UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME CATTANI DE CASTRO

CG Guide: a Modern OpenGL and Computer Graphics teaching application

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Eduardo Simões Lopes Gastal

Porto Alegre
December 2021

ABSTRACT

Learning computer graphics is hard. Not only it involves a multitude of skills: spatial reasoning, mathematics, and physics. It is also a vast field with many different topics such as lighting, texture mapping, 3D transformations, etc. Added to it, there is also the complexity of its basics, with modern OpenGL it takes considerable effort to draw a shape on-screen. Given that it deals with intrinsically visual content, teaching computer graphics interactively is recommended. We did not find a teaching tool that was easy to use, extensible, and open sourced when looking for related work. We aim to fix this issue with an application called CG Guide, an interactive tool that runs on modern OpenGL and shows computer graphics scenes that can be changed in real-time. Its scenes are extensible and the code is open sourced. CG Guide comes with premade scenes that are ready to use and were created aiming to elucidate a specific concept of the principles of computer graphics, these include: texture mapping, shaders, matrix transformations, rendering, animations and 3D drawing.

Keywords: Computer graphics. OpenGL. teaching.

CG Guide, uma aplicação voltada para o ensino de conceitos de computação gráfica e de OpenGL moderno

RESUMO

Aprender computação gráfica é difícil. São envolvidas muitas habilidades: raciocínio espacial, matemática e física. É uma área vasta com muitos tópicos diferentes como iluminação, mapeamento de textura, transformações 3D, etc. Existe também a complexidade do seu básico, com OpenGL moderno é necessário um esforço considerável para desenhar uma forma na tela. Como computação gráfica é uma área intrinsicamente visual, ensiná-la interativamente é recomendado. Procurando por trabalhos com esse objetivo, não foram encontradas ferramentas de ensino que eram fáceis de usar, extensíveis e com o código fonte aberto. Nós nos propusemos a consertar esse problema com uma aplicação chamada CG Guide, uma ferramenta interativa que roda em OpenGL moderno e mostra cenas de computação gráfica que podem ser alteradas em tempo real. Suas cenas são extensíveis e o código fonte é aberto. O CG Guide já conta com cenas que foram criadas com o objetivo de clarificar conceitos específicos e princípios da computação gráfica, incluindo: mapeamento de textura, shaders, transformações matriciais, rendering, animação e desenho 3D.

Palavras-chave: computação gráfica, OpenGL, aprendizado.

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

GPU	Graphics Processing Unit
GUI	Graphic User Interface
API	Application Programming Interface
OpenGL	Open Graphics Library
GLSL	OpenGL Shading Language
GLFW	Graphics Library Framework
CG	Computer Graphics
NDC	Normalized Device Coordinates
VBO	Vertex Buffer Object
VAO	Vertex Array Object
MVPV	Model-View-Perspective-Viewport

CONTENTS

1 INTRODUCTION

Computer graphics is one of the most exciting and fast-evolving areas of computer science. New knowledge is being published every day at a high rate (BALREIRA; WALTER; FELLNER, 2017). It is a very expansive topic that can range from simple shapes on a screen to realistic scenes that are basically indistinguishable from reality. Given its visual appeal, it is easy to see how it evolved to its current groundbreaking state.

