Lassi Väisänen

Creating and Using OpenGL Shader Effects in a 2D Game

Business Information Technology

School of Business

Fall 2015

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tekijä: Väisänen Lassi

Työn nimi: OpenGL-varjostinefektien toteutus ja käyttö 2D-pelissä

Tutkintonimike: Tradenomi (AMK), tietojenkäsittely

Asiasanat: 2D, OpenGL, varjostin, ohjelmointi

Opinnäytetyön tavoitteena oli esitellä, kuinka varjostimia voidaan käyttää 2D-peliympäristössä käyttäen apuna SFML-multimediakirjastoa. Aihe valittiin tekijän oman kehityksen tukemiseksi. Opinnäytetyössä esitellään, mitä OpenGL-rajapinta ja GLSL-varjostinohjelmakieli ovat ja kuinka ne liittyvät pelien kehitykseen.

Opinnäytetyön toiminnallisessa osassa luotiin kaksiulotteinen ympäristö varjostinefektien kehitystä, testausta ja esittelyä varten käyttäen Visual Studio 2010-kehitysympäristöä ja SFML-multimediakirjastoa. Työssä käytettiin C++-kieltä ohjelmointikielenä.

Työssä käytettiin onnistuneesti erilaisia varjostinefektejä, joista yksinkertainen valaistus luotiin turvautumatta valmiisiin esimerkkeihin. Bloomiksi kutsutun efektin luomiseen käytettiin useita erilaisia pikselivarjostinohjelmia, jotka olivat valmiiksi olemassaolevia.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author(s): Väisänen Lassi

Title of the Publication: Creating and Using OpenGL Shader Effects in a 2D Game

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: 2D, OpenGL, shader, programming

The goal of the thesis was to show how shaders can be used in a 2D game environment with the help of SFML multimedia library. The subject of the thesis was chosen to support the learning of the author. OpenGL and the GLSL shading language and their relation to game development is presented in the thesis.

In the practical part of the thesis a 2D environment was created for developing and testing shaders, and showing how they effect a game scene. The tools utilised for the development of the 2D environment were Visual Studio 2010 integrated development environment and SFML multimedia library. C++ was used as the programming language.

Shader effects were successfully implemented in the practical part of the thesis and a simple lighting effect was created without relying on examples. Several pre-existing custom fragment shader effects were applied to create the bloom effect.

CONTENTS

LIST OF SYMBOLS

API	Application Programming Interface.
Fragment shader	Shader that modifies the attributes of fragments one fragment at a time. Usually used for post-processing effects.
Fragment	Contains the data required for rendering a single pixel
GIMP	GNU Image Manipulation Program, a free image editor
GLSL	OpenGL Shading Language used for creating shader effects.
GPU	Graphics Processing Unit. Used for graphics rendering on computers, consoles, and mobile devices.
IDE	Integrated Development Environment.
OpenGL	Open Graphics Library, a cross-platform graphics application programming interface.
SFML	Simple Fast Multimedia Library, a cross-platform library for application development.
Shader	A program that executes on the graphics processing unit of a device. Used for graphics rendering and creating special effects.
Texture	Two dimensional image that represents the surface of an object.
Vertex	Point in 2D or 3D space. In computer graphics, vertices create surfaces that are usually triangles and arrays of these surfaces describe a rendered 2D or 3D object.
Vertex shader	Shader that transforms attributes of a vertex such as colour, texture and position.

1 INTRODUCTION

Shaders are used in nearly every modern computer, mobile, and console game. Most games would look dull if there was no lighting or filtering of some kind. 2D and 3D Game engines usually have built-in shaders for the most common use-cases so a game developer might not have any idea how the visual effects are implemented into a game.

At least programmers and graphics artists should have a general idea of how shaders work because shaders greatly affect the final outcome of virtual game scenes. Assets used in games should be able to be tested with the special effects a game is using to make sure they look like how the developers want them to look like.

The theory section is about shaders, OpenGL and GLSL. It gives a basic understanding of the OpenGL graphics application programming interface, the OpenGL Shading Language, and how they work together. The reader is expected to have general understanding of programming and graphics programming.

A 2D game environment was created to show shader effects in action and explain why they are used in game development. Visual Studio 2010 integrated development environment and Simple Fast Multimedia Library, known as SFML, were chosen as the tools utilised in the development of the 2D environment because of previous experience using these tools. SFML provided the functionality for OpenGL graphics rendering and using vertex and fragment shaders.

OpenGL and GLSL were chosen for the graphics programming because of previous experience with OpenGL and an interest towards graphics programming. All commonly used shader languages are similar with little differences in syntax so learning to use one language helps in using other shading languages as well.

2 OPENGL

OpenGL is a cross-platform graphics application programming interface utilised widely in desktop and mobile environments. It is employed in 2D and 3D graphics and is used in e.g. game development, computer-aided design and virtual reality applications. OpenGL enables graphics rendering utilising the graphics processing unit of a device. OpenGL supports several different types of programmable shader programs that modify the output of the rendering pipeline. The programmability of shaders allows creation of special effects. (OpenGL website 2014.)

The Khronos Group has developed both desktop and mobile implementations of OpenGL. The mobile version is called OpenGL for Embedded Systems. The desktop and mobile versions of OpenGL are compatible between each other to an extent. (Segal & Akeley 2013, 2 – 3.)

OpenGL provides low-level rendering functionality, giving the programmer a lot of control and flexibility. OpenGL routines can be utilised to build libraries for high-level functionality allowing for better usability. It is used for e.g. graphics engines in games. It is only a graphics library and has no support for user input in itself. (Astle & Hawkins 2002, 10.)

