Lappeenranta University of Technology
Faculty of Industrial Engineering and Management
Degree Program in Information Technology

Master's thesis

Mikko Kaistinen

OPEN SOURCE SOLUTIONS FOR MULTIPLATFROM GRAPHICS PROGRAMMING

Examiners: Associate Professor Uolevi Nikula
D.Sc. (Tech.) Jussi Kasurinen

Supervisors: Associate Professor Jouni Ikonen
D.Sc. (Tech.) Jussi Kasurinen

ABSTRACT

Lappeenranta University of Technology
Faculty of Industrial Engineering and Management
Degree Program in Information Technology

Mikko Kaistinen

Open source solutions for multiplatform graphics programming

Master's thesis

24.11.2015

59 page, 9 figures, 5 tables, 3 appendices

Examiners: Associate Professor Uolevi Nikula
D.Sc. (Tech.) Jussi Kasurinen

Supervisors: Associate Professor Jouni Ikonen
D.Sc. (Tech.) Jussi Kasurinen

Keywords: OpenGL, WebGL, computer graphics

New emerging technologies in the recent decade have brought new options to cross platform computer graphics development. This master thesis took a look for cross platform 3D graphics development possibilities. All platform dependent and non real time solutions were excluded. WebGL and two different OpenGL based solutions were assessed via demo application by using most recent development tools. In the results pros and cons of the each solutions were noted.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Tuotantotaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Mikko Kaistinen

Avoimen lähdekoodin tarjoamat mahdollisuudet monialusta grafiikkaohjelmointiin

Diplomityö

24.11.2015

59 sivua, 9 kuva, 5 taulukko, 3 liite

Tarkastajat: Tutkijaopettaja Uolevi Nikula
Tutkijatohtori Jussi Kasurinen

Ohjaajat: Tutkijaopettaja Jouni Ikonen
Tutkijatohtori Jussi Kasurinen

Hakusanat: OpenGL, WebGL, tietokonegrafiikka

Uudet teknologiset ratkaisut viimeisen vuosikymmenen aikana ovat tuoneet uusia mahdollisuuksia monialusta grafiikkaohjelmointiin. Tämä diplomityö tarkastelee monialusta 3D grafiikkaohjelmoinnin mahdollisuuksia. Alustariippuvaiset ja ei-reaaliajassa suoritettavat ratkaisut on rajattu tutkimuksen ulkopuolelle. WebGL ja kaksi eri OpenGL totuustapaa arvioitiin kehittämällä demo-ohjelma käyttämällä uusimpia kehitystyökaluja. Kaikkien ratkaisu vaihtoehtojen vahvuudet ja heikkoudet arvioitiin.

TABLE OF CONTENTS

SYMBOL AND ABBREVIATION LIST

API	Application Programming Interface
COM	Component Object Model
CPU	Central Processing Unit
CSS	Cascading Style Sheet
DLL	Dynamic-Link Library
ES	ECMAScript
FOV	Field of View
FPS	Frames Per Second
GLSL	OpenGL Shading Language
GLM	OpenGL Mathematics
GLEW	OpenGL Wrangler Library
GLUT	OpenGL Utility Toolkit
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
HLSL	High Level Shading Language
HTML	Hyper Text Markup Language
LHS	Left Hand Side
NPM	Node Package Manager
OpenGL ES	OpenGL for Embedded Systems
RHS	Right Hand Side
STL	Standard Template Library
SVG	Scalable Vector Graphics
XML	Extensible Markup Language

1 INTRODUCTION

Nowadays high-level programming languages dominate lower level languages according the Tiobe index and Github info [6, 27]. Lower level languages like Assembly, C and C++ are still de facto development languages for industrial level game engines, which people can see on their Steam account or on retail store. Direct3D 11 is modern high performance 3D graphics library on Windows environment [17]. Direct3D is low-level API (Application Programming Interface), which means that it models closely the hardware it controls. Game industry is dominant end user of Direct3D graphics library. Usually graphic engines are built on top of Direct3D, which controls graphics rendering of the game engine implementation. Medical and architectural industries also use Direct3D for high performance visualization like magnetic resonance imaging.

Direct3D is normally programmed with unmanaged C++, but any .NET based language e.g. C++/CLR, C# and Visual Basic can be used as well. Additional every new PC with modern GPU (Graphics Processing Unit) utilizes GPU in 2D applications like Windows Aero for intensive calculations. Direct3D 11 offers a compute shader API general-purpose GPU programs. Direct3D 11 also plays a key role in developing high performance Metro applications on Windows 8. To develop Direct3D applications programmer needs DirectX 11 SDK, which can be found on Microsoft's web site http://msdn.microsoft.com/en-us/directx/default.aspx.

Graphic libraries have long been a battle of the two manufacturers. DirectX library set by Microsoft and OpenGL by Khronos group (original author Silicon Graphics). OpenGL is cross-language and multiplatform API for 2D and 3D computer graphics, where as DirectX is set of programming libraries and APIs. Direct3D is one of those libraries and is closest counterpart to OpenGL graphics library. Direct3D supports only Microsoft's Windows operating systems. VC++ and C# can are the only supported programming languages with Direct3D.

OpenGL currently have multiple implementations like standard OpenGL, OpenGL ES and WebGL. OpenGL ES is mobile platform version of OpenGL and it has more limited access to low level resources compared to vanilla OpenGL. WebGL is based on OpenGL ES for 3D rendering in modern web browsers. Unlike DirectX OpenGL consist only Graphic API specification of 3D graphics programming although there are libraries for cross platform context creation and input handling.

In summer the 2014 current version of OpenGL is version 4.5 released in August 2014. Current Direct3D version of the moment is version 11.1. Newest version of Direct3D 12 was released in fall 2014. [18]

This master thesis examines 3D graphics implementation with cross platform capable technologies. The idea for this master came from bachelor work, which is a 3D model loader using Direct3D 10. Project is available GitHub https://github.com/Laastine/3dmodelloader. The goal is to investigate possibilities of the cross platform development via creating interactive windowed 3D demo application with each technology.

1.1 Objectives and scope

The research questions of this master thesis are:

1. How does WebGL differ from OpenGL?
2. What are the pros and cons of each chosen solutions?
3. Which kind of project is suitable for each implementation technology?

This master's thesis focuses to comparing WebGL and OpenGL libraries. The thesis describes WebGL and OpenGL programming interface. Readers are expected to have basic understanding of 3D graphics, high school mathematics and decent level of knowledge about programming.

1.2 Structure

This master's thesis consists of seven chapters. The second chapter takes a look at the scientific literature for this master's thesis. The third chapter investigates OpenGL and WebGL libraries and frameworks. The fourth chapter introduces the theory behind 3D graphics and also takes short look at the graphics engine architecture. The fifth chapter puts theory into practice with both technologies and describes technical details of the implementation. Sixth and seventh chapter discuss the obtained results and the final chapter summarizes conclusions of the work.

2 LITERATURE REVIEW OF MODERN 3D GRAPHICS

The topic of this master's thesis sets some challenges to literature and scientific material since the field of study requires latest information about libraries and frameworks. Generally all computer graphics material is relevant in some level, but the literature research focuses on most recently published papers.

Computer graphics application consists of many different elements. This master's thesis examines those different elements and implements working prototype version of 3D graphics application. To achieve this goal various implementation technologies are examined and tested. Literature research is also performed to get research data about various implementation technologies.

2.1 Literature search

Literature review was done for investigation what others were done before and for gathering references for this thesis work. Firstly, good keywords were needed to be defined for systematic literature review. The keywords are OpenGL, Direct3D, WebGL, computer graphics, game programming and game engine. The source literature was included if contained comparison knowledge about Direct3D, OpenGL or WebGL. To gather more keywords for search queries, a number of Internet forums were read and promising keywords were tests via test queries. Also manufacturer's documentation was used in practical part of the master's thesis. The source of the thesis consists of well-known computer science libraries such as IEEE Xplore, ACM DL and Science Direct. "Introduction to Game Studies" by Frans Mäyrä lists following sources that were used in this master's thesis:

- Game Studies Journal - www.gamestudies.org
- Journal of Game Development - www.jogd.com
- Google Scholar – http://scholar.google.com

This list was only a starting point for research. Search queries were performed via AND and OR queries to combine search results and screen out irrelevant articles. Time range of research articles were limited to 2012 to 2014, because of rapidly evolving technologies.

2.2 Inclusion and exclusion criteria

The inclusion and exclusion criteria were applied to the literature search results. The inclusion criteria in this study were discussion of the following topics:

- Game or other real-time graphics with WebGL
- WebGL API discussion
- 3D web applications

The excluded categories in the papers were:

- Medical or geographical modeling.
- Non real-time rendered techniques

As a result of filtering 8 articles were selected for reference material. Results of literature search were poor due to a strict filtering of literature material. This was also preliminary forecast.

Table 1. Summary of literature research queries to selected databases

Source database	Search parameters and Filtering	Included / Total
IEEE Xplore	WebGL AND HTML5 AND graphics, journals and conference publications between 2012-2014	4/22
ACM DL	WebGL AND HTML5 AND graphics, journals and conference publications between 2012-2014	1/11

Science Direct	WebGL, journals and conference, Computer&Graphics, publications between 2012-2014	2/4
IEEE Xplore	Direct3D OR OpenGL AND game engine, journals and conference publications between 2012-2014	1/4
Google Scholar	WebGL, webgl	1/1
Total		8/39

2.3 Literature search result analysis

Database search offered vast amount of research papers. After filtering research result a total amount of 39 conference and journal articles were reached. Literature research results were reviewed by reading topic and abstract.

2.4 Data extraction

Virtual Heritage to Go by Nils Nichaelis, Yvonne Yung and Johannes Behr takes a conceptual approach to implement web application with 3D graphics. The paper discusses the usage of WebGL and X3DOM in distributed application model. Both technologies are based on HTML5, CSS3, DOM scripting and Ajax. X3DOM and WebGL provides rendering completely on client side. Paper also provides concrete example on various execution platforms like iPad.

3D for The Web by Michael Macedonia is short article about 3D web applications. Article discusses about X3DOM and WebGL technologies and addresses past obstacles and future sights of three dimensional web applications. This article provides insights to WebGL alternatives, although X3DOM aims to serve easier usage with less performing solution a compared to WebGL.

3D Graphics on the Web a Survey by Alun Evans et al. Journal presents different WebGL libraries to support development. Since web browser plugins and declarative methods (SVG) are out of scope this paper those were not considered. Paper discusses different remote rendering techniques in web browser to produce 3D graphics. There is also discussion about major obstacles of 3D graphics in web browsers.

A WebGL Based Method for Visualization of Space Environment by Yang Yikang, Liu Xinxing and Liu Lei describes technical implementation of WebGL. Paper also presents three.js library and shows Three.js example application. The three.js library eases WebGL initialization and usage. 50 percent of web sites that uses WebGL are using Three.js helper library.

Cloud And Mobile Web Based Graphics and Visualization by Haim Levkowitz and Curran Kelleher. Article describes mobile graphic rendering via WebGL and argues that WebGL is going to be market leader in too-distant future. Authors also argue that this revolutionize data visualization for end users.

Graphics Performance in Rich Internet by Rama C. Hoetzlein. Article describes RIA (Rich Internet Applications) performance. The author describes performance tests for each RIA implementation method: WebGL, Flash, OpenGL and HTML5. HTML5 and Flash are out of scope of this thesis work due to their limitations to render real time 3D graphics efficiently. The article also provides testing results with major web browsers for each different implementation framework. In the summary section of the article, the author points out that JavaScript engine implementation is a huge factor in WebGL performance.

