SDL2 Texture sometimes empty after loading multiple 8 bit surfaces

Asked 5 years, 8 months ago Modified 5 years, 8 months ago Viewed 573 times

4

I'll try to make that question as concise as possible, but don't hesitate to ask for clarification.

I'm dealing with legacy code, and I'm trying to load thousands of 8 bit images from the disk to create a texture for each.

I've tried multiple things, and I'm at the point where I'm trying to load my 8 bits images into a 32 bits surface, and then create a texture from that surface.

The problem : while loading and 8 bit image onto a 32 bit surface is working, when I try to `SDL_CreateTextureFromSurface`, I end up with a lot of textures that are completely blank (full of transparent pixels, 0x00000000).

Not all textures are wrong, thought. Each time I run the program, I get different "bad" textures. Sometimes there's more, sometimes there's less. And when I trace the program, I always end up with a correct texture (is that a timing problem?)

I know that the loading to the `SDL_Surface` is working, because I'm saving all the surfaces to the disk, and they're all correct. But I inspected the textures using *NVidia NSight Graphics*, and more than half of them are blank.

Here's the offending code :

```
int __cdecl IMG_SavePNG(SDL_Surface*, const char*);

SDL_Texture* Resource8bitToTexture32(SDL_Renderer* renderer, SDL_Color*
palette, int paletteSize, void* dataAddress, int Width, int Height)
{
  u32 uiCurrentOffset;
  u32 uiSourceLinearSize = (Width * Height);
  SDL_Color *currentColor;
  char strSurfacePath[500];

  // The texture we're creating
  SDL_Texture* newTexture = NULL;

  // Load image at specified address
  SDL_Surface* tempSurface = SDL_CreateRGBSurface(0x00, Width, Height, 32,
0x00FF0000, 0x0000FF00, 0x000000FF, 0xFF000000);

  SDL_SetSurfaceBlendMode(tempSurface, SDL_BLENDMODE_NONE);

  if(SDL_MUSTLOCK(tempSurface)
    SDL_LockSurface(tempSurface);
  for(uiCurrentOffset = 0; uiCurrentOffset < uiSourceLinearSize;
uiCurrentOffset++)
  {
    currentColor = &palette[pSourceData[uiCurrentOffset]];
    if(pSourceData[uiCurrentOffset] != PC_COLOR_TRANSPARENT)
    {
      ((u32*)tempSurface->pixels)[uiCurrentOffset] = (u32)((currentColor->a <<
24) + (currentColor->r << 16) + (currentColor->g << 8) + (currentColor->b <<
0));
    }
  }

  if(SDL_MUSTLOCK(tempSurface)
    SDL_UnlockSurface(tempSurface);

  // Create texture from surface pixels
  newTexture = SDL_CreateTextureFromSurface(renderer, tempSurface);

  // Save the surface to disk for verification only
  sprintf(strSurfacePath, "c:\\tmp\\surfaces\\%s.png", GenerateUniqueName());
  IMG_SavePNG(tempSurface, strSurfacePath);

  // Get rid of old loaded surface
  SDL_FreeSurface(tempSurface);

  return newTexture;
}
```

Note that in the original code, I'm checking for boundaries, and for NULL after the `SDL_Create*`. I'm also aware that it would be better to have a spritesheet for the textures instead of loading each texture individually.

EDIT : Here's a sample of what I'm observing in NSight if I capture a frame and use the Resources View.

The first 3186 textures are correct. Then I get 43 empty textures. Then I get 228 correct textures. Then 100 bad ones. Then 539 correct ones. Then 665 bad ones. It goes on randomly like that, and it changes each time I run my program.

Again, each time the surfaces saved by `IMG_SavePNG` are correct. This seems to indicate that something happens when I call `SDL_CreateTextureFromSurface` but at that point, I don't want to rule anything out, because it's a very weird problem, and it smells undefined behaviour all over the place. But I just can't find the problem.

c sdl-2

Share
Improve this question
Follow

edited Apr 9, 2019 at 15:32

asked Apr 9, 2019 at 13:31
MartinVeronneau
1,306 ● 7 ● 24

It sounds like you're dealing with Undefined Behavior somewhere. I would start by getting rid of the `Lock/Unlock Surface` calls; since the surface is not RLE optimized, they aren't needed. Also, I think you can lose the `memset` call to zero out the pixels. They are already zero. – Mark Benningfield Apr 9, 2019 at 14:25

Thanks for your comment! You're right, I'll remove the `SDL_memset`. The help page for SDL_LockSurface seems to indicate that we should lock before accessing the pixels, not because the surface is RLE-optimized. I'll add a check to `SDL_MUSTLOCK` before locking and unlocking. – MartinVeronneau Apr 9, 2019 at 14:53

1 See this page One thing to note about the SDL documentation is that it is pretty spotty, so you pretty much have to read the entire thing, and then read the source code. – Mark Benningfield Apr 9, 2019 at 15:11

I tested with and without the surface locking, and I'm still getting the same problem. I'll edit my question to add a sample of what kind of problem I'm observing. – MartinVeronneau Apr 9, 2019 at 15:22

By "bad", you mean the pixels are mapped to the wrong color? – Mark Benningfield Apr 9, 2019 at 18:05

1 Answer

Sorted by: Highest score (default)

4

With the help of @mark-benningfield, I was able to find the problem.