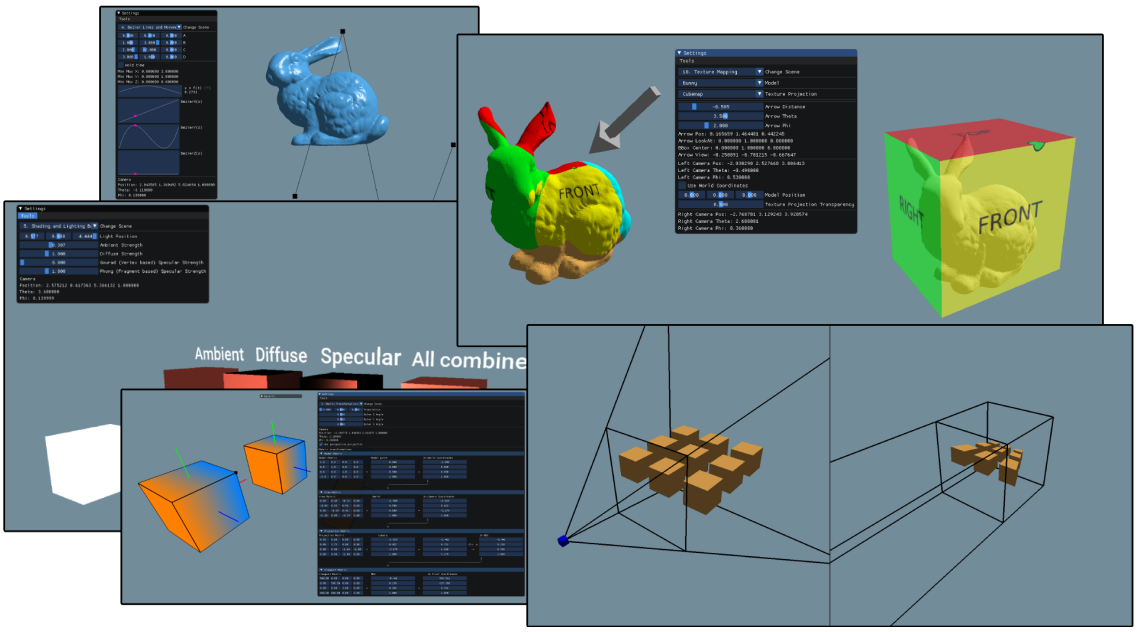
As computer graphics requires interdisciplinary skills, it can often be discouraging and hard for students to face it. According to Suselo, Wünsche and Luxton-Reilly (2017), insufficient skills in mathematics, programming, and spatial reasoning can impact the learning of computer graphics. In our institution, UFRGS[1], an introductory course on Computer Graphics (CG) had a failure rate of over 10% (NETO, 2021).

While the end result is no stranger to students (most have been exposed to a Graphic User Interface (GUI), three-dimensional animations, realistic renders, computer games, etc.), the approach of teaching is most commonly bottom up (SUNG; SHIRLEY, 2003), so lessons are initially more focused on setting a solid understanding of the theory before showing a visual result. As He and Zhao (2012) suggest, *computer graphics is a practical area that needs to combine theory study and practice for a better result.*

With this motivation, we have developed an application called **CG Guide** fig. 1.1, that uses a top-down approach and illustrates what goes on behind the scenes in the building blocks of 3D and 2D scenes. The software is interactive and serves as means to make computer graphics lessons more appealing and relatable to students. At the time of writing of this work, it comes with ten different scenes that cover different aspects of computer graphics programming. It is also extensible and open-sourced[2].We review some fundamental mathematical and computer graphics concepts, required to understand the source code of CG Guide, in Appendix A and B.

[1]Federal University of Rio Grande do Sul
[2]Available in https://github.com/guicattani/cg-guide

Figure 1.1 – CG Guide screenshots collage, showing what the application is capable of



Source: our system

2 RELATED WORK

Computer graphics is a very dynamic field with new knowledge being created everyday, considering well-known journals, the computer graphics community publishes an average of 2.75 papers a day (BALREIRA; WALTER; FELLNER, 2017).

Knowledge is also publicly available on the Internet. On websites like SIGGRAPH's (Special Interest Group on GRAPHics and Interactive Techniques) (OWEN, 2005), it is possible to find introductory mathematics for computer graphics, surface mapping and even ray tracing explanations. On Eurographics's (European Association for Computer Graphics) website it is possible to find material curated for teaching computer graphics.

While tutorials and explanations enable students to learn on their own, teaching such dynamic field is also a challenge. Added to this difficulty, the question of what to teach can also arise. Balreira, Walter and Fellner (2017)'s work on what is being taught in introductory computer graphics elucidates how courses throughout the world handle this problem, according to it most common subjects are Rendering (75%), Modeling (14%), Animation (7%), Fundamentals (3%), and Visualization (1%). Knowing this distinction was important to decide to focus CG Guide development almost completely on rendering.

Other applications with the same motivations of teaching computer graphics interactively as CG Guide have been built in the past, the rest of this chapter will describe this previous work relates our system.