The Last Eternity, a 3D Role-Playing Game with a Cross-Platform Development by Yodthong Rodkaew describes development of Last Eternity for mobile and PC platform. Cross-platform nature of The Last Eternity gives insight of various runtime platforms.

Interpretive OpenGL for computer graphics discusses about an OpenGL library called Ch. Ch is an embedded scripting tool for cross platform application development. This allows rapid prototyping for 3D graphic application powered by OpenGL. Applications developed via Ch can be executed in web browser.

Resource materials were very technically oriented, but as such they were suitable for this master's thesis. Most of the articles considered WebGL one way or another. WebGL and real-time web browser graphics is relatively new to research topic, but many of computer graphics principles apply also to WebGL since it's only an OpenGL wrapper in web browser. Unfortunately there weren't much technical info about WebGL. One article referenced to Tony Parisi's WebGL books that were used as source material for this master's thesis.

3 MODERN WEBGL AND OPENGL

Both Direct3D and OpenGL offer robust graphics library to implement two or three-dimensional computer graphics with hardware acceleration meaning GPU (Graphics Processing Unit). Both API are well supported by graphics card manufacturers and newer graphics cards are backward-compatible meaning that one can execute applications based on DirectX 9 on more recent hardware which is DirectX 11 capable. Under the surface these libraries differ from each other. While Direct3D is based on Windows COM (Component Object Model) binary-interface, the OpenGL is based low level C-style graphics API. OpenGL API doesn’t depend on any particular programming language and there are OpenGL bindings for most of the common programming languages. Direct3D is and has always been a proprietary API meaning that API’s source code is not available for the end users.

Direct3D applications can only be executed on Windows environment although there is a Direct3D compatibility layer on Linux side, which converts the Direct3D API commands to corresponding OpenGL API commands on runtime. On the contrary OpenGL is open standard API, which provides 2D and 3D graphics API for most of the modern operating systems like Windows, Linux and Mac OS X. Both Direct3D and OpenGL are implemented in the display driver level, which provides interface between hardware and programming API. There is quite a difference in a way, which these drivers are implemented. Direct3D uses Windows CLR (Common Language Runtime) communicate to DDI (Device Driver Interface) to implement its API [2, 17], while in OpenGL side graphics card vendors implement whole OpenGL API. OpenGL implementations can vary a bit between different graphic card vendors. Both APIs uses own shading language, which doesn’t differ much from each other.

3.1 OpenGL and OpenGL ES

OpenGL was developed by SGI in 1991 and released on the following year. It was designed to group up to be 3D graphics rendering API. Today Khronos group governs the development of OpenGL. OpenGL has little brother implementation called OpenGL ES, which sort for OpenGL for Embedded Systems. OpenGL ES has limited subset of features compared to original OpenGL. Khronos group announced project Vulkan, which is going to be replacement for OpenGL and OpenGL ES. Vulkan is a grounds-up redesign of the OpenGL API. Vulkan is not going to backward portable with OpenGL. OpenGL is ported to various programming languages like C, C++, Java, Python, Perl and Fortran.

OpenGL supports extensions that bring new features to core OpenGL and still applications backward compatible. Whenever a graphics company comes up with a new technique or a new large optimization for rendering this is often found in an extension implemented in the drivers. Developer can check if the specific feature is implemented for current hardware and so maintain backwards compatibility. Direct3D has feature levels for handling hardware diversity problems. While feature levels are fast way to develop new features to graphics cards using OpenGL is also fragments the hardware support, because developers have to ensure support for each feature individually.

3.2 WebGL

In the early days of modern Internet web pages where only text and static images. After the release of Netscape Navigator and Internet Explorer in the mid 90s web development exploded to various web technologies like HTML, CSS, XML, Java, JavaScript. Ten years later in 2005 Ajax (Asynchronous JavaScript + XML) made possible to implement SPA (Single Page Application) side web pages. Before WebGL and canvas element Adobe's Flash was primary web technology to create interactive graphics, audio and animations to the web browser. While Flash needed a browser plugin WebGL is integrated to browser itself at becoming the standard technology in all modern web browsers. Competition between browser manufacturers increased JavaScript performance on the level where 3D graphics is possible to implement in web browser without any browser plugins [3].

WebGL is a JavaScript API for implementing 3D and 2D real-time graphics in modern web browser. WebGL was originally an experiment of Vladimir Vukićević at Mozilla. He developed first three-dimensional Canvas experiments in 2006. Vukićević based new

Canvas 3D to OpenGL ES (Embedded Systems) which was gaining its popularity. OpenGL ES is a subset of OpenGL graphics API. On the next year both Mozilla Firefox and Opera Internet browsers had own implementation of Canvas 3D. In 2009 WebGL Apple, Opera and Google within Khronos Group to govern new standard created working group. Khronos Group also governs OpenGL and COLLADA specifications. Khronos Group maintains WebGL specification to this date. Currently WebGL uses OpenGL ES 2.0 specification. WebGL doesn't require any plug-ins or external decencies and it's standardized method of displaying 3D graphics on web browser. Current WebGL browser support status can be seen in table 2. Next version of the WebGL is in the standardization process and currently it will be based on OpenGL ES 3.0 [19].

Table 2, WebGL browser support status in late 2014 [19]

	Version	Additional notes
Internet Explorer	IE11+	Partially supported
Google Chrome	9.0+	
Firefox	4.0+	
Safari	5.1+	Needs to activate separately
Opera	12+	Needs to activate separately

WebGL API is accessed exclusively via JavaScript. There aren't any HTML tags for WebGL except the canvas element where WebGL output is rendered. WebGL elements can be mixed with HTML elements and can also be composed with other parts of page or its background. Notable WebGL web applications are for example Farmville and Google Earth.

WebGL by its nature provides low-level API for graphics development. This can be tedious for large applications, which use WebGL heavily. Most notable WebGL libraries or frameworks are Three.js, Babylon.js and X3DOM.js. Three.js is oldest of three libraries and also most popular at the moment.

3.3 Three.js

Three is the oldest of the three libraries released in April 2010. Originally Three.js was written with ActionScript and later ported to JavaScript. Three is made available under MIT license. Ricardo Cabello Miguel also known as Mr. doob created Three.js. Roots of the Three.js are in the demoscene an interactive computer art subculture that is specialized in producing computer demos. Demo is a small self-contained computer programs that produce audiovisual presentation of makers coding skills. Three.js was first major graphic library available to WebGL. Three.js is easy to get start with, because it hides gory details of three graphics and provide lots of examples and good documentation. Gorescript game runs in the users browser and is implement with Three.js. Figure 1 shows screen capture of the gorescript game. Project is available in GitHub (https://github.com/timeinvariant/gorescript).

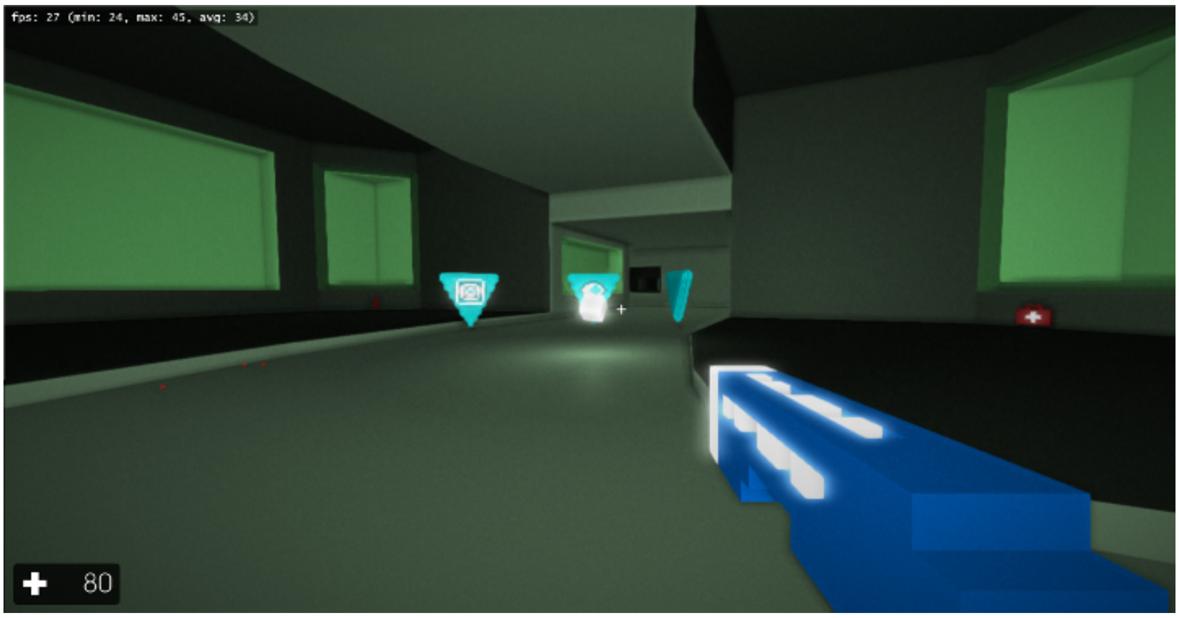


Figure 1, Screenshot of gorescipt 3D shooter made with Three.js

Three.js can also render 2D graphics via 2D canvas element and SVG (Scalable Vector Graphics) / CSS (Cascading Style Sheet). While Three.js does many things it is not a game engine, but graphics library which allows 3D graphics rapid development.

3.4 X3DOM

X3DOM (pronounced X-Freedom) is dual licensed under MIT and GPL licenses. X3DOM provides declarative way to present 3D graphics in the user's web browser. Declarative in a way that 3D content is HTML tags in the HTML page instead of canvas element like in the pure WebGL implementation. There is no need for additional plugins and it support most the HTML events like onclick and onsubmit. [3]

Table 3, X3DOM browser support in 15.3.2015

	Version	Additional notes
Internet Explorer	IE11+	Workarounds for older versions
Google Chrome	9.x+	
Firefox	All available versions	
Safari	5.1 and OS X 10.6+	Needs to activate separately
Opera	Unknown	

3.5 Babylon.js

Babylon.js is much more recently released WebGL library in summer 2013. Babylon.js originates from Microsoft's Redmond's labs, but has made open to public later. Babylon.js was first introduces with Internet Explorer 11. Babylon.js is more focused to game development compared to more generic Three.js.

In the table 3 below shows comparison of all three WebGL frameworks presented earlier in this chapter. Size minified was executed via Google's Closure compiler option using --language in=ECMASCRIPT5 flag.

Table 4 WebGL framework comparison in 15.3.2015.

	X3DOM (1.6.2)	Three.js (r70)	Babylon.js (v2.0)
Github stars	237	18639	1698
Contributors	35	414	44
Size minified	700KB	401KB	635KB
License	MIT and GPL	MIT	Apache License 2.0

3.6 Alternatives and portability

The Direct3D is implemented officially only on Microsoft's own products like Windows operating system and Xbox gaming consoles. Third parties have made several unofficial reimplementation of Direct3D API. The most famous of them is Wine (Wine Is Not an Emulator). Acronym points that Wine uses portability layer to execute Direct3D commands. Other known portability layer for Direct3D in Linux environment is Gedega a proprietary source code fork from Wine. Gedega is concentrated to bring Windows games playable in Linux environment. Both these are implemented by reverse engineering Direct3D proprietary code.

