COMP220: Graphics & Simulation

08: Post-processing

Learning outcomes

- **Understand** Post-processing in computer graphics
- **Implement** graphical effects from COMP120 in realtime
- **Implement** other Post-processing effects in your application

Introduction

What is Post-Processing?

- Is a series of techniques that are carried out after a scene has been rendered
- Traditionally this could only be done as an offline process
- With the advent of faster GPUs and programmable shaders, we could implement some of the effects in real time
- The key to the effect is to switch from drawing to the back buffer to a texture
- This texture will contain the scene
- We then use a shader to implement a post-processing effect
- We then map the processed texture onto a full screen quad, which is rendered to the backbuffer

Rendering To Texture

Brief Overview

1. Create a Texture of the required dimensions
2. Create Depth Buffer Object
3. Create a Frambuffer Object (FBO)
4. Bind the texture and the Depth Buffer Object into the FBO
5. Bind the FBO to the pipeline
6. Render the scene to the new framebuffer

Creating a Texture

```
//The texture we are going to render to
GLuint renderTextureID;
glGenTextures(1,&renderTextureID);

//Bind Texture
glBindTexture(GL_TEXTURE_2D, renderTexture);

//fill with empty data
glTexImage2D(GL_TEXTURE_2D,0, GL_RGB, 840,680,0, ↩
    GL_RGB, GL_UNSIGNED_BYTE,0);

//Add any texture states (filtering etc)
```

Creating Depth Buffer Object

```
//The depth buffer
GLuint depthBufferID;
glGenRenderbuffers(1,&depthBufferID);

//Bind the depth buffer
glBindRenderbuffer(GL_RENDERBUFFER, depthBufferID) ↩
    ;
//Set the foremat of the depth buffer
glRenderbufferStorage(GL_RENDERBUFFER, ↩
    GL_DEPTH_COMPONENT,840,680);
```

Creating Frame Buffer

```
//The frambuffer
GLuint frameBufferID;
glGenFramebuffers(1,&frameBufferID);
```

Bind Texture and Depth Buffer

```
//Bind the framebuffer
glBindFramebuffer(GL_FRAMEBUFFER,frameBufferID);

//Bind the texture as a colour attachment 0 to the ↩
     active framebuffer
glFramebufferTexture(GL_FRAMEBUFFER,  ↩
    GL_COLOR_ATTACHMENT0, renderTextureID, 0);

//Bind the depth buffer as a depth attachment
glFramebufferRenderbuffer(GL_FRAMEBUFFER,  ↩
    GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,  ↩
    depthBufferID);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) !=  ↩
    GL_FRAMEBUFFER_COMPLETE)
{
    //error message!
}
```

Render to Framebuffer

```
//Bind the framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, frameBufferID);

//Drawn everything as normal!
```

Using Our Texture

Brief Overview

- Now we have our scene store on a texture
- We need to map this texture onto a surface
- This is usually a screen-aligned quad, but it can be any 3D object!
- In the fragment shader, we can do some processing

Steps

1. Create a Vertex Buffer Object (VBO) for our quad
2. Create a Vertex Array Object (VAO)
3. Load in a ‘pass through‘ Vertex Shader and a Fragment shader which takes in a texture
4. Render the quad and send across the texture that was bound to the framebuffer

Creating our Vertex Buffer Object

```
float vertices[] =
{
    -1, -1,
    1, -1,
    -1, 1,
    1, 1,
};

GLuint screenQuadVBOID;
glGenBuffers(1, &screenQuadVBOID);
glBindBuffer(GL_ARRAY_BUFFER, screenQuadVBOID);
glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(float),
    vertices, GL_STATIC_DRAW);
```

Creating our Vertex Array

```
GLuint screenVAOID;
glGenVertexArrays(1, &screenVAOID);
glBindVertexArray(screenVAOID);
glBindBuffer(GL_ARRAY_BUFFER, screenQuadVBOID);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, ↩
    NULL);
```

Pass Through Vertex Shader

```
#version 330 core

layout(location=0) in vec2 vertexPosition;

out vec2 textureCoords;

void main()
{
    //Calculate Texture Coordinates for the Vertex
    textureCoords = (vertexPosition + 1.0) / 2.0;
    gl_Position = vec4(vertexPosition, 0.0, 1.0);
}
```

Example Fragment Shader

```
#version 330 core

out vec4 color;
in vec2 textureCoords;

uniform sampler2D texture0;

void main()
{
    //Read the texture and do some processing!
    color = texture(texture0, textureCoords);
}
```

