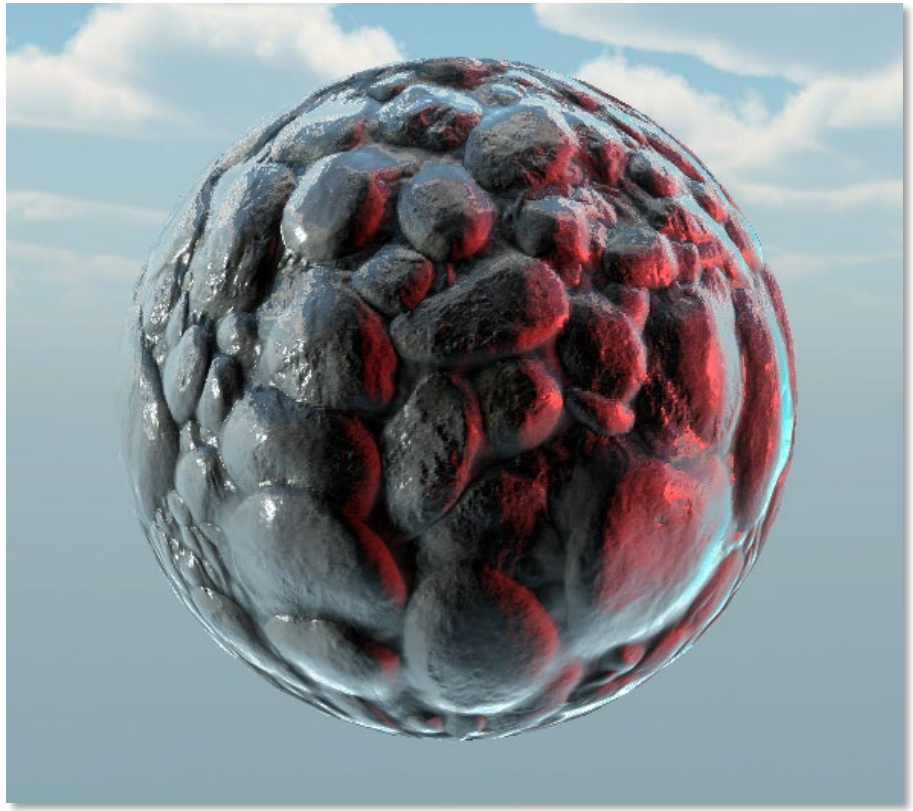
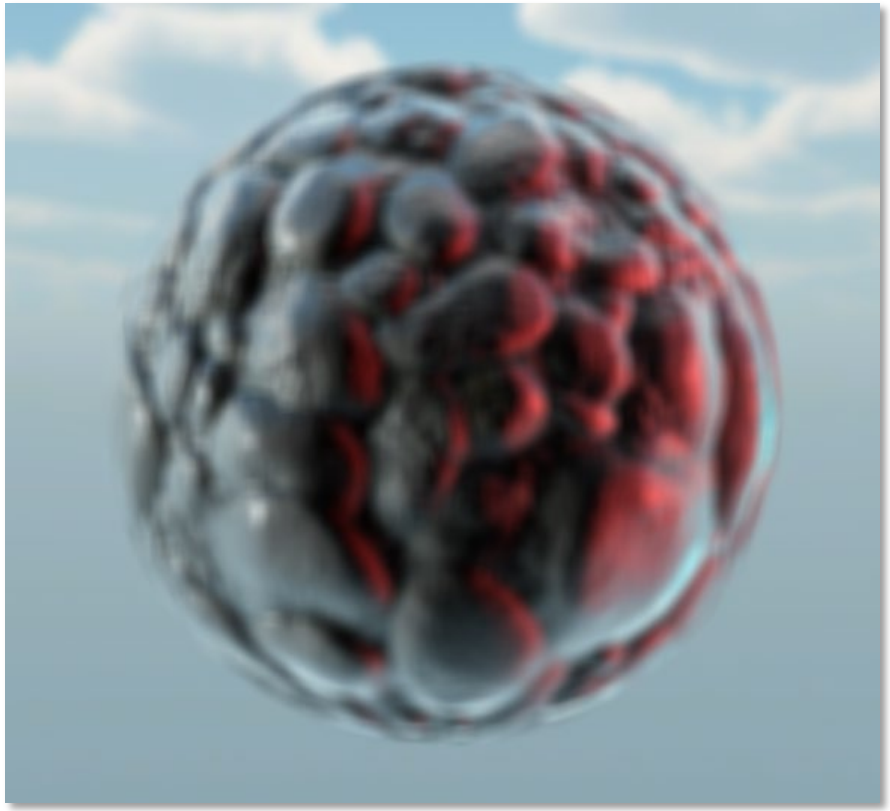
Post Processing

Manipulating the results of rendering before displaying them

Post process example: Blur

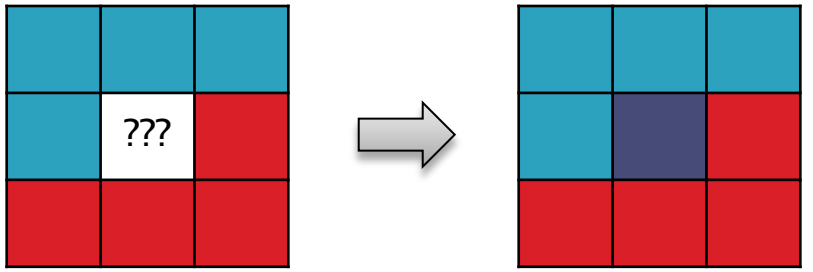




How do you blur an image?

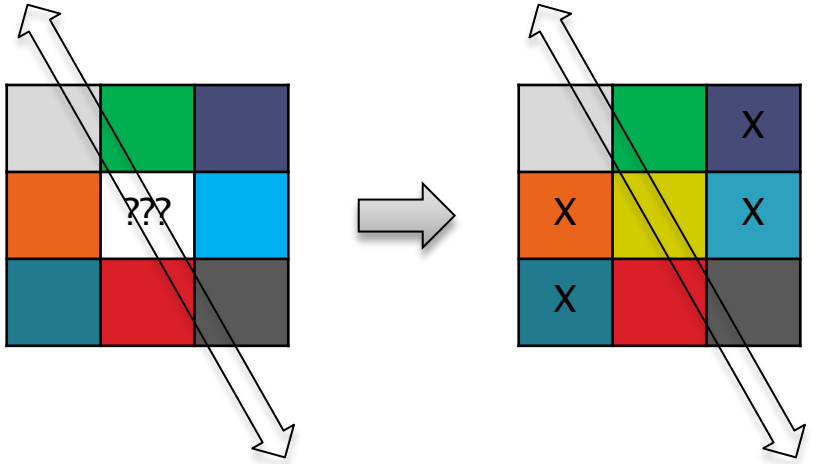
- Basic “box” blur

```
foreach(pixel)
{
   Average w/ surrounding pixel colors
}
```

