Abertay University

Post Processing

CMP301 Graphics Programming with Shaders

This week

- Post processing
- Multi pass rendering
- Blur
  - Box blur
  - Gaussian blur

What is it?

- Post processing
  - Manipulate the image buffer before displaying on the screen
  - Predominately handled by the pixel / fragment shader
  - Allows the creation of effects that require awareness of the entire scene / image
  - Such as …

What can we do?

- Blur
- Motion blur
- Bokeh (shallow depth of field)
- Glow
- Lens flare
- Bloom
- Soften shadows
- Under water effects
- Colour filtering
    - Contrast, saturations, grey scale
- Anti-aliasing
- Etc
- etc

Post processing

- Manipulating EVERY pixel in the render
  - This requires many Render-to-Textures
- “Normally” we render our scene to the frame buffer
  - Last class when we looked at render to texture
- For post processing we render the whole scene, in it’s entirety to a texture
  - We then process this image with a collection of shaders before rendering to the screen
  - Here we handle the scene as a whole
    - During normal rendering we are dealing with objects individually

Multi pass rendering

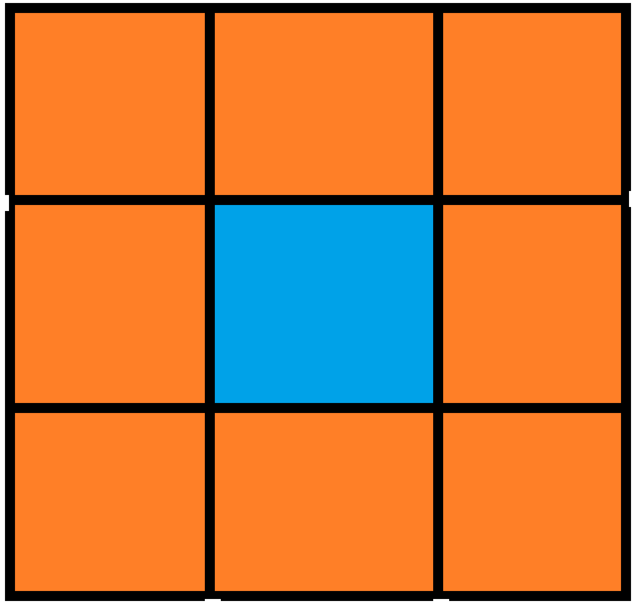
- The process of rendering different attributes of your scene separately
  - Your scene has to be rendered multiple times to create the final scene
  - Many uses and post processing is one of them
    - Specular pass (isolates specular highlights)
    - Reflection pass
    - Shadow pass
    - Lighting pass
    - Effects pass
    - Depth pass
    - etc

Blur an example

- I'm not talking about every form of post processing
- We will use blur as an example
  - Blur is the base for most forms of post processing
  - Will give a good introduction to the technique
- There are many many types of blur
  - The main differences between blurring techniques is how you combine the pixel colours

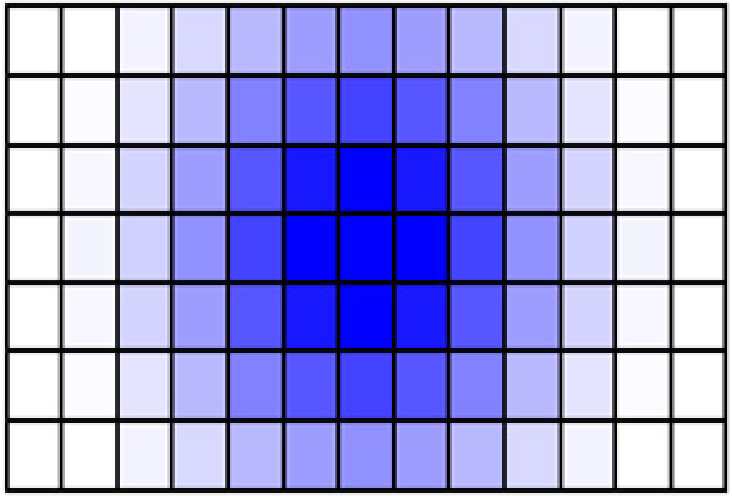
Box blur

- The simplest of blurs
- Blur colour with surrounding colour values
- Equal weighting for every colour
- Fast and easy
- Could look much better



Gaussian blur

- Same idea as box blur
  - Combine colour values from surrounding pixels
- Major difference
  - Colour values are weighted
  - The closer to the centre the more effect on the final colour they have
  - Usually more surrounding pixels are used



Neighbours

- The number of neighbours is one aspect that can control the blur
  - 0 neighbours
  - 3 neighbours
  - 10 neighbours

Weightings

- The amount each neighbour influences the final colour is controlled by weightings
  - Weightings should add up to 1
    - If not the resulting image will be darker or lighter
  - Neighbours further away have less weighting
    - They have less influence over the final colour, but still play a part
  - Like the neighbours these weightings can have a great effect of the final blur
  - You can choose any weightings you want
    - In the case of the Gaussian blur, the weightings are selected based on a Gaussian function

