

An introduction to SDL

Simon Kågström

Department of Systems and Software Engineering
Blekinge Institute of Technology
Ronneby, Sweden

<http://www.ipd.bth.se/ska/>



- SDL (**S**imple **D**irect media **L**ayer)



- A C-based, cross platform low-level media library
- Available for many platforms
 - Linux, Windows, BSD UNIX, Mac OS-X etc., etc.
- Has bindings to many different languages
 - C, C++ natively (C)
 - Python, Erlang (!) etc., etc. also available

Some SDL history



- SDL was created by Sam Lantinga for LOKI games (now defunct)
- The idea was to create a high-performance library which allowed easy porting between platforms, without sacrificing low-level access and performance
- LOKI ported a few popular windows games to Linux, but went bankrupt in 2001
- SDL is available under the GNU LGPL and is still actively developed

What should we learn today?

- You already know Java
- This will be an introduction to the SDL library
 - Illustrating common operations and popular add-on libraries
 - We will introduce **SDL**, **SDL_image**, **SDL_ttf** and **SDL_mixer**.
 - Include `SDL_image.h` etc. for the libraries
- The C++ programming project will be presented, as well as some hints regarding that

SDL portability

- SDL operates at a low level, which increases portability
- If you use SDL and related libraries exclusively, your game can run almost anywhere



- Be careful with endianness issues (for instance in binary file formats)
- SDL specifies integer types to use when the size is important
 - Unsigned: Uint32, Uint16, Uint8
 - Signed: Sint32, Sint16, Sint8
 - Sometimes: Uint64, Sint64 (only if SDL_HAS_64BIT_TYPE is defined)
 - Also SDL_Bool (either SDL_FALSE or SDL_TRUE)

Init and cleanup

Description

- Before you do *anything* with SDL, you need to initialize it
- The argument specifies what to initialize
- Before you exit, you should also cleanup after SDL by calling `SDL_quit()`
 - Call this at the end of `main`
 - or add `atexit(SDL_quit)` and it will be called automatically on program termination

Example

```
/* Initialize SDL */
if (SDL_Init(SDL_INIT_EVERYTHING) < 0) {
    fprintf(stderr, "Error initializing SDL: %s\n", SDL_GetError());
    /* Fail */
}
```

Description

- After initializing SDL, we need to create a window
- This is done with `SDL_SetVideoMode`
- The example will create a 640x480 window with 32 bit color depth
- Useful flags (last argument):
 - `SDL_HWSURFACE`/`SWSURFACE`: Use video/system memory
 - `SDL_DOUBLEBUF`: Use double buffering
 - `SDL_FULLSCREEN`: Use full screen mode
- `SDL_SetVideoMode` returns a `SDL_Surface`-pointer (NULL on failure)

Example

```
if ( !(p_screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE)) ) {  
    fprintf(stderr, "Error setting video mode: %s\n", SDL_GetError());  
    /* Fail! */  
}
```

The SDL_Surface

- The `SDL_Surface` describe graphical surfaces
 - E.g. the screen, sprites, background tiles etc.
 - This is a *very* important structure, you'll use it a lot
- A surface has a width, a height (attributes **w**, **h**), a pixel format (color depth, alpha-channel, attribute **format**) etc.
- Many SDL functions operate on `SDL_Surface`'s
 - `SDL_FillRect`: Fill a rectangle on a surface with a specified color
 - `SDL_BlitSurface`: Draw one surface on another (e.g. a sprite on the screen)
- `SDL_Surface`'s **must** be freed with `SDL_FreeSurface` after use

A complete SDL application

```
#include <SDL/SDL.h> /* SDL defs */
int main(int argc, char *argv[]) {
    SDL_Surface *p_screen;

    /* Initialize SDL */
    if (SDL_Init(SDL_INIT EVERYTHING) < 0) {
        /* Print a nice error string */
        fprintf(stderr, "Error initializing SDL: %s\n", SDL_GetError());
        return 1;
    }
    if ( !(p_screen = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE)) ) {
        fprintf(stderr, "Error setting video mode: %s\n", SDL_GetError());
        return 1;
    }

    /* Simulate MS-crash colors, flip the screen and wait 5 secs */
    SDL_FillRect(p_screen, NULL, SDL_MapRGB(p_screen->format, 0,0,255));
    SDL_Flip(p_screen);
    SDL_Delay(5*1000);

    /* Cleanup SDL */
    SDL_FreeSurface(p_screen);
    SDL_Quit();
}
```