2.1 Edugraph

Developed by Battaiola, Elias and Domingues (2003), shown in fig. 2.1, Edugraph is an educational software focused on teaching concepts of computer graphics through interactivity and a high level of experimentation. The user controls an avatar, represented by a robot in a 3rd person view, that is teleported to different worlds. Each world has a task to be done to proceed to other worlds with other tasks, all tasks have the subject of computer graphics and aim to teach concepts through exposition and playfulness.

Although the goal of teaching is similar, the approach is different from our application, since we focused on end results that do not have an abstraction of a more ludic approach.

Figure 2.1 – Edugraph screenshot

Source: (BATTAIOLA; ELIAS; DOMINGUES, 2003)

2.2 GAIN: An interactive program for teaching interactive computer graphics programming

Developed by Towle and DeFanti (1978), GAIN is the first of its kind, a software capable of teaching computer graphics to students without direct supervision of tutors. The lessons consist in drawing pictures with 3D coordinates via macros similar to FORTRAN and ASSEMBLY. Once the student finishes the drawing, an evaluation system assesses if the user missed any step. Interactive macros provide experience in command usage and real-time experience with drawing in 3D.

GAIN is built with an automated assessment system, the need for such is a limiting factor and our application focuses on learning through experimentation rather than assessing the student directly.

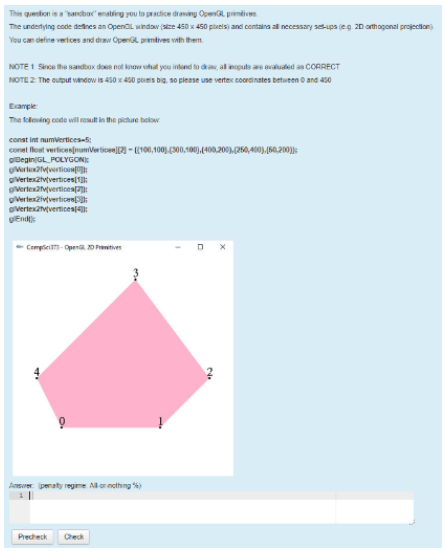
2.3 CodeRunnerGL

Developed by Wunsche et al. (2019), shown in fig. 2.2, CodeRunnerGL is an adaptation of CodeRunner, a plug-in for Moodle. It enables Moodle to support Open Graphics Library (OpenGL) programming through commands sent to a Virtual Machine, which then has its display snapshotted and shown to the user. It supports automated feedback and assessment.

Our approach with CG Guide is focused on having results in real-time that match

what parameters the user chooses, opposed to CodeRunnerGL, which does not support real-time rendering.

Figure 2.2 – CodeRunnerGL screenshot



Source: (WUNSCHE et al., 2019)

2.4 TERA, a Tool for Exploring Rendering Algorithms

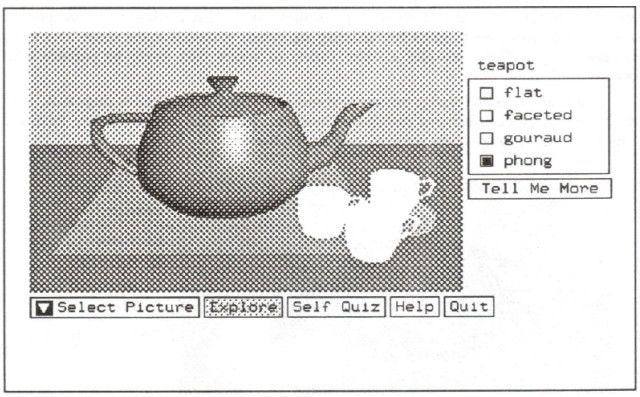
Developed by Wolfe and Sears (1996), shown in fig. 2.3, TERA is a tool focused on comparing rendering algorithms. In its initial version, it was only capable to show the difference between Flat, Phong, and Gouraud shading and quiz the student on them. Later versions of TERA are capable of showing more shading techniques such as ray-tracing, texture mapping, bump mapping, and lighting. (EBER; WOLFE, 2000)

TERA focuses more on showing different algorithms but not on letting the user change what is on-screen. Having quizzes helps students cement their knowledge but only teaches students to differentiate the content, and not interact with it.