Challenges and solutions

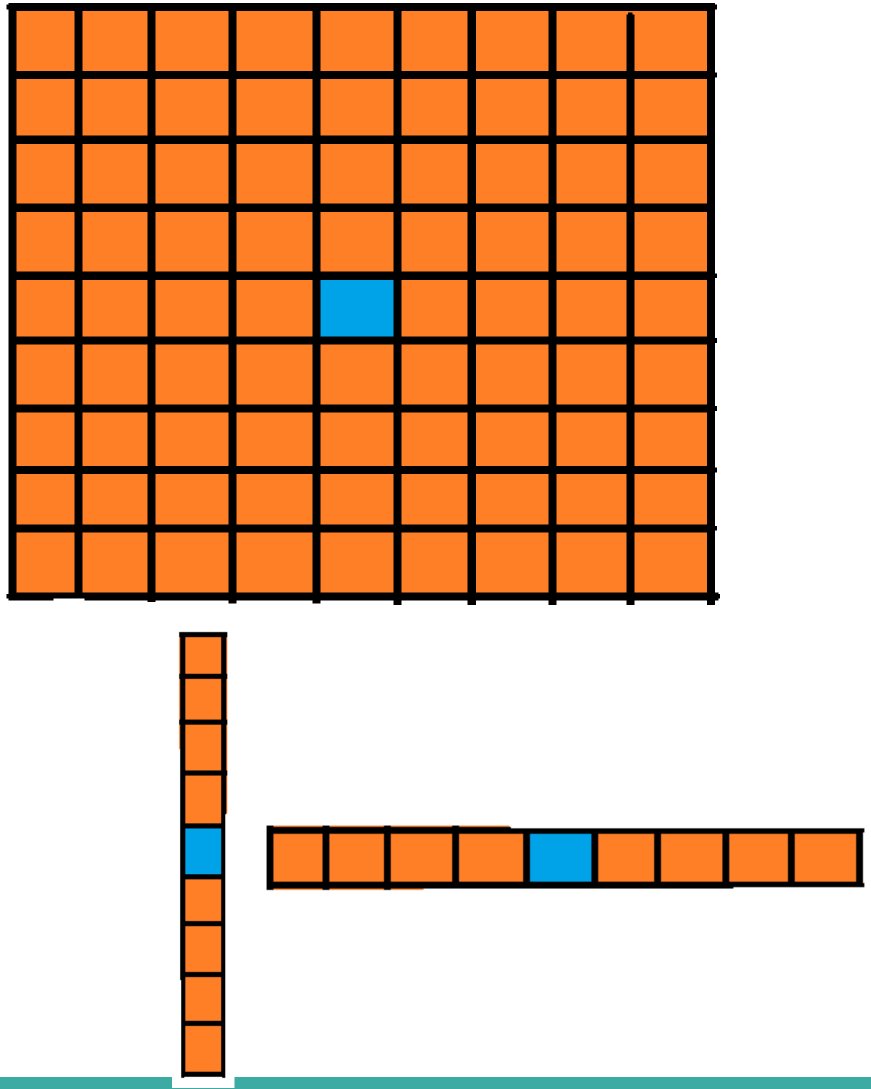
- The major challenge with any post processing is the large number of calculations being done
  - Rendering the scene multiple times
  - Processing every pixel
  - Multiple sample per pixel
- Solutions
  - Down scale for processing
  - Divide blur into horizontal and vertical blurring

Down scaling

- Why?
  - Greatly reduces the number of pixel we have to process
  - For example, half width and half height
    - Only a ¼ of the pixels to process
- Have to up scale the processed image before final render
- Helps with blurring
  - Just down scaling and then up scaling provides a simple weak blur