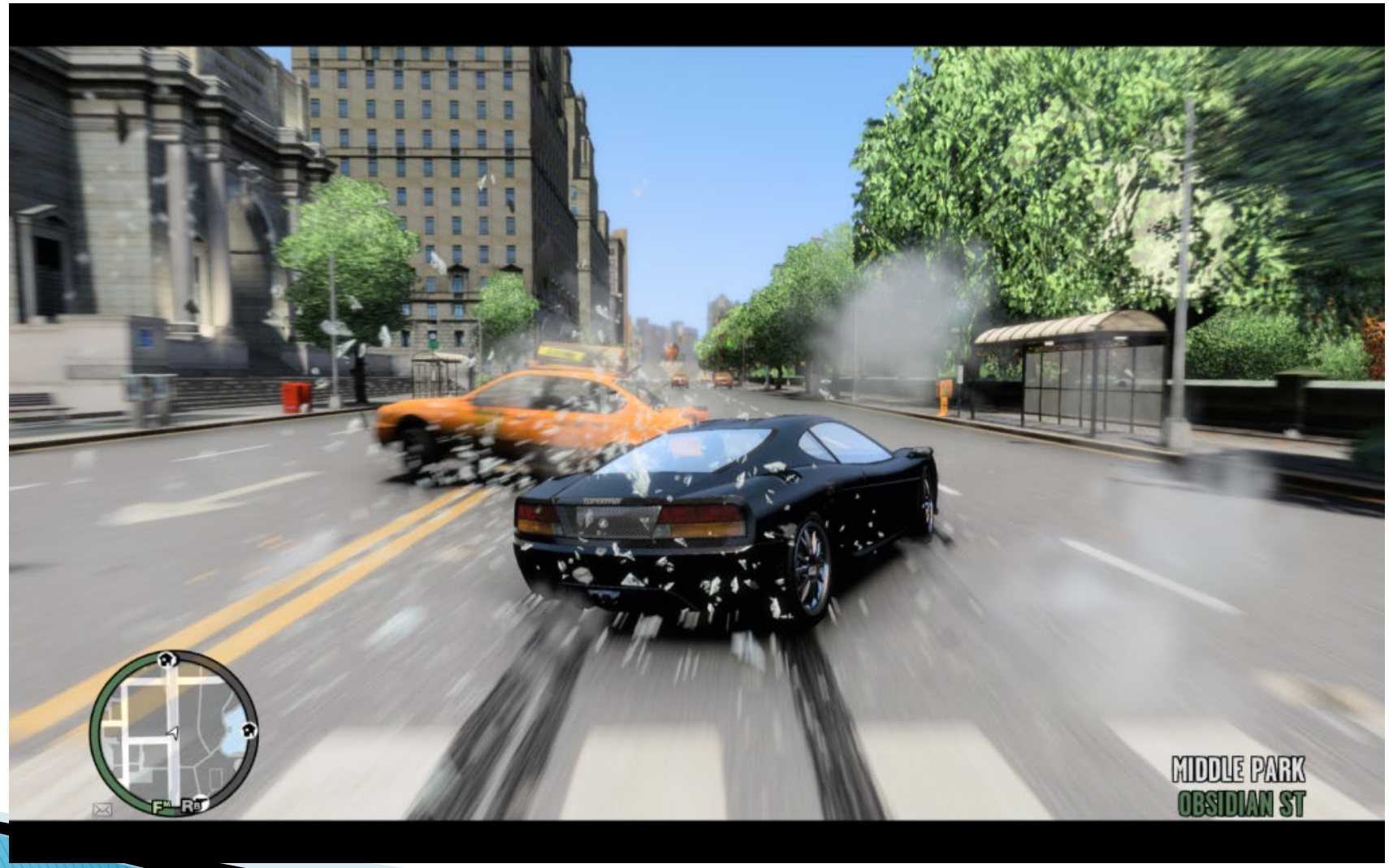


- Directional blur

```
foreach(pixel)
{
   Average w/ pixels along a screen-space vector
}
```