TL;DR

There's a bug (or at least, an undocumented feature) in SDL with the DX11 renderer. There's a work-around ; see at the end.

CONTEXT

I'm trying to load around 12,000 textures when my program start. I know it's not a good idea, but I was planning on using that as a stepping-stone to another more sane system.

DETAILS

What I realized while debugging that problem is that the SDL renderer for DirectX 11 does that when it creates a texture :

```
result = ID3D11Device_CreateTexture2D(rendererData->d3dDevice,
        &textureDesc,
        NULL,
        &textureData->mainTexture
        );
```

The Microsoft's ID3D11Device::CreateTexture2D method page indicates that :

> If you don't pass anything to pInitialData, the initial content of the memory for the resource is undefined. In this case, you need to write the resource content some other way before the resource is read.

If we're to believe that article :

> Default Usage The most common type of usage is default usage. To fill a default texture (one created with D3D11_USAGE_DEFAULT) you can :
>
> [...]
>
> After calling ID3D11Device::CreateTexture2D, use ID3D11DeviceContext::UpdateSubresource to fill the default texture with data from a pointer provided by the application.

So it looks like that `D3D11_CreateTexture` is using the second method of the default usage to initialize a texture and its content.

But right after that, in the SDL, we call `SDL_UpdateTexture` (without checking the return value ; I'll get to that later). If we dig until we get the the D3D11 renderer, we get that :

```
static int
D3D11_UpdateTextureInternal(D3D11_RenderData *rendererData, ID3D11Texture2D
*texture, int bpp, int x, int y, int w, int h, const void *pixels, int pitch)
{
    ID3D11Texture2D *stagingTexture;
[...]
    /* Create a 'staging' texture, which will be used to write to a portion of
the main texture. */
    ID3D11Texture2D_GetDesc(texture, &stagingTextureDesc);
[...]
    result = ID3D11Device_CreateTexture2D(rendererData->d3dDevice,
&stagingTextureDesc, NULL, &stagingTexture);
[...]
    /* Get a write-only pointer to data in the staging texture: */
    result = ID3D11DeviceContext_Map(rendererData->d3dContext, (ID3D11Resource
*)stagingTexture, 0, D3D11_MAP_WRITE, 0, &textureMemory);
[...]
    /* Commit the pixel buffer's changes back to the staging texture: */
    ID3D11DeviceContext_Unmap(rendererData->d3dContext, (ID3D11Resource
*)stagingTexture, 0);

    /* Copy the staging texture's contents back to the texture: */
    ID3D11DeviceContext_CopySubresourceRegion(rendererData->d3dContext,
(ID3D11Resource *)texture, 0, x, y, 0, (ID3D11Resource *)stagingTexture, 0,
NULL);

    SAFE_RELEASE(stagingTexture);

    return 0;
}
```

Note : code snipped for conciseness.

This seems to indicate, based on that article I mentioned, that SDL is using the second method of the Default Usage to allocate the texture memory on the GPU, but uses the Staging Usage to upload the actual pixels.

I don't know that much about DX11 programming, but that mixing up of techniques got my programmer's sense tingling.

I contacted a game programmer I know and explained the problem to him. He told me the following interesting bits :

- The driver gets to decide where it's storing staging textures. It usually lies in CPU RAM.
- It's much better to specify a pInitialData pointer, as the driver can decide to upload the textures asynchronously.
- If you load too many staging textures without commiting them to the GPU, you can fill up the RAM.

I then wondered why SDL didn't return me a "out of memory" error at the time I called `SDL_CreateTextureFromSurface`, and I found out why (again, snipped for concision) :

```
SDL_Texture *
SDL_CreateTextureFromSurface(SDL_Renderer * renderer, SDL_Surface * surface)
{
[...]

    SDL_Texture *texture;

[...]

    texture = SDL_CreateTexture(renderer, format, SDL_TEXTUREACCESS_STATIC,
                                surface->w, surface->h);
    if (!texture) {
        return NULL;
    }

[...]

        if (SDL_MUSTLOCK(surface)) {
            SDL_LockSurface(surface);
            SDL_UpdateTexture(texture, NULL, surface->pixels, surface->pitch);
            SDL_UnlockSurface(surface);
        } else {
            SDL_UpdateTexture(texture, NULL, surface->pixels, surface->pitch);
        }

[...]

    return texture;
}
```

If the creation of the texture is successful, it doesn't care whether or not it succeeded in updating the textures (no check on `SDL_UpdateTexture`'s return value).

WORKAROUND

The poor-man's workaround to that problem is to call `SDL_RenderPresent` each time you call a `SDL_CreateTextureFromSurface`.

It's probably fine to do it once every hundred textures depending on your texture size. But just be aware that calling `SDL_CreateTextureFromSurface` repeatedly without updating the renderer will actually fill up the system RAM, and the SDL won't return you any error condition to check for this.

The irony of this is that had I implemented a "correct" loading loop with percentage of completion on screen, I would never had that problem. But fate had me implement this the quick-and-dirty way, as a proof of concept for a bigger system, and I got sucked into that problem.

Share Improve this answer Follow

answered Apr 11, 2019 at 20:56

MartinVeronneau
1,306 ● 7 ● 24