2.5 Mental Vision

Developed by Peternier, Thalmann and Vexo (2006), shown in fig. 2.4, Mental Vision is a pedagogical-oriented graphics engine. It offers a set of tools, called modules, to better visualize computer graphics abstract notions. Parameters are changeable and the

Figure 2.3 – TERA screenshot

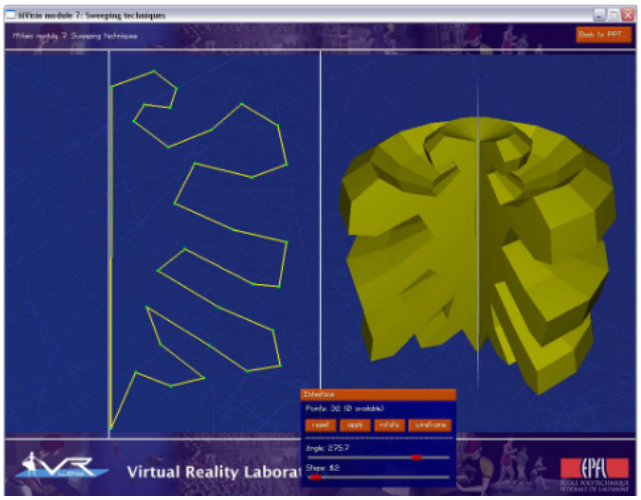


Source: (WOLFE; SEARS, 1996)

scene acts accordingly. When used in a class, teachers have privileged control over the shown scene, but students can control the scene when given permission, this is useful when the teacher asks the student to solve a problem or show publicly their results.

Mental Vision is not open sourced and support appears to be dropped as of 2009. Documentation is unreachable at the time of writing this work. Our application has the code open for contributions and extensions.

Figure 2.4 – Mental Vision screenshot



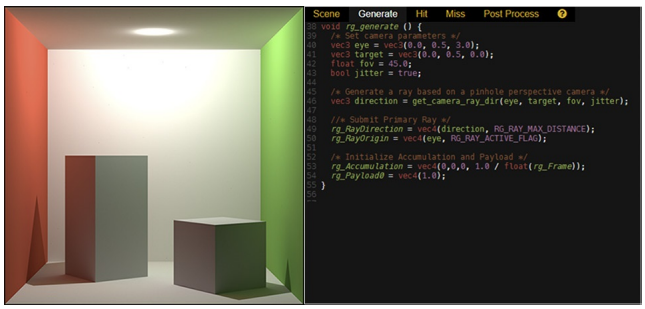
Source: (PETERNIER; THALMANN; VEXO, 2006)

2.6 Rayground

Developed by Vitsas et al. (2020), shown in fig. 2.5, Rayground is an educational tool for a richer in-class teaching or self-study that provides an introduction to ray tracing. The underlying mechanics of the rendered scene are abstracted by Rayground, while still allowing the user to change parameters pertinent to ray tracing.

While Rayground is similar to CG Guide in the sense both render scenes in real-time based on the user input, it is focused on teaching ray tracing concepts and its intended audience are students who are already familiar with the basics of computer graphics. CG Guide aims to help students of all levels while being able to teach a multitude of concepts.

Figure 2.5 – Rayground screenshot



Source: (VITSAS et al., 2020)

2.7 SIECG

Developed by GOMES and Manssour (2003), shown in fig. 2.6, SIECG is an interactive tool designed to aid Computer Graphics teaching. It has interactive examples and texts that describing the objective of them. Algorithms and commented source codes of the implementation using OpenGL and Virtual Reality Modeling Language (VRML) are also available.

Both CG Guide and SIECG aim to help teach concepts of computer graphics interactively while using OpenGL. SIECG has a smaller roster of scenes compared to CG Guide and is not available to download at the time of writing this work.

