Lesson 05 – Textures

Creating a Texture | Texture Settings | Rendering Textures | Pixels | Render to Texture | Destroying a Texture

Example Program

Download

```
#include <iostream>
#include <SDL.h>
#include <cmath>

using namespace std;

bool init();
void kill();
bool loop();

// Pointers to our window, renderer,
and texture
SDL_Window* window;
SDL_Renderer* renderer;
SDL_Texture* texture;

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

	if ( !init() ) return 1;

	while ( loop() ) {
		// wait before processing
the next frame
		SDL_Delay(10);
	}

	kill();
	return 0;
```

Creating a Texture

As mentioned in the last lesson, textures are the GPU rendering equivalent of surfaces. Hence, textures are almost always created from surfaces, using the function

```
}

bool loop() {

        static const unsigned char*
keys = SDL_GetKeyboardState(
NULL );

        SDL_Event e;
        SDL_Rect dest;

        static int mx = -1, my = -1;
        static double rot = 0;

        // Clear the window to white
        SDL_SetRenderDrawColor(
renderer, 255, 255, 255, 255 );
        SDL_RenderClear( renderer );

        // Event loop
        while ( SDL_PollEvent( &e ) !=
0 ) {
                switch ( e.type ) {
                        case SDL_QUIT:
                                return false;
                        case
SDL_MOUSEMOTION:
                                mx = e.motion.x;
                                my = e.motion.y;
                                break;
                }
        }

        if ( mx != -1 ) {
                // Distance across window
                float wpercent = mx /
640.0f;
                float hpercent = my /
480.0f;

                // Color
                unsigned char r = round(
wpercent * 255 );
                unsigned char g = round(
hpercent * 255 );

                // Color mod (b will always
be zero)
                SDL_SetTextureColorMod(
texture, r, g, 0 );
```

```
		mx -= 320;
		my -= 240;
		rot = atan((float)my /
(float)mx) * (180.0f / 3.14f);
		if (mx < 0)
			rot -= 180;
	}
	mx = my = -1;

	// Render texture
	dest.x = 240;
	dest.y = 180;
	dest.w = 160;
	dest.h = 120;
	SDL_RenderCopyEx(renderer,
texture, NULL, &dest, rot, NULL,
keys[SDL_SCANCODE_F] ?
SDL_FLIP_VERTICAL :
SDL_FLIP_NONE);

	// Update window
	SDL_RenderPresent( renderer
);

	return true;
}

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

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

	// Load bitmap into surface
	SDL_Surface* buffer =
SDL_LoadBMP("test.bmp");
	if ( !buffer ) {
		cout << "Error loading
image test.bmp: " << SDL_GetError()
<< endl;
		return false;
	}

	// Create texture
	texture =
SDL_CreateTextureFromSurface(
renderer, buffer );
	// Free surface as it's no longer
needed
	SDL_FreeSurface( buffer );
	buffer = NULL;
	if ( !texture ) {
		cout << "Error creating
texture: " << SDL_GetError() << endl;
		return false;
	}

	return true;
}

void kill() {
	SDL_DestroyTexture( texture );
	SDL_DestroyRenderer(
renderer );
	SDL_DestroyWindow( window );
	texture = NULL;
	window = NULL;
	renderer = NULL;
	SDL_Quit();
}
```

SDL_CreateTextureFromSurface(). This function more or less does what you'd expect—the parameters are the rendering context and a surface to create the texture from. As with other creation functions, it will return NULL on failure.

When creating textures in this manner, the data will be copied to the texture, allowing you to free the surface used for loading the image.

```
// create window, renderer

SDL_Surface* image = SDL_LoadBMP("image.bmp");

SDL_Texture* texture = SDL_CreateTextureFromSurface( renderer,
image );

SDL_FreeSurface( image );
image = NULL;

if( !texture ) {
	// etc
}
```

You can also create blank textures with the function SDL_CreateTexture(). This function takes the rendering context, pixel format, width, and height of the texture to create.

```
SDL_Texture* texture = SDL_CreateTexture( renderer,
SDL_PIXELFORMAT_RGBA8888, SDL_TEXTUREACCESS_STATIC,
1024, 1024 );
```

This isn't useful for rendering images, but is necessary to edit pixel data or to render to a texture.

Texture Settings

SDL provides several texture properties that affect how they are rendered. These settings mirror the rendering options from the last lesson.