Valve Software has developed own translation layer called ToGL for Direct3D -> OpenGL, which is focused to bring Windows games to Linux platform. ToGL is open source project available on Valve's GitHub account https://github.com/ValveSoftware/ToGL. Currently ToGL only supports limited subset of Direct3D 9c and project is in experimental stage.

Google has developed own translation called Angle project (https://github.com/stammen/angleproject) layer mainly for Google Chrome web browser to convert OpenGL ES 2.0 application to Direct3D 9 or 11 API calls. Google's translation layer also adds security for WebGL so that possible bugs that are in OpenGL drivers cannot be exploited via browser. Angle project is also available in the Github https://github.com/ericmckean/angleproject.

Both Direct3D and OpenGL are stable and well documented APIs. While Direct3D was designed mainly for gaming on Windows environment, the OpenGL is much broader API and it has many applications in scientific research, modeling and planning.

4 BASICS OF 3D GRAPHICS

3D graphics includes lots of math and physics. This chapter describes basics of 3D computer graphics and its math. A scene that is rendered users computer display as 3D graphics is composed of many separate objects. These objects have geometrical forms, which are represented by a set of vertices and a particular set of graphics primitives. These indicate how vertices are connected to each other. Interconnected vertices produce a mesh, which forms 3D object to screen buffer.

4.1 Coordinate system

Computer graphics is based on several interlaced coordinate system, which dictates positions and orientations of the 3D objects in worldview. World coordinate system is where every object is located; beside this there are local coordinates for each object or surface for their own use. These local coordinate systems are then transformed to world coordinate system so that local manipulation is easy and there aren't so many coordinate system transformations while rendering the final image to users screen. [13]

In 3D world every pixel is expressed via triplet (x, y, z), which describes its location in the three-dimensional world. Direct3D uses LHS (Left Hand Side) coordinate system while OpenGL uses RHS (Right Hand Side). In the RHS coordinate system Z-axel increase towards to the viewer and LHS opposite direction in the LHS coordinate system. Direct3D 9 introduced an optional RHS coordinate system for Direct3D. To convert the RHS system to the LHS, system matrix (v0, v1, v2) is converted to matrix (v0, v2, v1) and z-axel is multiplied with -1 to invert direction. [13, 15, 17, 18]

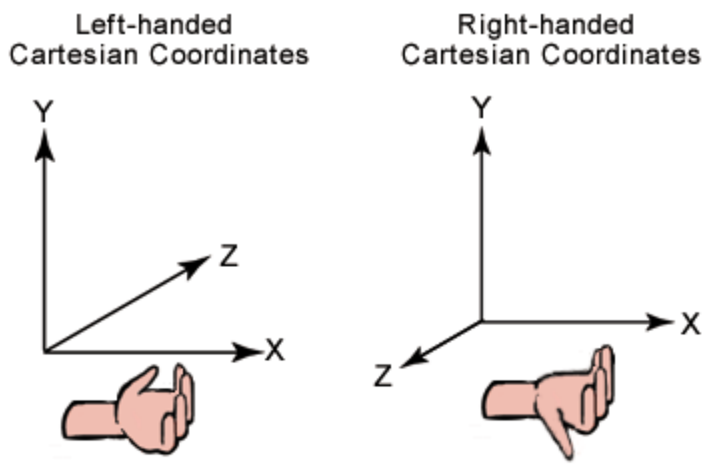


Figure 2, LHS and RHS coordinate systems

4.2 Translation and rotation of objects

View matrix is a 4x4 normalized matrix which contains X, Y and Z vertices. Object translations move object on the screen in x, y or z-axis. The following examples are implemented via orthographic projection to keep things simple.

```
function translation(x, y, z) {
        return [
                1, 0, 0, 0,
                0, 1, 0, 0,
                0, 0, 1, 0,
                x, y, z, 1]
}
```

Object rotations in 3D scene are handled the following way.

```
function rotateX(angleInRadians) {
 var c = Math.cos(angleInRadians)
 var s = Math.sin(angleInRadians)

 return [
  1, 0, 0, 0,
  0, c, s, 0,
  0, -s, c, 0,
  0, 0, 0, 1
 ]
}

function rotateY(angleInRadians) {
 var c = Math.cos(angleInRadians)
 var s = Math.sin(angleInRadians)
 return [
  c, 0, -s, 0,
  0, 1, 0, 0,
    s, 0, c, 0,
    0, 0, 0, 1
  ]
}

function rotateZ(angleInRadians) {
  var c = Math.cos(angleInRadians)
  var s = Math.sin(angleInRadians)
  return [
     c, s, 0, 0,
    -s, c, 0, 0,
     0, 0, 1, 0,
     0, 0, 0, 1,
  ]
}

function scale(x, y, z) {
  return [
    sx, 0,  0,  0,
    0, sy,  0,  0,
    0,  0, sz,  0,
    0,  0,  0,  1,
  ]
}
```

Functions rotateX, rotateY and rotateZ are only capable of rotating around predefined axis in 3D. To freely rotate camera around the object, which is located in a fixed point. There is a following equation [15].

$$R = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} + (1-\cos\theta) * \begin{matrix} x^2 & x*y & x*z \\ x*y & y^2 & y*z \\ x*z & y*z & z^2 \end{matrix} * \sin\theta \begin{matrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{matrix}$$

In JavaScript code:

```
var axis = {
  x: 0,
  y:0,
  z:0
}
function rotateZ(angleInRadians, axis) {
  var c = Math.cos(angleInRadians)
  var s = Math.sin(angleInRadians)
  var identityMatrix = [
     1, 0, 0,
     0, 1, 0,
     0, 0, 1
  ]
  var dualMatrix = [
     0, -axis.z, axis.y,
     axis.z, 0, -axis.x,
     -axis.x, axis.x, 0
```

```
 ]
 var axisTranspose = [
   axis.x * axis.x, axis.x * axis.y, axis.x * axis.z,
   axis.x * axis.y, axis.y * axis.y, axis.y * axis.z,
   axis.x * axis.z, axis.y * axis.z, axis.z * axis.z
 ]
 return identityMatrix * cos + axisTranspose * (1-cos) + dualMatrix * sin
}
```

4.3 Camera projection

Every computer graphics application has a camera projection, which consists of field of view x- and y-axis of the scenery rendered to users screen. Figure 3 shows how the camera projection works. It takes near- and far clipping planes, which delimits the area rendered to screen. This produces a (viewing) frustum, a pyramid like object, which upper part gets clipped away. Camera is located in the tip of the clipped pyramid and looks at center of both clipping planes. Without frustum culling every triangle in the world coordinate system gets rendered even though they are not visible in screen i.e. user is not looking at them. This separation of objects permits to lesser amount of computation for each frame that gets rendered to end-users screen. Camera projection is defined with four parameters distance to near- and far clipping planes, aspect ratio and FOV (Field of View). These parameters are visualized in figure 3. [15, 17]

Camera projection described above is a perspective projection, which basically means that objects that are farther seems to be smaller compared to objects that are close to the viewer. There is another projection called orthographic projection, which ignores this depth effect. In orthographic projection elements on the different position in the Z-axel are considered in the same size. In the perspective projection elements that are farther are scaled down.

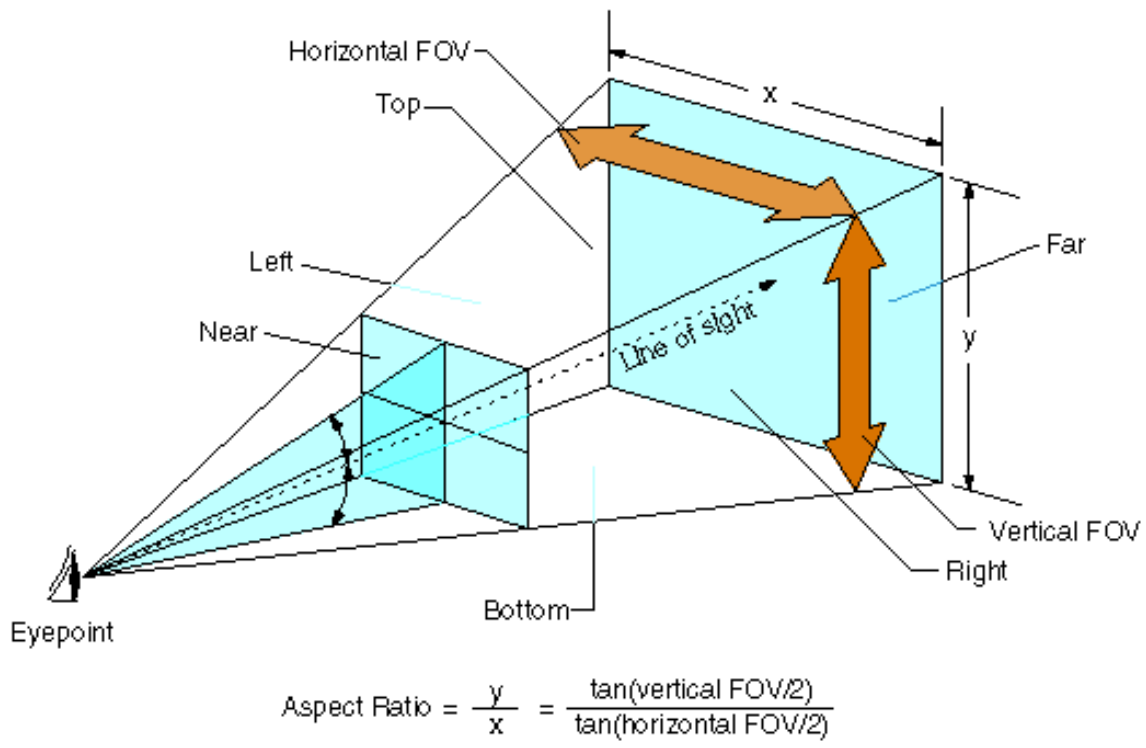


Figure 3, Frustum culling visualized

In 3D graphics the average frame rate for smoothly updating screen is about 30 FPS meaning that screen gets updated 30 times in a second. This is enough for human eye not to detect individual frame from among the set of frames that get rendered to the screen. One mechanism for smooth screen updating process is double buffering. Double buffering draws each frame on the buffer before they are presented to screen. This allows every frame to be drawn to the end. If an incomplete frame is send to the screen there could be a screen tearing as result.

Light element provides realistic views for 3D graphics. Lighting calculation are the single most expensive operations in the computer graphics. Lighting calculations have been and still are some sort of simplifications of the actual lighting calculations. Shaders play a big part in the lighting calculations due to fast GPU floating point arithmetic operations.

