SDL

**S**imple **D**irect-media **L**ayer is a cross platform development library designed to provide low level access to audio, keyboard, mouse, joystick and graphics hardware via openGL + direct3D.

INSTALL & SETUP

* download from libsdl.org and unzip
* in Visual Studio project settings, make sure you add the include folder in C++ > general > additional include directories
* add the lib/x86 in linker > general > additional library directories
* add SDL2.lib and SDL2main.lib in linker > input > additional dependencies
* add SDL2.dll in to the root project directory

CREATE A WINDOW

#include “SDL.h”

Int main() {

SDL\_Init(SDL\_INIT\_EVERYTHING);

SDL\_Window\* window = SDL\_CreateWindow(“title”, SDL\_WINDOWPOS\_CENTERED, SDL\_WINDOWPOS\_CENTERED, 600, 400, SDL\_WINDOW\_SHOW);

SDL\_Renderer\* renderer = SDL\_CreateRenderer(window, -1, 0);

SDL\_SetRenderDrawColor(renderer, 0, 255, 0, 255);

SDL\_RenderClear(renderer);

SDL\_RenderPresent(renderer);

SDL\_Delay(3000);

Return 0;

}

GAME LOOP

Class Game {

Private:

Bool isRunning;

SDL\_Window\* window;

SDL\_Renderer\* renderer;

Public:

Game() {}

~Game() {}

Void init(const std::string title, int xPos, int yPos, int w, int h, bool fullscreen) {

Int flags = 0;

If (fullscreen) {

Flags = SDL\_WINDOW\_FULLSCREEN;

}

If (SDL\_Init(SDL\_INIT\_EVERYTHING) == 0) {

Window = SDL\_CreateWindow(title, xPos, yPos, w, h, flags);

Renderer = SDL\_CreateRenderer(window, -1, 0);

If (renderer) {

SDL\_SetRenderDrawColor(renderer, 255, 255, 255, 255);

}

isRunning = true;

} else {

isRunning = false;

}

}

Void handleEvents() {

SDL\_Event event;

SDL\_PollEvent(event);

Switch(event.type) {

Case SDL\_QUIT:

isRunning = false;

break;

default:

break;

}

}

Void update() {}

Void render() {

SDL\_RenderClear(renderer);

// put stuff here to render

SDL\_RenderPresent(renderer);

}

Void clean() {

SDL\_DestroyWindow(window);

SDL\_DestroyRenderer(renderer);

SDL\_Quit();

}

Bool isRunning() { return isRunning; }

}

Int main() {

Game = new Game();

Game->init(“Rezaa engine”, SDL\_WINDOWPOS\_CENTERED, SDL\_WINDOWPOS\_CENTERED, 800, 600, false);

// game loop

While (game->isRunning()) {

Game->handleEvents();

Game->update();

Game->render();

}

Game->clean();

Return 0;

}

SDL API SO FAR…

|  |  |
| --- | --- |
| **Member** | **Description** |
| SDL\_Init(*uint32*) | Used to initialize specific SDL subsystems. It can be used interchangeably with SDL\_InitSubSystem();  Returns 0 on success and negative on error  Constants accepted as arguments:  SDL\_INIT\_AUDIO  SDL\_INIT\_VIDEO  SDL\_INIT\_EVENTS  SDL\_INIT\_EVERYTHING  …  Chain these arguments with OR (|) e.g. SDL\_INIT\_AUDIO | SDL\_INIT\_VIDEO |
| SDL\_GetError() | More info on error/s |
| SDL\_RenderPresent() | Use this to update the screen with any rendering performed since the previous call |
| SDL\_Event | Structure to be filled with the next event from the queue, or null. E.g. move events, window events, input events, … |
| SDL\_PollEvent() | Used to poll for currently pending events, e.g. window close |
| SDL\_CreateTextureFromSurface(  *Renderer,*  *SDL\_Surface*  ) |  |
| SDL\_Surface | A structure that contains a collection of pixels |
| SDL\_FreeSurface() | Free an rgb surface |
| SDL\_RenderCopy() | Copy a portion of the texture to the current rendering target  Args:  Renderer\*  SDL\_Texture\*  SDL\_Rect\* srcRect  SDL\_Rect\* destRect |
| SDL\_Rect | A structure that contains the definition of a rectangle, with the origin at the upper left  h = height of rectangle  w = width of rectangle  y = y location of upper left corner  x = x location of upper left corner |
| SDL\_Texture | A structure that contains an efficient, driver specific representation of pixel data |