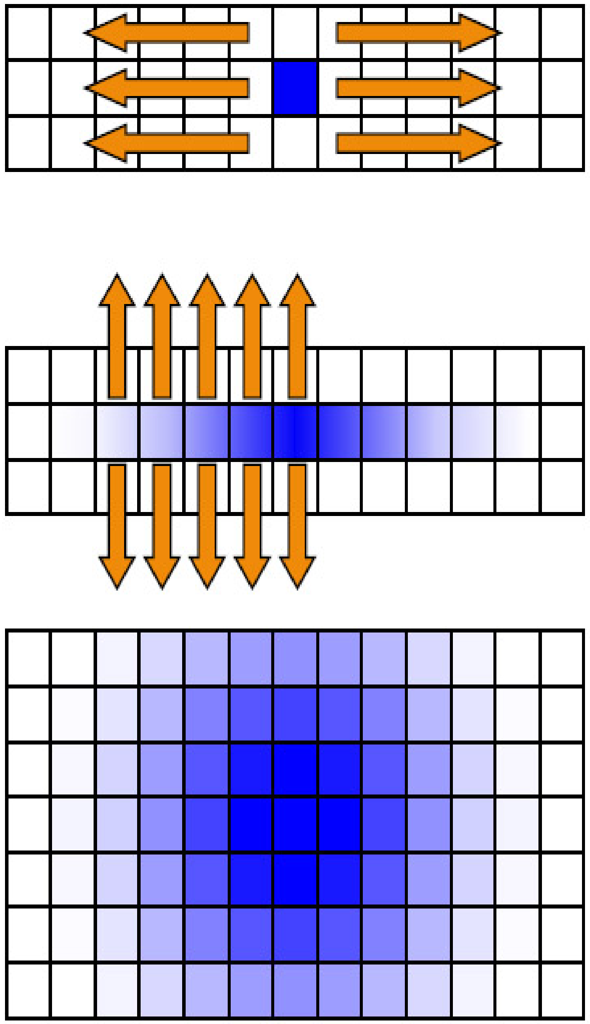
Divide the processing

- Separate horizontal and vertical blurring vs all-in-one blur
- 9x9 blur matrix
- All in one blur requires 81 samplers per pixel
- Horizontal requires 9 samples
- Vertical requires 9 samples
  - Total of 18 samples



Divide the process

- Perform two blurs
  - A horizontal blur
    - For the current pixel get the neighbouring colours (left and right)
    - Based on weightings they are combined to give a final colour
    - This colour is a combination of the original pixel and small amounts its neighbours
  - A vertical blur
    - Same idea as horizontal but up and down
    - Each pixel colour is combined with top and bottom neighbours



The whole process

- Render scene to texture
- Down sample texture
- Apply horizontal blur
- Apply vertical blur
- Up sample texture
- Render image (using 2D object / orthoMesh)

- Almost every stage requires a render to texture
  - This is why post processing is so resource expensive
  - You can output most stages to the screen to see the effect of each stage

Examples

- Box blur
- Gaussian blur

Box blur

- Overview
  - 2 sets shaders
  - 1 render target
  - 1 ortho mesh
- Render
  - Render scene to texture
  - Render that texture with blur shader

Box blur vs

```
cbuffer MatrixBuffer : register(cb0)
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};

cbuffer ScreenSizeBuffer : register(cb1)
{
    float screenWidth;
    float screenHeight;
    float2 padding;
};

struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};
```

Box blur vs

```
struct OutputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float2 texCoord1 : TEXCOORD1;
    float2 texCoord2 : TEXCOORD2;
    float2 texCoord3 : TEXCOORD3;
    float2 texCoord4 : TEXCOORD4;
    float2 texCoord5 : TEXCOORD5;
    float2 texCoord6 : TEXCOORD6;
    float2 texCoord7 : TEXCOORD7;
    float2 texCoord8 : TEXCOORD8;
};
```

Box blur vs

```
OutputType main(InputType input)
{
    OutputType output;
    float texelWidth, texelHeight;

    input.position.w = 1.0f;
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    output.tex = input.tex;

    // Determine the floating point size of a texel for a screen with this specific width.
    texelWidth = 1.0f / screenWidth;
    texelHeight = 1.0f / screenHeight;
```