For programmers OpenGL enables the modification of geometry and textures in 2D and 3D space. To use OpenGL (the programmer) one must first open a rendering window in which the program will draw. After opening the window an OpenGL context is created and associated with the window. After initializing the OpenGL context calls to define shaders, geometry and textures can be made and then draw calls are applied to draw specific geometry passed to the shader program. Drawing commands include geometry such as points, lines and polygons. The geometry is modifiable by the shaders. (Segal & Akeley 2013, 2 – 3.)

OpenGL uses rasterisation for rendering graphics. Rasterisation is transforming data, usually points describing triangles, into a 2D image. The 2D image resulting from rasterisation is restricted to the screen's view. In 3D space matrices are used to transform received data into screen space. Screen space is the space where data is presented in pixel coordinates of the screen. As the primitives are being rendered to the screen they are also shaded. Rasterisation clips the data that is to be rendered, meaning processing data that is out of the screen space can be avoided. (Sherrod 2008, 117 – 118.)

2.1 Where is OpenGL used

OpenGL is a multi-platform API and has both desktop and mobile implementations avail-able. OpenGL is the desktop version and OpenGL ES is for mobile applications. Nearly all current mobile phones and tablets e.g. Android and iOS devices use OpenGL for graphics rendering. On Microsoft Windows platforms DirectX is the most common graphics API but OpenGL is becoming more widely adopted. Linux operating systems use OpenGL for graphics rendering and they are becoming viable as gaming platforms and hence game development. (OpenGL website 2014.)

2.2 OpenGL and other graphics libraries

The main competitor of OpenGL is the DirectX graphics API implemented in Microsoft Windows operating systems. DirectX is a set of APIs providing access to hardware in Windows op-erating systems. DirectX is not just for graphics rendering as it also supports reading input from devices, mixing and sampling sound, and networking. (Astle & Hawkins 2002, 10.)

Both OpenGL and DirectX use the same kind of graphics rendering pipeline. The pipeline has not changed much since the start of 3D graphics APIs. Both APIs also use vertices as a data set of coordinates in space that define vertex-related data. Graphics primitives (points, lines, triangles) are described as a set of vertices. (Astle & Hawkins 2002, 17.)

3 SHADER EFFECTS AND GLSL

Graphics Application Programming Interfaces are used to communicate directly with graphics hardware of a computer, gaming console or a mobile device. The most common graphics APIs are Direct3D and OpenGL. In older versions of graphics APIs there was no possibility for programmable shaders and custom-made graphical effects. The rendering pipeline was something called a fixed-function pipeline that only allowed for modification of predetermined states used for rendering. Programmable shaders enabled developers to be able to create their own graphical effects using different kinds of shader programs in the rendering pipeline. (Sherrod 2008, 173.)

Shaders are programs that are run on the graphics processing unit. They are used in games to e.g. create lighting and shadows, and apply post-processing effects such as bloom or anti-aliasing. Different types of shaders process different parts of data sent to a GPU. The vertex shader receives its data from the application it is utilised in and the following shaders receive their data from their preceding shader in the rendering pipeline. Shaders can also receive data from the application with uniform variables. The combination of shaders used to create an effect is called a shader program. (Sherrod 2008, 173 – 174; Haller, Hansson & Moreira 2013, 32; Wolff 2011, 50.)

The OpenGL Shading Language is a high-level shading language used for creating shader programs. It actually contains several different closely related languages that create shaders for different tasks. A program is a set of shaders that are compiled and linked together, completely creating one or more of the programmable stages of the OpenGL pipeline. Different types of shader processors are vertex, tessellation, geometry, fragment, and compute shaders. (Kessenich, Baldwin & Rost 2014, 10 - 12.)

3.1 GLSL language

Shaders became a part of OpenGL version 2.0, presenting programmability into OpenGL's fixed-function pipeline. GLSL is the language used to create rendering algorithms that run directly on the GPU of a computer and take advantage of its processing power. GLSL is similar to the C programming language. Shaders make some of the parts of OpenGL rendering pipeline programmable replacing the fixed-function pipeline functionality. (Wolff 2011, 48.)

Since the fixed-function pipeline is deprecated in newer versions of OpenGL and OpenGL ES, using it is still supported but the functionality will be removed in future versions. Sometimes developers only need to implement the basic shading techniques of the old fixed-function pipeline for development or to use the basic shaders as a starting point for developing shader effects. Multimedia libraries such as SFML use OpenGL version 2.0 or later for rendering graphics and obscure the shaders from the developer until custom shading effects are used. If no custom shader effects are applied, SFML implements default shading replicating the old fixed-function pipeline. (Wolff 2011, 7; SFML tutorials c.)

GLSL defines certain data types to make use of in shader programming. Scalar data types such as bool, integer, unsigned integer, float, and double are used much in the same way as they would be in C++. GLSL also defines vectors of each scalar types, even boolean, with two to four components. The same math operators such as multiplication and dividing can be applied to vectors as well as scalar types, but vector types have to have the same amount of components for both vectors. Matrix types are also defined in GLSL. All matrix types use floating point values. Matrices have two to four columns and rows. Matrices are generally utilised for describing views, scenes, positions and transformations. (GLSL Data Types 2014.)

3.2 Vertex processor

Vertex processor is a programmable unit that processes the incoming vertices and their data. Compilation units written in GLSL to run on the vertex processor are called vertex shaders. Vertex shaders are composed of source code that is compiled on the vertex processor. The vertex processor processes one vertex at a time. It handles the following functions: vertex transformations, normal transformations, modifying texture coordinates, vertex lighting and vertex colours. (Kessenich, Baldwin & Rost 2014, 10.)

As shown in Figure 1, a default vertex processor consists of gl_Position, gl_TexCoord[0], and gl_FrontColor variables. The variable gl_Position is for transforming the processed vertex, gl_TexCoord[0] for transforming texture coordinates and passing them to the fragment shader, and gl_FrontColor for setting the colour of the vertex. (SFML tutorials c.)

