OULUN AMMATTIKORKEAKOULU

Aatu Mikkonen

GRAPHICS PROGRAMMING THEN AND NOW

How the ways of showing pixels on screen have changed

GRAPHICS PROGRAMMING THEN AND NOW

How the ways of showing pixels on screen have changed

Aatu Mikkonen
Bachelor's Thesis
Spring 2021
Bachelor's Degree of Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Bachelor's degree of Information technology

Author(s): Aatu Mikkonen
Title of the thesis: Graphics Programming Then and Now
Thesis examiner(s): Kari Laitinen
Term and year of thesis completion: Spring 2021 Pages: 20

Graphics programming is relatively unknown form of programming even though everyone uses its capabilities every day, e.g., writing documents, creating art on a computer and playing or creating video games. Graphics programming, at least in Finland, does not have much literature written for it. The purpose of this report is to examine how the graphics programming has evolved from the old times to modern times, and examine how the tools for writing graphics programming has changed.

The author applied background knowledge learned in school and in free time about the research. The work was carried out by using the internet with search engines and online documentation searches. A variety of books, videos, articles and documentation pages were used as materials.

The main results are a brief overview on the differences in graphics programming from the old times to modern times, descriptions on the areas graphics programming handles and a small example on the implementation of graphics program.

Keywords:
graphics, graphics programming, graphics history, api

CONTENTS

TERMINOLOGY AND ABBREVIATIONS

AMD	Advanced Micro Devices
API	Application Programming Interface
C64	Commodore 64
GLSL	OpenGL Shading Language
GLUT	OpenGL Utility Toolkit
GPU	Graphics Processing Unit
HLSL	High-Level Shading Language
MSL	Metal Shading Language
NES	Nintendo Entertainment System
OS	Operating System
ROM	Read-only Memory

1 INTRODUCTION

This thesis is an explanation about graphics programming in game development in the old times and about the development of a simple graphics program without the need of a game engine, using modern technology and tools. There is no client for the project. The author's objective is to learn about graphics programming and improve general programming knowledge.

This thesis is going to cover topics on how the graphics programming in game development has changed from circa 1980 to modern days, also the development and implementation of a graphics program written from the ground up. The reason for the author to write graphics programming with game development in mind is that the author feels that graphics programming is more prevalent in game development than in desktop application on web application development in general. The program's source code will be hosted on web via git.

The program is going to display two simple textures, where both will be fading in and out using a shader program. The author wrote the program in conjunction to a tutorial on the Internet to learn about how to draw a shape and how to apply a texture to it. The language of the program used is C. OpenGL and its shader language was used to display a texture on the program window.

2 GRAPHICS PROGRAMMING

Graphics programming is a form of programming where the developer implements everything that is visually shown on the screen. Graphics programming could be considered a fairly low level programming language for front-end developers, even though in modern times and depending on the dimensions the developer works on, the requirements of graphics programming more relate to displaying 3D and shading correctly then and project them to the camera of the 3D scene.

As the programming language goes lower level, the involvement from the developer rises as there are less abstractions which help the programmer to write the code more safely and make the code more readable. Every modern language is considered high level as basically the only low-level programming language is assembly where each line resembles a machine code instruction. In assembly there can be unique instructions on each different processor architecture, though there can be similarities.

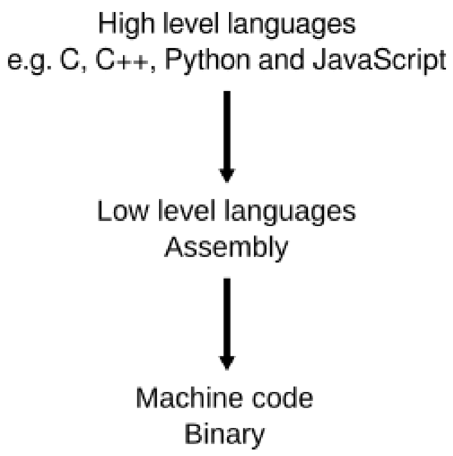


FIGURE 1. Hierarchy of programming language levels

2.1 Graphics in the 1980s

Graphics in the years late 1970s to early 1980 was primitive as the computers, especially commercial ones, were simple, where the most common microprocessor was an 8-bit microprocessor. The most famous example of early computers, and which used an 8-bit microprocessor was the Commodore 64, C64 [1]. The microprocessor that the C64 ran on was a variation of an 8-bit microprocessor called “MOS 6502”, which had variations for other well-known computers and consoles such as the Atari 2600 and the original NES.

