

}

return 1;

}

static void pg\_hline(pg\_struct \*pg, u8g\_t \*u8g)

{

pg\_word\_t x1, x2, y;

x1 = pg->pge[PG\_LEFT].current\_x;

x2 = pg->pge[PG\_RIGHT].current\_x;

y = pg->pge[PG\_RIGHT].current\_y;

if ( y < 0 )

return;

if ( y >= u8g\_GetHeight(u8g) )

return;

if ( x1 < x2 )

{

if ( x2 < 0 )

return;

if ( x1 >= u8g\_GetWidth(u8g) )

return;

if ( x1 < 0 )

x1 = 0;

if ( x2 >= u8g\_GetWidth(u8g) )

x2 = u8g\_GetWidth(u8g);

u8g\_DrawHLine(u8g, x1, y, x2 - x1);

}

else

{

if ( x1 < 0 )

return;

if ( x2 >= u8g\_GetWidth(u8g) )

return;

if ( x2 < 0 )

x1 = 0;

if ( x1 >= u8g\_GetWidth(u8g) )

x1 = u8g\_GetWidth(u8g);

u8g\_DrawHLine(u8g, x2, y, x1 - x2);

}

}

static void pg\_line\_init(pg\_struct \* pg, uint8\_t pge\_index)

{

struct pg\_edge\_struct \*pge = pg->pge+pge\_index;

uint8\_t idx;

pg\_word\_t x1;

pg\_word\_t y1;

pg\_word\_t x2;

pg\_word\_t y2;

idx = pge->curr\_idx;

y1 = pg->list[idx].y;

x1 = pg->list[idx].x;

idx = pge->next\_idx\_fn(pg, idx);

y2 = pg->list[idx].y;

x2 = pg->list[idx].x;

pge->curr\_idx = idx;

pge\_Init(pge, x1, y1, x2, y2);

}

static void pg\_exec(pg\_struct \*pg, u8g\_t \*u8g)

{

pg\_word\_t i = pg->total\_scan\_line\_cnt;

/\* first line is skipped if the min y line is not flat \*/

pg\_line\_init(pg, PG\_LEFT);

pg\_line\_init(pg, PG\_RIGHT);

if ( pg->is\_min\_y\_not\_flat != 0 )

{

pge\_Next(&(pg->pge[PG\_LEFT]));

pge\_Next(&(pg->pge[PG\_RIGHT]));

}

do

{

pg\_hline(pg, u8g);

while ( pge\_Next(&(pg->pge[PG\_LEFT])) == 0 )

{

pg\_line\_init(pg, PG\_LEFT);

}

while ( pge\_Next(&(pg->pge[PG\_RIGHT])) == 0 )

{

pg\_line\_init(pg, PG\_RIGHT);

}

i--;

} while( i > 0 );

}

/\*===========================================\*/

/\* API procedures \*/

void pg\_ClearPolygonXY(pg\_struct \*pg)

{

pg->cnt = 0;

}

void pg\_AddPolygonXY(pg\_struct \*pg, u8g\_t \*u8g, int16\_t x, int16\_t y)

{

if ( pg->cnt < PG\_MAX\_POINTS )

{

pg->list[pg->cnt].x = x;

pg->list[pg->cnt].y = y;

pg->cnt++;

}

}

void pg\_DrawPolygon(pg\_struct \*pg, u8g\_t \*u8g)

{

if ( pg\_prepare(pg) == 0 )

return;

pg\_exec(pg, u8g);

}

pg\_struct u8g\_pg;

void u8g\_ClearPolygonXY(void)

{

pg\_ClearPolygonXY(&u8g\_pg);

}

void u8g\_AddPolygonXY(u8g\_t \*u8g, int16\_t x, int16\_t y)

{

pg\_AddPolygonXY(&u8g\_pg, u8g, x, y);

}

void u8g\_DrawPolygon(u8g\_t \*u8g)

{

pg\_DrawPolygon(&u8g\_pg, u8g);

}

void u8g\_DrawTriangle(u8g\_t \*u8g, int16\_t x0, int16\_t y0, int16\_t x1, int16\_t y1, int16\_t x2, int16\_t y2)

{

u8g\_ClearPolygonXY();

u8g\_AddPolygonXY(u8g, x0, y0);

u8g\_AddPolygonXY(u8g, x1, y1);

u8g\_AddPolygonXY(u8g, x2, y2);

u8g\_DrawPolygon(u8g);

}