DRAWING TEXTURES / SPRITES TO THE SCREEN

* You will need to download ‘SDL image’ which is an image file loading library
* Copy and paste zlib.dll, libpng16-16.dll and SDL\_image.dll in to the project root directory
* Update VS project settings to include these libraries
* You will also need to create some pixel art (using aseprite) as your texture, save as png
* Store these images in an assets folder within the root directory

SDL\_Rect srcR, destR; // this will be updated in an update function (see next para)

SDL\_Surface\* tmpSurface = IMG\_LOAD(“path/to/img”);

SDL\_Texture\* playerTexture = SDL\_CreateTextureFromSurface(renderer, tmpSurface);

SDL\_FreeSurface(tmpSurface);

SDL\_RenderCopy(renderer, playerTexture, null, &destR);

Static int x = 0; // when this increases, the destR.x will increase and the texture will move across the screen

destR.h = 64; // pixels size – if created as 32\*32, then this is 2\*

destR.w = 64;

destR.x = x;

All textures should essentially be managed by its own class

HOW TO LIMIT FRAME RATE

Int main() {

// include the game var from the game loop above

Const int fps = 60;

Const int frameDelay = 1000 / fps; // ms

Uint32 frameStart; // massive unsigned integer

Int frameTime;

While (game->isRunning()) {

frameStart = SDL\_GetTicks(); // no. of ms since sdl init

game->handleEvents();

game->update();

game->render();

frameTime = SDL\_GetTicks() – frameStart;

// if frame updated too quick, then delay before the next frame

if (frameDelay > frameTime) {

SDL\_Delay(frameDelay - frameTime); // wait no. of ms

}

}

}

TEXTURE MANAGER

Class TextureManager {

Public:

Static SDL\_Texture\* loadTexture(const char\* filename) {

SDL\_Surface\* tmpSurface = IMG\_Load(filename); // sdl image lib needed

SDL\_Texture\* texture = SDL\_CreateTextureFromSurface(Game::renderer tmpSurface);

SDL\_FreeSurface(tmpSurface);

Return texture;

}

Static void draw(SDL\_Texture\* texture, SDL\_Rect src, SDL\_Rect dest) {

SDL\_RenderCopy(Game::renderer, texture, &src, &dest);

// where Game::renderer is a static public method on the Game class so multiple renderer instances aren’t created

}

}

GAME OBJECT

\*refactor some of the code above in order to incorporate this\*

Class GameObject {

Public:

GameObject(const char\* textureSheet, SDL\_Renderer\* renderer, int x, int y)

: renderer(renderer), xPos(x), yPos(y)

{

objectTexture = TextureManager::loadTexture(textureSheet, renderer);

}

Void update() {

yPos++;

xPos++;

srcRect.h = 32;

srcRect.w = 32;

srcRect.x = 0;

srcRect.y = 0;

destRect.h = srcRect.h \* 2;

destRect.w = srcRect.w \* 2;

destRect.x = xPos;

destRect.y = yPos;

}

Void render() {

SDL\_RenderCopy(renderer, objectTexture, &srcRect, &destRect);

}

Private:

Int xPos;

Int yPos;

SDL\_Texture\* objectTexture;

SDL\_Rect srcRect, destRect;

SDL\_Renderer \* renderer;

}

Player = new GameObject(“assets/player.png”, renderer, 0, 0);

Player->update();

Player->render();

IMPLEMENTING A TILE MAP

In main function, change the resolution to 800 \* 600, so our 32 bit individual tile maps fit exact to our window. (800/32 = 25, 600/32 = 20 – our tile map will be 25 \* 20).

