Sprites and Text in Direct3D 11

Frank Luna
December 21, 2011
www.d3dcoder.net

Unlike previous versions of Direct3D, Direct3D 11 does not have a utility library for drawing sprites and fonts. This sample shows one strategy for drawing sprites and fonts in Direct3D 11. In this document we highlight some of the conceptual details of the sample, but the reader is expected to study the code.

Sprites

Drawing sprites is easy enough; we simply draw a textured 2D quad on the screen. The texture coordinates of the quad defines the subset of the texture to map onto the quad and the quad spatial 3D coordinates can be defined in NDC space with depth value 0 so that the quad is drawn in the foreground. Note that many sprite images are normally placed on one large "texture atlas" so that we can draw different sprites with the same texture bound to the rendering pipeline with a single draw call; the draw call contains the geometry for multiple quads, where the texture coordinates identify the sprite on the atlas to map onto a given quad.

When drawing in screen space, we normally like to use pixel coordinates rather than NDC coordinates or normalized texture coordinates. For example, given a 1024x512 texture, we might want to draw the sub-rectangle of the texture defined by pixels (x=128, y=256, width=128, height=128) to the screen rectangle defined by pixels (x=600, y=400, width=128, height=128). The `SpriteBatch` class we have implemented, which is based off XNA's `SpriteBatch` class, provides an API for issuing draw calls in screen space using pixel coordinates. Internally, our `SpriteBatch` implementation converts texture pixel coordinates to the normalized texture space coordinates $[0,1]^2$. Similarly, pixel screen coordinates are converted to NDC space:

```
XMFLOAT3 SpriteBatch::PointToNdc(int x, int y, float z)
{
	XMFLOAT3 p;

	p.x = 2.0f*(float)x/mScreenWidth - 1.0f;
	p.y = 1.0f - 2.0f*(float)y/mScreenHeight;
	p.z = z;

	return p;
}

void SpriteBatch::BuildSpriteQuad(const Sprite& sprite, SpriteVertex v[4])
{
	const CD3D11_RECT& dest = sprite.DestRect;
	const CD3D11_RECT& src  = sprite.SrcRect;

	// Dest rect defines target in screen space.
	v[0].Pos = PointToNdc(dest.left,  dest.bottom, sprite.Z);
	v[1].Pos = PointToNdc(dest.left,  dest.top,    sprite.Z);
	v[2].Pos = PointToNdc(dest.right, dest.top,    sprite.Z);
	v[3].Pos = PointToNdc(dest.right, dest.bottom, sprite.Z);

	// Source rect defines subset of texture to use from sprite sheet.
	v[0].Tex = XMFLOAT2((float)src.left   / mTexWidth,
	                    (float)src.bottom / mTexHeight);
	v[1].Tex = XMFLOAT2((float)src.left   / mTexWidth,
	                    (float)src.top    / mTexHeight);
	v[2].Tex = XMFLOAT2((float)src.right  / mTexWidth,
	                    (float)src.top    / mTexHeight);
	v[3].Tex = XMFLOAT2((float)src.right  / mTexWidth,
	                    (float)src.bottom / mTexHeight);

	v[0].Color = sprite.Color;
	v[1].Color = sprite.Color;
	v[2].Color = sprite.Color;
	v[3].Color = sprite.Color;

	// Quad center point.
	float tx = 0.5f*(v[0].Pos.x + v[3].Pos.x);
	float ty = 0.5f*(v[0].Pos.y + v[1].Pos.y);

	XMVECTOR scaling = XMVectorSet(sprite.Scale, sprite.Scale, 1.0f, 0.0f);
	XMVECTOR origin = XMVectorSet(tx, ty, 0.0f, 0.0f);
	XMVECTOR translation = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
	XMMATRIX T = XMMatrixAffineTransformation2D(scaling,
		origin, sprite.Angle, translation);

	// Rotate and scale the quad in NDC space.
	for(int i = 0; i < 4; ++i)
	{
		XMVECTOR p = XMLoadFloat3(&v[i].Pos);
		p = XMVector3TransformCoord(p, T);
		XMStoreFloat3(&v[i].Pos, p);
	}
}
```

Text

To draw text, we need to generate character data. A common strategy is to use the GDI or GDI+ text drawing API to render the characters of a font to a texture atlas. We call a texture atlas of fonts a *font sheet* (see Figure 1).

