# Lesson 08 – Timing: Frame Rate, Physics, Animation

Home

Timing | Performance Counters | Frame Rate | Capping Frame Rate | VSync | Physics | Animation

## Timing

SDL Provides a simple but convenient API for timing. Timing has many applications, including FPS calculation and capping, profiling what parts of your program take the most time, and any simulations that should be based on time, such as physics and animation.

The most basic form of timing is SDL_GetTicks(). This function simply returns the number of ticks that have elapsed since SDL was initialized. One tick is one millisecond, a tolerable resolution for physics simulation and animation.

```
Uint32 ticks =
```

## Example Program

Download | Download Timing Class | Download Animation Example

```
#include <iostream>
#include <string>
#include <vector>
#include <time.h>

#include <SDL.h>
#include <SDL_image.h>
#include <SDL_ttf.h>

using namespace std;

// Gravity in pixels per second squared
const float GRAVITY = 750.0f;

bool init();
void kill();
void loop();
void renderText(string text
```

SDL_GetTicks()

Ticks are always increasing—SDL doesn't provide a way to time between intervals, pause the global timer, or anything like that. However, all these features are relatively straightforward to implement yourself. For example, you could create a timing class that manages separate and pause-able timers. All you really need is what SDL gives you; the global tick timer.

For example, to time an interval in ticks, simply request the time at the start and end...

```
Uint32 start =
SDL_GetTicks();

// Do long operation

Uint32 end =
SDL_GetTicks();

float secondsElapsed =
(end - start) / 1000.0f;
```

# Performance Counters

While SDL_GetTicks() is good enough for most purposes, the minimum interval it can time is one millisecond. But what if you want to time sub-millisecond operations, or want more precision than simply a thousandth of a second? This is where SDL_GetPerformanceCounter() comes in. The performance

```
void renderText(string text,
SDL_Rect dest);

SDL_Window* window;
SDL_Renderer* renderer;
SDL_Texture* box;
TTF_Font* font;

struct square {
	float x, y, w, h, xvelocity,
yvelocity;
	Uint32 born, lastUpdate;
};

int main(int argc, char** args) {

	if ( !init() ) {
		system("pause");
		return 1;
	}

	loop();

	kill();
	return 0;
}

void loop() {

	srand(time(NULL));

	// Physics squares
	vector<square> squares;

	bool running = true;
	Uint32 totalFrameTicks = 0;
	Uint32 totalFrames = 0;
	while(running) {

		// Start frame timing
		totalFrames++;
		Uint32 startTicks =
SDL_GetTicks();
		Uint64 startPerf =
SDL_GetPerformanceCounter();

		SDL_Event e;


SDL_SetRenderDrawColor(
renderer, 255, 255, 255, 255 );
```

counter is a system-specific high-resolution timer, usually on the scale of micro- or nano-seconds.