- We can use `SDL_BlitSurface` to draw “sprites”
- How do we get a `SDL_Surface` then?
 - Load it from a file with `SDL_LoadBMP(filename)`: `SDL`
 - or with `IMG_Load(filename)`: **SDL_image**, supports PNG, JPG, XCF, LBM, XPM etc.
 - Both return a `SDL_Surface` pointer or `NULL` on failure
- `SDL_BlitSurface(SDL_Surface *src, SDL_Rect *src_rect, SDL_Surface *dst, SDL_Rect *dst_rect)`:
draw **dst** on **src** at **dst_rect** (inside **src_rect**)

```
typedef struct {  
    Sint16 x, y;  
    Uint16 w, h;  
} SDL_Rect;
```

- SDL has no built-in functionality for sprites, tilemaps, collision handling or even getting and setting pixels
- Sprites is simple: just add coordinates to a `SDL_Surface`
- Collisions: principles described in the first course
 - There is an example on Idenet, based on code by Amir Taaki
 - The example does pixel-perfect and bounding box collisions
 - Pixel-perfect collision is implemented with the alpha-channel for transparency
- Pixel operations are more complex than it seems
 - Color depth, endianness, etc., matters for the pixel address
 - Examples in the SDL documentation and in the collision example (`getpixel`)

Description

- Not implemented in SDL, but see **tilemap.hh**
- Tilemaps are fairly simple (centering around the player)
- Is there something missing in the code? World/tilemap borders?
 - How is `view_x` and `view_y` selected?

Example

```
Uint8 map[] = {0,1,2,0,1
               1,2,0,1,0};

for (y = view_y / TILE_H; y <= (view_y + screen_h) / TILE_H; y++) {
    for (x = view_x / TILE_W; x <= (view_x + screen_w) / TILE_W; x++) {
        Uint8 tile = map[y * tilemap_w + x];
        SDL_Rect where;

        where.x = x*TILE_W - view_x;
        where.y = y*TILE_H - view_y;

        SDL_BlitSurface(p_game->p_tiles[tile], NULL, p_game->p_screen, &where);
    }
}
```

- What about collisions against the tilemap?
- With small tiles (e.g., 8x8), we can check collisions against the entire tile
 - Often we use several tiles for a single structure (house, road etc)
- With large tiles, it's hard to limit every tile to one structure - what do we do?
 - A big `switch()`-statement dealing with tile types?
 - Variable-sized tiles?
 - Works, but more complex lookup, map drawing and map creation
- We can use a “mask map” in addition to the “graphical” tilemap

Graphics: tilemap collisions (more)



- With a mask map, every tile has both
 - A graphical representation (what you see on the screen)
 - A mask, which specifies the “material” of the tile
- Count pixels with pixel-perfect collision between the sprite and the background to see what we are “standing on”
 - In a car-game, do this between the car and the background (road)
 - If all pixels are red we can travel full-speed (tarmac)
 - If some pixels are purple, we slow down (grass)

- Keyboards, mice and joysticks all generate events (stored in a `SDL_Event` structure)
 - There are separate events for key presses and releases (`SDL_KEYDOWN` and `SDL_KEYUP`)
- Events are entered onto a queue in FIFO order
- `int SDL_PollEvent(SDL_Event *ev)` fills in **ev** with the first event
 - 1 is returned if there are any events, 0 otherwise
- `SDL_Event` is a union of different event types (given by the **type** member)

Input example

Description

- `event.key` is a `SDL_KeyboardEvent` structure
- `event.key.keysym.sym` is the “virtual” SDL scancode for keys (`SDLK_?` (both on presses and releases))