Box blur vs

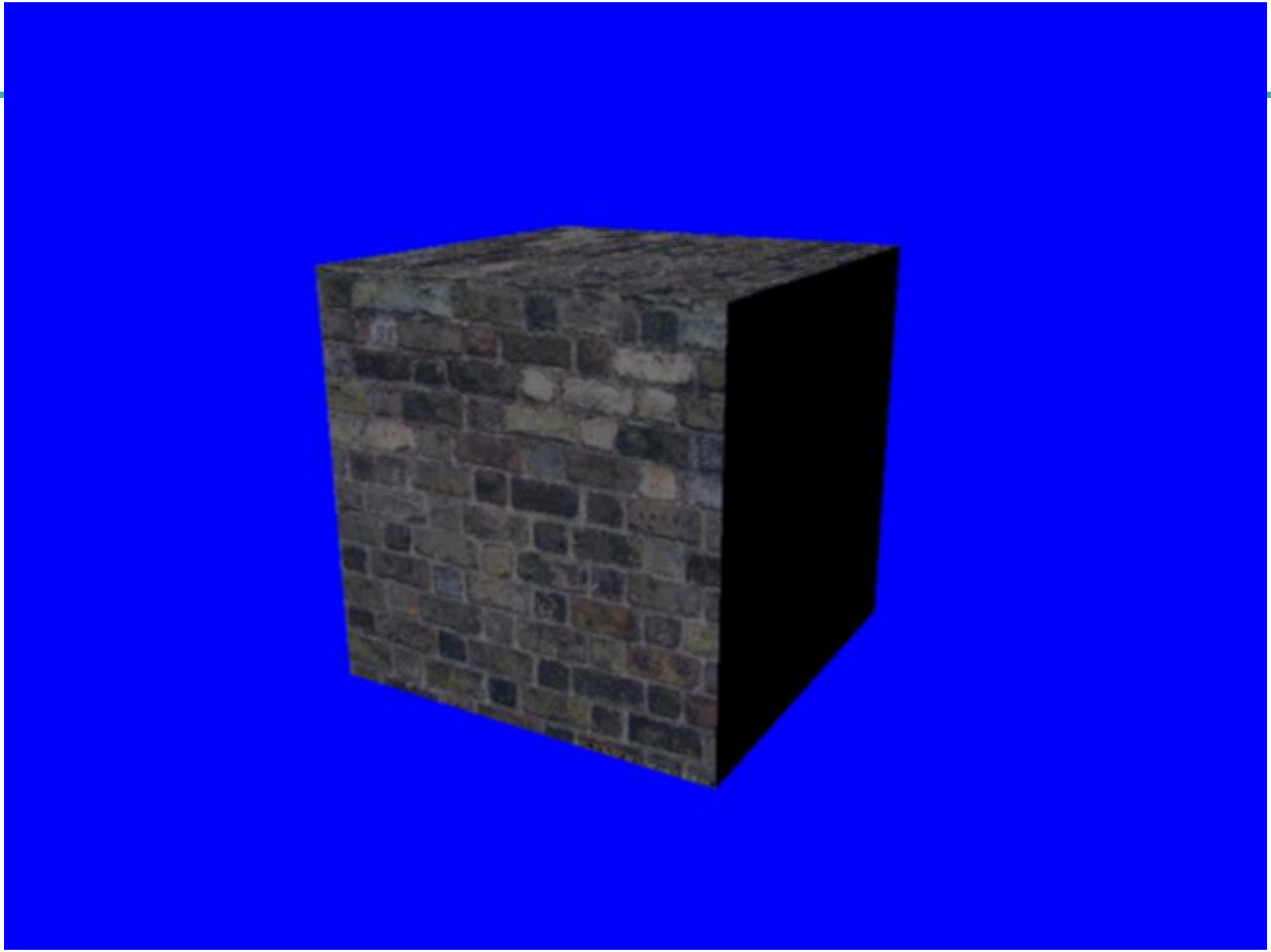
```
    // Create UV coordinates for the pixel and its four horizontal neighbors on either side.
    output.texCoord1 = input.tex + float2(-texelWidth, -texelHeight);
    output.texCoord2 = input.tex + float2(0.0f, texelHeight);
    output.texCoord3 = input.tex + float2(texelWidth, -texelHeight);
    output.texCoord4 = input.tex + float2(-texelWidth, 0.0f);
    output.texCoord5 = input.tex + float2(texelWidth, 0.0f);
    output.texCoord6 = input.tex + float2(-texelWidth, texelHeight);
    output.texCoord7 = input.tex + float2(0.0f, texelHeight);
    output.texCoord8 = input.tex + float2(texelWidth, texelHeight);

    return output;
}
```

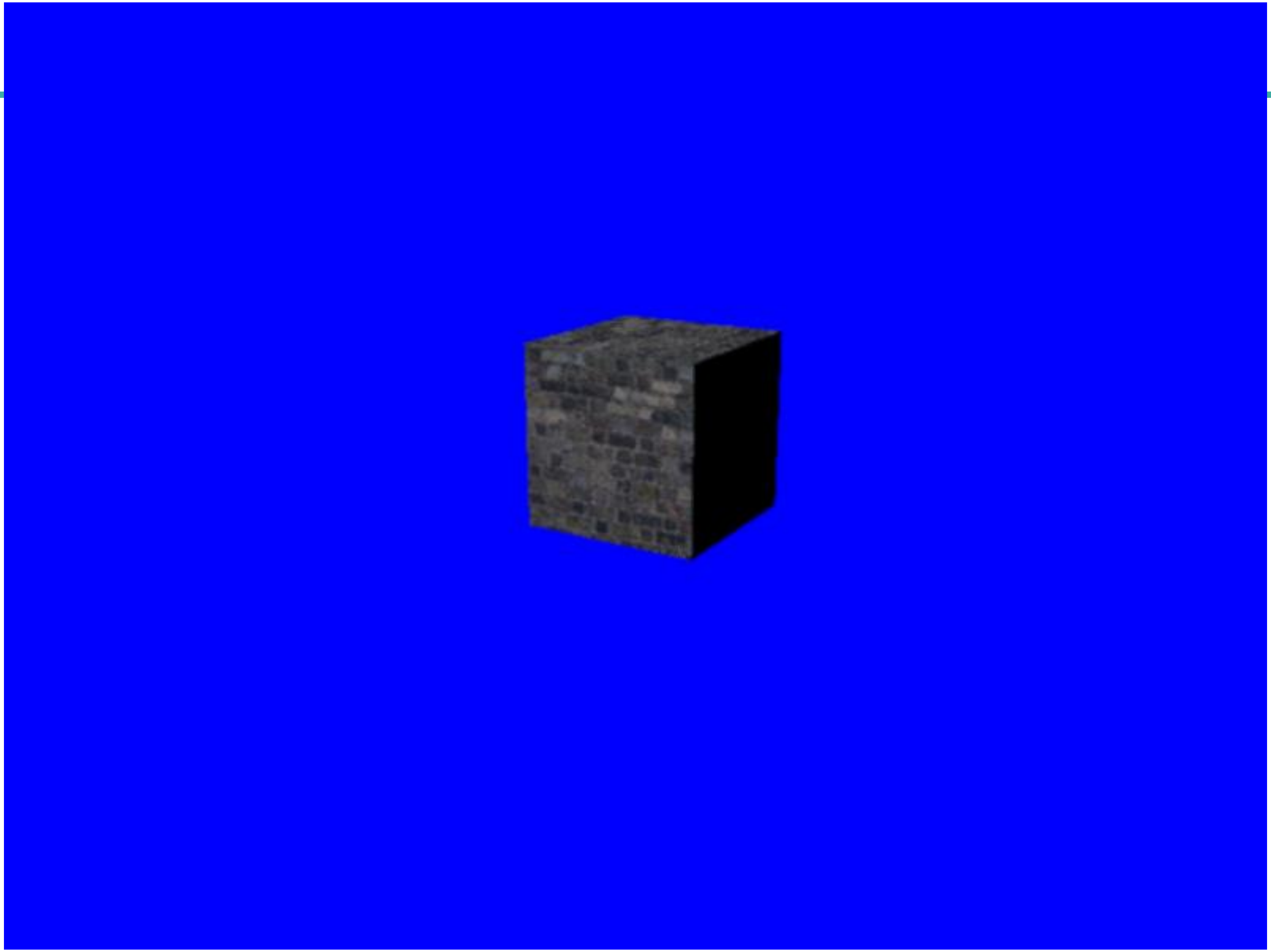
Box blur ps

```
Texture2D shaderTexture : register(t0);
SamplerState SampleType : register(s0);

struct InputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float2 texCoord1 : TEXCOORD1;
    float2 texCoord2 : TEXCOORD2;
    float2 texCoord3 : TEXCOORD3;
    float2 texCoord4 : TEXCOORD4;
    float2 texCoord5 : TEXCOORD5;
    float2 texCoord6 : TEXCOORD6;
    float2 texCoord7 : TEXCOORD7;
    float2 texCoord8 : TEXCOORD8;

};
```

