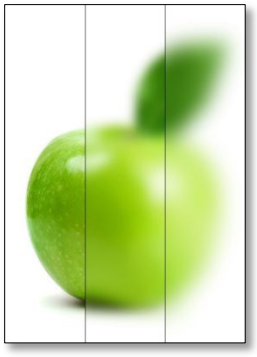
Post Processing

Altering Rendering Results Before Presenting Them



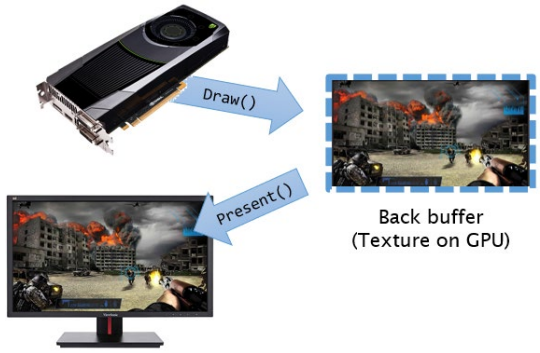
Blurring an Image

Before jumping into post processing, let's look at a simple image effect: blur. Blurring an image involves averaging surrounding pixel colors. As more and more pixels are included in the average, the overall blurriness of the image increases. See the image to the right for an example. The left slice is the unblurred image, the middle slice has been blurred a certain amount and the right slice is even blurrier.

This is fairly simple with a static image, but is much more complex in a real-time rendering engine. To discuss why, let's review the basic rendering process and where those results are stored.

Review: The Back Buffer

In a basic rendering engine, the results of each draw call are stored in a special texture on the GPU called the *back buffer*. At the beginning of a frame, we clear the contents of the back buffer to a solid color. Each successive draw operation might replace, or be combined with, colors previously drawn this frame. At the end of the frame, the contents of the back buffer are *presented* to the user in the application window.



Back Buffer Problems

Rendering directly to the back buffer has worked great thus far. Sometimes, however, we may want to further tweak the final rendering results *before* presenting them to the user. For instance, what if we want to blur our rendering results? See the street in the screenshot of *Need for Speed Heat* below. Pixels of the street are blurred along a vector from the center of the screen outward, leading to a *radial motion blur* effect.

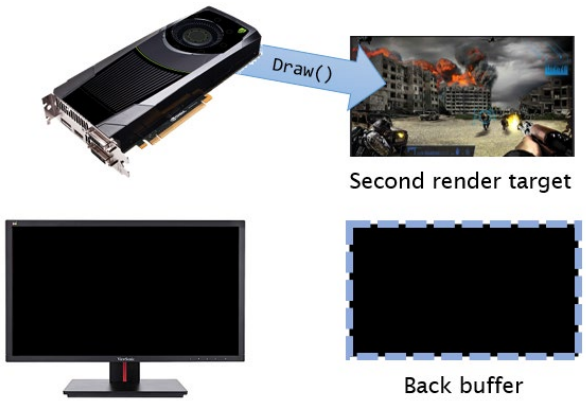
So, what's the problem? Each pixel of a single draw call is executed simultaneously. This means that we cannot average a pixel with its neighbors until *after* the pixel shader is complete for the entire object! Complicating this even further is the fact that other objects will be drawn later in the same frame, so we cannot feasibly blur the screen until every single object has been fully drawn. However, once the results are in the back buffer, we cannot easily alter them (other than to render something else on top).

Post Processing

Altering rendering results before the user sees them is known as *post processing*. This requires the entire scene to be fully rendered (minus any user interface elements) so we can modify those results in another shader. In other words, we essentially need a static image of what the user will see this frame. Then, one or more post process shaders can apply various image effects, such as blur, to that final image.

Render Targets

The results of the rendering pipeline are always stored in a texture (that's the whole point). A texture that can be used to hold rendering results is known as a *render target*. The back buffer has been our only render target thus far. However, since rendering directly to the back buffer precludes us from reading those pixel colors in another shader, we need to render *somewhere else* first.

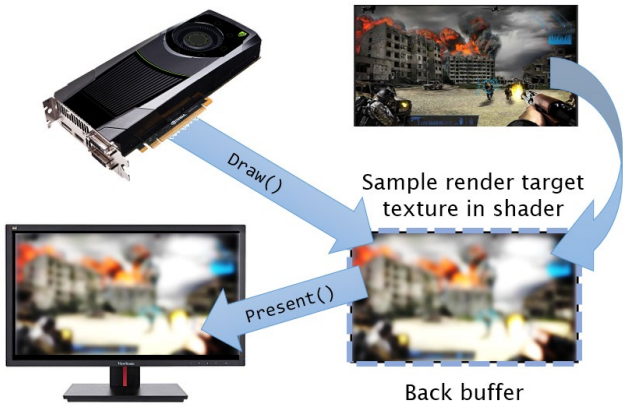


Whenever we choose, we can change the active render target so that rendering results are stored there *instead of* the back buffer. See the diagram above for an example. This is only half the battle, though, as presenting the frame at this point will simply display a blank screen to the user.