Example

```
SDL_Event event;

/* Handle all pending events */
while( SDL_PollEvent(&event) ) {
    switch(event.type) {
        case SDLK_KEYDOWN: /* Key pressed */
            switch (event.key.keysym.sym) {
                case SDLK_LEFT:
                    p_game->player.dx = -1;
                    break;
                case SDLK_RIGHT:
                    p_game->player.dx = 1;
                    break;
            }
            break;
        case SDLK_KEYUP: /* Key released */
            ...
        case SDLK_QUIT: /* Ctrl-C etc */
            ...
    }
}
```


Description

- For mice, there are two types of events: motion and key press/release
- You can check the current mouse state with `SDL_BUTTON`
- `SDL_QUIT` is enqueued on Ctrl-C, window close or process termination
- There are also events for window resizing, joysticks etc.

Example

```
switch(event.type) {
    case SDL_MOUSEMOTION:
        /* event.motion is a SDL_MouseMotionEvent */
        player.x = event.motion.x; /* Xpos */

        /* Change y if left button is pressed */
        if (event.motion.state & SDL_BUTTON(SDL_BUTTON_LEFT))
            player.y = event.motion.y;
    case SDL_MOUSEBUTTONDOWN:
        /* event.button is a SDL_MouseButtonEvent */
        ...
}
```

Description

- There is an alternative interface to keyboard input, based on a vector of the current key state
- The vector is read-only and should not be free:d

Example

```
SDL_PumpEvents();  
Uint8 *keystate = SDL_GetKeyState(NULL);  
  
if (keystate[SDLK_DOWN])  
    ; /* Handle a pressed DOWN key*/  
...
```

The SDL game loop

- The game structure is similar in most game engines
- You construct your game around a loop, performing
 - Check input (`SDL_PumpEvents()`)
 - Update the world
 - Draw sprites etc. (`SDL_BlitSurface(...)`)
 - Flip the back buffer (`SDL_Flip(p_screen)`)
 - Delay for a while (`SDL_Delay(...)`)
- Basically the same structure as for other comparable libraries

The SDL game loop (2)

Example

```
while(1) {
    SDL_Event event;
    Uint32 before = SDL_GetTicks();

    /* Poll events */
    while( SDL_PollEvent(&event) ) {
        switch(event.type) {
            case SDL_KEYDOWN:
                handle_key_down_event(&game, &event.key); break;
            case SDL_KEYUP:
                handle_key_up_event(&game, &event.key); break;
            ...
            default:
                break;
        }
    }
    move_player( &game );
    for (i=0; i<game.n_monsters; i++)
        move_monster( &game, &game.monster[i]);

    draw_game( &game );
    SDL_Flip(game.p_screen);

    if (SDL_GetTicks() - before < MS_TO_SLEEP )
        SDL_Delay( MS_TO_SLEEP - (SDL_GetTicks() - before) );
}
```

- With **SDL_ttf** you can print text with truetype fonts
- Text is rendered on a `SDL_Surface` which can be blitted to the screen
 - If the same text is printed repeatedly on the screen this can increase performance by allowing reuse of rendered text
- You can render text both “raw” and blended (using an alpha channel)
 - The difference can be quite large:



Tjoho! ÅÄÖ
Second string 0123456789

- **SDL_ttf** must first be initialized with `TTF_Init()`
- Load a font with `TTF_OpenFont(char *name, int pts)` (returns a `TTF_Font` pointer)
- `TTF_RenderText_?(TTF_Font*, char *, SDL_Color)` generates a `SDL_Surface` with some text `char*` and a specified color
 - ? == Solid will print the font without alpha-blending
 - ? == Blended uses alpha-blending (slower but better results)
- Free the font memory with `TTF_FreeFont(TTF_Font *)`
- Finally, cleanup with `TTF_Quit()`

- There is some built-in sound functionality in SDL, but **SDL_mixer** provides much better support
- **SDL_mixer** allows you to mix music and sound samples freely
- It supports many formats
 - Music: mod, s3m, it, xm, mp3, Ogg Vorbis, MIDI
 - Samples: wav, voc
- You can apply effects to the sounds
 - Distance from listener
 - 3D effects (the direction the sound comes from)
 - Fading music in/out
- Plus all the standard play/pause etc.