Because the performance counter is system specific, you don't actually know what the resolution is. Hence, the function

```
            SDL_RenderClear(
renderer );

            // Event loop
            while ( SDL_PollEvent( &e
) != 0 ) {
                  switch (e.type) {
                        case SDL_QUIT:
                              running =
false;
                              break;
                        case
SDL_MOUSEBUTTONDOWN:
                              square s;
                              s.x =
e.button.x;
                              s.y =
e.button.y;
                              s.w = rand()
% 50 + 25;
                              s.h = rand()
% 50 + 25;
                              s.yvelocity =
-500;
                              s.xvelocity =
rand() % 500 - 250;

s.lastUpdate = SDL_GetTicks();
                              s.born =
SDL_GetTicks();

squares.push_back(s);
                              break;
                  }
            }

            // Physics loop
            for (int index = 0; index <
squares.size(); index++) {
                  square& s =
squares[index];

                  Uint32 time =
SDL_GetTicks();
                  float dT = (time -
s.lastUpdate) / 1000.0f;

                  s.yvelocity += dT *
GRAVITY;
                  s.y += s.yvelocity *
dT;
```

```
                    s.x += s.xvelocity *
dT;

                    if (s.y > 480 - s.h) {
                        s.y = 480 - s.h;
                        s.xvelocity = 0;
                        s.yvelocity = 0;
                    }

                    s.lastUpdate = time;
                    if (s.lastUpdate >
s.born + 5000) {

squares.erase(squares.begin() +
index);
                        index--;
                    }
                }

                // Render loop
                for (const square& s :
squares) {
                    SDL_Rect dest = {
round(s.x), round(s.y), round(s.w),
round(s.h) };

SDL_RenderCopy(renderer, box,
NULL, &dest);
                }

                // Delay for a random
number of ticks - this makes the
frame rate variable,
                // demonstrating that the
physics is independent of the frame
rate.
                SDL_Delay(rand() % 25);

                // End frame timing
                Uint32 endTicks =
SDL_GetTicks();
                Uint64 endPerf =
SDL_GetPerformanceCounter();
                Uint64 framePerf =
endPerf - startPerf;
                float frameTime =
(endTicks - startTicks) / 1000.0f;
                totalFrameTicks +=
endTicks - startTicks;

                // Strings to display
```

```
            string fps = "Current FPS:
" + to_string(1.0f / frameTime);
            string avg = "Average
FPS: " + to_string(1000.0f /
((float)totalFrameTicks /
totalFrames));
            string perf = "Current Perf:
" + to_string(framePerf);

            // Display strings
            SDL_Rect dest = { 10, 10,
0, 0 };
            renderText(fps, dest);
            dest.y += 24;
            renderText(avg, dest);
            dest.y += 24;
            renderText(perf, dest);

            // Display window

SDL_RenderPresent(renderer);
        }
}

void renderText(string text,
SDL_Rect dest) {
        SDL_Color fg = { 0, 0, 0 };
        SDL_Surface* surf =
TTF_RenderText_Solid(font,
text.c_str(), fg);

        dest.w = surf->w;
        dest.h = surf->h;

        SDL_Texture* tex =
SDL_CreateTextureFromSurface(renderer,
 surf);

        SDL_RenderCopy(renderer, tex,
NULL, &dest);
        SDL_DestroyTexture(tex);
        SDL_FreeSurface(surf);
}

bool init() {
        if ( SDL_Init(
SDL_INIT_EVERYTHING ) < 0 ) {
            cout << "Error initializing
SDL: " << SDL_GetError() << endl;
            return false;
        }
```

```
        if ( IMG_Init(IMG_INIT_JPG) < 0 ) {
                cout << "Error initializing SDL_image: " << IMG_GetError() << endl;
                return false;
        }

        if ( TTF_Init() < 0 ) {
                cout << "Error initializing SDL_ttf: " << TTF_GetError() << endl;
                return false;
        }

        window = SDL_CreateWindow( "Example", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, 640, 480, SDL_WINDOW_SHOWN);
        if ( !window ) {
                cout << "Error creating window: " << SDL_GetError()  << endl;
                return false;
        }

        renderer = SDL_CreateRenderer( window, -1, SDL_RENDERER_ACCELERATED );
        if ( !renderer ) {
                cout << "Error creating renderer: " << SDL_GetError() << endl;
                return false;
        }

        SDL_Surface* buffer = IMG_Load("box.jpg");
        if ( !buffer ) {
                cout << "Error loading image box.jpg: " << SDL_GetError() << endl;
                return false;
        }

        box = SDL_CreateTextureFromSurface( renderer, buffer );
```

```
        SDL_FreeSurface( buffer );
        buffer = NULL;
        if ( !box ) {
                cout << "Error creating
texture: " << SDL_GetError() << endl;
                return false;
        }

        font = TTF_OpenFont("font.ttf",
24);
        if ( !font ) {
                cout << "Error loading font:
" << TTF_GetError() << endl;
                return false;
        }

        return true;
}

void kill() {
        TTF_CloseFont( font );
        SDL_DestroyTexture( box );
        font = NULL;
        box = NULL;

        SDL_DestroyRenderer(
renderer );
        SDL_DestroyWindow( window );
        window = NULL;
        renderer = NULL;

        TTF_Quit();
        IMG_Quit();
        SDL_Quit();
}
```

SDL_GetPerformanceFrequency(): it gives you the number of performance counter ticks per second.