Box blur ps

```
float4 main(InputType input) : SV_TARGET
{
    float4 colour;
    // Initialize the colour to black.
    colour = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Add the nine surrounding pixels to the colour.
    colour += shaderTexture.Sample(SampleType, input.tex);
    colour += shaderTexture.Sample(SampleType, input.texCoord1);
    colour += shaderTexture.Sample(SampleType, input.texCoord2);
    colour += shaderTexture.Sample(SampleType, input.texCoord3);
    colour += shaderTexture.Sample(SampleType, input.texCoord4);
    colour += shaderTexture.Sample(SampleType, input.texCoord5);
    colour += shaderTexture.Sample(SampleType, input.texCoord6);
    colour += shaderTexture.Sample(SampleType, input.texCoord7);
    colour += shaderTexture.Sample(SampleType, input.texCoord8);

    colour /= 9.0f;

    // Set the alpha channel to one.
    colour.a = 1.0f;

    return colour;
}
```





Gaussian blur

- Overview
  - 4 sets shaders
  - 5 render targets
  - 2 ortho meshs
- Render process
  - Render scene to texture
  - Down scale
  - Horizontal blur
  - Vertical blur
  - Up scale
  - Render to frame buffer

Render to textures

- We need at least 5 render to textures
  - Whole Scene
  - Down sample
  - Horizontal blur
  - Vertical blur
  - Up sample

Rendering

- Divided up the rendering into stages
  - Render scene to texture()
  - Down sample texture()
  - Render horizontal blur to texture()
  - Render vertical blur to texture()
  - Up sample texture()
  - Render 2D texture scene()

Code

```
bool App::Render()
{
	renderToTexture();

	downSample();

	horizontalBlur();

	verticalBlur();

	upSample();

	renderScene();

	return true;
}
```

Down sample ()

- This is where the image is shrunk before post processing takes place
- Set the render target
  - To our down sample texture
- Using the ortho matrix and Z buffering turned off
  - Render with a simple texture shader (no need for lighting)
- We render the scene texture to an ortho mesh
  - This *renderTexture* is half the window/screen size
  - To fit the object the renderer down samples the image for us
    - Nice and quick
  - downSampleTexture = new RenderTexture(renderer->getDevice(), screenWidth/2, screenHeight/2, SCREEN_NEAR, SCREEN_DEPTH);

No blur

Down sample

Horizontal blur ()

- Now we perform the horizontal blur
- To do so we render to a texture
  - Surprise!
  - Horizontal blur texture
- We also render to a down sample target window
  - No Z buffer
- Using our horizontal blur shaders
  - Here is where we do the real processing
- The shader requires
  - Position
  - Texture coordinates
  - Additional constant buffer with the width of the object

Horizontal blur vs

```
cbuffer MatrixBuffer : register(cb0)
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};

cbuffer ScreenSizeBuffer : register(cb1)
{
    float screenWidth;
    float3 padding;
};

struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
};
```