Game engine’s main loop contains everything that the game engine updates during its life time like AI (Artificial Intelligence), game physics, update animations, render the scene, play sound effects and background music. A game engine has to constantly iterate over its main loop for at least 30 times per second i.e. 30 FPS (Frames per second) so that graphics rendering won't hitch notably. When the game engine exits its main loop program shut downs. Because of this 30 FPS constant performance requirement main loop, is highly tailored for each particular game engine. A game engine differs many ways compared to normal programming, which executes its states and waits new input for next action or state. Even simple card game needs to run AI update function while game goes on and still render card movements according AI's decisions. [9, 26]

4.4 Multithreading vs. process based main loop

The simplest case of the main loop is the single loop, which handles every action in each update round. This basic solution has many flaws, when there is a lot of stuff to handle in the game engine. There are two possible workarounds: a multi-threaded solution and process based solution. Performance is one of the main advantages on the multi-threaded solution; on the down side there is much added complexity in the solution. Multi-threaded solution makes solution more complex, because solution has to take care of communication between threads and solve thread's timing issues on the same time. In case of multithreaded design game engine has to implement in a way that there aren't many mutable variable with state data. Figures 4 and figure 5 below show the sketch of game loop high-level architecture implemented with pure multithreaded model and a process based hybrid model. [9, 26]

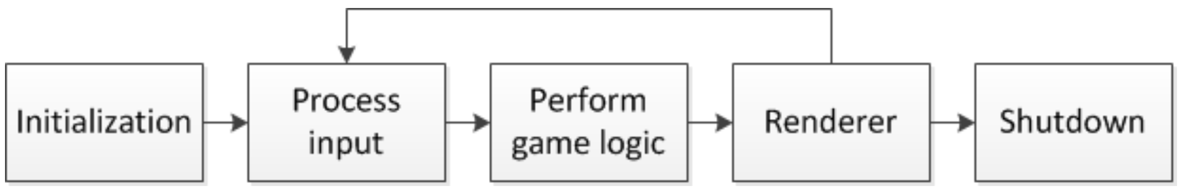


Figure 4, Game loop

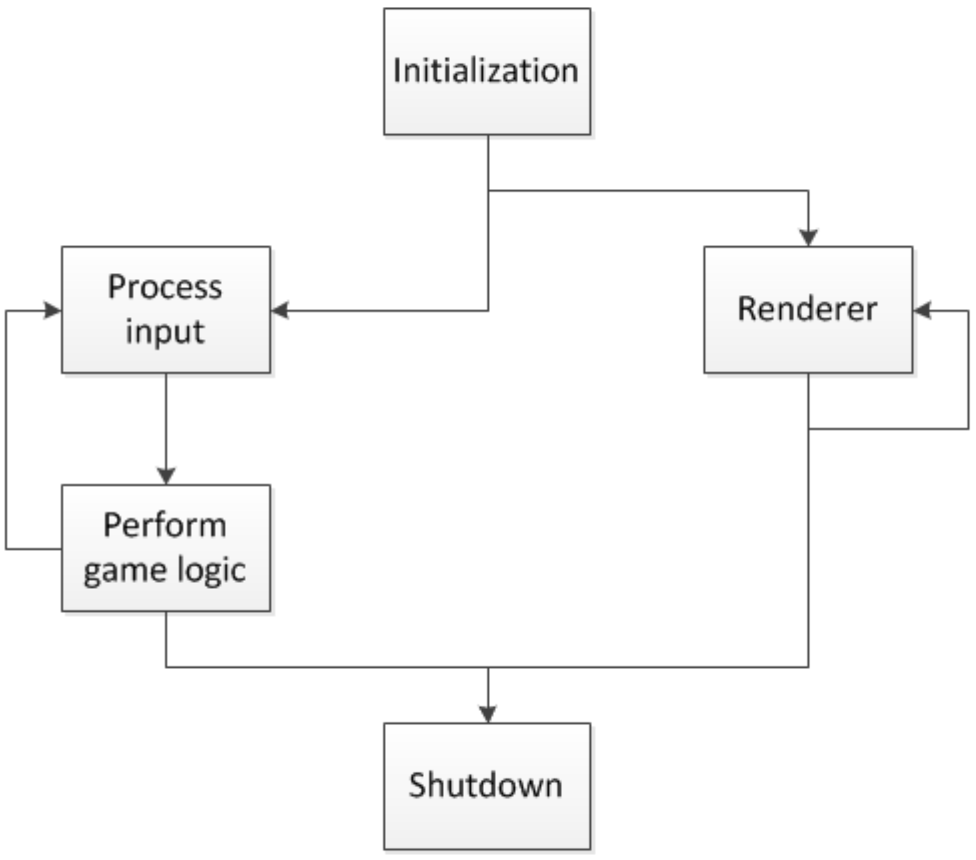


Figure 5, multithreaded loop

Threaded and hybrid model (Process based). In hybrid model each sub process is itself a threaded process. This model takes pros and cons of the both multi-threaded and synchronous main loop. It’s not as efficient as multithreaded and faster than synchronous solution.

4.5 Graphics pipeline

Graphics pipeline is a sequence of calculation steps to produce a 2D raster representation of a 3D scene. Basically this means to render a three dimensional representation to 2D computer display. In the early days of 3D computer graphics, fixed purpose hardware was used to accelerate the steps of the graphics pipeline through a fixed pipeline, but the hardware has evolved since early days of computer graphics, and the fixed-function pipeline has evolved to programmable one. While the graphics pipeline evolved so did the Direct3D and OpenGL specifications. Nowadays both Direct3D and OpenGL use programmable graphics pipeline. Programmable graphics pipeline gives much more control over graphics pipeline. Basically fixed function pipeline first set states of every pipeline states and those cannot be altered later. Programmable graphics pipeline allows the changes in the pipeline on runtime providing more control for every pixel that gets rendered to end users screen. [9, 18, 26]

3D graphics pipeline implementations differ a bit between OpenGL and Direct3D. Both contain generation 3D primitives e.g. triangles, modeling and transformation, camera transformation, lighting calculations, projection transformations, clipping, rasterization and texturing + shaders pixel, geometry and vertex.

4.6 Rendering pipeline

Shaders are scripts that are executed on graphics card HLSL (High Level Shading Language) and GLSL (OpenGL Shading Language). Shaders are scripts that perform GPU calculation. Both HLSL and GLSL are C-like script languages, which are loaded among other implementation code and send to GPU with vertex data to process. Those scripts are preprocessed and send to GPU, which performs floating-point mathematics much faster than CPU. [4]

Next there is simple fragment shader example, which colors every pixel to red without transparency:

```
out vec4 color;
void main(void)
{
        color = vec4(1.0, 0.0, 0.0, 1.0);
 }
```

There are several kinds of shader types; the most common types are vertex shader, fragment shader, tessellation shader and geometry shader. The most common ones are vertex and fragment shaders. In HLSL the fragment shader is called a pixel shader, which describes it better because fragment/pixel shader is operating via pixels, the tiniest single element in the display [18]. OpenGL programmable graphics pipeline can be seen in figure 6. Dotted boxed presented programmable stages in the picture.

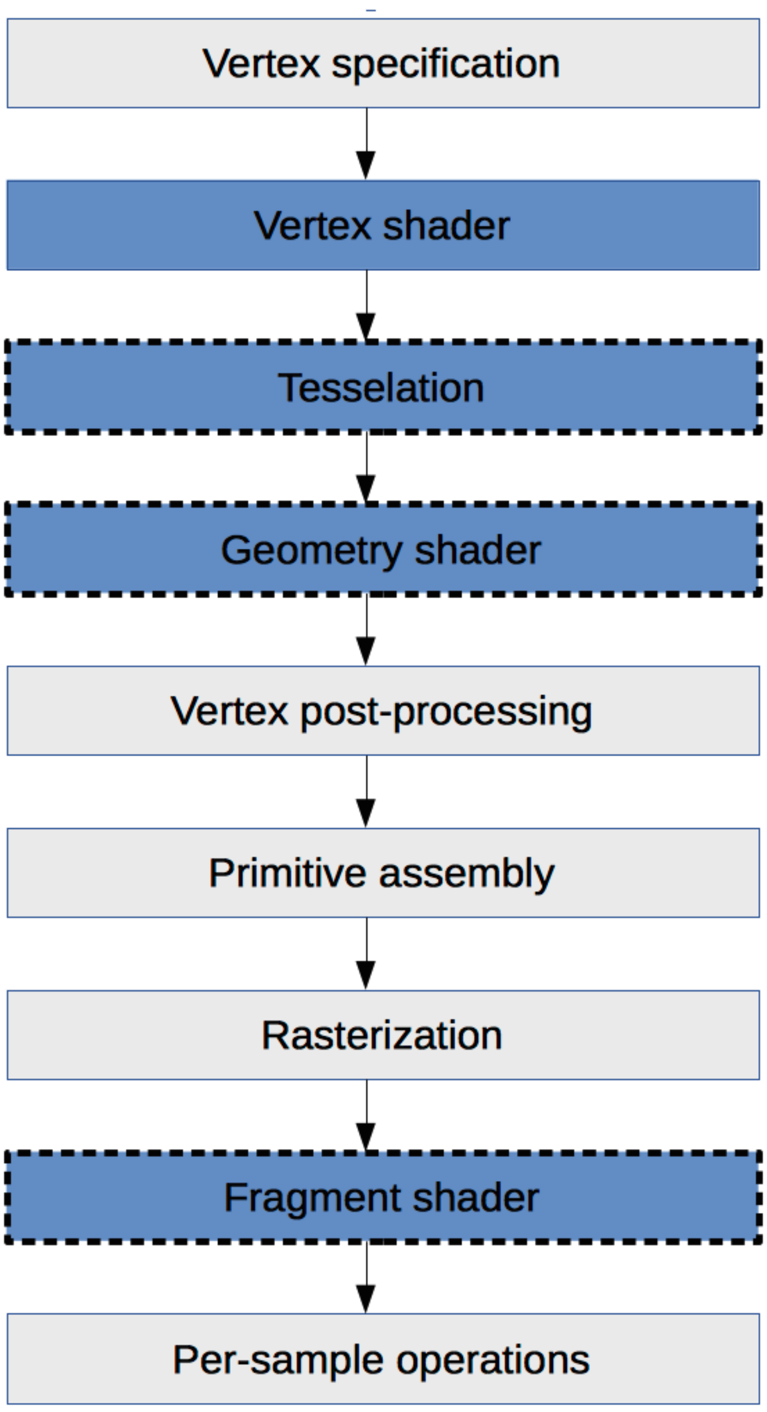
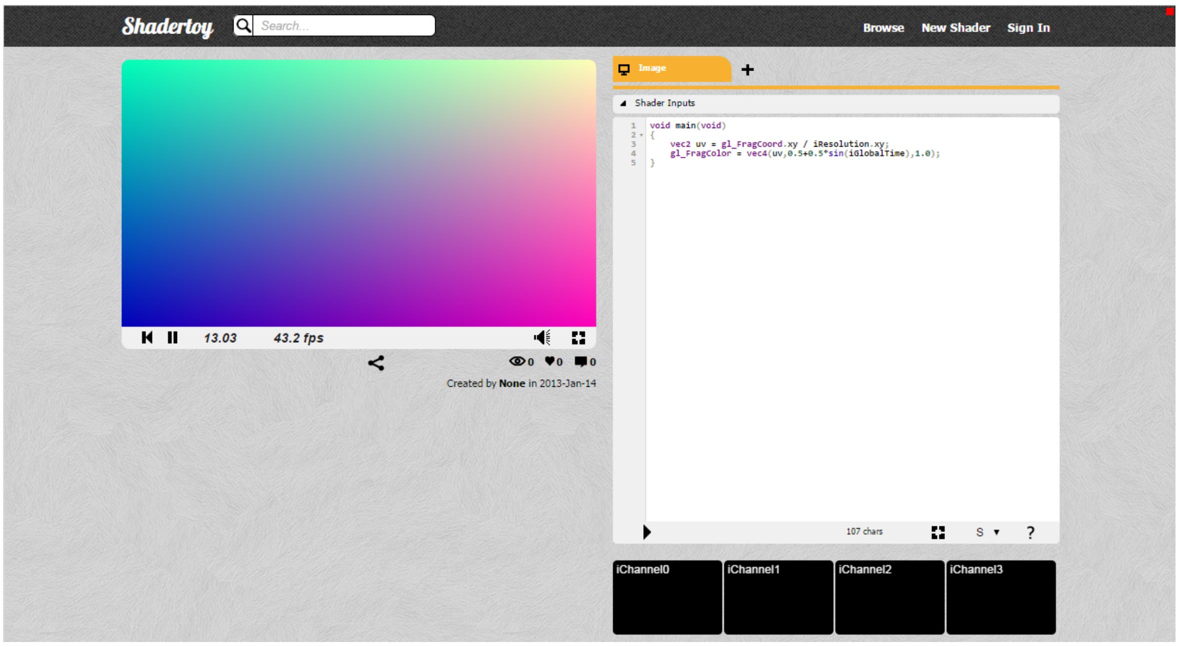


Figure 6, OpenGL 4 rendering pipeline (dotted boxes are optional stages) [11]

Vertex specification stage prepares vertex array data for shader pipeline (GPU). Vertex array is created in the main application. It contains 3D objects to be drawn to screen buffer. Objects are mage from drawing primitives like triangles, dots and lines. How this vertex mesh is actually interpreted is decided in shader pipeline.