Writing any sort of software to old hardware such as the MOS 6502 required the developer to write the code in assembly language. There were made some high-level programming languages for the processors of that time, such as the BASIC language, which depended on the platform, their own modifications and possibly were not compatible with each other. Even though there were higher level languages to help the programmer, they were not applicable to write anything else, but a simple program. If the software needed speed, talking machine code via the help of assembly was required as it could be highly optimized.

2.1.1 Character sets and sprite sheets

The most common way to draw graphics was to print the characters written to a character ROM, or overwritten to a character set in C64’s case. Consoles such as the original NES generally had the sprite sheets flashed to a character ROM chip on the game cartridge.

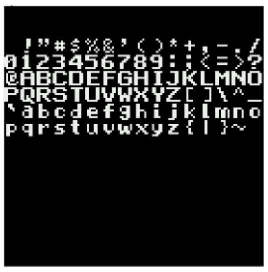


FIGURE 2. A .chr file containing a very simple tileset for the NES

In the case of NES, the character ROM consists of two individual sprite sheets, one for foreground sprites and the other for background elements. Each sprite can be made of 4 different colors and the colors can be changed to a specific palette to make the colors different. There could be up to four unique palettes, where each of them could hold up to three unique colors taken from a fixed palette like in FIGURE 3, with one shared background color [2].

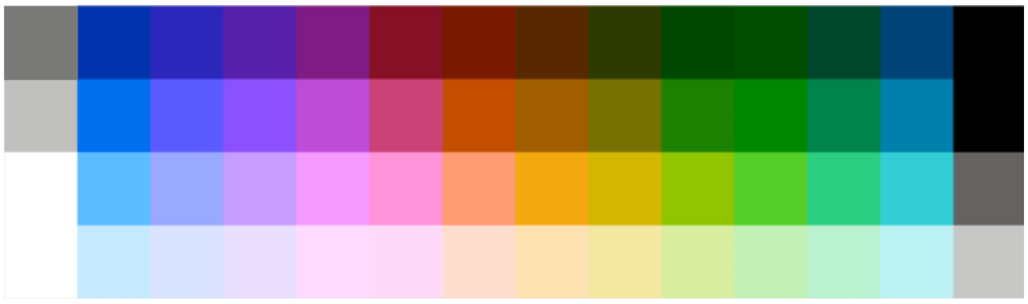


FIGURE 3. An example range of available colors for the NES

NES sprite sheets could hold 256 sprites each, 16 sprites horizontally and 16 sprites vertically as seen in FIGURE 2. Each sprite sheet took four kilobytes of space and generally there was eight kilobytes of character ROM available, enough for a foreground and background sprite sheet. A NES cartridge can hold up to 40 kilobytes of ROM in total without modifications, so that makes the game logic have 32 kilobytes of space [3].

Even though in modern times as the technology has evolved and the processors are more powerful, sprite sheets are still used to increase performance by decreasing load calls and to organize images to a single file.

2.2 Modern time graphics

Graphics in modern time can be split into two different types of graphics rendering modes: software and hardware rendering. While software rendering is still usable, using it instead of hardware rendering is not advisable as hardware rendering gives the developer more performance overall compared to software rendering.

2.2.1 Software rendering

Software rendering is graphics rendering that happens on the CPU. Software rendering has an advantage over hardware rendering that the graphics implementation is not tied to the graphics processor's implementations of graphics APIs, which gives the developer more freedom on how the drawing is made.

A CPU is designed to handle a wide variety of different calculations and instructions given to it. Even though a CPU can display graphics, it cannot do it as efficiently as a GPU, because they are not specifically designed to calculate graphics related calculations such as vertex and pixel calculations.

2.2.2 Hardware rendering

Hardware rendering is happening on the GPU and uses the help of graphics APIs to help draw to the screen. While **technically** possible, writing graphics without the help of an API is not feasible as the GPUs implemented APIs do most of the heavy lifting to make graphics code run on the unit.

GPUs are designed to handle a very high amount of parallel calculations consisting of models, textures, camera positions, camera projection, lights calculation and the pixels that displays on the screen [4]. A GPU can have thousands of processing cores for the spatial calculations in the GPU die itself.