Rendering

```
glBindFramebuffer(GL_FRAMEBUFFER,0);
glBindVertexArray(screenVAOID);

//Bind our Postprocessing Program

//Send across any values to the shader

//Draw the quad!
```

Live Coding

Post Processing Setup

COMP120 Refresher

Overview

- Some of the algorithm you learned in COMP120 can be applied to GLSL
- The only difference is we don't use an array to access the pixel data
- Remember that a Texture Lookup uses the texture coordinates passed to the fragment shader
- Additionally, the fragment shader only processes a one fragment at a time

COMP120 Techniques

- Colour Replacement
- Luminance calculation
- Colour Correction
- Black & White and Sepia Tone
- Edge Detection

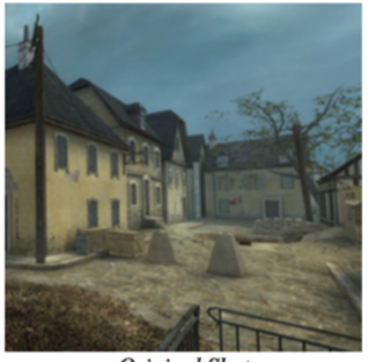
Implementing these effects

- Remember we can't access a pixel array, we use the texture coordinates
- If we want to access a pixel adjacent to the current one, we can offset the current texture coordinates
- This is especially useful for edge detection
- Lastly, GLSL has lots of inbuilt functions (e.g distance) which can aid in creating post-processing effects

Other Post Processing Techniques

Colour Correction

- Process of taking an image, convert each colour in the image to a some other colour
- Mimic a specific film stock, provide coherent look or provide a mood

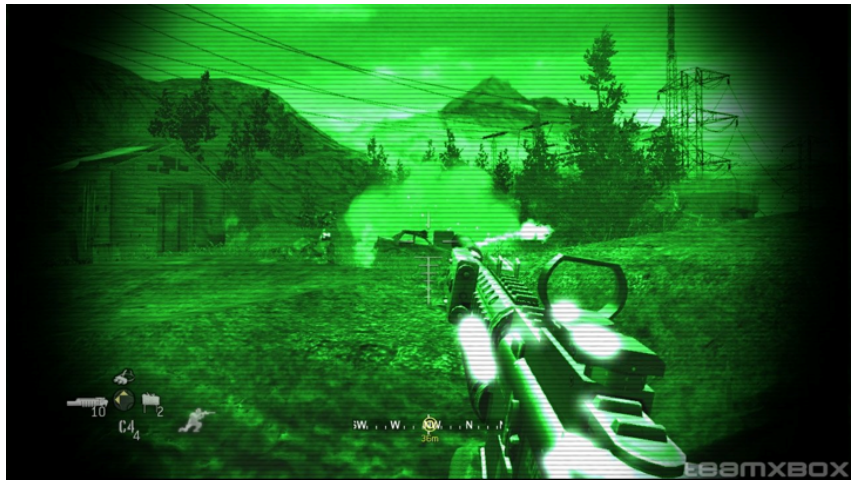


Original Shot

Day-for-Night Color Corrected shot

Colour Correction Again

- Depending on the technique this could involve manipulating the colours using a Filter Kernel
- Or we could use a colour palette (see Colour Grading)

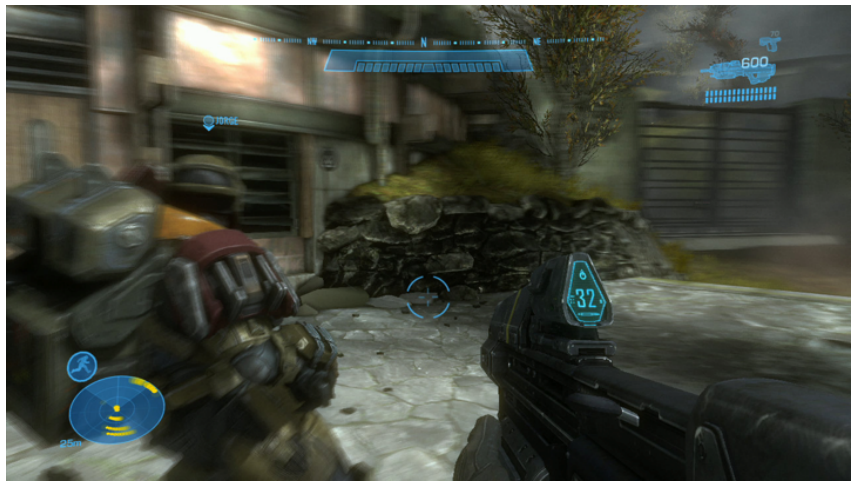


Blur

- This often used as a basis for other techniques such as Depth of Field and HDR
- To achieve blurring we apply a filter to the texture lookup of the texture render target
- We can apply many different filters, one of the simplest is a Box Filter
- With a box filter we sample 4 points around the point we are interested in