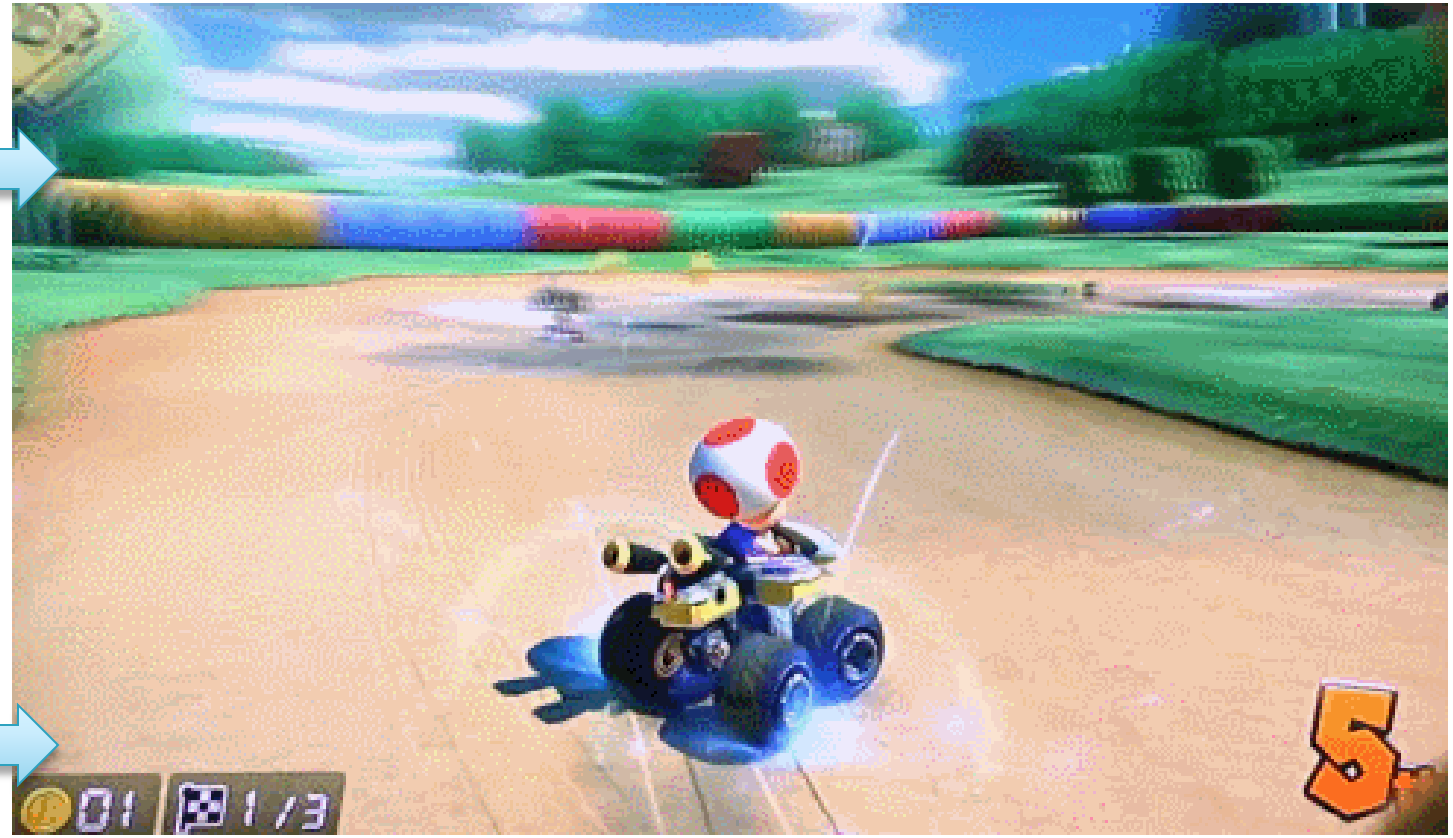
Directional blur example: Radial blur



Radial blur example

Blur in motion

Even in “cartoony” games



Freeze frame

Horizontal blur when cornering

Radial blur when going forward

How do we blur what we're rendering?

- We can't blur *while rendering* an object
  - Each pixel is processed in parallel
  - Can't get "neighbors" during pixel shader
- Objects rendered one at a time
  - Can't get pixel colors that haven't been rendered yet
  - Won't know final neighbor colors until all objects are rendered
- Can't really blur until the frame has been completed

Blurring the screen

- Need all of the colors before you can blur
  - Must render entire frame
  - Then apply a blur
- But the pixels are already in the back buffer…
  - How do we manipulate colors after rendering them?
- We need to render them to an intermediate location
  - Instead of the screen
  - We need a new *Render Target*

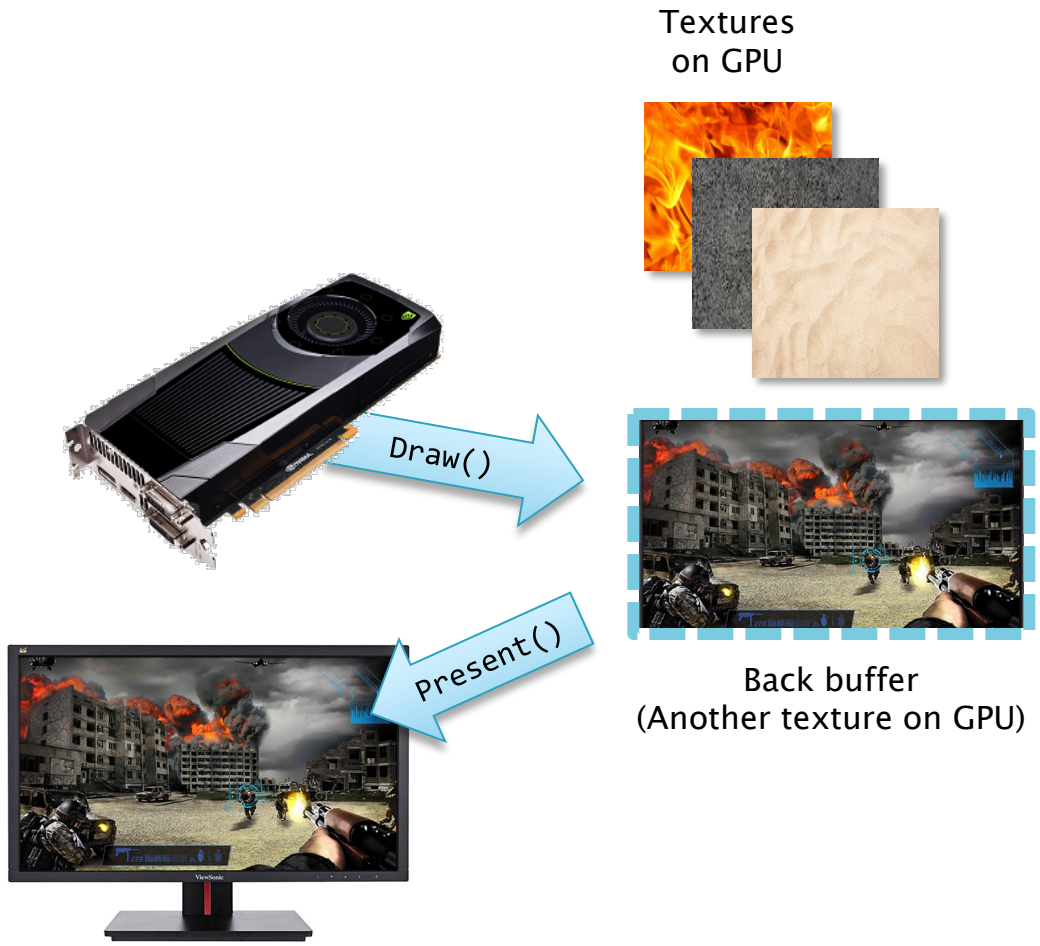
Render Targets

Render targets

- A texture in memory where rendering results are stored
- All rendering is always put in a texture
  - Usually just the “back buffer”
  - Back buffer is the render target used by the swap chain
  - Set up for you in the starter code
- We can make any number of render targets
- And swap which one is active at any time

Current rendering process

- Draw() objects to default render target
- The *back buffer*
  - A texture in GPU memory
  - Dimensions match window
- Calling Present() displays back buffer contents