3 APPLICATION PROGRAMMING INTERFACES FOR GRAPHICS

Application Programming Interface, or API, is as the name suggests, an interface that provides the developer with a set of functions to help the developer to write code with [5, p. 10], [6]. The implementation of the API may not be ever known to the developer at all. In the case of graphics programming, the APIs are only specifications, which tell what the API should do [see 7]. API implementation is usually written by the graphics card manufacturers (e.g. NVIDIA and AMD) partly to the driver on the OS and partly on the GPU. The manufacturers themselves decide what parts of the specification they want to support [see 8].

Modern graphics programming almost always happened through graphics APIs as described in the earlier chapters.

3.1 Comparison of Graphics APIs

This section compares the following graphics APIs: OpenGL, Vulkan, Direct3D, Metal and WebGL. These APIs are the most used in graphics development on desktop computers.

3.1.1 OpenGL

Open Graphics Library, or OpenGL is a graphics API developed by the Khronos group. The first release of OpenGL was in 1992 and from since it has become the most widely used and supported graphics API [9].

OpenGL is free to use for application developers, but generally closed source platform distributors (e.g. graphics card manufacturers) must pay Khronos Group a license fee [10], [11].

OpenGL in the old times had a programming mode called immediate mode which was easy to use, as the OpenGL hid most of the functionality and calculations which limited the flexibility for the developers [5, p. 10]. This functionality was deprecated in 2008 when the version 3.0 of OpenGL specification was released in favour of OpenGL core mode which disallows the usage of deprecated functions and forces the use of modern practices [12].

OpenGL has a shading language, called GLSL which resembles C, but has some functions to help on maths and calculations related to graphics programming.

3.1.2 Vulkan

Vulkan is a new generation graphics API which aims to bring better performances and more control for the developer with less abstraction between the application layer and the GPU. Vulkan is based on older technology which was made by AMD in conjunction with the game studio DICE. [13]

As with OpenGL, Vulkan is free to use for application developers, but if a developer wants the Vulkan API implemented and advertise it as being a Vulkan implementation, they are required to pay a conformance fee for Khronos Group. [11]

Vulkan uses GLSL as its shading language, such as OpenGL.

Vulkan can run on most of the main desktop OSs, including MacOS with MoltenVK which runs on top of Apple's own graphics API, Metal.

3.1.3 DirectX

DirectX is Microsoft's collection of propertiary API. This collection of components can be used to develop multimedia applications, but it is more geared towards game development. Some of its components include APIs for 3D and 2D graphics rendering, text rendering, audio and lately also raytracing.

The most relevant APIs in graphics programming perspective from the DirectX are Direct2D and Direct3D, both of which handle their own area of graphics. Direct2D handles 2D graphics, such as 2D geometry, bitmaps and text and Direct3D handles drawing of triangles lines or points per frame to the screen.

DirectX's latest version is 12, which mostly enables the use of ray tracing and performance boosts like with Vulkan compared to its predecessor, DirectX11. DirectX 12 is only available for users with Windows 10.

DirectX's shading language is called High-Level Shading Language, or HLSL. It is closely similar to C programming language.

3.1.4 Metal

Metal is Apple's own implementation of a low-level graphics API. Metal is used to enable high performance graphics on most of the Apple designed products: iOS, macOS and tvOS.

Metal uses it’s own shading language, MSL, which is closely similar to C++14 and can use similar directives such as overloading, templates and pre-processing directives. [14]

3.1.5 WebGL

A JavaScript API which enables the web developer to use hardware-accelerated 3D graphics technology in a browser. WebGL has been implemented on every major browser according to “caniuse”, a utility, which displays up-to-date support table for browser features [15]. Many game engines also support compiling a program to WebGL.

The main application for WebGL is the ability to publish games online and make them playable on a browser. Although applications can also be made into WebGL, doing so would not be feasible due to general user interface and website programming is done way easier and simpler with traditional HTML and JavaScript.

WebGL also uses GLSL as its shading language, like OpenGL and Vulkan.

3.2 Graphics API comparison results

Different graphics APIs have different use cases and support different system, so choosing a fitting graphics API requires consideration from the developer. The developer must choose whether they want ease of use or more control on how the graphics run and what platforms they want to support.