Sampling Render Targets

The final step is to draw the contents of our second render target (which is just a regular texture) to the back buffer. During this draw, we use a special pixel shader that reads, and further alters, the data in the render target.

Here is where we can apply image effects like blur. Since all of the pixels of the frame are finalized and captured in our render target, we can grab neighboring pixels and average their colors.



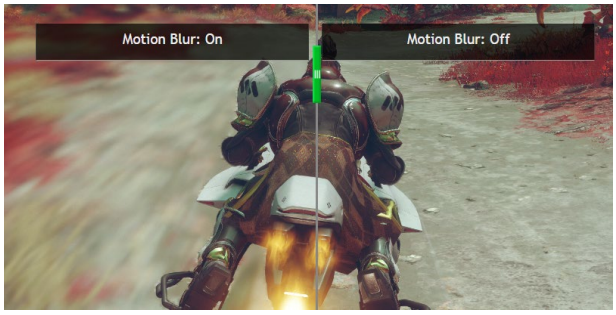
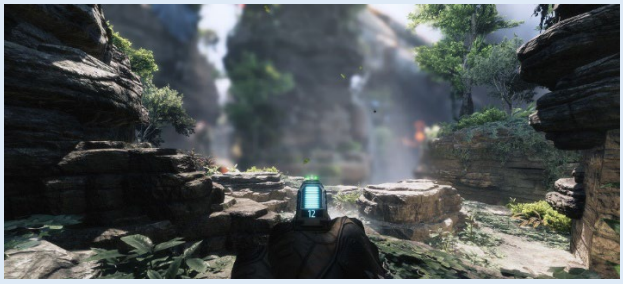
How Many Render Targets?

To implement a simple post process, only one extra render target is required. Complex effects are comprised of multiple post processing steps (and, therefore, require numerous render targets). Chaining multiple post processes together, one after the other, is accomplished by feeding the results of one into the next, also requiring extra render targets (or smartly reusing targets from previous steps).

The only limit to the number of render target resources that can exist at once is the amount of memory on the GPU. Advanced techniques can even *render to* multiple render targets simultaneously.

Post Processing Examples

Below you'll find examples of common effects that are implemented as post processes.

Post Process	Example
Bloom Bloom, or Light Bloom, approximates the scattering of light to make bright areas of the screen appear to "glow". This is a multiple-pass post process: 1. Extract bright colored pixels only 2. Blur the "bright colors" target 3. Combine that result with the original frame render	 *Rise of the Tomb Raider*
Motion Blur This post process blurs objects based on their motion and/or the movement of the camera in relation to those objects. Any object that isn't moving relative to the camera, such as the player, generally remains in focus (unblurred).	 *Destiny 2*
Depth of Field This post process approximates the focus of a camera lens, causing objects in the foreground, background or both to appear out of focus (or blurred). Modern depth of field post processes often requires separating fore- & background elements into separate render targets.	 *Titanfall 2*
Screen Space Reflections SSR, for short, produces approximate reflections of other surfaces that appear within the same frame. This is accomplished by ray marching through the frame. SSR works very well for reflections on horizontal flat surfaces like floors or bodies of water.	 *Control*

Implementation

Render targets, like any other GPU resource, should be created during initialization. When drawing, we swap the active render target to control the destination of the rendering pipeline. At the end of the frame, we have one final step of plastering the entire screen with the post processing results. Below we'll dig into these individual steps.

Required Resources

A render target is simply a texture under the hood, so we'll need to create a texture that matches the window size. However, to actually use that texture as either output or input, corresponding views must also be created: a *render target view* (RTV) for rendering into the texture and a *shader resource view* (SRV) for sampling from it in a shader. Since the views track the texture internally, we don't explicitly need to keep a reference to the texture itself.

We'll also need post-process-specific vertex and pixel shaders. The same vertex shader can be used for *all* post processes, but each requires a specialized pixel shader to perform a specific image effect.