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;

    gl_FrontColor = gl_Color;
}
```

Figure 1. Basic vertex shader in SFML graphics module

3.3 Fragment processor

Fragment processor is a programmable unit that processes fragment values and their data. Compilation units written in GLSL to run on the fragment processor are called fragment shaders. Fragment shaders run on the fragment processor. The fragment processor processes texture information and fragment colour one fragment (pixel) at a time. A fragment shader cannot modify the fragments position and cannot access its neighbouring fragments data. The values processed by the fragment shader either update the frame buffer or texture memory depending on the state of OpenGL rendering and the command that caused the fragments to be generated. (Kessenich, Baldwin & Rost 2014, 11.)

As shown in Figure 2. The default fragment shader used by SFML consists of a texture passed to the shader, a vector4 type variable for getting the colour of a specific pixel in an image in a location that is defined by gl_TexCoord[0]. The final colour of the pixel being processed is the value in the pixel variable multiplied with the gl_Color value passed on from the vertex shader. (SFML tutorials c.)

```
uniform sampler2D texture;

void main()
{
    vec4 pixel = texture2D(texture, gl_TexCoord[0].xy);

    gl_FragColor = gl_Color * pixel;
}
```

Figure 2. Basic fragment shader in SFML graphics module

3.4 Graphics performance

Drawing each object on the screen separately potentially hinders performance of a program if there are a lot of objects to draw. Each separate draw call to the GPU introduces overhead in the performance of the CPU. A program should be able to determine what it should draw by using techniques such as depth testing or occlusion culling. The Z-buffer, also known as the depth buffer, is used to determine how much of a currently processed object should be drawn to the screen by comparing its position to other rendered objects e.g. if there is a fully opaque object rendered in front of the currently processed object, the current object is not visible and therefore should not be drawn. Graphics Processing Units usually have some kind of implementation of depth testing as built-in functionality. Occlusion culling is determining which objects in a scene occlude others and only rendering the visible objects. Occlusion culling is generally meant to be utilised to save CPU processing power by limiting draw calls sent to the GPU. (Sherrod 2008, 537.)

4 CREATING A 2D ENVIRONMENT

In order to display shader effects, an application using OpenGL for rendering graphics is required. The application needs to be able to load graphical assets, update game objects, render the game state into a window, and process user input.

In this project SFML library is used together with the Microsoft Visual Studio 2010 integrated development environment to create an environment in which to present various graphical effects. SFML is employed for opening a window, graphics rendering and user input and Visual Studio for programming the game application using C++ as the programming language.

4.1 Visual Studio 2010

Visual Studio is an IDE from Microsoft for developing programs for Microsoft Windows, creating web sites, web applications, and web services. Visual Studio has built-in support for C, C++, C++/CLI, VB.NET, C#, and F# programming languages. Visual Studio has a built-in code editor. Code editors make writing and reading code easier with features such as automatic indentation and automatic code completion. (Microsoft Developer Network: Solutions and Projects.)

For applications Visual Studio creates a solution containing a project. A solution is a container that includes the items required for creating an application. Projects make up the items contained in the application and produce executable files or dynamically linked libraries. (Microsoft Developer Network: Solutions and Projects.)

4.2 SFML

Simple Fast Multimedia Library is a cross-platform library that contains modules for vector operations, handling time, threading, opening application windows, collecting user input, rendering 2D-graphics, playing sounds, and networking. In this project, the System, Window, and Graphics modules are required to set up the development environment and have the required functionality. SFML's modules are all inside namespace sf to avoid conflicts with other possible classes or libraries used in an application. (Haller, Hansson & Moreira 2013, 7 – 9; SFML tutorials d.)

4.3 Setting up the development environment

At the time of writing the latest version of SFML is 2.1. It is available in sfml-dev.org/download/sfml/2.1 for several operating systems and compilers. To set up SFML modules in Visual Studio 2010, the static or the dynamic libraries of the required modules are linked both for debug and release modes. If the static version is linked, a preprocessor definition SFML_STATIC must also be added. (SFML tutorials a.)

The path to include SFML headers is set up in Visual Studio 2010 project properties C/C++ » General » Additional Include Directories as (<sfml-install-path>/include). The path to include SFML libraries is set up in project properties Linker Settings » General » Additional Library Directories as <sfml-install-path>/lib>). Both of these paths are shown in Figure 3.

Both these paths are the same in Debug and Release configurations and can be set up globally in the project properties. Linking the libraries needs to be done separately for Debug and Release modes as they use different libraries for each mode. The Debug-libraries use a suffix "-d" at the end e.g. "sfml-graphics-d.lib" and release mode "sfml-graphics.lib" when linking the libraries. Libraries are added in the project properties in Linker » Input » Additional Dependencies. (SFML tutorials a.)

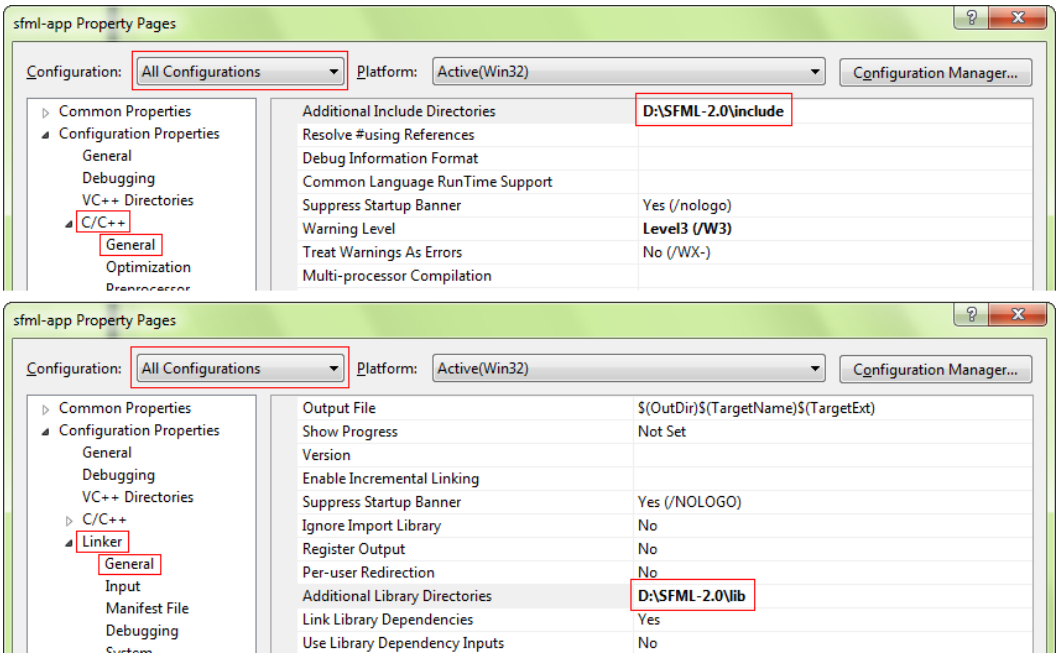

Figure 3. SFML include and library directories configuration (SFML tutorials a.)

A simple way to test if the setup was successful is to try a minimal example code of opening a SFML window and drawing a circle to a screen as shown in Figure 4.

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }

    return 0;
}
```