Horizontal blur vs

```
struct OutputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float2 texCoord1 : TEXCOORD1;
    float2 texCoord2 : TEXCOORD2;
    float2 texCoord3 : TEXCOORD3;
    float2 texCoord4 : TEXCOORD4;
    float2 texCoord5 : TEXCOORD5;
};


OutputType main(InputType input)
{
    OutputType output;
    float texelSize;

    // Change the position vector to be 4 units for proper matrix calculations.
    input.position.w = 1.0f;
```

Horizontal blur vs

```
    // Calculate the position of the vertex against the world, view, and projection matrices.
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Store the texture coordinates for the pixel shader.
    output.tex = input.tex;

    // Determine the floating point size of a texel for a screen with this specific width.
    texelSize = 1.0f / screenWidth;

    // Create UV coordinates for the pixel and its four horizontal neighbours on either side.
    output.texCoord1 = input.tex + float2(texelSize * -2.0f, 0.0f);
    output.texCoord2 = input.tex + float2(texelSize * -1.0f, 0.0f);
    output.texCoord3 = input.tex + float2(texelSize *  0.0f, 0.0f);
    output.texCoord4 = input.tex + float2(texelSize *  1.0f, 0.0f);
    output.texCoord5 = input.tex + float2(texelSize *  2.0f, 0.0f);

    return output;
}
```

Horizontal pixel shader

```
Texture2D shaderTexture : register(t0);
SamplerState SampleType : register(s0);

struct InputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float2 texCoord1 : TEXCOORD1;
    float2 texCoord2 : TEXCOORD2;
    float2 texCoord3 : TEXCOORD3;
    float2 texCoord4 : TEXCOORD4;
    float2 texCoord5 : TEXCOORD5;

};

float4 main(InputType input) : SV_TARGET
{
    float weight0, weight1, weight2;
    float4 colour;
```

Horizontal pixel shader

```
    // Create the weights that each neighbour pixel will contribute to the blur.
    weight0 = 0.4062f;
    weight1 = 0.2442f;
    weight2 = 0.0545f;

    // Initialize the colour to black.
    colour = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Add the nine horizontal pixels to the colour by the specific weight of each.
    colour += shaderTexture.Sample(SampleType, input.texCoord1) * weight2;
    colour += shaderTexture.Sample(SampleType, input.texCoord2) * weight1;
    colour += shaderTexture.Sample(SampleType, input.texCoord3) * weight0;
    colour += shaderTexture.Sample(SampleType, input.texCoord4) * weight1;
    colour += shaderTexture.Sample(SampleType, input.texCoord5) * weight2;

    // Set the alpha channel to one.
    colour.a = 1.0f;

    return colour;
}
```

No blur

Just horizontal blur

Vertical blur

- Render the down sample window again
- This time using the horizontal blur texture
- Render with vertical blur shaders
  - This takes the horizontal blur and adds the vertical blur
  - No Z buffer again

Vertical blur vs

```
cbuffer MatrixBuffer : register(cb0)
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};

cbuffer ScreenSizeBuffer : register(cb1)
{
    float screenHeight;
    float3 padding;
};


struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
};
```

Vertical blur vs

```
struct OutputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float2 texCoord1 : TEXCOORD1;
    float2 texCoord2 : TEXCOORD2;
    float2 texCoord3 : TEXCOORD3;
    float2 texCoord4 : TEXCOORD4;
    float2 texCoord5 : TEXCOORD5;
};

OutputType main(InputType input)
{
    OutputType output;
    float texelSize;

    // Change the position vector to be 4 units for proper matrix calculations.
    input.position.w = 1.0f;
```

Vertical blur vs

```
    // Calculate the position of the vertex against the world, view, and projection matrices.
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Store the texture coordinates for the pixel shader.
    output.tex = input.tex;

    // Determine the floating point size of a texel for a screen with this specific height.
    texelSize = 1.0f / screenHeight;

    // Create UV coordinates for the pixel and its four vertical neighbors on either side.
    output.texCoord1 = input.tex + float2(0.0f, texelSize * -2.0f);
    output.texCoord2 = input.tex + float2(0.0f, texelSize * -1.0f);
    output.texCoord3 = input.tex + float2(0.0f, texelSize *  0.0f);
    output.texCoord4 = input.tex + float2(0.0f, texelSize *  1.0f);
    output.texCoord5 = input.tex + float2(0.0f, texelSize *  2.0f);


    return output;
}
```