> *If we were implementing multiple, sequential post processes, each would need its own RTV, SRV and pixel shader.*

Lastly, we'll most likely need a sampler that clamps at the edge of textures rather than wraps around, so that colors on one edge of the screen don't bleed into the opposite edge.

```
// Resources that are shared among all post processes
Microsoft::WRL::ComPtr<ID3D11SamplerState> ppSampler;
std::shared_ptr<SimpleVertexShader> ppVS;

// Resources that are tied to a particular post process
std::shared_ptr<SimplePixelShader>  ppPS;
Microsoft::WRL::ComPtr<ID3D11RenderTargetView>   ppRTV; // For rendering
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> ppSRV; // For sampling
```

Sampler Setup

The usual sampler for post processing requires *clamp* addressing rather than *wrap*. Since it will be used for a texture that we're viewing "head on", anisotropic filtering is unnecessary.

```
// Sampler state for post processing
D3D11_SAMPLER_DESC ppSampDesc = {};
ppSampDesc.AddressU = D3D11_TEXTURE_ADDRESS_CLAMP;
ppSampDesc.AddressV = D3D11_TEXTURE_ADDRESS_CLAMP;
ppSampDesc.AddressW = D3D11_TEXTURE_ADDRESS_CLAMP;
ppSampDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
ppSampDesc.MaxLOD = D3D11_FLOAT32_MAX;
device->CreateSamplerState(&ppSampDesc, ppSampler.GetAddressOf());
```

Render Target Creation

The creation of the underlying texture is straightforward: fill out the description and create. The width and height generally need to match the window size, though some post processes will render to targets that are half or quarter size and then blown back up to get a slight blur “for free”.

```
// Describe the texture we're creating
D3D11_TEXTURE2D_DESC textureDesc = {};
textureDesc.Width     = windowWidth;
textureDesc.Height    = windowHeight;
textureDesc.ArraySize = 1;
textureDesc.BindFlags = D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE;
textureDesc.CPUAccessFlags = 0;
textureDesc.Format    = DXGI_FORMAT_R8G8B8A8_UNORM;
textureDesc.MipLevels = 1;
textureDesc.MiscFlags = 0;
textureDesc.SampleDesc.Count = 1;
textureDesc.SampleDesc.Quality = 0;
textureDesc.Usage = D3D11_USAGE_DEFAULT;

// Create the resource (no need to track it after the views are created below)
Microsoft::WRL::ComPtr<ID3D11Texture2D> ppTexture;
device->CreateTexture2D(&textureDesc, 0, ppTexture.GetAddressOf());
```

Now we need to create the corresponding views so we can bind the texture to the pipeline in various ways. Note that the SRV creation is simplified by passing in null (zero) for the description, which results in a default SRV that can access any and all sub-resources.

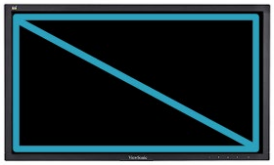
```
// Create the Render Target View
D3D11_RENDER_TARGET_VIEW_DESC rtvDesc = {};
rtvDesc.Format = textureDesc.Format;
rtvDesc.Texture2D.MipSlice = 0;
rtvDesc.ViewDimension = D3D11_RTV_DIMENSION_TEXTURE2D;
device->CreateRenderTargetView(
	ppTexture.Get(),
	&rtvDesc,
	ppRTV.ReleaseAndGetAddressOf());

// Create the Shader Resource View
// By passing it a null description for the SRV, we
// get a "default" SRV that has access to the entire resource
device->CreateShaderResourceView(
	ppTexture.Get(),
	0,
	ppSRV.ReleaseAndGetAddressOf());
```

Note that if the window size changes, any and all “screen-sized” render targets need to be destroyed and re-created, along with their associated views. This could be handled with a helper function that first .Reset()’s the SRV & RTV ComPtrs, then re-creates the texture (using the new window size), the RTV and the SRV. If you have multiple screen-sized render targets, they all need to be resized!

Post Process Shaders

The end goal of our post process shaders is to fill the screen with new pixel data. This means we need to run a pixel shader for every pixel of the window. To do this in a rasterization pipeline, we need geometry that fully covers the screen so that it rasterizes every possible pixel.