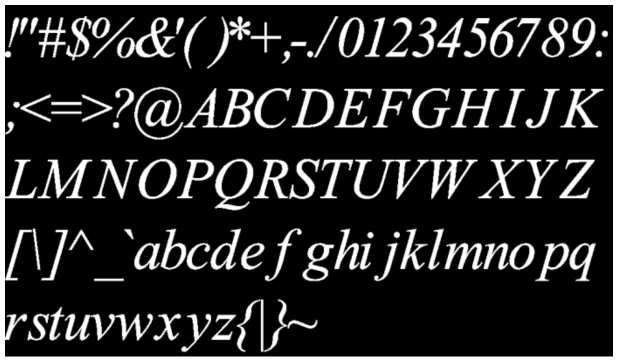
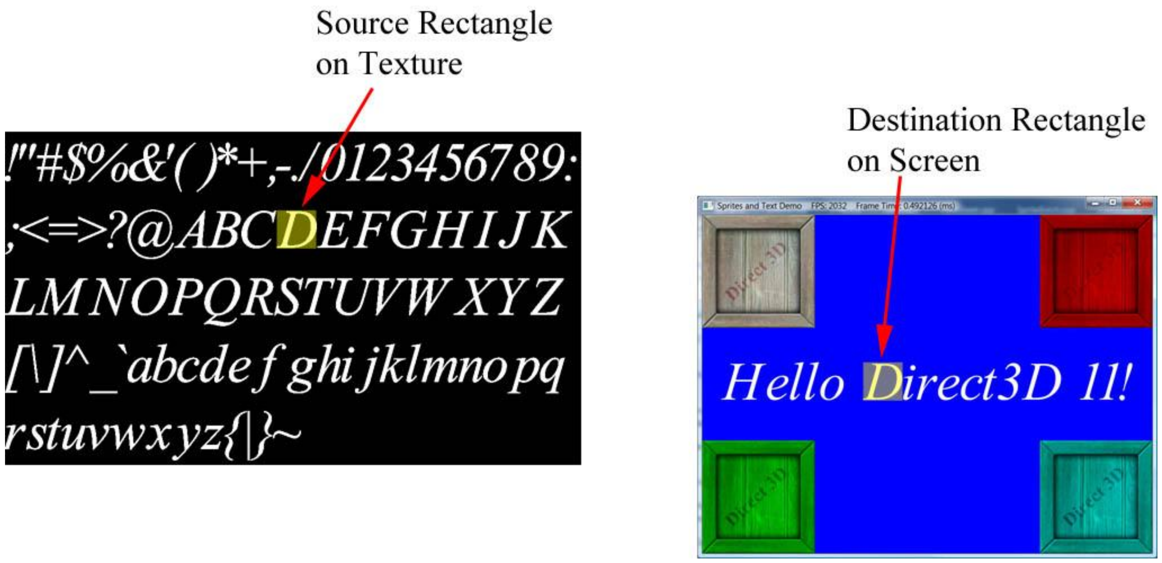


Figure 1: Example of a font sheet.

A font sheet is generated for a particular font family, size, and other display properties like bold, and italics. Therefore, you will need a different font sheet for each kind of font your application needs. It is relatively slow to generate a font sheet, so all the font sheets an application needs should be generated at initialization time (or level load time). As the font characters are being rendered out to the texture atlas, we cache the bounding rectangle for each character on the texture atlas. Thus, given any character, we can obtain its bounding rectangle on the texture atlas. Now, drawing a string simply amounts to drawing a sprite for each character, where each sprite is textured with the character image from the font sheet (see Figure 2).


Source Rectangle on Texture
Destination Rectangle on Screen
Sprites and Text Demo FPS: 2032 Frame Time: 0.492126 (ms)
Hello Direct3D 11!
Figure 2: To render the 'D' character, we obtain its bounding rectangle on the texture atlas (which the FontSheet caches when being built); this rectangle is the "source" rectangle in the SpriteBatch::Draw call. The destination rectangle on the screen is the rectangle we draw the quad over textured with the 'D' image.

The following code illustrates how a string is drawn using our `SpriteBatch` class given a `FontSheet`:

```
void SpriteBatch::DrawString(ID3D11DeviceContext* dc,
                             FontSheet& fs,
                             const std::wstring& text,
                             const POINT& pos,
                              XMCOLOR color)
{
        BeginBatch(fs.GetFontSheetSRV());

        UINT length = text.length();

        int posX = pos.x;
        int posY = pos.y;

        // For each character in the string...
        for(UINT i = 0; i < length; ++i)
        {
                WCHAR character = text[i];

                // Is the character a space char?
                if(character == ' ')
                {
                        posX += fs.GetSpaceWidth();
                }
                // Is the character a newline char?
                else if(character == '\n')
                {
                        posX  = pos.x;
                        posY += fs.GetCharHeight();
                }
                else
                {
                        // Get the bounding rect of the character on the fontsheet.
                        const CD3D11_RECT& charRect = fs.GetCharRect(character);

                        int width = charRect.right - charRect.left;
                        int height = charRect.bottom - charRect.top;

                        // Draw the character sprite.
                        Draw(CD3D11_RECT(posX, posY,
                            posX + width,
                            posY + height),
                            charRect,
                            color);

                        // Move to the next character position.
                        posX += width + 1;
        }
    }

        EndBatch(dc);
}
```