Shader code is basically C-like language, which is the programming language of the shaders. Figure 7 shows fragment shader, which colors the display with RGB color shades.



```
void main(void)
{
    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    gl_FragColor = vec4(uv,0.5+0.5*sin(iGlobalTime),1.0);
}
```

Figure 7, Simple shader illustrated with www.shadertoy.com

Vertex shader is the main shader stage; it operates for each vertex in the render scene. The main functionality of vertex shader is to operate each vertex on the render scene. Vertex shader receives the inputs from the main application and converts each input vertex into a single output vertex based on an arbitrary, user-defined program a shader program. If there are not any shader stages, which mutate vertices, then the vertex shader output is considered to fill in that position with the clip-space position of the vertex for screen buffer. [23]

Vertex shader takes places in post-projection space before fragment shader stage. Vertex shader takes vertex attribute data as an input and outputs modified vertex data. If the input data hasn’t changed i.e. input and output data are identical then data isn’t reprocessed, but fetched in the post-transform cache, which saves the previous calculation result.

Tessellation shader provides possibility to subdivide vertex meshes to even smaller ones so that those can have new geometric transformations. Tessellation shader consists of Tessellation Control Shader (TCS) and Tessellation Evaluation Shader (TES). Former comes first and applies given amount of tessellation. Practical example would be paved street or rubble.

Geometry shader became available with Direct3D 10 and OpenGL 3.2. Geometry shader can manipulate graphics primitives that were sent to GPU. Geometry shaders are user-defined programs that process each incoming primitive, returning zero or more output primitives [17].

The output of a geometry shader is zero or more simple primitives, much like the output of the primitive assembly. The geometry shader is able to remove primitives, or tessellate them by outputting many primitives for a single input. The geometry shader can also tinker with the vertex values themselves, either doing some of the work for the vertex shader, or just to interpolate the values when tessellating them. Geometry shaders can even convert primitives to different types; input point primitives can become triangles, or lines can become points. Geometry shader gets executed after vertex and tessellation shaders. [14]

Graphics primitives who survive to this stage in the pipeline are rasterized in given order. Rasterization stage outputs a sequence of fragments for fragment shader. A fragment is a set of a state that is used to compute the final data for a pixel or a sample if multisampling is enabled in the output frame buffer. The state for a fragment includes its position in screen-space, the sample coverage if multisampling is enabled, and a list of arbitrary data that was outputted from the previous vertex or geometry shader. This last set of data is computed by interpolating between the data values in the vertices for the fragment. The shader that outputted those values defines the style of interpolation. [14]

Fragment shader or pixel shader operates on each pixel that is rendered to the screen. In HLSL, the fragment shader is called as a pixel shader and in GLSL it is called a fragment shader. Pixel shader manipulates the pixel color and the alpha value (translucent). Pixel shader also requires heavy calculation since the amount of pixels is huge compared to the amount of graphics primitives. Fragment shader is optimal and without it the depth- and stencil values get their original values. [17]

4.7 WebGL rendering pipeline

Figure 8 shows the WebGL rendering pipeline. In comparison to the OpenGL rendering pipeline WebGL's counterpart is rather simplified. Due to OpenGL ES 2.0 specification all the modern techniques are not available due to hardware constraints.

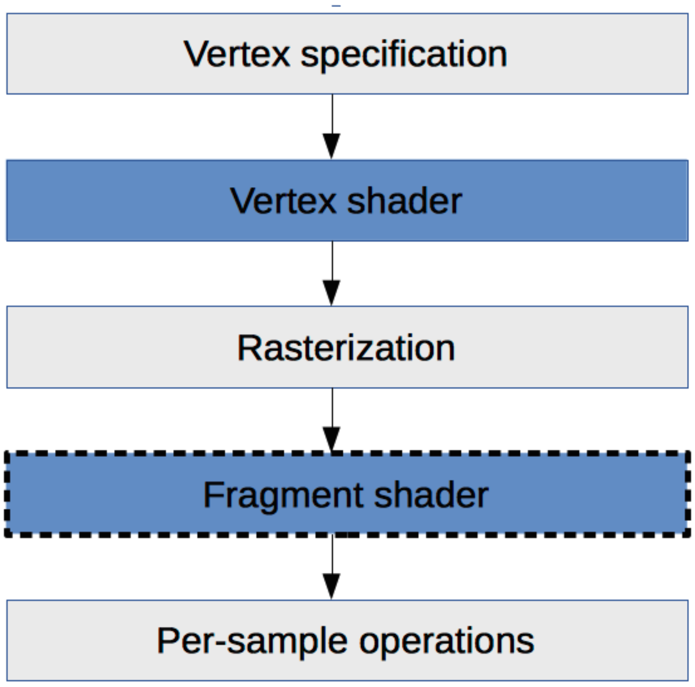


Figure 8, WebGL rendering pipeline (dotted boxes are optional stages)

Newer shader stages are not present in OpenGL ES specification. WebGL rendering pipeline has only vertex- and fragment shader compared to OpenGL's rendering pipeline. [19]

5 IMPLEMENTATION COMPARISON

This chapter discusses implementation of the simple three-dimensional application in various implementation techniques. Criteria for selection implementation are: multiplatform, widely adopted and freely available private and even commercial use. Multiplatform means in this context Windows, Linux and Mac OS X, whereas freely available means that there aren't any license costs. Widely adopted vaguely means that there are good existing documentation and community support. Khronos group is developing new rewritten version of OpenGL, but it has not been released yet. Microsoft got its Direct3D API, which is market leader, but unfortunately limited only to Windows platform so it's out of question. Apple released its own Metal 3D graphics API in June 2015, but like Direct3D it's restricted to the publishers own platforms OS X and iOS.

After a short market study there are only one viable option for implementation; OpenGL and its variations like OpenGL ES. OpenGL is developed by extensions, which are merged to new releases, if developer community accepts them. [20] Compared to Microsoft's Direct3D, OpenGL is developed by community whereas Microsoft manages Direct3D development process. [18]

WebGL provides two- and three-dimensional graphics for modern web browser. WebGL utilizes HTML canvas element to draw graphics. WebGL ecosystem provides many viable 3D graphics programming libraries. Most known of them are Three.js and Bablylon.js libraries. Both Three.js and Babylon.js raise the abstraction level of the coding high compared to the raw OpenGL ES API. WebGL uses OpenGL ES 2.0 specification, which has a wide hardware support. On the other hand, the calculation heavy effects and features are not available via WebGL. For example multisampling anti-aliasing is not available via WebGL. OpenGL ES 2.0 is loosely based on old OpenGL 2.0 specification, which was originally released in September 2004. Shader code is written either in HTML file using <script> tag is straight into JavaScript code in String format.

Original WebGL standard was released in March 2011. WebGL 2.0 specification started few years later 2013. WebGL is based on OpenGL ES 2.0. [11]

5.1 Solution comparison

In this chapter we are going to create a simple 3D application via WebGL and OpenGL. The main characteristics of both implementations are noted below.

1. Create renderer
2. Creating a scene
3. Implementing run loop

Both WebGL and OpenGL implementations have roughly the same abstraction level. This is mainly because of comparison purpose. OpenGL and WebGL will be compared between each other, but their frameworks will not. WebGL is JavaScript API and the WebGL is limited set of OpenGL API calls. Application initialization is pretty straightforward with JavaScript WebGL. The developer has to initialize a HTML canvas element and set view matrices, draw an example element like a cube, which is a kind of "Hello world" in computer graphics coding.

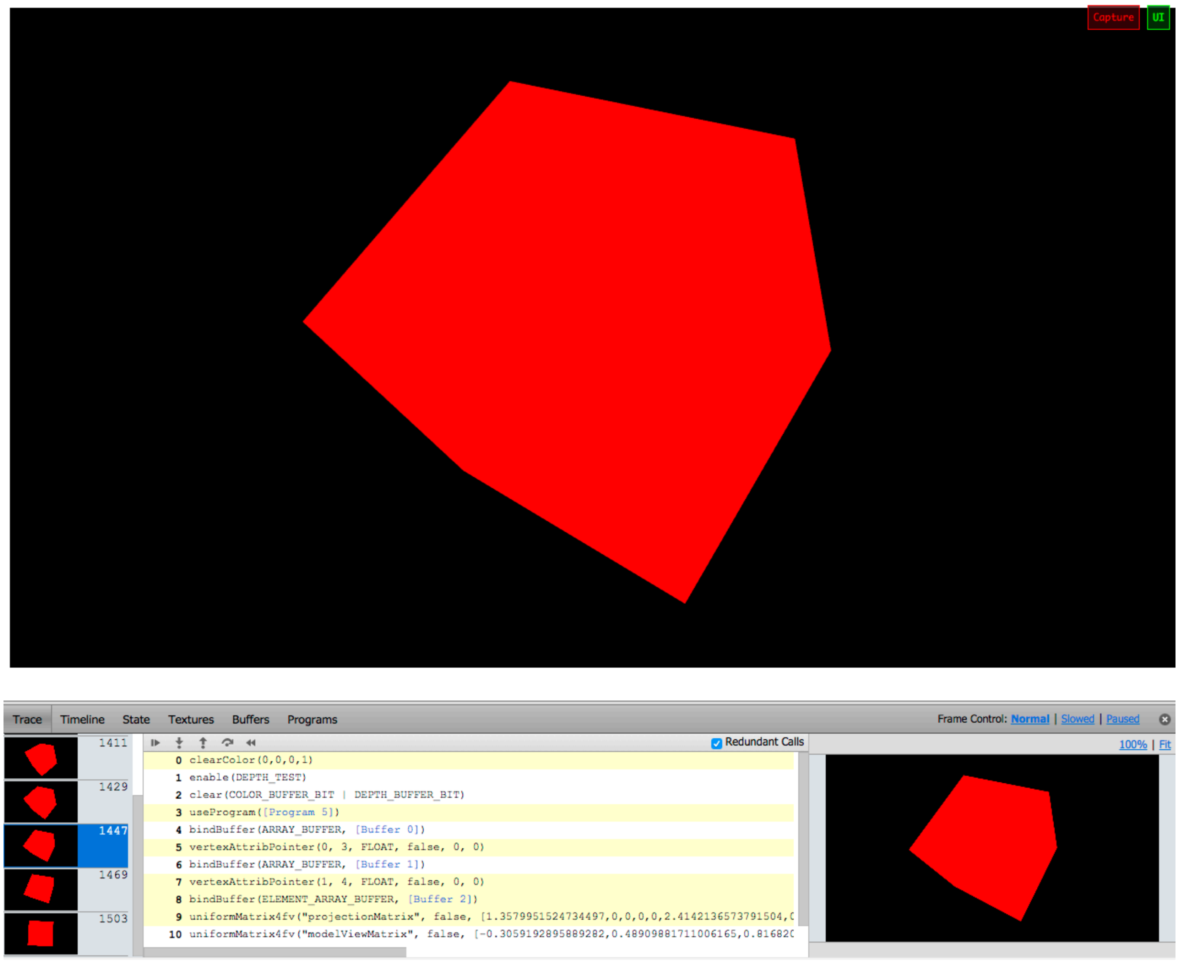
OpenGL version is implemented in both C++ and Rust. Rust is a new open source compiled low level programming language developed by Mozilla. Rust is not just another "academic language", since its original release in 2010 it has received a good community, which drives it forward. Rust brings memory safety to low level programming while keeping performance nearly at level of the C-code. In Rust safety means not only type safety, but memory safety too. Rust doesn't have garbage collection mechanism like Java has, but compiler takes care of the memory allocation and frees resources after they are no longer needed. Rust puts lots of effort to memory safety and frees developer's to solve other problems.