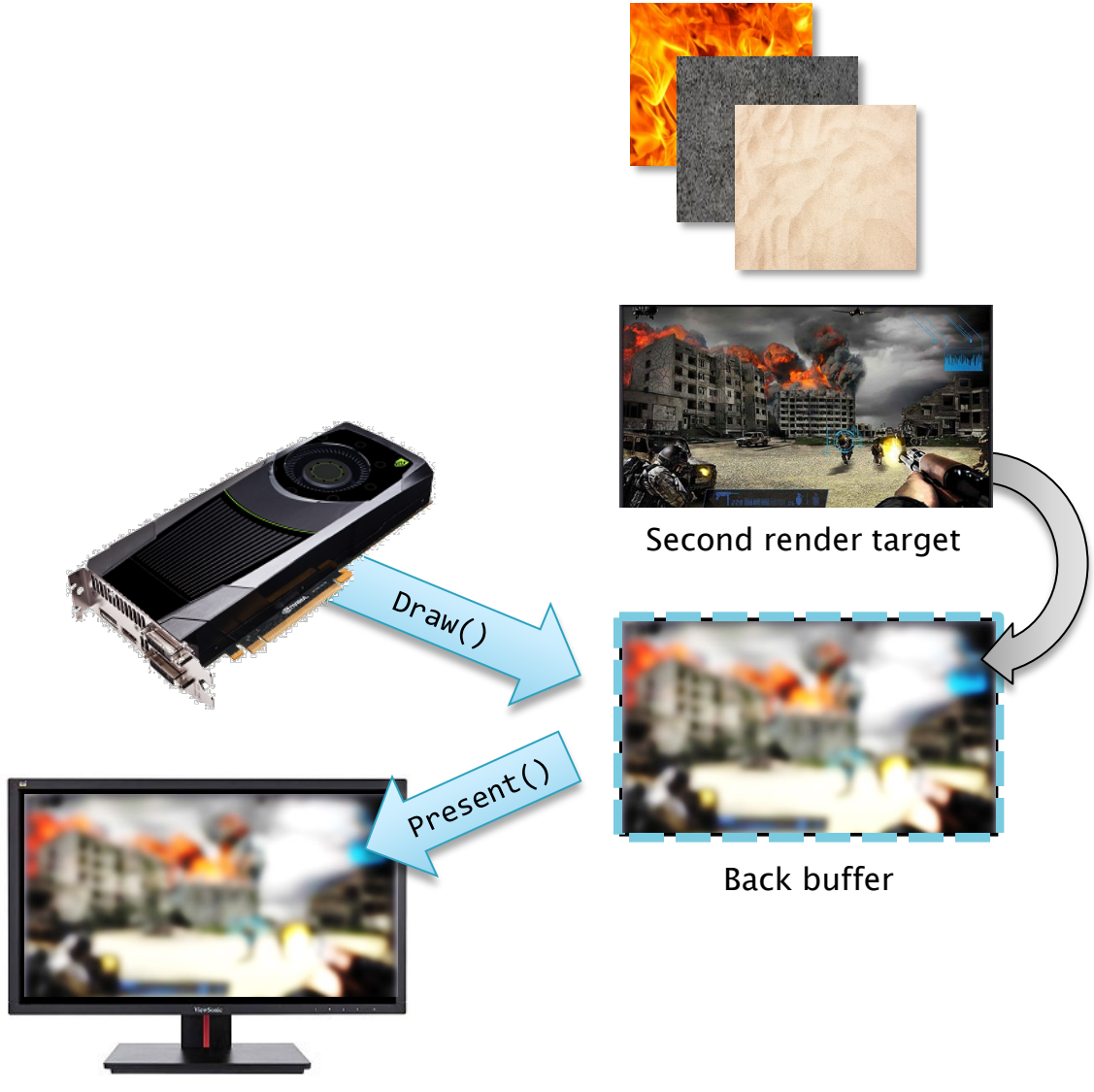


Render target example

- Suppose we had a second render target
  - Another texture in GPU memory
  - Same dimensions as back buffer
- Could redirect rendering there
  - `context->OMSetRenderTargets();`

What good is another render target?

- It's a texture
  - Like any other texture on GPU
  - Can be sampled in a pixel shader
- Draw one more time
  - Redirect output to back buffer
  - Use a pixel shader that further alters the image
- Present() final image after



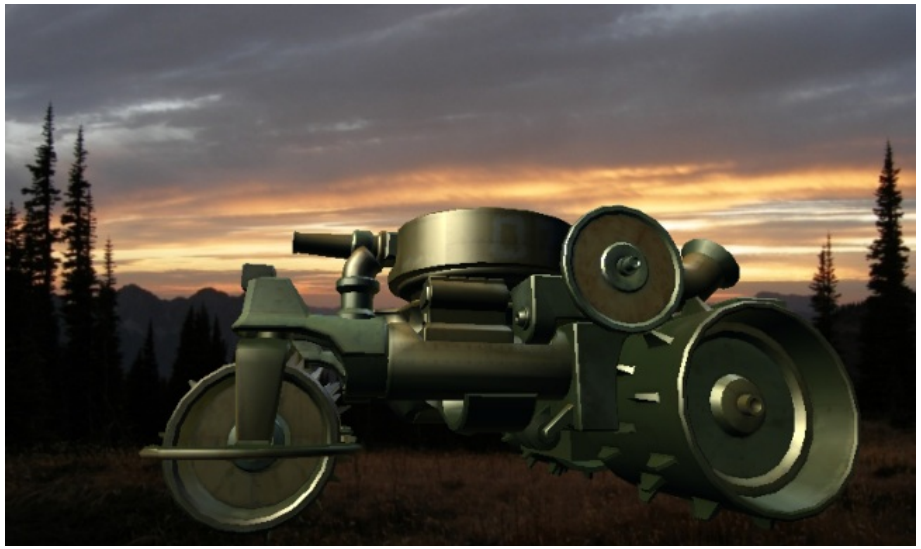
Rendering to a texture

- When would we want to render to a texture?
- **Real-time maps** – Data about this frame
  - Shadows
  - Reflections
- **Full-screen special effects** – Post Processing
  - Motion Blur
  - Bloom/Glow
  - Depth of Field
- **Deferred Rendering** – Apply lighting after all geometry is rendered

Post Processing

Post processing

- Full-screen effects added after scene has been fully rendered



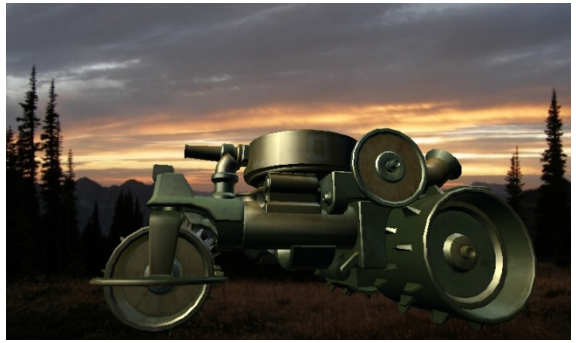
Before

After

Post process example: Blur

Scene Objects (meshes)

Draw() to
Render Target



Render Target

Use shader that
blurs final image

Draw() to back buffer

Back Buffer

Resources required for post processing

- ID3D11Texture2D*
  - The texture into which you'll be rendering
- ID3D11RenderTargetView*
  - Allows us to bind a texture as a target for rendering
  - Uses the above Texture2D
- ID3D11ShaderResourceView*
  - Allows us to bind a texture for sampling
  - Also uses the above Texture2D

Pre-scene-render

- Steps to take before any actual rendering

- 1. Set the render target on the device
  - `context->OMSetRenderTargets()`
  - Requires the depth/stencil view as well

- 2. Clear
  - All render targets
  - Depth buffer

Rendering scene geometry

- Render normally!
  - Use your existing rendering code
  - No changes necessary
- Output will go to the new render target instead of screen
- This means there's another step…