Figure 4. Minimal SFML example. (SFML tutorials a.)

If everything was linked correctly, by using the code snippet in Figure 4, a small window with the title “SFML Works!” rendering a green circle should appear by running the application as shown in Figure 5.

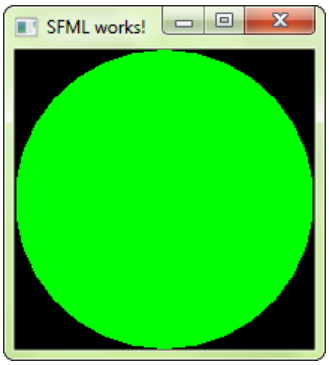

Figure 5. SFML application window (SFML tutorials a.)

4.4 Creating the game

A minimal game is created to show off the effects created in the next chapter. In the game there will be a background, a player turret and some targets for the player to destroy. In the game the player controls a turret that shoots boxes falling from the sky. To show the differences between normal rendering and using different shader effects the game has a state for switching between rendering modes. The game was made to have as simple functionality as possible as its purpose was to demonstrate the effects of shader programs created for the application.

The following classes were created to make the game work:

- Game

  The Game class handles initialisation of assets, creating and keeping track of game objects, and updating game logic. In the main function of the application an object of the Game class is created and updated until the application is closed. Game handles user input to switch between the rendering modes utilised in the application and the player turret's movement and shooting input. To switch between graphics rendering modes, Game uses an enum called DrawState that defines values called Normal, Lights, Bloom and LightsAndBloom. The function used in rendering the game scene is decided with this enum.

- Effect

  Effect is a class that loads a shader effect and can modify the loaded shader's parameters. Objects of the Effect class are used to apply the shader effects into drawable objects, windows or render textures.

- BloomEffect

  BloomEffect is an extension of the Effect class and specialises in creating a bloom effect. It loads multiple fragment shaders employed in creating the bloom effect and handles everything needed to apply the effect into the game. Game creates and initialises a single object of BloomEffect when the application is started and uses it in the rendering of the game scene when the drawing state of Game has bloom enabled.

- Bullet

  Bullet is an object spawned by the Game class and is spawned from the player's turret in short intervals when the left mouse button is clicked. It has a hitbox like the enemy and bullets are only able to collide with the boxes that fall from the sky and game class is checking if that happens or if the bullets go out of the screen causing them to get destroyed. Game also assigns textures for bullets and updates their movement. The direction of the bullet is relative to the player turret's direction.

- Enemy

  The Enemy class is used for the destroyable boxes in the game. It keeps track of the position, collider and if it is destroyed.

- Light

  Light is applied for creating multiple lights with their own positions and different qualities such as intensity, colour and range. Game applies the lights that are created to the game scene automatically if the drawing state has lighting enabled.

- Player

  Player is a class for creating the turret controlled by the player with the mouse. It also handles the loading of the player's graphical assets for the turret as there is only need to load the player's textures once unlike other assets. Player updates the rotation of the turret.

- PrimitiveShape

  PrimitiveShape class is utilised for an optimised way of drawing multiple textures with the same spritesheet and shader effect and used in the application for drawing the background as it is easy to batch static objects to a single draw call. It is not required to draw the background the way it is done in this class as it runs smoothly even on low-end hardware but for 3D applications and even really large and complex 2D scenes batching the static objects' draw calls saves a lot of processing power.

- Renderer

  The Renderer class renders game objects onto a window or a texture. Each game object can be rendered separately into a texture or the application window. By calling draw separately for each game object, CPU overhead is introduced through the graphics API. The game objects can also be batched by adding their vertex and texture information into a single vertex array provided they are using the same texture and shader effect. Even when drawing hundreds of objects separately, there should be no discernible performance problems on 2D applications or games even on low-end hardware so optimisation through batching all the game objects using SFMLs VertexArrays might not be required, but it is supported in the Renderer class.