5.2 Tools

Implementations presented in previous chapter are platform independent. OpenGL's GLEW library handles platform specific window opening routines. Both solutions were implemented with MacBook Pro (Retina, 15-inch, Late 2014), which has Nvidia's GT 750M graphics card. GT750M is capable of OpenGL 4.1. The example code was executed on Google Chrome version 45.0.2454.101 (64-bit) in OS X operating system. Due to browser development the code might not be executable in the near future.

While coding the implementations, community support was relied heavily. The toughest part was the shader debugging, since unlike Microsoft's Direct3D OpenGL counterpart doesn't provide good tooling for profiling frames and shader code execution in OS X. In OS X the only viable shader debugger is XCode's OpenGL ES shader debugger for iOS. Valve Software is developing promising VoGL OpenGL capture / playback debugger. (https://github.com/ValveSoftware/vogl). Currently VoGL only supports Windows and Linux operating systems.

When a desktop version of the OpenGL is hard to debug the web browser version WebGL is much easier to debug. Ben Vanik from Google has made a WebGL Inspector tool, which is available as [28], but in a short test WebGL inspector was superior over its competitors. WebGL inspector can highlight redundant WebGL calls for potential optimization. WebGL haven't been updated in a while latest update is from July 2012. There is a mention in project's home page that the author is currently working on with version 2, which is supposed to be a total rewrite for the WebGL Inspector. In the figure 9 there is picture of the WebGL inspector in action. Yellow lines in the console means possible dead code lines.



```
0 clearColor(0,0,0,1)
1 enable(DEPTH_TEST)
2 clear(COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT)
3 useProgram([Program 5])
4 bindBuffer(ARRAY_BUFFER, [Buffer 0])
5 vertexAttribPointer(0, 3, FLOAT, false, 0, 0)
6 bindBuffer(ARRAY_BUFFER, [Buffer 1])
7 vertexAttribPointer(1, 4, FLOAT, false, 0, 0)
8 bindBuffer(ELEMENT_ARRAY_BUFFER, [Buffer 2])
9 uniformMatrix4fv("projectionMatrix", false, [1.3579951524734497,0,0,0,0,2.4142136573791504,0
10 uniformMatrix4fv("modelViewMatrix", false, [-0.3059192895889282,0.48909881711006165,0.81682
```

Figure 9, WebGL inspector

Rust is the most recent technology of the chosen implementation languages and so its development tools support is poor. After 1.0 release on May 2015 there are only simple auto completion named racer (https://github.com/phildawes/racer), which works weakly in real use. Coding with Rust is basically text editor + compiler in terminal screen this is evidendently the weakest point of graphics programming with Rust. All new technologies suffer this and Rust is not an exception in this area. Rust compiler kind of nuisance since it takes care of many possible error scenarios and makes some precautions for null pointers and type checking.

5.3 Development libraries

Each implementation is done with same abstraction level so that comparison between them is somewhat fair. Each solution handles matrix calculation via libraries. WebGL uses gl-matrix.js library, which basically wraps matrix operations like normalization, view matrix creation through perspective function and matrix manipulation functions like translate, rotate and scale.

C++ OpenGL uses GLEW (OpenGL Extension Wrangler Library), GLUT (OpenGL Utility Toolkit) and GLM (OpenGL Mathematics) libraries. GLEW library handles the hardware feature detection for OpenGL extensions, which provides mechanism to support various hardware setups. GLEW provides information about hardware capability, cross platform window creation routines and IO-operations. GLEW exposes hardware capability through global variables like GLEW_VERSION_4_1, which basically provides info if OpenGL 4.1 is support in current runtime platform. Window creation is also handy, because all three operating systems use different window creation routines.

Rust example application uses glium library for OpenGL wrapping. Glium is an API layer between OpenGL3+ and Rust code. Glium API tries to overcome OpenGL's error prone API. Glium uses high-level stateless API, which doesn't have any set methods. Everything is passed as a parameter value, so that values aren't mutable. Mutable values are the root of all evil since it's not guaranteed that initialization values are used in later stages of the program. Immutable values (counterpart of the mutable values) are not free since they require new stack- or heap memory allocation by compiler. Memory operations are way heavier operations than simple mutations in the already allocated values. Glium is constructed above GLUT (OpenGL Utility Toolkit) and so it exposes some functionality like its C++ counterpart. Glium provides good documentation and many examples for one-year-old GitHub project.

Comparing code snippets (see the attachments 1, 2 and 3) one can easily see that native OpenGL solution is much longer. OpenGL solution is longer because of arcane syntax of C+. There is also quite a bit similar code lines when it comes to actual GL code. GLSL version of C++ solution is much newer since WebGL is OpenGL ES 2.0, which in turn is basically OpenGL 3.0. Table 5 shows simple feature comparison of the each implementation method.

Table 5, statistics

	WebGL	OpenGL / Rust	OpenGL / C++11
Release year	2007	2010	1983/2011
Variable typing	Dynamic typing	Static typing	Strong static typing
OpenGL version	ES 2.0	GLSL 4.1	GLSL 4.1
GLSL version	Old	Modern	Modern

5.4 Code

WebGL was easy to start with subset of modern OpenGL functions and interpreted programming language guaranteed the fast prototyping of different techniques, where as the plain OpenGL solution with C++ and Rust required compilation of the whole project, if the shader code was included to source (as it was for the sake of simplicity). With WebGL there was also possible to live edit the shader, when the developer didn't instantiate new variables in the shader code. Mozilla Firefox has a feature in its development tools to live edit the shader code.

Compiled code solutions didn't show their best potential in the small example application. Static typing of both Rust and C++ reduces the type errors in the application, but can also be quite cumbersome to develop. C++ type safety is kind of hard to achieve since the language doesn't force types be default, but require developer attention to manage the type safety. Compiler does the type checking and makes executable, which can be debugged via GDB (GNU Project Debugger) or LLDB debugger. Both debuggers can examine runtime behavior of the program. Rust programs are nearly impossible to crash at run time if the exceptions and return values are handled correctly. Rust doesn't have checked exceptions (exceptions that are forced at compile time). So the developer might miss the error handling and then crash the program at run time.

6 DISCUSSION

After through out examination of each solution and their advantages and disadvantages, it seems that there isn’t a clear winner in the comparison. Each solution has its pros and cons, but it seems that open source alternatives to domination Direct3D are a viable option. Although the goal of this research is to give an objective review of each solution, many developers prefer to stay on their comfort zone and this master’s thesis points out other possibilities outside what developers are used to doing and not to give a definitive rule to decide, which of the implementation technologies is the best. Like many areas in computer science there isn’t a single silver bullet, but each solution has its strengths and weaknesses. Some use cases other solution is better than other one, while in the other scenarios it’s terrible choice.

6.1 WebGL

WebGL got the best tooling in the test set for building, debugging and development. Due to JavaScript’s popularity community support is great. There is a WebGL Fundaments (http://webglfundamentals.org/) site, which gives good instructions to people new to WebGL development. JavaScript has many options of compile-to-JavaScript language like typescript and Babel, if a developer wants to use all modern ES6 (ECMAScript) features. NPM (Node Package Manager) provides good scripting for various builds and NPM got very broad amount of libraries for developers. All in all WebGL solution is evidently best for fast prototyping of the applications, due to dynamic typing and good tooling.

6.2 OpenGL C++

C++ is the oldest of the three technologies although C++11 and C++14 give wide variety of modern features to it. Legacy of the C++ weights heavily upon community documentation since much of the C++ OpenGL example on the internet are written before OpenGL 3.0, when the fixed function pipeline was way to go and there wasn’t programmable shaders like nowadays. After searching and investigating Internet tutorials there are at least Learn OpenGL (www.learnopengl.com) site, which provides good OpenGL examples implemented with C++. Biggest drawback with the C++ solution is the missing package manager for the library dependencies like matrix calculation libraries. Compared to WebGL's node package manager the static importing of the C++ libraries is laborious, error prone and also platform dependent. Tooling for the C++ code is code, because of the popularity of the language. Memory leaks and the dangling pointers are the constant nuisance with C++ development. These are the issue, which attempts to solve and succeeds with quite good.

6.3 OpenGL Rust

Rust is the newest of the three technologies. Basically Rust implementation has the same characteristics as C++ solution, but with better memory safety and less error prone programming language. While Rust solution frees developer from memory handling the compiler is quite nuisance since it tries to make sure that developer doesn't code any stupid things. Developer's time goes with the compiler warnings and errors whereas C++ code breaks at run time. Rust solution also has a good package manager. Packages in the Rust's world are called crates. Cargo tool is the Rust's package manager that handles crate downloading and install platform independently. Glium library, which was used with the Rust solution, seemed to be good wrapper over GLUT. It provided fast application development mainly due to good documentation.

7 CONCLUSION

This master’s thesis took a look to modern computer graphics via open source and multiplatform solutions. The objective was to create a simple example with each technology and compare the development experience, tools and results. Research started with literature research from the University Library’s database to get scientifically approved resource material for the research. After the literature research it was clear that there was much info about modern technologies, but each research paper got few years old data. This was good enough to get some information about technologies. The literature research was accompanied with literature and web resources to get most recent information.

After the literature research the theory part took a look at the about computer graphics in general. How the code constructs visible frames to the computer’s display. There was a graphics theory section, which dealt with implementation patterns and discussed the 3D graphics APIs, libraries and frameworks. Libraries and frameworks section tried to give information about current state of the development libraries and frameworks like Three.js and Babylon.js. There were also some frameworks like X3DOM, which were mentioned in the research literature results.

In the implementation part three different kinds of solutions were implemented, examined and crosschecked. Development technologies were selected from most common ones and there was Rust for give some insight of the newer low level language. Native solutions were based on OpenGL and browser version to pure WebGL. Each solution was implemented roughly in same abstraction level so that the results would be comparable.

Abstraction level of the implementation was that matrix operations were implemented in the library level and not in the actual source code. Results point out pros and cons of the solutions. WebGL got the best tooling and wide variety development libraries due to popularity of the JavaScript, C++ got lots of existing documentation, but many of the resources were obsolete in current standards. Rust has lot to offer, but is quite new programming language, but certainly less error prone compared to C++ counterpart.