Post-scene-render

- Steps to apply the post process

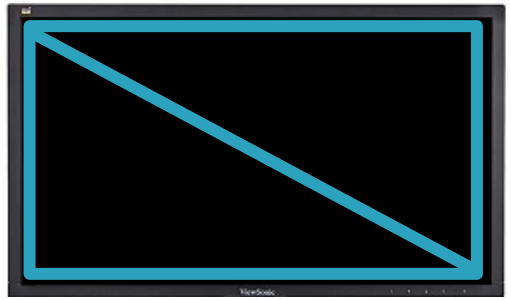
- 1. Reset rendering to the back buffer
  - OMSetRenderTargets() again
  - Set depth/stencil view to null

  (Whatever you draw now goes to the screen)

- 2. Draw geometry that covers the whole viewport
  - Bind render target as a shader resource
  - Use a pixel shader that further alters the results

Drawing a texture to the screen

- You have a frame's worth of data in a texture
- Need to "draw" that texture to the screen
  - Passing it through a custom Pixel Shader
- What geometry is used when drawing?
  - Need triangles that exactly cover the screen
  - Could create a whole new mesh in C++
  - Or…



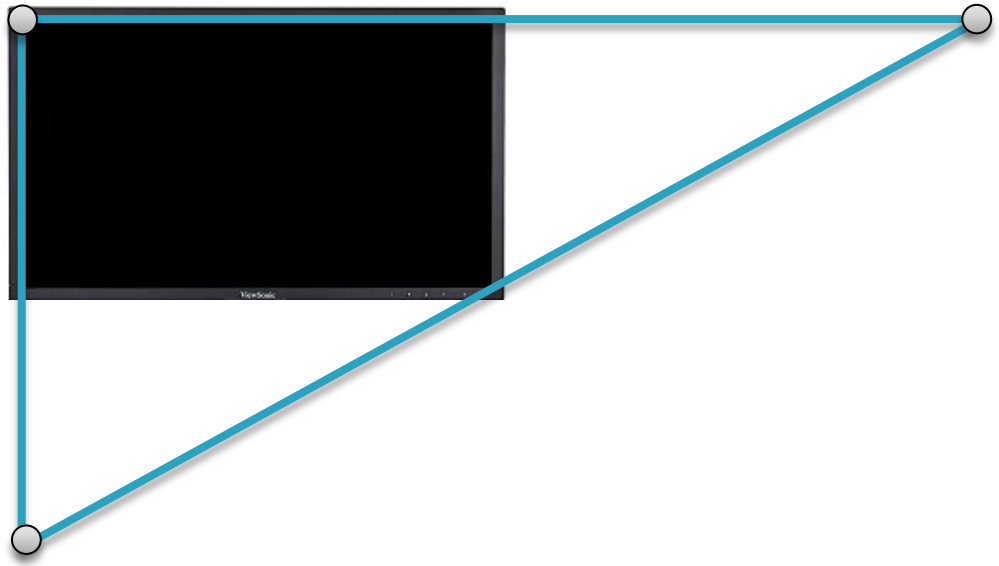
Rendering without geometry

- Draw() technically doesn't need buffers
  - Vertex/index buffers
- If no buffers are active
  - Can still call `context->Draw(N, 0);`
  - Vertex shader simply executes *N* times
- What good is this? What's our VS input?
  - Set up VS to take an unsigned int
  - Use `SV_VertexID` semantic – the index of the vertex

Full-screen geometry trick

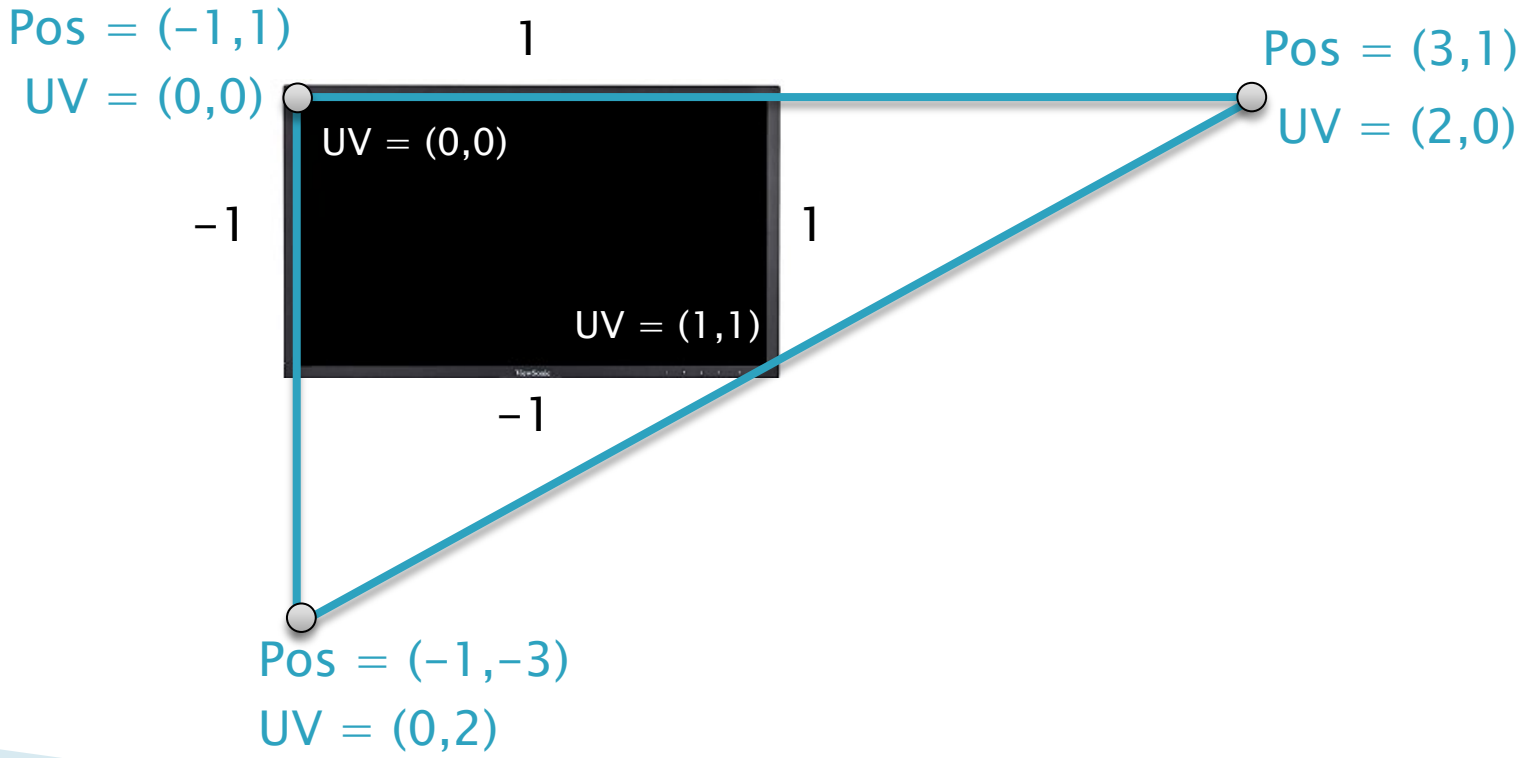
- If we have an index in a vertex shader
  - Perhaps we can generate a UV from it
  - And a corresponding screen position