5 USING SHADERS WITH SFML

SFML abstracts shaders to a class called sf::Shader. The user does not need a lot of knowledge about OpenGL to use SFML for compiling GLSL shader programs as shader programs are ready to be applied after specifying the shader program paths in the sf::Shader objects constructor or calling its loadFromFile-function. SFML only supports vertex and fragment shaders and that is usually all that is needed for 2D games. (SFML tutorials c.)

Classes for handling shaders have been implemented into the project called Effect and BloomEffect. They use SFMLs built-in sf::Shader class for loading the shader programs but also implement their own rendering functions and sf::RenderStates for using different kinds of OpenGL blending states. Both classes can render objects to a sf::RenderWindow or a sf::RenderTexture. In the project most of the rendering is done by rendering objects to sf::RenderTextures and further applying shader effects into the textures and finally drawing them in a sf::RenderWindow and displaying them.

5.1 Graphics rendering

To render images on the screen, first loading the files to memory is required. The SFML graphics module has classes supporting loading images from Portable Network Graphics image files. First a texture should be created of type sf::Texture. An image can be loaded into the texture by using the loadFromFile-function defined for the sf::Texture class by specifying the path the texture file is in. If texture loading fails the error should be handled by the programmer. After the texture loading is successful an object of type sf::Sprite can be created to set the texture in. The sf::Sprite object is a drawable representation of a texture, which has its own transformations and colour. (SFML tutorials c.)

SFML contains a class called RenderWindow. An object of type RenderWindow needs to be created to open a window for graphics rendering. To render objects to a window, it first has to be cleared with some color to fill the rendering surface with pixels. If the window is not cleared for every frame rendered and contains space where objects are not drawn, pixels from previous frames stay where nothing else is rendered. However a game window is usually fully covered with a background and game objects and clearing the window would not be necessary. After clearing the window, objects such as sprites, texts, shapes, and user defined vertex arrays can be drawn to it by calling the window's draw-function that takes drawable objects as parameters. The objects are not drawn directly into the window, but to a hidden buffer. To display what has been rendered to the window, calling the display function of the window object is required. Modern GPUs and APIs are made for this kind of clear, draw and display loop described where everything is completely redrawn for each frame rendered. (SFML tutorials c.)

SFML also has a way to render objects to a texture instead of drawing directly to a window. Instead of using an object of type sf::RenderWindow for rendering, sf::RenderTexture can be used to first render objects to an offscreen texture and then drawing the RenderTexture in the RenderWindow. It has the same drawing functions as sf::RenderWindow. RenderTextures are usually used in post processing effects such as motion blur and bloom. (SFML tutorials d.)

5.2 Loading and applying shaders

SFML supports loading shaders in its Graphics module. Shaders can be loaded into memory and do not need to be compiled separately since it is done internally by SFML. When loading a shader, both the vertex and fragment shader may be loaded at once into a single shader to be applied or only one of them can also be loaded. In case only one type of shader is loaded, SFML creates a shader program with the chosen custom shader and with one default shader. (SFML tutorials c.)

Modification of the uniform variables in the shader is possible using SFMLs Shader classes' setParameter-function which supports floats, vectors, colors, matrices, and textures. The setParameter-function takes the name of the variable to be changed as the first parameter and the value of the variable as the second parameter. Changing the location of the light with the fragment shader shown in Figure 6 e.g. would be called using shader.setParameter("lightPos", sf::Vector2f(300.0f, 300.0f)) as the variable of light position is called lightPos in the shader and is type of vector2 taking in x and y-coordinates. Doing this would set the position of the light to (300, 300) in the game screen from the top left corner. (SFML tutorials c.)

6 IMPLEMENTING THE SHADERS

The goal of special effects in games is essentially to make games look better and portray sensations. Different types of games need different types of shader effects, e.g. stealth based games use lighting effects for both gameplay mechanics representing visibility or vision of game characters, and to create atmosphere. First person shooters often make the screen shake and create a bleeding effect when the player is harmed. In 2D games sometimes just different kind of blending of drawable objects is all it takes to e.g. create a fake light or any kind of desired effect with transparent textures.

6.1 Lighting

To create a simple lighting effect in a two dimensional environment only the fragment shader needs to be modified. Lighting in 2D can essentially be just defining a position coordinate, a range value, and the colour of a light.

The lighting fragment shader as shown in Figure 6 defines the source texture to apply the lighting effect on, light position as x and y-coordinates, the radius of a light, and the colour of the light.

```
uniform sampler2D texture;
uniform vec2 lightPos;
uniform float lightRadius;
uniform vec4 lightColor;

void main() {
    vec4 pixel = texture2D(texture, gl_TexCoord[0].xy);
    float dist = length(lightPos - gl_FragCoord.xy);
    float lightIntensity = (1.0 - (dist / lightRadius));
    if(dist >= lightRadius) lightIntensity = 0.0;
    gl_FragColor = pixel * lightIntensity * lightColor;
}
```

Figure 6. The light fragment shader

The farther a point in the texture is from the center of the light the darker it will be. The dis-tance is calculated using the Pythagorean Theorem ($a^2 + b^2 = c^2$) e.g. if a fragment of the texture2D is in coordinates (x1, y1) and the lights position is (x2, y2) the distance between them is distance =$\sqrt{((x2 - x1)^2 + (y2 - y1)^2)}$. In the shader this distance variable is called dist and the function used to calculate it is built into GLSL called length() which calculates the distance between two vectors.

The effect is completed by multiplying the colour of the fragment, intensity of the light, and the colour of the light. By rendering the game scene using only SFML's default shaders, the game objects in the scene look as their texture files, some with applied rotation (Figure 7).