Otherwise, this system is used in exactly the same way as ticks. To time an interval more precisely, capture the starting and ending performance counter values.

```
Uint64 start = SDL_GetPerformanceCounter();

// Do some operation

Uint64 end = SDL_GetPerformanceCounter();
```

```
float secondsElapsed = (end - start) /
(float)SDL_GetPerformanceFrequency();
```

# Frame Rate

A common application of timing is to calculate the FPS, or frames per second, your program is running at. A frame is simply one iteration of your main game or program loop. Hence, timing it is quite straightforward: log the time at the start and end of each frame. Then, in some form output the elapsed time or its inverse (the FPS).

```
bool running = true;
while (running) {

	Uint64 start = SDL_GetPerformanceCounter();

	// Do event loop

	// Do physics loop

	// Do rendering loop

	Uint64 end = SDL_GetPerformanceCounter();

	float elapsed = (end - start) /
(float)SDL_GetPerformanceFrequency();
	cout << "Current FPS: " << to_string(1.0f / elapsed) << endl;

}
```

# Capping Frame Rate

Aside from performance profiling, you may want to calculate your FPS in order to cap it. Capping your FPS is useful because if you try to update the screen too many times per second, frames will start drawing on top of each other—this is screen tearing. Further, capping your FPS allows your program to not use all CPU resources given to it, freeing the user's computer to work on other tasks. Although ideally, those extra resources can be put towards improving gameplay or graphics.

Capping your FPS is quite simple: just subtract your frame time from your desired time and wait out the difference with SDL_Delay(). However, this function only takes delay in milliseconds—unfortunately, you cannot cap your FPS with very much precision. (At least with SDL—look at std::chrono for more.)

You will usually want to cap your FPS to 60, as this is by far the most common refresh rate. This means spending 16 and 2/3 milliseconds per frame. Note that you can easily change this cap.

```
bool running = true;
while (running) {

	Uint64 start = SDL_GetPerformanceCounter();

	// Do event loop

	// Do physics loop

	// Do rendering loop

	Uint64 end = SDL_GetPerformanceCounter();

	float elapsedMS = (end - start) /
(float)SDL_GetPerformanceFrequency() * 1000.0f;

	// Cap to 60 FPS
	SDL_Delay(floor(16.666f - elapsedMS));

}
```

# VSync

Another way to prevent screen tearing is with VSync, or vertical sync. This technique simply makes calls to SDL_RenderPresent() wait for the correct time interval before showing the window. Basically, it will cap your FPS for you.

To use it, you must enable VSync when creating your renderer. To do so, simply pass the flag SDL_RENDERER_PRESENTVSYNC to SDL_CreateRenderer(). Subsequent calls to SDL_RenderPresent() will wait before showing the window.

```
SDL_Renderer* renderer = SDL_CreateRenderer(window, -1,
SDL_RENDERER_PRESENTVSYNC |
SDL_RENDERER_ACCELERATED );
```

# Physics

As I've mentioned, you need timing for physics simulation. As of yet, everything you could do was based on frames. For example, your player could move 20

pixels per frame. However, this is not a good way to do things. If your frame rate changes, if it's variable, your physics "time" will run slower or faster as well. The solution is simply to simulate based on time—this way, whenever you update your physics, it will automatically use the correct time interval.

Doing this is quite simple: store the last time your physics was updated (either per entity or globally) and calculate how much you need to update to get to the current time. This is your delta time (dT) value—multiply it into calculations based on time (e.g. position += velocity * dT).

```
bool running;
Uint32 lastupdate = SDL_GetTicks();

while (running) {

	// Event loop

	// Physics loop
	Uint32 current = SDL_GetTicks();

	// Calculate dT (in seconds)

	float dT = (current - lastUpdate) / 1000.0f;
	for ( /* objects */ ) {
		object.position += object.velocity * dT;
	}

	// Set updated time
	lastUpdate = current;

	// Rendering loop

}
```

# Animation

Note: we will go over this in more detail during class.

As with physics, correct animation also relies on timing. While it's not as bad to have your sprite animation FPS depend on your global FPS, this makes animation look off, forces you to cap your FPS, and requires the same FPS for each animated object.

The solution is almost exactly the same as with physics; calculate a dT value and use it to decide what frame to draw. However, here you must track last updated times for every animated object, and must only do the update when enough

time has passed for a frame to flip. Download an example.

```
float animatedFPS = 24.0f;
bool running;

while (running) {

	// Event loop

	// Physics loop

	// Rendering loop
	Uint32 current = SDL_GetTicks();

	// Calculate dT (in seconds)

	for ( /* objects */ ) {
		float dT = (current - object.lastUpdate) / 1000.0f;

		int framesToUpdate = floor(dT / (1.0f / animatedFPS));
		if (framesToUpdate > 0) {
			object.lastFrame += framesToUpdate;
			object.lastFrame %= object.numFrames;
			object.lastUpdate = current;
		}

		render(object.frames[object.lastFrame]);
	}
}
```

---

Made by Maxwell Slater © 2015–2017 | Contact me at mslater@nevada.unr.edu |

View this project on GitHub