- Make a single triangle big enough to fill the entire screen
  - Twice the width
  - Twice the height
  - UV's in the range (0,0) to (2,2)



Full-screen triangle

- We have a screen with these coordinates
- We need a triangle to cover the screen



Full-screen triangle VS

```
VertexToPixel main(uint id : SV_VERTEXID)
{
   VertexToPixel output;

   // Calculate the UV (0,0 to 2,2) via the ID
   // x = 0, 2, 0, 2, etc.
   // y = 0, 0, 2, 2, etc.
   output.uv = float2((id << 1) & 2, id & 2);

   // Convert uv to the (-1,1 to 3,-3) range for position
   output.position = float4(output.uv, 0, 1);
   output.position.x = output.position.x * 2 - 1;
   output.position.y = output.position.y * -2 + 1;

   return output;
}
```

Render to texture – Unbind

- Unbind render target's SRV
  - At end of the frame
  - Or you get DirectX run-time errors.
- The texture shouldn't be bound as both a render target and input to another shader
  - At the same time
  - Will be bound to one or the other for different steps
- DirectX doesn't like rendering to a texture that is currently bound somewhere else

Advanced: Multiple render targets

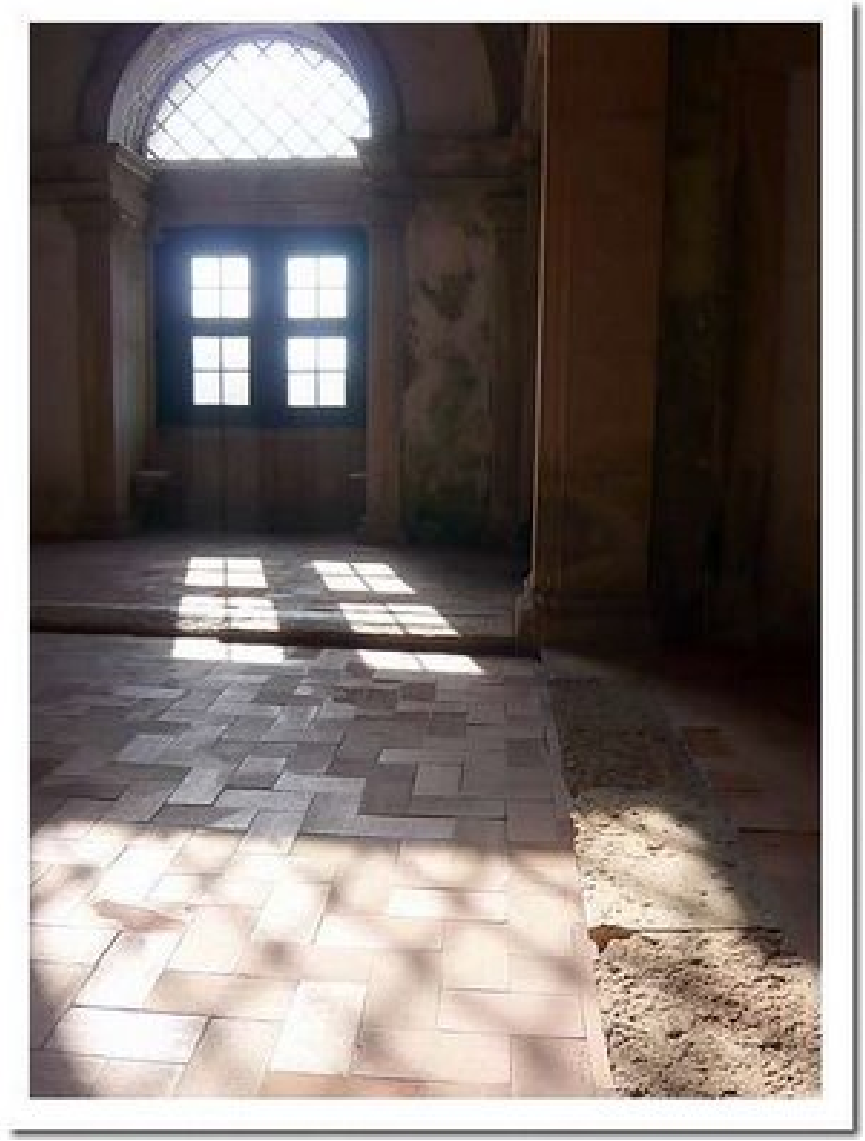
- Can set up to 8 *active* render targets at once
- A single pixel shader can output up to 8 colors
  - Return a struct of colors instead of a single float4
  - Each color is written to a separate target
- Necessary for advanced techniques like deferred rendering
  - Writing out information to be combined later

Bloom

A post-process for making things glow

Bloom / Glow

- Simulates light glowing or “bleeding” into other objects in the scene
- Like the windows in the photo to the right



Glow “baked” into texture

- Just looks washed out

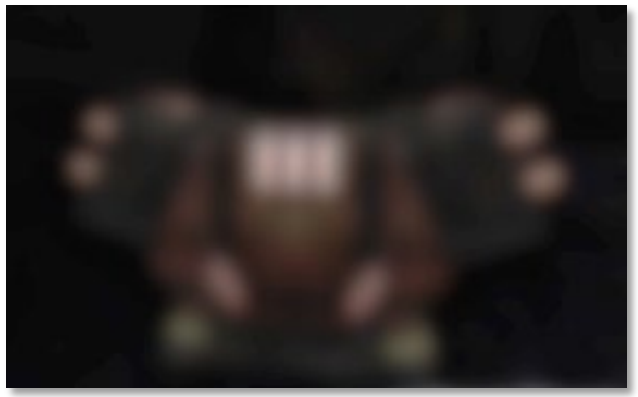
Glow as a post process

- Bright areas bleed into surrounding pixels
  - And outside the geometry

Bloom

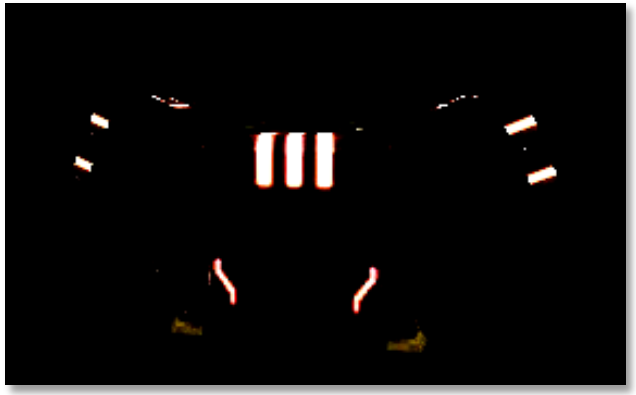
- Bright areas need to “glow”
  - Requires a blur
  - Hence: Post processing

- However, we can’t just blur the whole screen



Bloom – Extract pixels to blur

- We only want to blur brightest parts of the image
- Extract “bright” pixels of the image
  - An intermediate post-process step
  - Shader that returns either a color or black
  - Depending on the pixel’s “brightness”
- What does “bright” mean here?