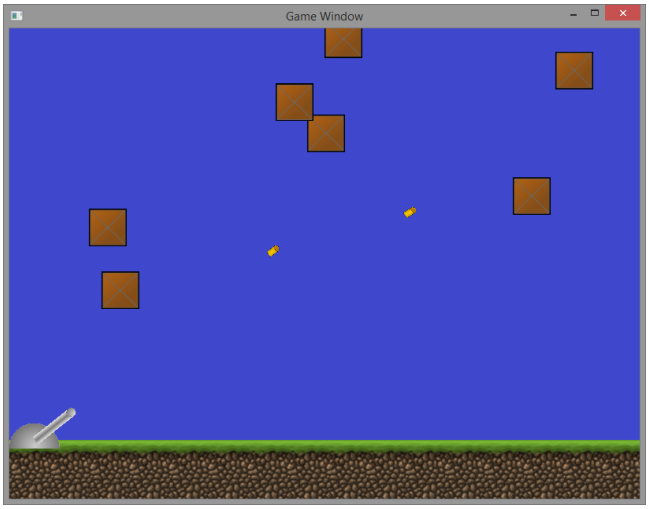

Figure 7. The scene without additional shader effects

The light effect demonstrated in Figure 8 is achieved by adding a single white light to the game scene and using the lights rendering state. To add multiple lights as shown in Figure 9, first the game objects in the scene are rendered to a texture without any additional effects. Then for each light defined in the application code the light position, radius, and colour variables contained in the Light objects are set in the lighting shader before each rendering pass. After setting the parameters, the scene is rendered to the texture again with the shader effect applied and ad-ditive blend mode enabled. Without changing the blend mode the lights would not blend properly and the scene would become too dark.

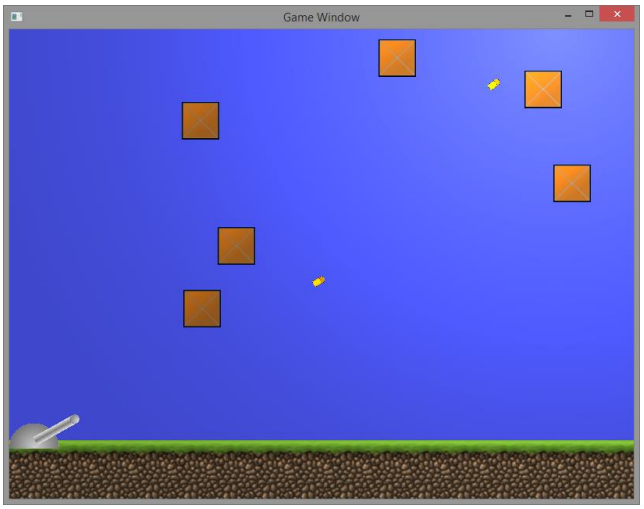

Figure 8. The scene with the light fragment shader applied

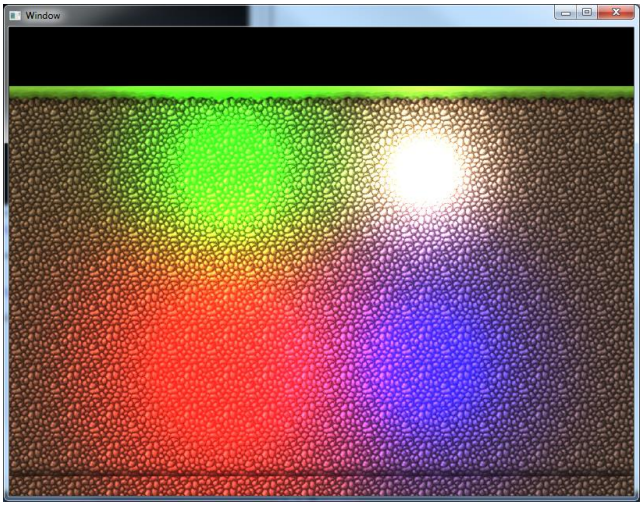

Figure 9. Multiple lights with different colours and radii

6.2 Bloom

Bloom is an effect used to simulate the effect watching a bright light or filming a light on a camera. The bloom effect will spread bright colours in a scene to make bright objects bleed light over their edges where there is no light. Bloom is a post-processing effect and it is applied to each rendered frame after everything else. The bloom effect in this project is achieved similarly as described in the book SFML Game Development with a few changes. (Haller, Hansson & Moreira 2013, 209 – 215.)

First a GLSL program to make a brightness pass from the game scene is required. The game scene is rendered to a texture and is passed to the brightness filtering GLSL program without any additional shader effects applied. The result is a texture that only shows the bright parts of the scene. The brightness filtering code is shown in Figure 10 and the result of a brightness pass applied to a scene with the player's turret, boxes and a light in the top right corner of the screen is shown in Figure 11. In Figure 11 the parts that are black in the image were originally transparent and paint black to make it easier to perceive how the scene is being filtered by the shader program.

```
uniform sampler2D source;
const float Threshold = 0.7;
const float Factor = 4.0;
void main()
{
    vec4 sourceFragment = texture2D(source, gl_TexCoord[0].xy);
    float luminance = sourceFragment.r * 0.2126 + sourceFragment.g * 0.7152 + sourceFragment.b * 0.0722;
    sourceFragment *= clamp(luminance - Threshold, 0.0, 1.0) * Factor;
    gl_FragColor = sourceFragment;
}
```

Figure 10. Brightness filter fragment shader

Figure 11. Result of a brightness pass in a scene with the player character, boxes, and a light.

This GLSL program filters what is bright, and not bright in the game scene. If the resulting texture from this pass would be applied to the game scene it would just make the already bright objects very bright or totally white instead of the desired effect of having the bright areas spread the brightness around them.