Helping the developers to make their choice are ready made game engines. Developers do not have to think about how to implement the graphics on a specific platform as the game engine developers have done the graphics API support for the developers themselves. The same could be applied to windowing systems and application developers.

TABLE 1. Graphics API comparison

	OpenGL	Vulkan	DirectX	Metal	WebGL
Year of Release	1992	2016	1996	2014	2011
Shading Language	GLSL	GLSL	HLSL	MSL	GLSL
Developers	Khronos Group	Khronos Group	Microsoft	Apple	Khronos Group
License	Open Source	Apache 2.0	Propertiary	Propertiary	Open Source
Desktop OS Support	Windows, Linux & MacOS	Windows, Linux & MacOS (via moltenVK)	Windows	MacOS	Almost every modern browser.

4 GRAPHICS PROGRAM IMPLEMENTATION

As mentioned in the Introduction, OpenGL is chosen to be used with a simple program to display how to use a graphics APIs without the use of a game engine to do everything for the developer. The reason for the author to go along with OpenGL and not with any other graphics API came down to wanting the program to be able to run cross-platform and because OpenGL is open source software.

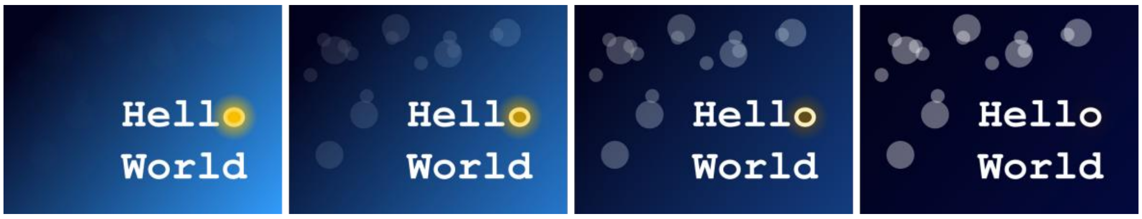


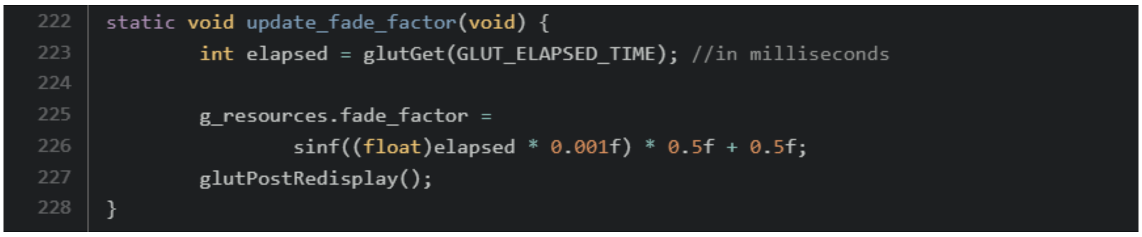
FIGURE 4. Program window flow

4.1 Initialization

Before starting to write graphics in OpenGL, a window must be displayed, and OpenGL needs to be initialized before any graphics could be drawn in. In this program, freeGLUT is used to help initialize and display a window and to minimize the requirement for boilerplate code. FreeGLUT, or Free OpenGL Utility Kit, is a library for OpenGL which, along with initializing OpenGL and create windows, also can handle input events and provide functions for creating an OpenGL program loop.

4.2 Shaders

All the listed graphics API in TABLE 1 uses a form of shading language. A shading language is a program that is resting on the GPU. A shader is a program that takes an input and does arbitrary modifications to it and pushes it to an output [16].