Bloom – Calculating brightness?

- Pixels with a high average color? `(R+G+B)/3`
- Pixels with any channel above a threshold?
  - `if(R > 0.9 || G > 0.9 || B > 0.9)`
- Pixels with a high luminance?
  - `luminance = dot(color, float3(0.21f, 0.72f, 0.07f));`
- Pixels where color goes above 1?
  - Wait…

Color values above 1.0?

- Aren't color values clamped between 0-1?
  - Yes…with certain color formats
  - Like DXGI_FORMAT_R8G8B8A8_UNORM (Unsigned NORMalized)
  - But not ALL color formats
  - DXGI_FORMAT_R16G16B16A16_FLOAT can hold arbitrary float values
- If our render target is a FLOAT format
  - We can store colors outside the 0-1 range
  - Unnecessary for standard rendering
  - Fantastic for representing areas with LOTS of light
  - AKA pixels that should be bloom'd

Bloom steps

- 1. Render the scene to a texture
- 2. Extract the “bright” areas to another texture
- 3. Blur the “bright” texture
- 4. Combine original scene render and “blurred bright” texture
  - Sample both textures
  - Add results

Bloom step 1: Render to texture

- Render your scene to a texture
- Nothing else fancy

Bloom step 2: Extract brightness

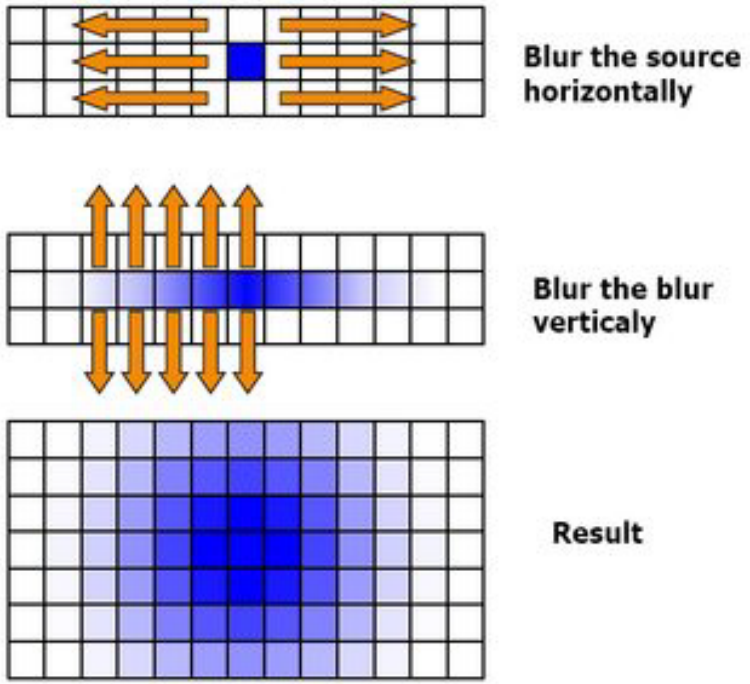
- Render step 1's result into a second texture
  - Only keep pixels above a certain brightness
  - Others can be black – we'll be additively blending

Bloom step 3: Blur bright pixels

- Blur the brightness texture
  - By rendering it to *yet another* render target
  - Usually half size of screen
- Which kind of blur?
  - Standard “box” blur: $O(n^2)$
  - Gaussian blur: $O(2n)$
- Half size target?
  - Downscale on render
  - Sampling results in upscale
  - An extra “free” blur

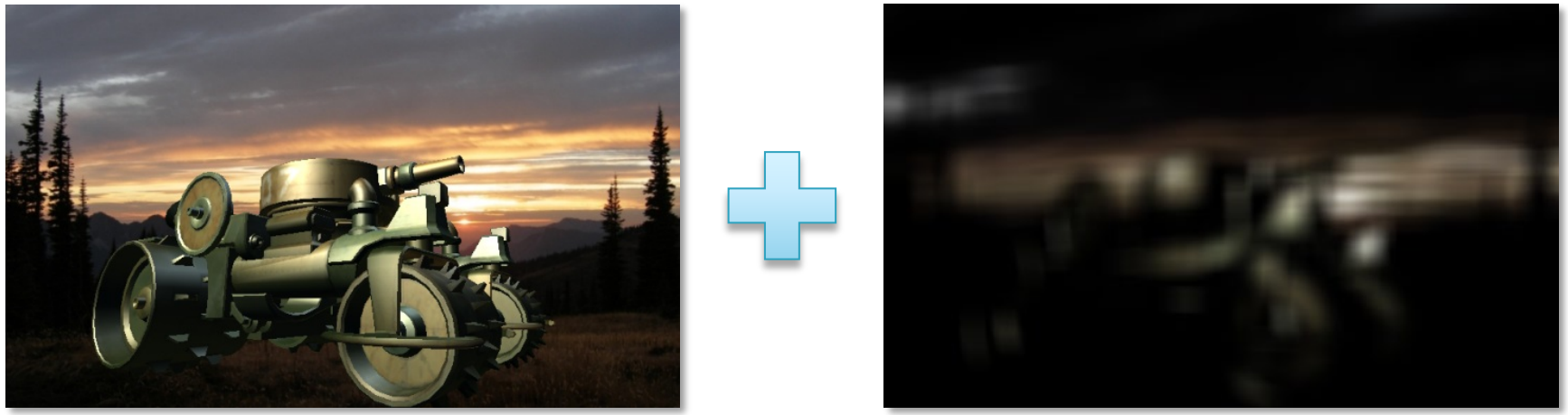
Gaussian – A separable blur

- Two-pass blur
  - Two pixel shaders
  - First blurs horizontally
  - Second blurs vertically
  - Result is full blur
- Way faster than a simple box blur!
- Gaussian kernel calculator