Nothing certain can be said about the validity of the results since they are highly subjective and somewhat limited. It would be good to get broader review board to this kind of research, but at least there are some viable options for proprietary direct3D.

REFERENCES

[1] Balbaert Ivo, Rust Essentials, Packt publishing, 2015

[2] Dickheiser Michael, C++ for Game Programmers 2007. USA: Wordware Publishing Inc.

[3] Evans Alun et al, 3D Graphics on the Web: A survey, Interactive Technologies Group, Universitat Pompeu Fabra, Barcelona, Spain 2014

[4] Feinstein, D, HLSL Development Cookbook, Packt Publishing 2013.

[5] Frustum Culling, Retrieved November 21 from http://cube3dev.blogspot.fi/2013/03/frustum-culling-unoptimized.html

[6] Github info, Retrieved September 27 2015 from http://githut.info/

[7] Glium project (2015, November 21), https://github.com/tomaka/glium

[8] Graphics Performance in Rich Internet Applications, Rama C. Hoetzlein Fraunhofer IGD, Darmstadt, Aalborg University Copenhagen 2012

[9] Gregory, J. (2009). Game Engine Architecture. A K Peters/CRC Press.

[10] Interpretive OpenGL for computer graphics, Bo Cheng, Integration Engineering

[11] Khronos Group, Retrieved 24.11.2015 from https://www.khronos.org/

[12] Laboratory, Department of Mechanical and Aeronautical Engineering, University of California 2005

[13] Lay, D. C. Linear Algebra and Its Applications. painopaikka tuntematon: Pearson Education.

[14] Learn OpenGL, Retrieved November 21 2015 from http://learnopengl.com/

[15] Lengyel, E. (2004). Mathematics for 3D Game Programming & Computer Graphics. USA: Delmar Cengage Learning.

[16] Levkowitz Haim, Kelleher Curran, Cloud and mobile Web-based graphics and visualization, 2012 XXV SIBGRAPI Conference on Graphics, Patterns and Images Tutorials

[17] Luna, F. D, Introduction to 3D Game Programming with DirectX 11, David Pallai 2012.

[18] Microsoft MSDN Library, DirectX reference. (Microsoft), Retrieved June 27 2012, from http://msdn.microsoft.com/en-us/library/ee663274(v=vs.85).aspx

[19] Mozilla Developer Network WebGL, Retrieved August 10 2015 from https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[20] Nichaelis Nils, Yung Yvonne, Fraunhofer Johannes, Virtual Heritage to Go, IGD, Darmstadt, Germany 2012

[21] OpenGL is Broken, Retrieved November 23 2015 from
http://www.joshbarczak.com/blog/?p=154

[22] Programming 3D Applications with HTML5 and WebGL, Tony Parisi O'Reilly Media 2014

[23] Puhakka Antti, 3D-grafiikka, Talentum 2008

[24] Resig John, Bibeault Bear, Secrets of the JavaScript Ninja, Manning Publications 2012

[25] Rodkaew Yodthong, The Last Eternity a 3D Role-Playing Game with a Cross-Platform Development International Computer Science and Engineering Conference 2013

[26] McShaffry Mike, Graham David, Game Coding Complete Course Technology 2013

[27] Tiobe Index, Retrieved September 27 2015 from http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[28] WebGL Debugging and Profiling Tools, Retrieved November 21 2015 from http://www.realtimerendering.com/blog/webgl-debugging-and-profiling-tools/

[29] WebGL-Inspector, Retrieved November 21 2015 from http://benvanik.github.io/WebGL-Inspector/

[30] WebGLStudio.js, Retrieved November 21 2015 from http://webglstudio.org/

[31] Yikang Yang, Xinxing Liu, Lei Liu, A WebGL Based Method For Visualization Of Space Environment, 2012 Fifth International Symposium on Computational Intelligence and Design

APPENDIX

7.1 Rust

```
#[macro_use]
extern crate glium;

fn main() {
    use glium::{DisplayBuild, Surface};
    let display = glium::glutin::WindowBuilder::new()
    .with_dimensions(1280, 720)
    .with_title(format!("Rust example"))
    .build_glium().unwrap();

    #[derive(Copy, Clone)]
    struct Vertex {
        position: [f32; 3],
    }

    #[derive(Copy, Clone)]
    struct Index {
        position: [f32; 2],
    }

    #[derive(Copy, Clone)]
    struct Normal {
        normal: (f32, f32, f32)
    };

    implement_vertex!(Vertex, position);
    implement_vertex!(Normal, normal);

    const VERTICES: [Vertex; 24] = [
    Vertex { position: [-0.5, -0.5, -0.5] },
    Vertex { position: [  -0.5, -0.5,  0.5] },
    Vertex { position: [  0.5, -0.5,  0.5] },
    Vertex { position: [   0.5, -0.5, -0.5] },
    Vertex { position: [  0.5,  0.5, -0.5] },
    Vertex { position: [  0.5,  0.5,  0.5] },
    Vertex { position: [ -0.5,  0.5,  0.5] },
    Vertex { position: [  -0.5,  0.5, -0.5] },

    Vertex { position: [     -0.5,  0.5, -0.5] },
    Vertex { position: [  -0.5,  0.5,  0.5] },
    Vertex { position: [ -0.5, -0.5,  0.5] },
    Vertex { position: [  -0.5, -0.5, -0.5] },

    Vertex { position: [  0.5, -0.5, -0.5] },
    Vertex { position: [  0.5, -0.5,  0.5] },
    Vertex { position: [  0.5,  0.5,  0.5] },
    Vertex { position: [  0.5,  0.5, -0.5] },

    Vertex { position: [  0.5, -0.5, -0.5] },
    Vertex { position: [   0.5,  0.5, -0.5] },
    Vertex { position: [ -0.5,  0.5, -0.5] },
    Vertex { position: [ -0.5, -0.5, -0.5] },

    Vertex { position: [ -0.5, -0.5,  0.5] },
    Vertex { position: [ -0.5,  0.5,  0.5] },
    Vertex { position: [  0.5,  0.5,  0.5] },
    Vertex { position: [ 0.5, -0.5,  0.5] },
    ];

    let indices: [u16; 36] = [
    0, 1, 2,
    0, 2, 3,
    4, 5, 6,
    4, 6, 7,
    8, 9,10,
    8,10,11,
    12,13,14,
    12,14,15,
    16,17,18,
    16,18,19,
    20,21,22,
    20,22,23u16
    ];

    const NORMALS: [Normal; 6] = [
    Normal { normal: (0.0, 0.0, 0.0) },
    Normal { normal: (0.0, 0.0, 0.0) },
    Normal { normal: (0.0, 0.0, 0.0) },
    Normal { normal: (0.0, 0.0, 0.0) },
    Normal { normal: (0.0, 0.0, 0.0) },
    Normal { normal: (0.0, 0.0, 1.0) },
    ];

    let positions = glium::VertexBuffer::new(&display, &VERTICES).unwrap();
    let normals = glium::VertexBuffer::new(&display, &NORMALS).unwrap();
    let indices = glium::IndexBuffer::new(&display, glium::index::PrimitiveType::TrianglesList,
&indices).unwrap();

    let vertex_shader_src = r#"
        #version 410
        in vec3 position;
        in vec3 normal;
        out vec3 v_normal;
        uniform mat4 perspective;
        uniform mat4 view;
        uniform mat4 model;
        void main() {
            mat4 modelview = view * model;
            v_normal = transpose(inverse(mat3(modelview))) * normal;
            gl_Position = perspective * modelview * vec4(position, 1.0);
        }
    "#;

    let fragment_shader_src = r#"
    #version 410
    in vec3 v_normal;
    out vec4 color;
    uniform vec3 u_light;

    void main() {
        float brightness = dot(normalize(v_normal), normalize(u_light));
        vec3 dark_color = vec3(0.5, 0.0, 0.0);
        vec3 regular_color = vec3(1.0, 0.0, 0.0);
        color = vec4(mix(dark_color, regular_color, brightness), 1.0);
    }
    "#;

    let program = glium::Program::from_source(&display, vertex_shader_src, fragment_shader_src,
None).unwrap();
    loop {
        let mut target = display.draw();
        target.clear_color(0.0, 0.0, 0.0, 1.0);

        let model = [
            [ 0.5, 0.0, 0.0, 0.0 ],
            [ 0.0, 0.5, 0.0, 0.0 ],
            [ 0.0, 0.0, 0.5, 0.0 ],
            [ 0.0, 0.0, 2.0, 1.0f32 ]
        ];

        let view = view_matrix(&[2.0, -1.0, 1.0], &[-2.0, 1.0, 1.0], &[0.0, 1.0, 0.0]);

        let perspective = {
            let (width, height) = target.get_dimensions();
            let aspect_ratio = height as f32 / width as f32;

            let fov: f32 = 3.141592 / 3.0;
            let zfar = 1024.0;
            let znear = 0.1;

            let f = 1.0 / (fov / 2.0).tan();

            [
            [f *   aspect_ratio   ,    0.0,              0.0              ,   0.0],
            [         0.0         ,     f ,              0.0              ,   0.0],
            [         0.0         ,    0.0,  (zfar+znear)/(zfar-znear)    ,   1.0],
            [         0.0         ,    0.0, -(2.0*zfar*znear)/(zfar-znear),   0.0],
            ]
        };
        let light = [1.4, 0.4, 0.7f32];
        target.draw((&positions, &normals), &indices, &program,
            &uniform! { model: model, view: view, perspective: perspective, u_light: light },
            &Default::default()).unwrap();
        target.finish().unwrap();

        for ev in display.poll_events() {
            match ev {
                glium::glutin::Event::Closed => return,
                _ => ()
            }
        }
    }
}

fn view_matrix(position: &[f32; 3], direction: &[f32; 3], up: &[f32; 3]) -> [[f32; 4]; 4] {
    let f = {
        let f = direction;
        let len = f[0] * f[0] + f[1] * f[1] + f[2] * f[2];
        let len = len.sqrt();
        [f[0] / len, f[1] / len, f[2] / len]
    };

    let s = [up[1] * f[2] - up[2] * f[1],
             up[2] * f[0] - up[0] * f[2],
             up[0] * f[1] - up[1] * f[0]];

    let s_norm = {
        let len = s[0] * s[0] + s[1] * s[1] + s[2] * s[2];
        let len = len.sqrt();
        [s[0] / len, s[1] / len, s[2] / len]
    };

    let u = [f[1] * s_norm[2] - f[2] * s_norm[1],
             f[2] * s_norm[0] - f[0] * s_norm[2],
             f[0] * s_norm[1] - f[1] * s_norm[0]];

    let p = [-position[0] * s_norm[0] - position[1] * s_norm[1] - position[2] * s_norm[2],
            -position[0] * u[0] - position[1] * u[1] - position[2] * u[2],
            -position[0] * f[0] - position[1] * f[1] - position[2] * f[2]];

    [
        [s[0], u[0], f[0], 0.0],
        [s[1], u[1], f[1], 0.0],
        [s[2], u[2], f[2], 0.0],
        [p[0], p[1], p[2], 1.0],
    ]
}
```