Filling the Screen

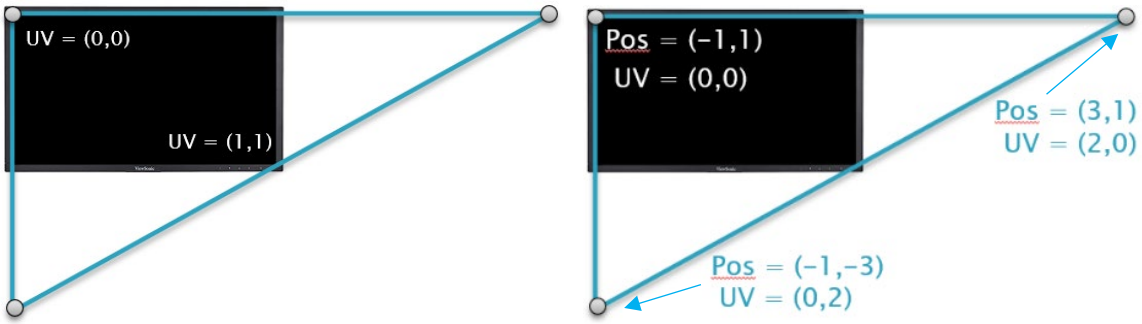


Not only do we need to fill the screen with geometry; we also need to ensure that the UV coordinates of that geometry are useful to us. Ideally, they should range from (0,0) in the top left corner to (1,1) in the bottom right corner. This means we can't just use an all-encompassing shape like a cube as we might for a skybox. Instead, we want geometry that perfectly fits the window.

Your first instinct might be a quad – two triangles – that covers the window (as seen above). This would absolutely work, but would probably require us to create new vertex and index buffers and swap the mesh before applying the post process. Not a lot of work, but we can do better.

A Fullscreen Triangle

The rasterizer will automatically clip (not rasterize) geometry outside the bounds of the window. This means we could potentially create a single triangle that is larger than the screen, such that it rasterizes every pixel. If we populate these three vertices with the correct data, we'll get the exact UV coordinates we need across those rasterized pixels.



The rasterizer measures the screen in normalized device coordinates (NDCs), which are [-1,1] on both X and Y axes. Thus, the triangle will need to be twice as wide and twice as tall – [-1, 3] on each axis – to fully cover the screen. The UV coordinates will also need to double across the triangle to ensure the rasterized pixels end up with UVs in the [0,1] range.

We could make a vertex buffer with exactly three vertices and use it when performing a post process. Even better: we can create this simple vertex data on the fly in the vertex shader, rather than relying on a buffer to permanently hold these values. Then we simply tell our Graphics API to draw exactly 3 vertices and let the shader figure out the details.

Fullscreen Vertex Shader

Below is the code for a simple "fullscreen triangle" vertex shader. Note the use of the SV_VertexID semantic. This denotes that the pipeline will provide a unique id (an unsigned integer) for each vertex. We can simply use this integer to decide the data for each vertex.

```
struct VertexToPixel
{
	float4 position : SV_POSITION;
	float2 uv       : TEXCOORD0;
};

VertexToPixel main(uint id : SV_VertexID)
{
	VertexToPixel output;

	if (id == 0)
	{
		output.position = float4(-1, 1, 0, 1);
		output.uv = float2(0, 0);
	}
	else if (id == 1)
	{
		output.position = float4(3, 1, 0, 1);
		output.uv = float2(2, 0);
	}
	else if (id == 2)
	{
		output.position = float4(-1, -3, 0, 1);
		output.uv = float2(0, 2);
	}

	return output;
}
```

That's it! In C++, we tell the GPU to draw exactly 3 vertices and we'll get a triangle that fills the screen.

Branchless Fullscreen VS

For completeness, here is the branchless vertex of the above vertex shader. It uses bitwise operators to very quickly and efficiently calculate a 0 or a 2 from the id. The following code would replace the if statements in the code above:

```
// Calculate the UV (0,0) to (2,2) using the ID
output.uv = float2(
	(id << 1) & 2, // Essentially: id % 2 * 2
	id & 2);

// Calculate the position based on the UV
output.position = float4(output.uv, 0, 1);
output.position.x = output.position.x * 2 - 1;
output.position.y = output.position.y * -2 + 1;
```

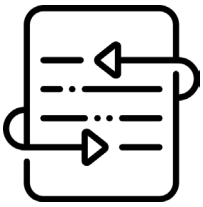
Post Process Pixel Shader

This is where the actual "processing" part of post processing takes place. It will need a texture representing the rendered frame and a clamp sampler. It will usually also need a cbuffer with any external data needed to control the overall effect (not explicitly shown below).

```
struct VertexToPixel
{
	float4 position : SV_POSITION;
	float2 uv       : TEXCOORD0;
};

Texture2D Pixels           : register(t0);
SamplerState ClampSampler  : register(s0);

float4 main(VertexToPixel input) : SV_TARGET
{
	float4 pixelColor = Pixels.Sample(ClampSampler, input.uv);

	// NOTE: Here is where you should actually "process" the image

	return pixelColor;
}
```

This example shader simply returns the existing color of the render target, which will make it look like we just rendered directly to the screen without any modifications. While not interesting to look at, this simplified shader makes it easy to tell if things are working before implementing a more complex effect.