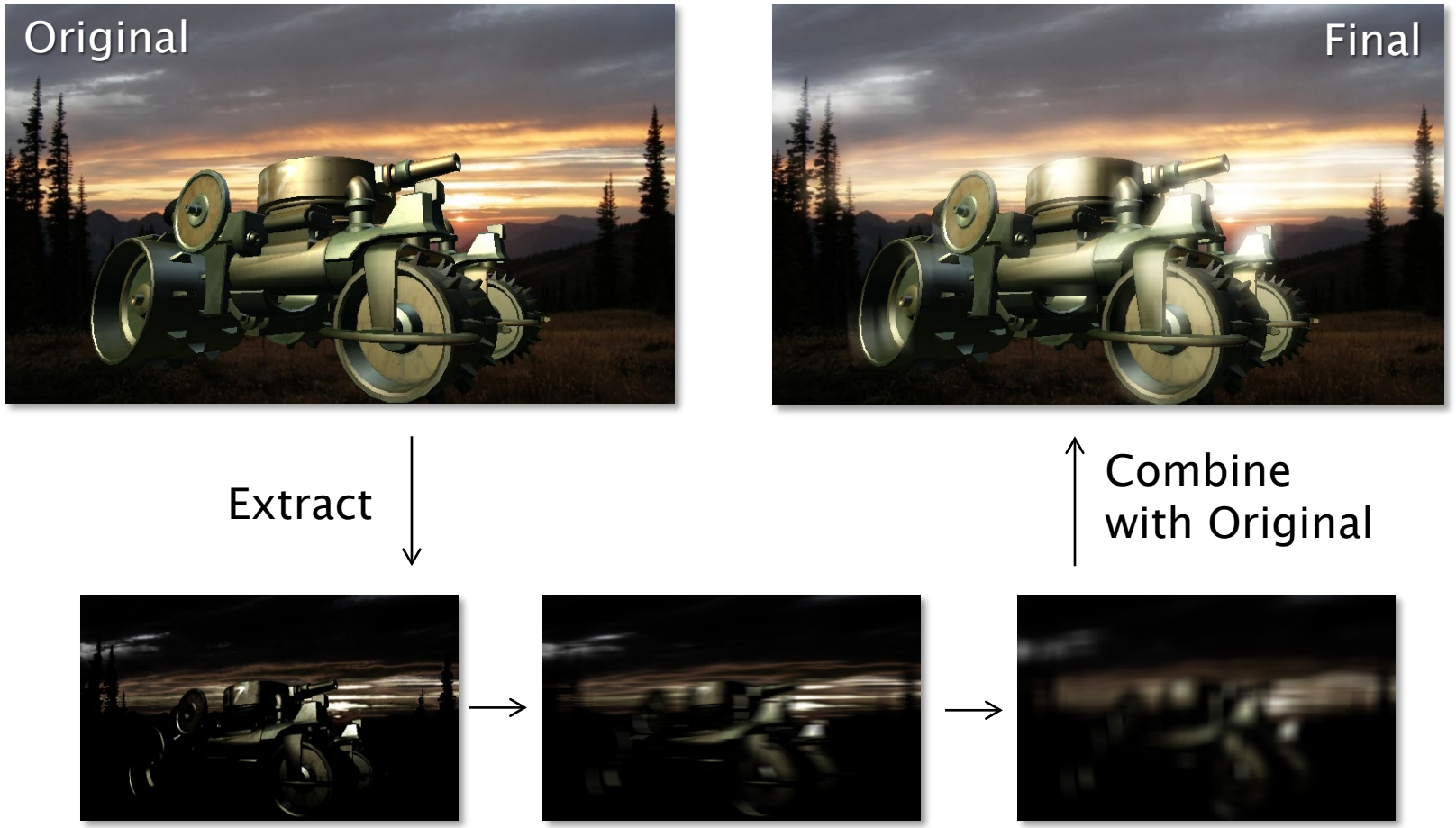


Bloom step 4: Combine

- Puts the pieces together on the screen
  - Combine original & blurred brightness:
  - `finalColor = originalTexture + brightTexture;`

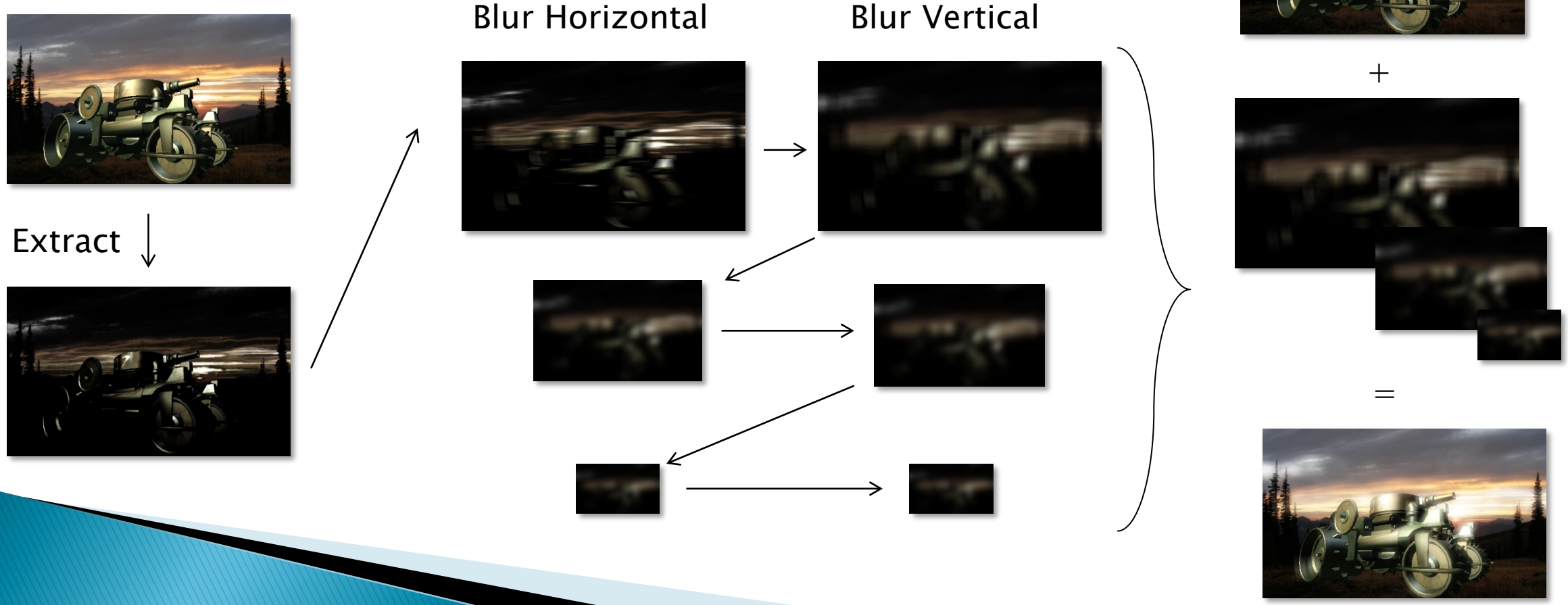


Overall process for basic bloom



Advanced bloom

- Multiple levels of blur all added together



Bloom should be subtle

- And only apply to exceptionally bright areas

Witcher 3

Unless you need a strong glow

But don’t pull an Oblivion