7.2 C++

```
#include <stdio.h>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

GLFWwindow *window;

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

using namespace glm;

#include "shader.hpp"

int main(void) {
    if (!glfwInit()) {
        fprintf(stderr, "Failed to initialize GLFW\n");
        getchar();
        return -1;
    }

    glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    window = glfwCreateWindow(1280, 720, "OpenGL", NULL, NULL);
    if (window == NULL) {
        fprintf(stderr,
                "Failed to open GLFW window\n");
        getchar();
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);

    glewExperimental = true;
    if (glewInit() != GLEW_OK) {
        fprintf(stderr, "Failed to initialize GLEW\n");
        getchar();
        glfwTerminate();
        return -1;
    }

    glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    GLuint VertexArrayID;
    glGenVertexArrays(1, &VertexArrayID);
    glBindVertexArray(VertexArrayID);

    GLuint programID = LoadShaders("vertex.glsl", "fragment.glsl");
    GLuint MatrixID = glGetUniformLocation(programID, "MVP");


    glm::mat4 Projection = glm::perspective(glm::radians(45.0f), 4.0f / 3.0f, 0.1f, 100.0f);

    glm::mat4 View = glm::lookAt(
            glm::vec3(4, 3, -3),
            glm::vec3(0, 0, 0),
            glm::vec3(0, 1, 0)
    );

    glm::mat4 Model = glm::mat4(1.0f);

    glm::mat4 MVP = Projection * View * Model;

    static const GLfloat g_vertex_buffer_data[] = {
            -1.0f, -1.0f, -1.0f,
            -1.0f, -1.0f, 1.0f,
            -1.0f, 1.0f, 1.0f,
            1.0f, 1.0f, -1.0f,
            -1.0f, -1.0f, -1.0f,
            -1.0f, 1.0f, -1.0f,
            1.0f, -1.0f, 1.0f,
            -1.0f, -1.0f, -1.0f,
            1.0f, -1.0f, -1.0f,
            1.0f, 1.0f, -1.0f,
            1.0f, -1.0f, -1.0f,
            -1.0f, -1.0f, -1.0f,
            -1.0f, -1.0f, -1.0f,
            -1.0f, 1.0f, 1.0f,
            -1.0f, 1.0f, -1.0f,
            1.0f, -1.0f, 1.0f,
            -1.0f, -1.0f, 1.0f,
            -1.0f, -1.0f, -1.0f,
            -1.0f, 1.0f, 1.0f,
            -1.0f, -1.0f, 1.0f,
            1.0f, -1.0f, 1.0f,
            1.0f, 1.0f, 1.0f,
            1.0f, -1.0f, -1.0f,
            1.0f, 1.0f, -1.0f,
            1.0f, -1.0f, -1.0f,
            1.0f, 1.0f, 1.0f,
            1.0f, -1.0f, 1.0f,
            1.0f, 1.0f, 1.0f,
            1.0f, 1.0f, -1.0f,
            -1.0f, 1.0f, -1.0f,
            1.0f, 1.0f, 1.0f,
            -1.0f, 1.0f, -1.0f,
            -1.0f, 1.0f, 1.0f,
            1.0f, 1.0f, 1.0f,
            -1.0f, 1.0f, 1.0f,
            1.0f, -1.0f, 1.0f
    };
    GLuint vertexbuffer;
    glGenBuffers(1, &vertexbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data,
GL_STATIC_DRAW);

    GLuint colorbuffer;
    glGenBuffers(1, &colorbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);

    do {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glUseProgram(programID);

        glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
        glEnableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
        glVertexAttribPointer(
                0,
                3,
                GL_FLOAT,
                GL_FALSE,
                0,
                (void *) 0
        );

        glEnableVertexAttribArray(1);
        glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
        glVertexAttribPointer(
                1,
                3,
                GL_FLOAT,
                GL_FALSE,
                0,
                (void *) 0
        );

        glDrawArrays(GL_TRIANGLES, 0, 12 * 3);
        glDisableVertexAttribArray(0);
        glDisableVertexAttribArray(1);
        glfwSwapBuffers(window);
        glfwPollEvents();

    }
    while (glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&
           glfwWindowShouldClose(window) == 0);

    glDeleteBuffers(1, &vertexbuffer);
    glDeleteBuffers(1, &colorbuffer);
    glDeleteProgram(programID);
    glDeleteVertexArrays(1, &VertexArrayID);

    glfwTerminate();

    return 0;
}
```

7.3 WebGL

```
<html>
<head>
  <title>WebGL</title>
  <script src="lib/gl-matrix.js"></script>
</head>
<body>
<canvas id="webglcanvas" style="border: none" width="1280" height="720"></canvas>
<script type="text/javascript">
  const initWebGL = (canvas) => {
    try {
      return canvas.getContext("webgl")
    } catch (e) {
      throw new Error("Error creating WebGL Context " + e.toString())
    }
  }

  const initViewport = (gl, canvas) => gl.viewport(0, 0, canvas.width, canvas.height)
  var projectionMatrix, modelViewMatrix, rotationAxis

  const initMatrices = (canvas) => {
    modelViewMatrix = mat4.create()
    mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -5])

    projectionMatrix = mat4.create()
    mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width / canvas.height, 1, 10000)

    rotationAxis = vec3.create()
    vec3.normalize(rotationAxis, [1, 1, 1])
  }

  const createCube = (gl) => {
    var vertexBuffer = gl.createBuffer()
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer)
    const verts = [
      // Front face
```

```
        -1.0, -1.0, 1.0,
        1.0, -1.0, 1.0,
        1.0, 1.0, 1.0,
        -1.0, 1.0, 1.0,

        // Back face
        -1.0, -1.0, -1.0,
        -1.0, 1.0, -1.0,
        1.0, 1.0, -1.0,
        1.0, -1.0, -1.0,

        // Top face
        -1.0, 1.0, -1.0,
        -1.0, 1.0, 1.0,
        1.0, 1.0, 1.0,
        1.0, 1.0, -1.0,

        // Bottom face
        -1.0, -1.0, -1.0,
        1.0, -1.0, -1.0,
        1.0, -1.0, 1.0,
        -1.0, -1.0, 1.0,

        // Right face
        1.0, -1.0, -1.0,
        1.0, 1.0, -1.0,
        1.0, 1.0, 1.0,
        1.0, -1.0, 1.0,

        // Left face
        -1.0, -1.0, -1.0,
        -1.0, -1.0, 1.0,
        -1.0, 1.0, 1.0,
        -1.0, 1.0, -1.0
    ]
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW)

    var colorBuffer = gl.createBuffer()
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer)
    const faceColors = [
        [1.0, 0.0, 0.0, 1.0], // Front face
        [0.0, 1.0, 0.0, 1.0], // Back face
        [0.0, 0.0, 1.0, 1.0], // Top face
        [1.0, 1.0, 0.0, 1.0], // Bottom face
        [1.0, 0.0, 1.0, 1.0], // Right face
        [0.0, 1.0, 1.0, 1.0]  // Left face
    ]
    var vertexColors = []
    faceColors.forEach((color) => {
        color.forEach(() => {
            vertexColors = vertexColors.concat(color)
        })
    })
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexColors), gl.STATIC_DRAW)

    var cubeIndexBuffer = gl.createBuffer()
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeIndexBuffer)
    const cubeIndices = [
        0, 1, 2, 0, 2, 3,    // Front face
        4, 5, 6, 4, 6, 7,    // Back face
        8, 9, 10, 8, 10, 11,  // Top face
        12, 13, 14, 12, 14, 15, // Bottom face
        16, 17, 18, 16, 18, 19, // Right face
        20, 21, 22, 20, 22, 23  // Left face
    ]
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(cubeIndices), gl.STATIC_DRAW)

    return {
        vertexBuffer,
        colorBuffer,
        cubeIndexBuffer,
        vertSize: 3,
        nVerts: 24,
        colorSize: 4,
        nColors: 24,
        nIndices: 36,
        primtype: gl.TRIANGLES
    }
```

```
  }

   const createShader = (gl, str, type) => {
    var shader
    if (type == "fragment") {
      shader = gl.createShader(gl.FRAGMENT_SHADER)
    } else if (type == "vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER)
    } else {
      return null
    }

    gl.shaderSource(shader, str)
    gl.compileShader(shader)

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      console.error(gl.getShaderInfoLog(shader))
      return null
    }
    return shader
  }

  var vertexShaderSource =
      "attribute vec3 vertexPos;" +
      "attribute vec4 vertexColor;" +
      "uniform mat4 modelViewMatrix;" +
      "uniform mat4 projectionMatrix;" +
      "varying vec4 vColor;" +
      "void main(void) {" +
      "   gl_Position = projectionMatrix * modelViewMatrix * vec4(vertexPos, 1.0);" +
      "   vColor = vertexColor;" +
      "}";

  var fragmentShaderSource =
      "precision mediump float;" +
      "varying vec4 vColor;" +
      "void main(void) {" +
      "   gl_FragColor = vec4(1,0,0,1);" +
      "}";
  var shaderProgram, shaderVertexPositionAttribute, shaderVertexColorAttribute,
      shaderProjectionMatrixUniform, shaderModelViewMatrixUniform

  const initShader = (gl) => {
    var fragmentShader = createShader(gl, fragmentShaderSource, "fragment")
    var vertexShader = createShader(gl, vertexShaderSource, "vertex")

    shaderProgram = gl.createProgram()
    gl.attachShader(shaderProgram, vertexShader)
    gl.attachShader(shaderProgram, fragmentShader)
    gl.linkProgram(shaderProgram)

    shaderVertexPositionAttribute = gl.getAttribLocation(shaderProgram, "vertexPos")
    gl.enableVertexAttribArray(shaderVertexPositionAttribute)

    shaderVertexColorAttribute = gl.getAttribLocation(shaderProgram, "vertexColor")
    gl.enableVertexAttribArray(shaderVertexColorAttribute)

    shaderProjectionMatrixUniform = gl.getUniformLocation(shaderProgram, "projectionMatrix")
    shaderModelViewMatrixUniform = gl.getUniformLocation(shaderProgram, "modelViewMatrix")

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      console.error("Could not initialise shaders")
    }
  }

  const draw = (gl, obj) => {
    gl.clearColor(0.0, 0.0, 0.0, 1.0)
    gl.enable(gl.DEPTH_TEST)
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)

    gl.useProgram(shaderProgram)

    gl.bindBuffer(gl.ARRAY_BUFFER, obj.vertexBuffer)
    gl.vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT, false, 0, 0)
    gl.bindBuffer(gl.ARRAY_BUFFER, obj.colorBuffer)
    gl.vertexAttribPointer(shaderVertexColorAttribute, obj.colorSize, gl.FLOAT, false, 0, 0)
```

```
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, obj.cubeIndexBuffer)

        gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix)
        gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix)

        gl.drawElements(obj.primtype, obj.nIndices, gl.UNSIGNED_SHORT, 0)
    }

    var duration = 5000
    var currentTime = Date.now()

    const animate = () => {
        var now = Date.now()
        var deltaT = now - currentTime
        currentTime = now
        var angle = Math.PI * 2 * deltaT / duration
        mat4.rotate(modelViewMatrix, modelViewMatrix, angle, rotationAxis)
    }

    const run = (gl, cube) => {
        requestAnimationFrame(() => run(gl, cube))
        draw(gl, cube)
        animate()
    }

    window.onload = () => {
        var canvas = document.getElementById("webglcanvas")
        var gl = initWebGL(canvas)
        initViewport(gl, canvas)
        initMatrices(canvas)
        var cube = createCube(gl)
        initShader(gl)
        run(gl, cube)
    }
</script>
</body>
</html>
```