Map.cpp

Int lvl1[20][25] = {}; // populate this with 0 (water), 1 (grass) & 2 (dirt)

Class Map {

Public:

Map() {

Dirt = TextureManager::loadTexture(“assets/dirt.png”);

Grass = TextureManager::loadTexture(“assets/grass.png”);

Water = TextureManager::loadTexture(“assets/water.png”);

loadMap(lvl1);

src.x = src.y = 0;

src.w = dest.w = 32;

src.h = dest.h = 32;

dest.x = dest.y = 0;

}

Void loadMap() {

Int type = 0;

For (int row = 0; row < 20; row++) {

For (int col = 0; col < 25; col++) {

Type = map[row][col];

Dest.x = col \* 32;

Dest.y = row\*32;

Switch (type) {

Case 0:

TextureManager::draw(water, src, dest);

Break;

Case 1: // grass

Case 2: // dirt

}

}

}

}

}

When the game renders, you can now draw the map: *map->drawMap();*

INPUT HANDLING

SDL\_Event event;

SDL\_PollEvent(&event);

If (event.type == SDL\_KEYDOWN) { // code here }

SDL\_Event is a union that contains structures for the different even types, e.g.; SDL\_WindowEvent, SDL\_KeyboardEvent, SDL\_MouseButtonEvent, SDL\_QuitEvent, etc…

Some common macros include:

* SDL\_MOUSEBUTTONDOWN
* SDL\_KEYDOWN
* SDL\_MOUSEWHEEL
* SDL\_QUIT
* Etc…

SDL\_KeyboardEvent

SDL\_KeyboardEvent is a struct (which is within SDL\_Event), whose members include:

* Type – e.g. SDL\_KEYDOWN, SDL\_KEYUP
* Keysym – SDL\_KeySym representing the key that was pressed or released

SDL\_KeySym

A struct within SDL\_KeyboardEvent (above), containing further nested structs:

* Scancode – SDL\_ScanCode // physical key code
* Sym – SDL\_KeyCode // virtual key code

Putting it together

If (event.type == SDL\_KEYDOWN) {

If (event.key.keysym.sym == SDLK\_w) { // SDLK\_w is a macro for ‘w’ keycode

Std::cout << “w key pressed”;

}

}

COLLISION (AABB)

Axis-Aligned Bounding Box (AABB) is an algorithm to detect collision in 2D games. It is one of the simpler forms of collision detection which detects collision between 2 rectangles which are axis-aligned (i.e. no rotation). The algorithm works by ensuring there is no gap between any of the 4 sides of the rectangles. Any gap means a collision does not exist.

w

A

h

B

We need to check:

1. A.x + A.w >= B.x (top right corner of A, overlaps B’s top left corner)
2. B.x + B.w >= A.x (top right corner of B, goes beyond A’s top left corner, without this condition, A could be on the right side of B, so this checks that is not the case)
3. A.y + A.h >= B.y (bottom left corner of A is below B’s top left corner on y axis)
4. B.y + B.h >= A.y (bottom left corner of B is below A’s top left corner, without this condition, A could be below and clear of B on the y axis)

Class Collision {

Public:

Bool AABB(const SDL\_Rect& rectA, SDL\_Rect& rectB) {

If (rectA.x + rectA.w >= rectB.x &&

rectB.x + rectB.w >= rectA.x &&

rectA.y + rectA.h >= rectB.y &&

rectB.y + rectB.h >= rectA.y

) {

return true;

}

Return false;

}

}

We can now compare 2 rectangles to if a collision has occurred.