This brightness filtered texture is then blurred by rendering the texture in a lower than original resolution, and blending pixels of a texture with their adjacent colour values. The fragment shader utilised in downsampling the brightness filtered texture is shown in Figure 12.

```
uniform sampler2D texture;
uniform vec2 sourceSize;

void main()
{
   vec2 pixelSize = vec2(1.0 / sourceSize.x, 1.0 / sourceSize.y);
   vec2 textureCoordinates = gl_TexCoord[0].xy;
   vec4 color = texture2D(texture, textureCoordinates);
   color += texture2D(texture, textureCoordinates + vec2( 1.0, 0.0) * pixelSize);
   color += texture2D(texture, textureCoordinates + vec2(-1.0, 0.0) * pixelSize);
   color += texture2D(texture, textureCoordinates + vec2( 0.0, 1.0) * pixelSize);
   color += texture2D(texture, textureCoordinates + vec2( 0.0, -1.0) * pixelSize);
   color += texture2D(texture, textureCoordinates + vec2( 1.0, 1.0) * pixelSize);
   color += texture2D(texture, textureCoordinates + vec2(-1.0, -1.0) * pixelSize);
   color += texture2D(texture, textureCoordinates + vec2( 1.0, -1.0) * pixelSize);
   color += texture2D(texture, textureCoordinates + vec2(-1.0, 1.0) * pixelSize);
   gl_FragColor = color / 9.0;
}
```

Figure 12. Downsampling fragment shader

After downsampling each processed texture, the textures are then blurred by using a fragment shader called Gaussian blur that uses a Gaussian function to reduce detail in an image. The shader program for this shader is shown in Figure 13.

Downsampling of the brightness filtered texture actually happens twice as does the blurring which is done after each downsampling pass. By making the source textures smaller, and blurring them, the desired effect of spreading the light around bright objects is achieved when combining the newly created processed textures with the original rendered game scene using an additive blending fragment shader shown in Figure 14. A total of four filtered textures are passed to the additive blending fragment shader, first combining the processed textures, and lastly blending them with the game scene waiting to be rendered.

```
uniform sampler2D texture;
uniform vec2 offsetFactor;
void main()
{
   vec2 texCoord = gl_TexCoord[0].xy;
   vec4 color = vec4(0.0);
   color += texture2D(texture, texCoord - 4.0 * offsetFactor) * 0.0162162162;
   color += texture2D(texture, texCoord - 3.0 * offsetFactor) * 0.0540540541;
   color += texture2D(texture, texCoord - 2.0 * offsetFactor) * 0.1216216216;
   color += texture2D(texture, texCoord - offsetFactor) * 0.1945945946;
   color += texture2D(texture, texCoord) * 0.2270270270;
   color += texture2D(texture, texCoord + offsetFactor) * 0.1945945946;
   color += texture2D(texture, texCoord + 2.0 * offsetFactor) * 0.1216216216;
   color += texture2D(texture, texCoord + 3.0 * offsetFactor) * 0.0540540541;
   color += texture2D(texture, texCoord + 4.0 * offsetFactor) * 0.0162162162;
   gl_FragColor = color;
}
```

Figure 13. Gaussian blur fragment shader

```
uniform sampler2D texture;
uniform sampler2D bloom;
void main()
{
   vec4 sourceFragment = texture2D(texture, gl_TexCoord[0].xy);
   vec4 bloomFragment = texture2D(bloom, gl_TexCoord[0].xy);
   gl_FragColor = sourceFragment + bloomFragment;
}
```

Figure 14. Additive blending fragment shader

The additive blending shader simply adds the color values of two textures it is given. This is also done twice like the downsampling and blurring passes as two separate textures from the brightness pass is created for them with 2 different sizes of downsampling. After all the textures processed have been blended with the game scene the effect is complete.

6.3 Combining shader effects

Combining the previously applied shader effects still takes some work. Since the lighting effect is an effect that is applied after the background and game objects have been rendered it might as well be a post processing effect as well as bloom.

The rendering loop as shown in Figure 15 is a bit simplified since the bloom effect actually makes several passes with four different shader programs. (Brightness filtering, downsampling, Gaussian blur, and additive). This is the order in which everything is rendered in the game scene when the drawing state that implements all the effects is used.

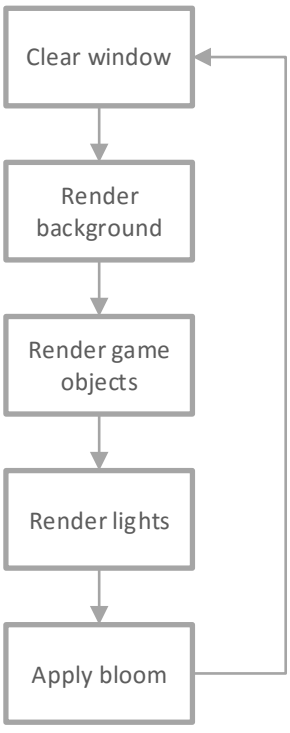

Figure 15. The applications rendering loop

In Figure 16 the combined effect of lighting and bloom is shown on the game screen. The player's turret is not affected by the light to increase the effect of bloom on the turret as it is too far away from the turret. The light in the game scene does have an effect on the bullets and boxes on the screen, and the objects closer to the light are visibly brighter. The bloom effect is stronger the brighter an object on the screen is so the light also makes the bloom more visible.