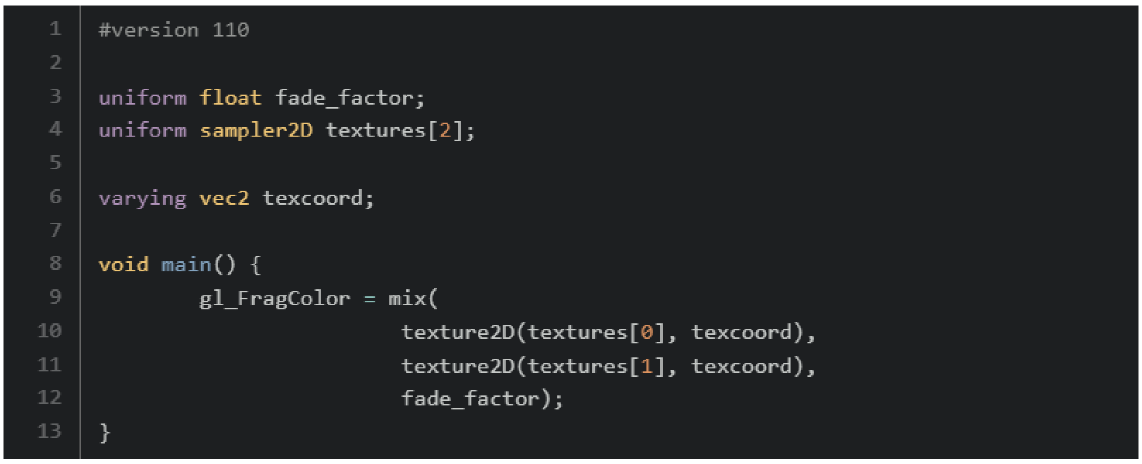
```
static void update_fade_factor(void) {
        int elapsed = glutGet(GLUT_ELAPSED_TIME); //in milliseconds

        g_resources.fade_factor =
                sinf((float)elapsed * 0.001f) * 0.5f + 0.5f;
        glutPostRedisplay();
}
```

FIGURE 5. Function responsible of the smooth transition of the images

Shaders can be divided to two different categories: fragment and vertex shaders. Shaders can be used to modify a model itself to a different look or change how the pixels themselves are displayed on the model, screen or texture. In the program, shaders are used to display a texture in a correct position and smoothly change the image in the program to a different image as in FIGURE 4.

4.2.1 Fragment shaders

Fragment shaders are programs that are run for each pixel in the given graphics state, whether it is the screen or a texture or face of a polygon. The implemented program uses the fragment shader to smoothly change every pixel of the texture in FIGURE 4.



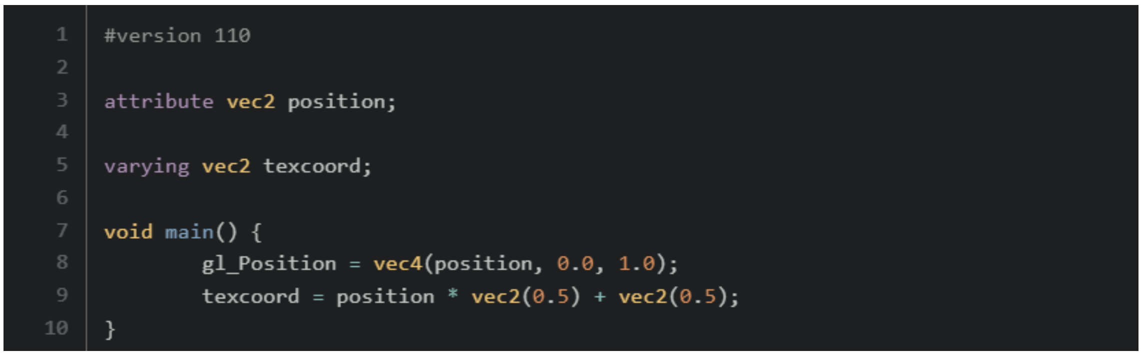
```
#version 110

uniform float fade_factor;
uniform sampler2D textures[2];

varying vec2 texcoord;

void main() {
        gl_FragColor = mix(
                        texture2D(textures[0], texcoord),
                        texture2D(textures[1], texcoord),
                        fade_factor);
}
```

FIGURE 6. Fragment shader program code

To get the shader program to change the pixels smoothly, a counter must be implemented in main program code to increment a floating-point counter every frame. This frame time counter could be used in the shader program itself, but in this implementation, it is used in the main program for simplicity's sake to change the shader program's texture fade value also, which ranges from 0 to 1. As the program uses freeGLUT, it can be used to update the counter via a function when there is nothing to be drawn on the screen. The fade value is then used in the shader program in FIGURE 6 to mix the two textures together which creates a fading effect.