Query

The function SDL_QueryTexture() is used to retrieve the basic settings of a texture, including the format, access, width, and height.

```
int w, h, access;
unsigned int format;

SDL_QueryTexture( texture, &format, &access, &width, &height );
```

Alpha

Textures may include alpha data, but SDL also provides a whole-texture alpha setting. Changing this value will simply cause the entire texture to fade in or out. The functions SDL_GetTextureAlphaMod() and SDL_SetTextureAlphaMod() are used for this purpose.

```
// decrease alpha by 25

unsigned char alpha;

SDL_GetTextureAlphaMod( texture, &alpha );

SDL_SetTextureAlphaMod( texture, alpha - 25 );
```

Blend Mode

The blend mode can modified per texture with the functions SDL_GetTextureBlendMode() and SDL_SetTextureBlendMode(). These functions operate using the SDL_BlendMode structure, which can include no blending, alpha blending (default), additive blending, and modulated blending.

```
SDL_SetTextureBlendMode( texture, SDL_BLENDMODE_ADD );
```

Color Mod

Again as with drawing, a color modifier can be set per texture. As you'd expect, the functions SDL_GetTextureColorMod() and SDL_SetTextureColorMod() are used for this purpose. Note that the color modifier does not include alpha—you have to get and set alpha separately.

```
// swizzle colors

unsigned char r, g, b;

SDL_GetTextureColorMod( texture, &r, &g, &b );

SDL_SetTextureColorMod( texture, g, b, r );
```

Rendering Textures

Actually drawing textures to the screen is very similar to blitting surfaces, except that you have a few more options. SDL_RenderCopy() is the direct parallel: it takes the rendering context, texture, source rectangle, and destination rectangle.

```
SDL_Rect dst;
dst.x = 0;
dst.y = 0;
dst.w = 100;
dst.h = 100;

SDL_RenderCopy( renderer, texture, NULL, &dst );
```

Rotation and Flips

The function SDL_RenderCopyEx() (Ex = Extended) provides a few more rendering options. These include a rotation around a point and a vertical/horizontal flip. These are simply passed as parameters. Note that the rotation angle is in degrees, but most math functions/libraries use radians.

```
SDL_Rect dst;
dst.x = 0;
dst.y = 0;
dst.w = 100;
dst.h = 100;

SDL_Point rotPoint;
rotPoint.x = 25;
rotPoint.y = 25;

SDL_RenderCopyEx( renderer, texture, NULL, &dst, 90, &rotPoint,
SDL_FLIP_HORIZONTAL );
```

Pixels

As with surfaces, you can directly edit the pixel data of textures. However, it's rather more complicated than simply accessing the "pixels" member.

Creating a texture from a surface uses the default SDL_TextureAccess value SDL_TEXTUREACCESS_STATIC, meaning the texture cannot be changed. If you plan to directly edit the pixel data of a texture, you must use SDL_TEXTUREACCESS_STREAMING instead. This flag tells SDL that the texture may be changed by external code.

However, that's not all you have to do. To access the pixel data of a texture, you must use the function SDL_LockTexture(). This function tells SDL to stop any other access to the texture until the user is done with it. The parameters include the region of the texture to edit, a pointer that will be set to the pixel data, and an integer that will be set to the pixel pitch. The "pitch" is simply the length (in bytes) of one horizontal line of pixels—the size of the first dimension, you could say. Note that the pixel data is for **write only**—it is not guaranteed to contain the previous correct values.

The pixel format specified at texture creation dictates the format of the pixel data. It's pretty straightforward: ...RGBA8888 means first red, then green, then blue, then alpha, and each value is eight bits (one byte). ...RGB444 means first red, then green, then blue, no alpha, and four bits per value.

```
SDL_Texture* texture = SDL_CreateTexture( renderer,
SDL_PIXELFORMAT_RGBA8888,
SDL_TEXTUREACCESS_STREAMING, 1024, 1024);

unsigned char* pixels;
int pitch;

SDL_LockTexture( texture, NULL, (void**)&pixels, &pitch );

// set pixels to solid white
for(int i = 0; i < pitch * textureHeight; i++) {
	pixels[i] = 255;
}
```

Finally, once you have finished writing to the texture, you must call SDL_UnlockTexture() to apply your changes and signal SDL that it can use the texture again.

```
SDL_UnlockTexture( texture );
```

Render to Texture

While *technically*, you can do any texture modification you'd like by editing pixel data, it is often much more convenient to render directly to a texture. Essentially, this means using the full rendering API, but instead of drawing to the screen, drawing to a texture.

A texture must be created with the access SDL_TEXTUREACCESS_TARGET to enable rendering to texture. Other than this, the texture is used the same as normal. The function SDL_SetRenderTarget() is used to change between rendering to textures or to the screen. The parameters are quite simple: the rendering context and either a texture to render to or NULL to render to the screen.

Note that you should not call SDL_RenderPresent() when rendering to a texture.

```
SDL_Texture* texture = SDL_CreateTexture( renderer,
SDL_PIXELFORMAT_RGBA8888,
SDL_TEXTUREACCESS_TARGET, 1024, 1024);

// draw to the texture
SDL_SetRenderTarget( renderer, texture );

// draw a point onto the texture
SDL_RenderDrawPoint( renderer, 123, 456 );

// now draw to the window
SDL_SetRenderTarget( renderer, NULL );
```

Destroying a Texture

Finally, as with surfaces, textures must be freed when your program shuts down (or they are no longer in use). This is simply done with the function SDL_DestroyTexture().

```
SDL_DestroyTexture( texture );
texture = NULL;
```