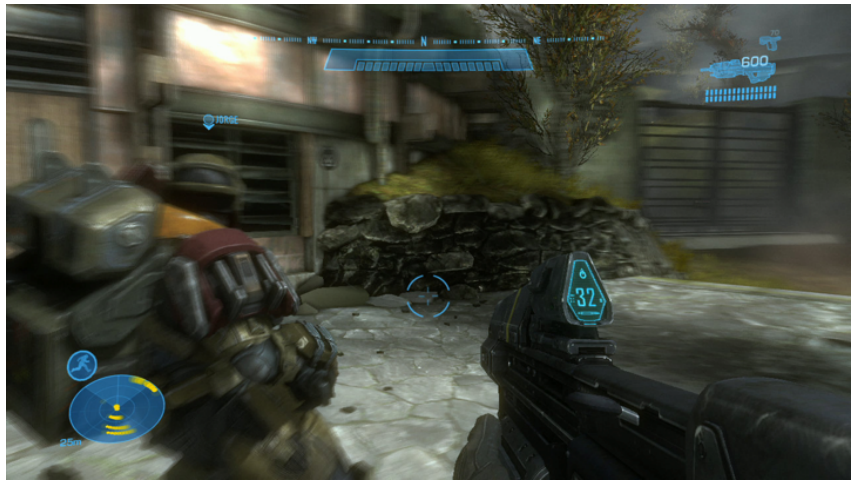
Motion Blur

- If an object moves very fast through you Field of view it can appear that the object leaves a slight ghost of itself
- There are several ways of implementing this, we can use the actual speed of the object to determine the amount of blur



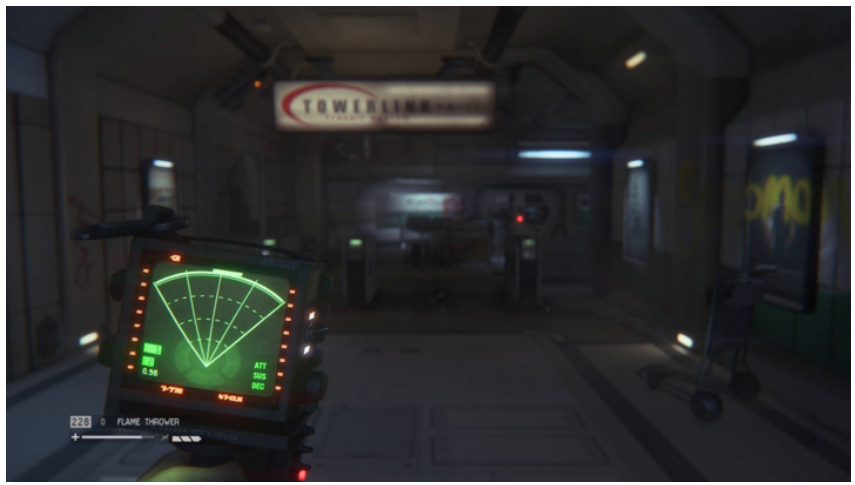
Motion Blur

- Or we can simply take the results of the last render update and blend them with current render results and blend based on a lerp



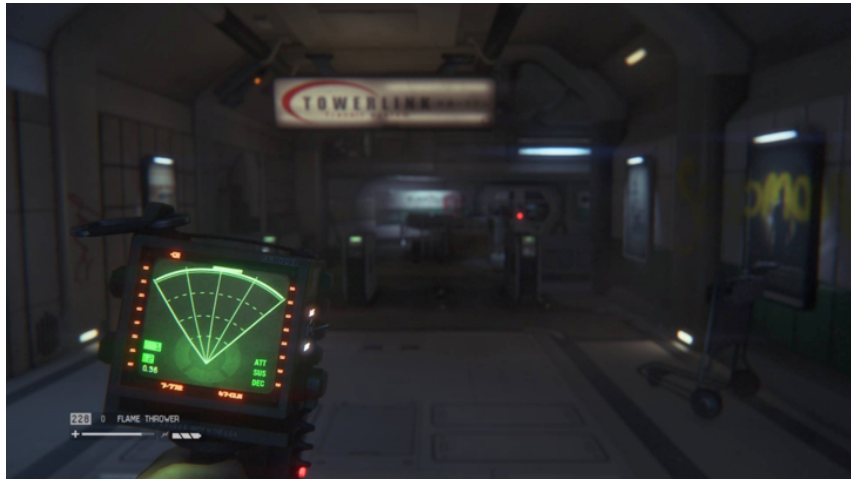
Depth of Field

- DOF attempts to simulate the effect where object that are too close or too far away appear out of focus
- We first have to consider in our scene what range of depth values will be considered in focus. We then need to blur everything that is outside this range