Vertical blur ps

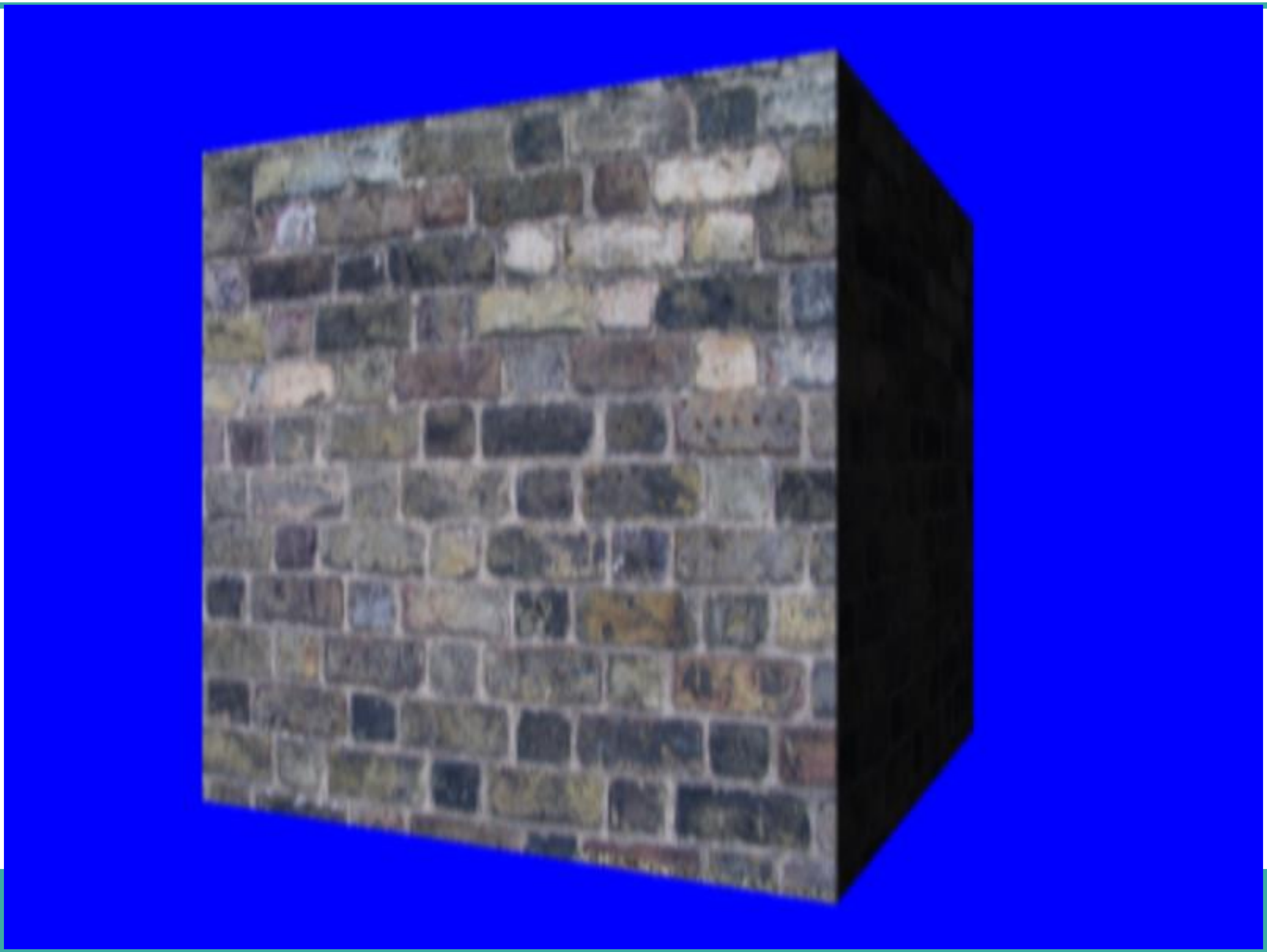
```
Texture2D shaderTexture : register(t0);
SamplerState SampleType : register(s0);

struct InputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float2 texCoord1 : TEXCOORD1;
    float2 texCoord2 : TEXCOORD2;
    float2 texCoord3 : TEXCOORD3;
    float2 texCoord4 : TEXCOORD4;
    float2 texCoord5 : TEXCOORD5;
};

float4 main(InputType input) : SV_TARGET
{
    float weight0, weight1, weight2;
    float4 colour;
```

Vertical blur ps

```
// Create the weights that each neighbour pixel will contribute to the blur.
    weight0 = 0.4062f;
    weight1 = 0.2442f;
    weight2 = 0.0545f;

    // Initialize the colour to black.
    colour = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Add the nine vertical pixels to the colour by the specific weight of each.
    colour += shaderTexture.Sample(SampleType, input.texCoord1) * weight2;
    colour += shaderTexture.Sample(SampleType, input.texCoord2) * weight1;
    colour += shaderTexture.Sample(SampleType, input.texCoord3) * weight0;
    colour += shaderTexture.Sample(SampleType, input.texCoord4) * weight1;
    colour += shaderTexture.Sample(SampleType, input.texCoord5) * weight2;

    // Set the alpha channel to one.
    colour.a = 1.0f;

    return colour;
}
```

No blur

Just vertical blur



Up sampling

- We aren’t done yet
- We have a blur
  - Combination of both horizontal and vertical
  - I just haven’t shown you it yet ☺
- But our image is still half the size
  - We need to make it bigger
- Reverse down sampling
  - Render using another render texture and ortho mesh set to full size
  - Render the texture we have after performing vertical blur
  - This up samples our texture
  - Still no Z buffer