- Start by initializing **SDL_mixer**: `Mix_OpenAudio(freq, format, mono/stereo, sample_size)` (returns `0` on error)
- Allocate the channels to mix with `Mix_AllocateChannels(int n)` (the number of concurrent sounds possible)
- Music is loaded with `Mix_Music *Mix_LoadMUS(char *name)` (returns a pointer to the `Mix_Music` structure or `NULL` on error)
- Play the music with `Mix_PlayMusic(Mix_Music *, int loops)` (pass `-1` for infinite looping)
- Sound samples are represented by the `Mix_Chunk` structure and loaded with `Mix_Chunk *Mix_LoadWAV(char *name)`
- The sample is played with `Mix_PlayChannel(int ch, Mix_Chunk *sample, int loops)` (pass `-1` as **channel** to use a unused channel)
- Issue `Mix_CloseAudio()` before quitting the program

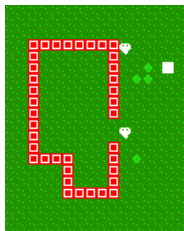
- The SDL homepage: **www.libsdl.org**
 - Documentation (doc project)
 - API documentation, tutorials etc.
 - Links to apps/games/libraries built with SDL (lots!)
- SDL_mixer: **www.libsdl.org/projects/SDL_mixer/**
- SDL_image: **www.libsdl.org/projects/SDL_image/**
- SDL_ttf: **www.libsdl.org/projects/SDL_ttf/**
- Examples on idenet: **idenet.bth.se**, search for **dvb021**
 - Most topics from this lecture are covered in the examples
- Man-pages (if applicable)

- You'll find the project specifications on idenet
- There is a suggested proposal (car game), but if you want to you can send in your own proposals
 - You need to describe your project in two pages then, what it does, what you will implement, etc.
- You can implement the project one by one or in groups of two
 - For groups of two, we expect a bit more (see the lab descriptions)
- The project is to be implemented in C++
 - A good object-oriented design is required
- This is a 2-credit project, i.e. we expect you to work 2 weeks *full time* on this

- Deadline to *send in* the project is **13/1**
 - The project should be sent with source code and a description to Senadin (senadin.alisic@bth.se)
 - You will present the project for Senadin *individually*
- The suggested car-game project requires some AI (computer-controlled cars)
- One way of controlling the cars is to use a FSM, which we will look at next

Situation

- We have a forest castle, patrolled by guards (NPCs)
 - The guards walk between waypoints
 - When a waypoint is reached, they aim for the next
- Our hero/heroine, *the Lorminator*, wants to get inside the castle.
- Guards will chase and capture any intruder
 - But will give up when too far from the intruder
- Guards cannot see through walls or bushes



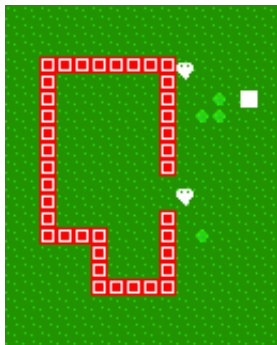
- The guards should behave “intelligently” (everything is relative)
 - I.e., avoid obstacles etc.
- But *not* too “intelligently”
 - If the player is out of sight, they should not know where he/she is
 - Perfect paths are not realistic
- All this should be implemented at interactive frame rates
 - On a 8-bit device (this was for Mophun)
 - ... with 60KB of free space
 - ... running a virtual machine
 - ... and with clean and good code

- How do we model the guard?
 - AI to the rescue!
- Finite state machines
 - Pages 155-169 of “Core Techniques and Algorithms in Game Programming”
 - We’ll introduce it here
 - Refer to courses in formal languages and automata for a complete understanding of the subject
 - (<http://www.statemachine.com/> ???)