4.2.2 Vertex Shaders

Vertex shaders are shader programs which are run for each point in a mesh. This allows the developer to manipulate a model in real time on the GPU. In the vertex shader program were used to make sure the texture displays correctly on the screen. It centres the vertices to the screen's viewport and gives both used textures correct coordinates to be displayed in the centre (see FIGURE 7).



```
#version 110

attribute vec2 position;

varying vec2 texcoord;

void main() {
        gl_Position = vec4(position, 0.0, 1.0);
        texcoord = position * vec2(0.5) + vec2(0.5);
}
```

FIGURE 7. Vertex shader program code

5 CONCLUSIONS

The objective of this thesis was to find out how the graphics programming has changed from the early years to today and how to create a graphics program without the need of a game engine to do most of the work for you. Finding information on graphics programming, especially from the old times, is hard to figure out. The most relevant information for the methods used in the old times came from people who shared their methods on creating games for old platform in modern times. During this thesis, the research on how the graphics programming worked before and works today was mostly successful. It came to realization that finding information relevant to 30 years in the past is hard, as official documentation can be almost impossible to find and the most relevant information in the area is upheld by an enthusiastic community.

Information on modern graphics APIs and how they are utilized is hard to write to people who are not in the graphics programming field already as the information there relies heavily on technical terms on the field. Graphics programming also is hard to get into, as the initial requirements to start writing more than a simple “Hello World”-demo requires learning more complicated maths such as matrix calculus to project an object to a 2D plane from 3D space.

Due to time constraints, learning more advanced form of graphics programming and doing more advanced programs was out of the question. It became clear during the thesis work that graphics programming takes a lot of time to learn, as there is a lot of information to learn on the get go and graphics programming is quite different from traditional programming. The author gained much needed knowledge on where to look for more information on how to start writing more advanced graphics programs. A good way would have been to design a usable program, a game or an application, and implement it well before the thesis to gain better understanding on the graphical programming.

REFERENCES

[1] I. Matthews. (2003). “Commodore 64 – the best selling computer in history” https://www.commodore.ca/commodore-products/commodore-64-the-best-selling-computer-in-history/ (visited on 05/06/2021).

[2] VblankEntertainment. (2013). “The making of: Rom city rampage” https://www.youtube.com/watch?v=Hvx4xXhZMrU (visited on 05/03/2021).

[3] Morphcat Games. (2018). “How we fit an nes game into 40 kilobytes” https://www.youtube.com/watch?v=ZWQ0591PAxM (visited on 03/28/2020).

[4] Computerphile. (2015). “Cpu vs gpu (what’s the difference?) - computerphile” https://www.youtube.com/watch?v=_cyVDoyI6NE (visited on 03/21/2020).

[5] J. de Vries, Learn OpenGL - Graphics Programming. 2020. https://learnopengl.com/book/book_pdf.pdf (visited on 04/23/2021).

[6] C. Hoffman. (2018). “What is an api?” https://www.howtogeek.com/343877/what-is-an-api/ (visited on 05/07/2021).

[7] Khronos Group. (2019). “Opengl 4.6 (core profile)” https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf (visited on 05/07/2021).

[8] NVIDIA Corporation. (2021). “Nvidia opengl specs” https://developer.nvidia.com/nvidia-opengl-specs (visited on 05/07/2021).

[9] Khronos Group. (2021). “Opengl overview” https://www.opengl.org/about/ (visited on 05/07/2021).

[10] Silicon Graphics International. (2012). “Opengl - licensing and logos” https://web.archive.org/web/20121101073722/http://www.sgi.com/products/software/opengl/license.html (visited on 04/29/2021).

[11] Khronos Group. (2021). “Api adopter program” https://www.khronos.org/adopters/conformance (visited on 04/29/2021).

[12] ——, (2016). “Legacy opengl” https://www.khronos.org/opengl/wiki/Legacy_OpenGL (visited on 05/07/2021).

[13] D. Altavilla. (2013). “Amd and dice to co-develop console style api for radeon graphics” https://www.forbes.com/sites/davealtavilla/2013/09/30/amd-and-dice-to-co-develop-console-style-api-for-radeon-graphics/ (visited on 05/07/2021).

[14] Apple inc. (2020). “Metal shading language specification” https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf (visited on 05/07/2021).

[15] Can I Use. (2020). “Webgl - 3d canvas graphics” https://caniuse.com/webgl (visited on 05/07/2021).

[16] J. de Vries. (2014). “Learnopengl – shaders” https://learnopengl.com/Getting-started/Shaders (visited on 04/23/2021).