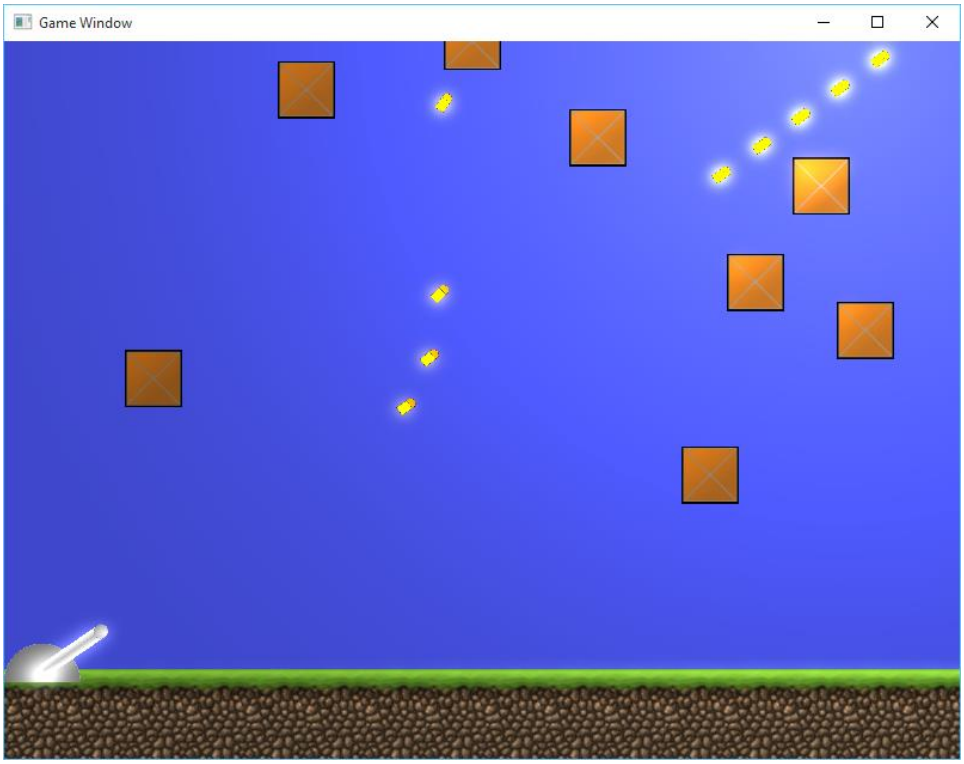

Figure 16. Lighting and bloom combined

The effect of bloom and lighting in this scene are very clear for demonstration purposes as games usually use bloom more subtly. It is best seen in the bullets and boxes nearest to the light. Grass in the right side of the scene is also lit, and has a minor glow as it is farther away from the light than most objects. Only some of the background on the left side of the screen is left completely unaffected by the light and bloom, and does not change as the rendering mode is switched.

7 CONCLUSIONS

Shaders are used almost everywhere when talking about computer graphics. Everyone dealing with special effects in games should have at least basic understanding of how shader effects work. Most of the time the shader effects you require for your game project have already been created, and available somewhere so most of the time you do not have to worry about writing your own shader programs, maybe just modifying them for your own purposes.

The creation of the project for showing off shaders could have gone better but it was enough to show how the pre-existing and newly created shaders might make a game look better. The game was originally meant to be more complex but the focus was mostly on implementing shader effects into the game after it had simple game mechanics and graphics. All the graphics assets except the ground sprite sheet, which was freely available, were made in GNU Image Manipulation Program. More graphics assets with varying sizes and qualities could have been employed for demonstrating the effects in various different scenarios.

The goal was to be able to create and implement different types of shader effects to a 2D game project. I only had a little prior experience working with writing shader programs from earlier game projects. Writing of a simple lighting shader was easy as it used only a few variables for controlling its position, size and colour. The shader had very simple mathematical functions utilised in it and some of it was built in to the GLSL language. Getting the bloom effect to work was the most demanding and time consuming part of the game project as it required employing multiple different shader programs to be applied in a specific order.

Having had previous experience with both Microsoft Visual Studio and SFML the process of creating a simple game and using the features of SFML’s graphics module was smooth. Originally creating more complex shaders than a simple lighting shader program was planned but it proved to be too much of a challenge to create and plan new effects from nothing in relatively short time.

REFERENCES

Kessenich, J., Baldwin, D. & Rost, R. 2014. The OpenGL Shading Language. URL: https://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf

Segal, M., & Akeley, K. 2013. The OpenGL® Graphics System: A Specification (Version 4.5 (Core Profile) – October 30, 2014) URL: https://www.opengl.org/registry/doc/glspec45.core.pdf

OpenGL website. 2014. URL: https://www.opengl.org/about/ (read 1.11.2014)

SFML tutorials a. 2014. URL: http://sfml-dev.org/tutorials/2.1/start-vc.php (read 1.11.2014)

SFML tutorials b. 2014. URL: http://sfml-dev.org/tutorials/2.1/window-window.php (read 1.11.2014)

SFML tutorials c. 2014. URL: http://sfml-dev.org/tutorials/2.1/graphics-shader.php (read 1.11.2014)

SFML tutorials d. 2014. URL: http://www.sfml-dev.org/tutorials/2.1/graphics-draw.php (read 1.11.2014)

GLSL Data Types. 2014. URL: https://www.opengl.org/wiki/Data_Type_%28GLSL%29 (read 17.11.2014)

Microsoft Developer Network: Solutions and Projects. URL: http://msdn.microsoft.com/en-us/library/b142f8e7.aspx (read 2.11.2014)

Astle, D., & Hawkins, K. 2002. OpenGL Game Programming. Boston, MA, USA: Course Technology. Retrieved from http://www.ebrary.com

Sherrod, A. 2008. Game Graphics Programming. Boston, MA, USA: Course Technology / Cengage Learning. Retrieved from http://www.ebrary.com

Wolff, D. 2011. OpenGL 4.0 Shading Language Cookbook. Olton, Birmingham, GBR: Packt Publishing. Retrieved from http://www.ebrary.com

Haller, J., Hansson, H. V. & Moreira, A. 2013. SFML Game Development. Olton, Birmingham, GBR: Packt Publishing. Retrieved from http://www.ebrary.com