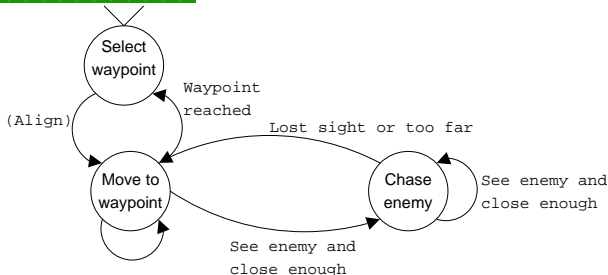
States for the guards

- The guards can be modeled using states
- The guards have three (or four) states:
 - Select a waypoint to walk to
 - Walk to the waypoint
 - Chase the player
 - (Optional) hug, tag, fight or *Lorminate* the player
- A guard will be in exactly one of these, but can change between them
- Easily modeled using an FSM!

Graphical representation



- FSMs are easily illustrated with a state graph
- Nodes are states
- Edges are transitions between states
 - Non-determinism can be added by allowing multiple transitions from a state (perhaps with probabilities)
- For the guards, it will look something like the following



How do we code the FSMs?

- The FSMs can be coded in a more or less automatic manner
- Each state is implemented as follows:

```
void guard_fsm(game_t *p_game, guard_t *p_guard) {  
    switch(p_guard->state) {  
        case STATE_NAME:  
            [Default actions]  
            if [Transition]  
                state=[New state]  
            break;  
        ...  
    }  
}
```

- The FSM is called regularly (e.g. in the game loop)

```
while(1) {  
    /* Handle input etc */  
    for (i=0; i<p_game->n_guards; i++)  
        guard_fsm(p_game, &p_game->guard[i]);  
    /* Draw everything etc. */  
}
```

Implementation example

- There is an example on the course homepage (`fsm.zip`)
- Tile-based world, sprites only moving horizontally and vertically
- The FSM implementation for guards is around 40 lines of code, not counting utility functions
- Utilities:
 - **`in_line_of_sight`**: Are there any walls on a line between two points?
 - **`sprite_move_to_dest`**: Move towards a destination
- Fairly simple and clean code

Implementation example, II

```
/* The states of a guard */
typedef enum
{
    SELECT_WAYPOINT,
    MOVE_TO_WAYPOINT,
    CHASE_ENEMY
} guard_state_t;

static void guard_fsm(game_t *p_game,
                     guard_t *p_guard)
{
    switch (p_guard->state) {
        default:
            /* Error ... */
            break;
        case SELECT_WAYPOINT: {
            p_guard->cur_waypoint++;
            if (p_guard->cur_waypoint >=
                p_guard->n_waypoints)
                p_guard->cur_waypoint = 0;
            p_guard->state = MOVE_TO_WAYPOINT;
            break;
        }
        case MOVE_TO_WAYPOINT: {
            waypoint_t *p_cur = &p_guard->p_waypoints[p_guard->cur_waypoint];

            /* Prioritize horizontal movement */
            sprite_move_to_dest(&p_guard->sprite, p_cur->x, p_cur->y);

            /* Close enough to an enemy? */
            if (in_line_of_sight(40, p_guard->sprite.x, p_guard->sprite.y,
                                p_game->player.x, p_game->player.y))
                p_guard->state = CHASE_ENEMY; /* Yes! Switch state */
            else if (p_guard->sprite.x == p_cur->x &&
                     p_guard->sprite.y == p_cur->y)
                p_guard->state = SELECT_WAYPOINT; /* Yes! Switch state */
            break;
        }
        case CHASE_ENEMY: {
            sprite_t *p_enemy = &p_game->player;

            /* Move towards the player */
            sprite_move_to_dest(&p_guard->sprite, p_enemy->x, p_enemy->y);

            if (!in_line_of_sight(40, p_guard->sprite.x, p_guard->sprite.y,
                                p_game->player.x, p_game->player.y))
                p_guard->state = MOVE_TO_WAYPOINT;
            break;
        }
    }
}
```

Questions?