Rendering Updates



Next, we'll look at how these new resources fit into our overall rendering work. Our rendering code can be categorized into three phases: **Pre-Render**, where we do our initial frame setup; **Render**, where we're actually drawing the scene; and **Post-Render**, where we apply post processing, draw the UI and present the frame to the user.

Pre-Render

Currently, our pre-render steps involve clearing the back buffer and depth buffer. We'll also need to clear any and all extra render targets.

```
context->ClearRenderTargetView(ppRTV.Get(), clearColor);
```

Lastly, before actually rendering, we need to swap the active render target. While some advanced techniques can make use of multiple simultaneous render targets, we only need one. (This is why the function is called OMSetRenderTargets and why it takes an array of RTVs.) As we'll be rendering our 3D meshes to this render target, we still need our one and only depth buffer to be active as well.

```
context->OMSetRenderTargets(1, ppRTV.GetAddressOf(), depthBufferDSV.Get());
```

Render

This phase is where we render all of the scene entities and the sky. Fortunately for us, it remains exactly the same! No changes are necessary here; the rendering pipeline will automatically place our results in the render target we set above.

Post-Render

This is where the fun begins. We first need to restore the back buffer, though we have no need for the depth buffer at this point in the frame.

```
context->OMSetRenderTargets(1, backBufferRTV.GetAddressOf(), 0);
```

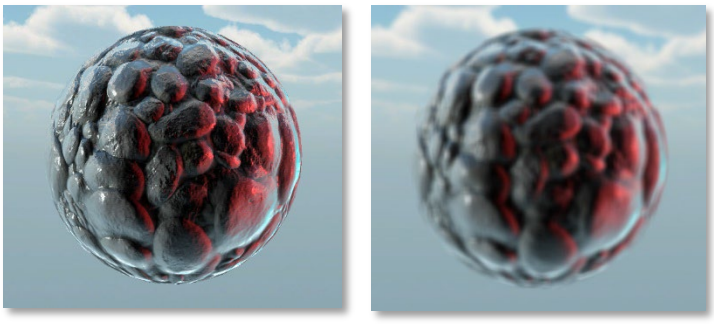
To apply a post process, we set our post process vertex and pixel shaders, as well as any required data, textures (SRVs) and samplers, and then "draw" it. This occurs before drawing any UI elements.

```
// Activate shaders and bind resources
// Also set any required cbuffer data (not shown)
ppVS->SetShader();
ppPS->SetShader();
ppPS->SetShaderResourceView("Pixels", ppSRV.Get());
ppPS->SetSamplerState("ClampSampler", ppSampler.Get());

context->Draw(3, 0); // Draw exactly 3 vertices (one triangle)
```

Lastly, our existing code presents the frame to the user.

Post Process Example: Box Blur

This listing details how to implement a basic "box blur" post process pixel shader. A box blur is a blur that takes into account pixels within a square (or box) shape around the current pixel. In other words, a 2D grid centered on the current pixel. The example images below show a frame rendered normally and the same frame with a box blur applied.



Box Blur Post Process Pixel Shader

This example requires three pieces of data from C++: the “radius” of the blur (how many pixels the box extends in each direction, with a radius of zero meaning no blur), 1.0 over the window width (the width of a single pixel in UV space) and 1.0 over the window height (the height of a pixel in UV space).

```
cbuffer externalData : register(b0)
{
	int blurRadius;
	float pixelWidth;
	float pixelHeight;
}

struct VertexToPixel
{
	float4 position : SV_POSITION;
	float2 uv       : TEXCOORD0;
};

Texture2D Pixels           : register(t0);
SamplerState ClampSampler  : register(s0);

float4 main(VertexToPixel input) : SV_TARGET
{
	// Track the total color and number of samples
	float4 total = 0;
	int sampleCount = 0;

	// Loop through the "box"
	for (int x = -blurRadius; x <= blurRadius; x++)
	{
		for (int y = -blurRadius; y <= blurRadius; y++)
		{
			// Calculate the uv for this sample
			float2 uv = input.uv;
			uv += float2(x * pixelWidth, y * pixelHeight);

			// Add this color to the running total
			total += Pixels.Sample(ClampSampler, uv);
			sampleCount++;
		}
	}

	// Return the average
	return total / sampleCount;
}
```