Finally

- After up sampling we finally have a render to texture that is
  - Full size
  - With our post processing done
- Now we can finally render it to the screen
  - 5 render to textures later
- We render another up sample target window with the texture output from the up sampling
  - But this time render to the buffer
  - Using a texture shader
  - No Z buffer

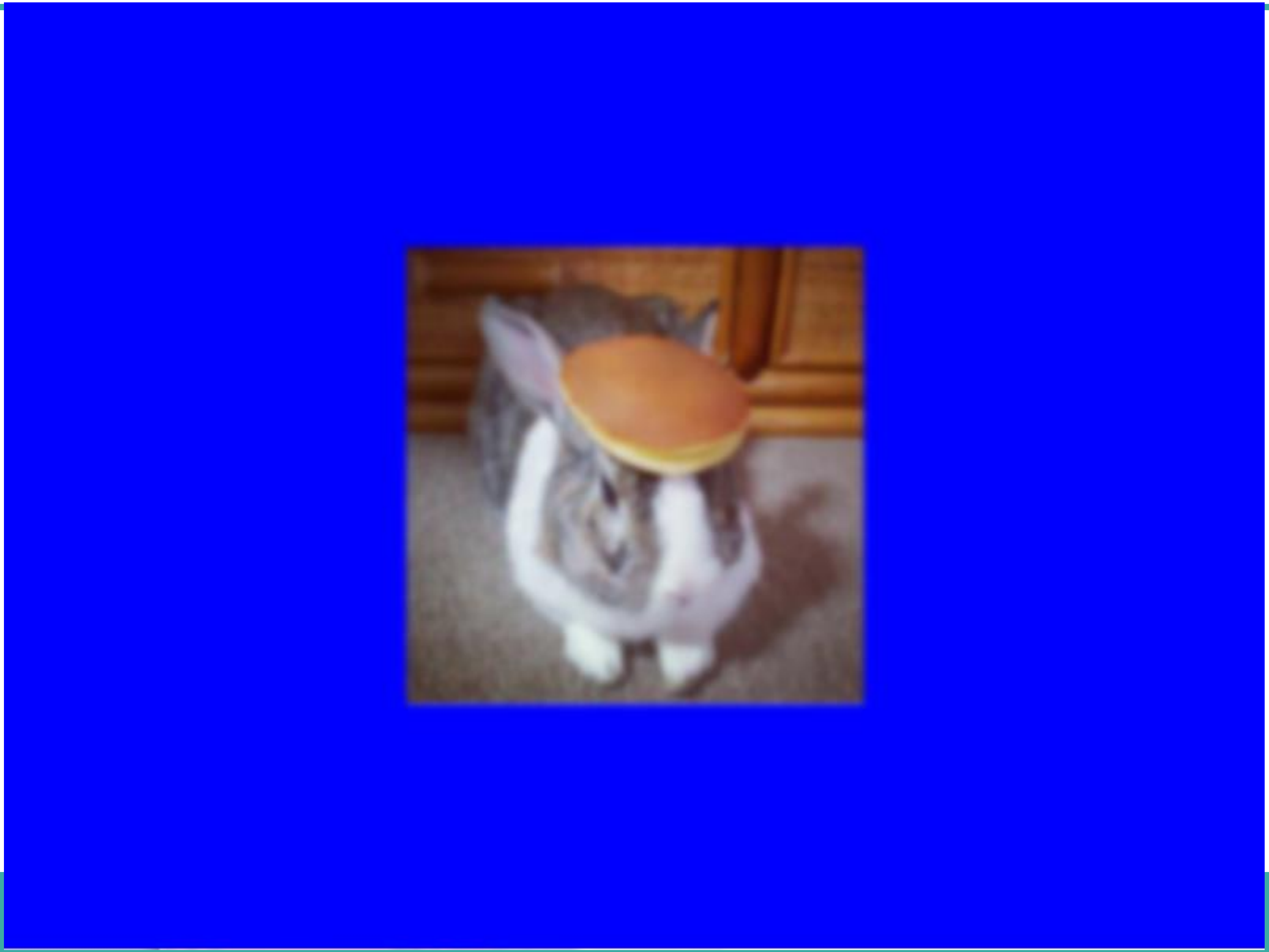
No blur

Full blur

Things to think about

- Changing the number of neighbours
- Changing the weightings
- Running the blur twice
  - Blur that blur
- Other types of blur
  - Box blur
  - Motion blur
  - Zoom blur
  - Spin blur
  - Variable blur
  - Bokeh

2 neighbours either side



With 4 neighbours either side

Motion blur

- Kind of done this
  - Single directional blur
  - Similar to a Horizontal blur but at an angle
  - Usually the colours only move in a single direction

Spin and zoom

Variable blur

Casting rays / God rays

In the labs

- Building box blur
- And Gaussian blur examples