Example Usage

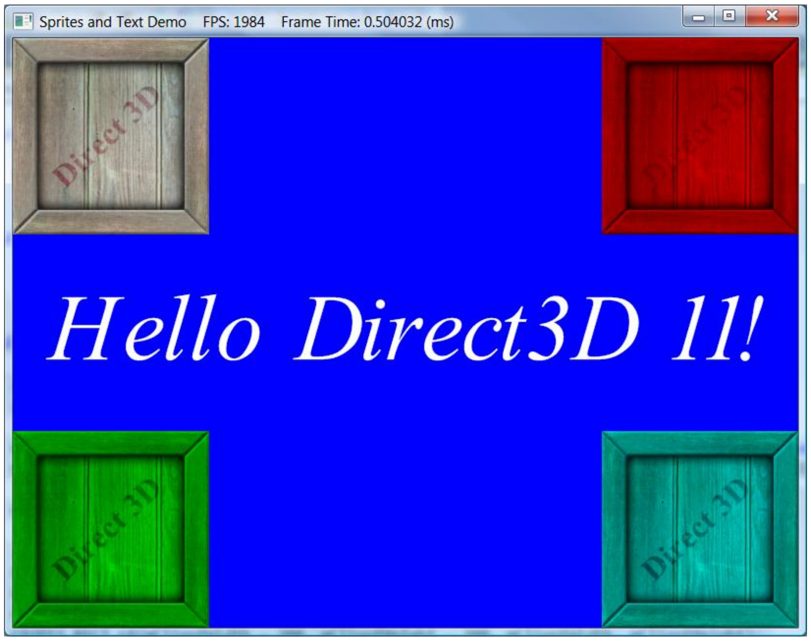


Figure 3: Screenshot of the "Sprites and Text" demo.

The `SpritesAndTextApp` demo application defines the following member variables:

```
FontSheet mFont;
SpriteBatch mSpriteBatch;
ID3D11ShaderResourceView* mImageSRV;
```

An application may need several `FontSheets`, but only one `SpriteBatch` is needed per application. The `mImageSRV` is another texture we use to illustrate drawing a texture using one of the `SpriteBatch::Draw` functions. These objects are initialized as follows:

```
bool InitDirect3DApp::Init()
{
	if(!D3DApp::Init())
		return false;

	Effects::InitAll(md3dDevice);
	RenderStates::InitAll(md3dDevice);

	HR(mFont.Initialize(md3dDevice,
		L"Times New Roman", 96.0f,
		FontSheet::FontStyleItalic, true));

	HR(mSpriteBatch.Initialize(md3dDevice));

	HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
		L"Textures/WoodCrate01.dds", 0, 0, &mImageSRV, 0 ));

	return true;
}
```

Our `FontSheet` generates a font sheet with characters 96 pixels high with Times New Roman, italics, and antialiased text.

We draw the crate texture 4 times, in each corner of the screen. We center the text in the middle of the screen. Note that we have to draw the text with transparency blending enabled to mask out the empty space of the character images (the font sheet does have an alpha channel for this).

```
void SpritesAndTextApp::DrawScene()
{
	assert(md3dImmediateContext);
	assert(mSwapChain);

	md3dImmediateContext->ClearRenderTargetView(
		mRenderTargetView, reinterpret_cast<const float*>(&Colors::Blue));
	md3dImmediateContext->ClearDepthStencilView(
		mDepthStencilView,
		D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL, 1.0f, 0);

	mSpriteBatch.BeginBatch(mImageSRV);

	// Draw the sprite in each corner of the screen with a different tint.
	CD3D11_RECT r1(0, 0, 200, 200);
	CD3D11_RECT r2(mClientWidth - 200, 0, mClientWidth, 200);
	CD3D11_RECT r3(0, mClientHeight - 200, 200, mClientHeight);
	CD3D11_RECT r4(mClientWidth - 200, mClientHeight - 200,
		mClientWidth, mClientHeight);

	mSpriteBatch.Draw(r1, XMCOLOR(0xffffffff));
	mSpriteBatch.Draw(r2, XMCOLOR(0xffff0000));
	mSpriteBatch.Draw(r3, XMCOLOR(0xff00ff00));
	mSpriteBatch.Draw(r4, XMCOLOR(0xff00ffff));

	mSpriteBatch.EndBatch(md3dImmediateContext);

	float blendFactor[4] = {1.0f};
	md3dImmediateContext->OMSetBlendState(
		RenderStates::TransparentBS, blendFactor, 0xffffffff);

	std::wstring text = L"Hello Direct3D 11!";

	// Calculate the text width.
	int textWidth = 0;
	for(UINT i = 0; i < text.size(); ++i)
	{
		WCHAR character = text[i];
		if(character == ' ')
		{
			textWidth += mFont.GetSpaceWidth();
		}
		else
		{
			const CD3D11_RECT& r = mFont.GetCharRect(text[i]);
			textWidth += (r.right - r.left + 1);
		}
	}

	// Center the text in the screen.
	POINT textPos;
	textPos.x = (mClientWidth - textWidth) / 2;
	textPos.y = (mClientHeight - mFont.GetCharHeight()) / 2 ;

	mSpriteBatch.DrawString(md3dImmediateContext, mFont,
            text, textPos, XMCOLOR(0xffffffff));

        // restore default
        md3dImmediateContext->OMSetBlendState(0, blendFactor, 0xffffffff);

        HR(mSwapChain->Present(0, 0));
}
```