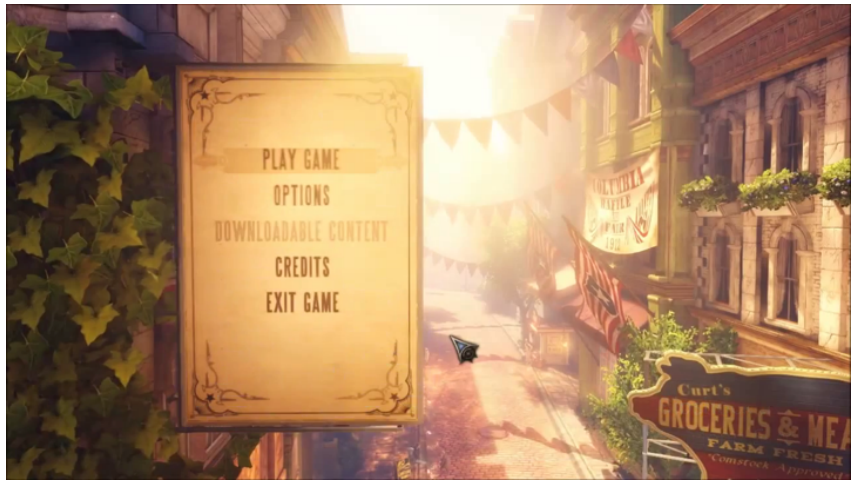
Depth of Field

- Once we have determined this we render a blurred scene to a render texture



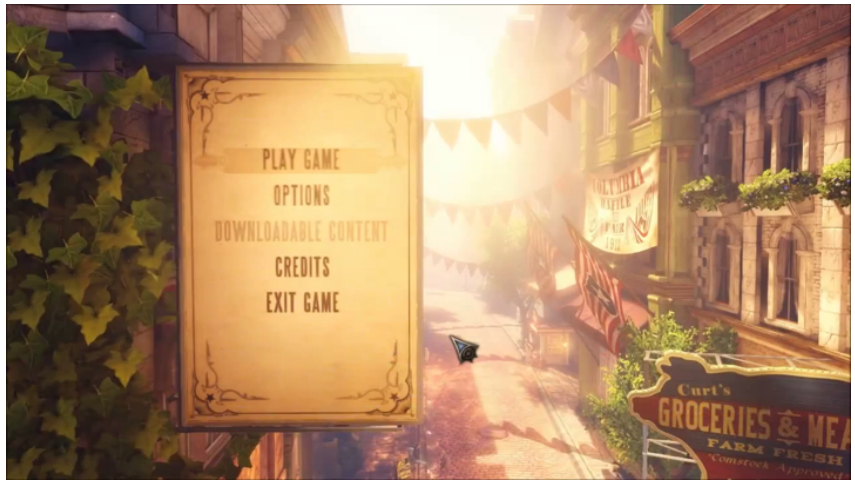
Bloom

- This attempts to simulate the overloading of the optic nerve if extremely bright areas of a surface are visible
- First of we render to a smaller render target usually 1/4 size



Bloom

- We then apply a filter(usually Gauss) to this reduced render target, if we apply this filter of multiple passes then we achieve a smoother blur
- We then blend the blurred image onto the screen with the original scene



Debrief

Post Processing Stack

- In theory we can chain post-processing effects together
- Essentially we keep rendering to a texturing using different fragment shaders each time
- These could be stored in a data structure such as a stack
- Once we have completed the last effect on the stack, we then switch to normal rendering and display the result

Exercise

Exercise 1 - COMP120 Algorithms

Implement the following algorithms:

1. Colour replacement
2. Black & white
3. Sepia Tone
4. Posterisation

Exercise 2 - Other Effects

Implement the following algorithms:

1. Blur (if you have camera controls, try to also implement motion blur)
2. Screen tear (hint. You will need to animate the texture coordinates using time)
3. Screen Static (hint. You will need some noise and the current time)