Figure 2.6 – SIECG screenshot

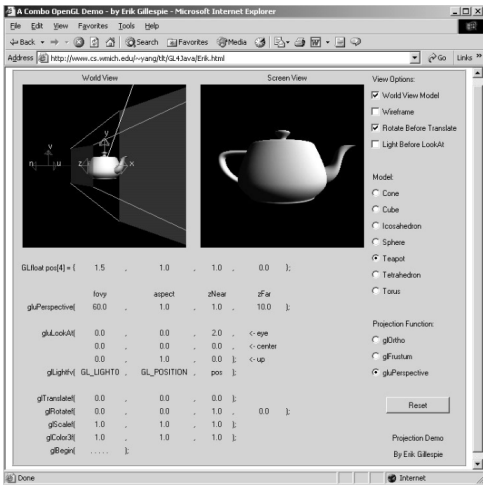
Source: (GOMES; MANSSOUR, 2003)

2.8 Web-Based Interactive 3D Visualization for Computer Graphics Education

Developed by Yang and Sanver (2003), shown in fig. 2.7, this work introduces a Web-Based Java wrapped OpenGL application capable of showing principles of computer graphics through interaction in a web browser.

This work is similar to CG Guide as it aims to supplement students knowledge on computer graphics interactively, but the application runs on legacy OpenGL and its dependencies have no support for modern hardware. At the time of writing this work, the demos available on the author's website could not be run.

Figure 2.7 – Web-Based Interactive 3D Visualization for Computer Graphics Education screenshot



Source: (YANG; SANVER, 2003)

3 SYSTEM DESIGN

This chapter focuses on presenting design goals and architecture decisions involved in implementing our application, whilst defining nomenclature of the building blocks of CG Guide scenes.

CG Guide is separated in what we called *scenes*, which are self-contained demonstrations of a particular computer graphics concept. The initial roster of scenes was developed considering that the user has no familiarity with CG so the most basic concepts behind computer graphics are explained in the first few scenes - how to send data to GPU, how to load Wavefront OBJ files, how shaders affect the scene, etc. - and the latter are focused on more practical examples - how to make objects move in a Bézier curve pattern, how to light objects, how perspective distorts objects, how a camera frustum works, and lastly how texture projections work. The scenes have built-in controls that allow the user to play with parameters for a holistic understanding.

3.1 Design Goals

Our system was created focusing on the easy creation and extension of such scenes. To achieve this, our application was built following some principles.

3.1.1 Ease of use

For an interactive user experience that is both easy and simple to understand, we opted to implement it with ImGui, a graphical user interface for C++. ImGui has multiple easy-to-use components, such as sliders, drop-downs, and graphs. Some of the components have been modified to better suit some of the scenes' requirements. ImGui is open-sourced and has an MIT license, so it is included in the project.

Other points considered in the ease of use are availability and ease of setup: Since the implemented scenes are designed to be lightweight, minimum requirements to run the application are low and it will run in any hardware that supports OpenGL 3.3 or newer.

At the time of writing this work, users can download the pre-release of CG Guide as single executable [1] without any external library dependencies. [2].

[1] only available for Windows x64, available in https://github.com/guicattani/cg-guide/releases
[2] models, textures, and shaders are used in run-time, so are separated from the executable

3.1.2 Modularity and Expandability

Each scene should be completely independent from each other. Scenes have their own objects, cameras and shaders, and GUIs, which enables an easy way to expand the roster of scenes without affecting existing ones.

3.1.3 Open-sourceness

From the start, the code was built with open-sourceness in mind. This means that code has been written with code quality standards, modularity and documentation. This enables anyone to update, improve or even copy the code for their own use. This also allows the code not to become outdated whilst contributing to the general quality of computer graphics learning materials.

3.1.4 Presentability

The scenes should be presentable to an audience, for example a class full of students. The interface scales to be bigger for better reading when presenting to a crowd. All scenes include scaling features to make scene objects appear bigger and more easily discernible when seen from a bigger distance.

3.2 Architecture

We used modern OpenGL for this project, namely OpenGL 3.3 and OpenGL Shading Language (GLSL) version 130 and although it is considered harder and less straightforward to learn compared to OpenGL 2.0, we can use programmable shaders, which allows shader code to be sent directly to the GPU.

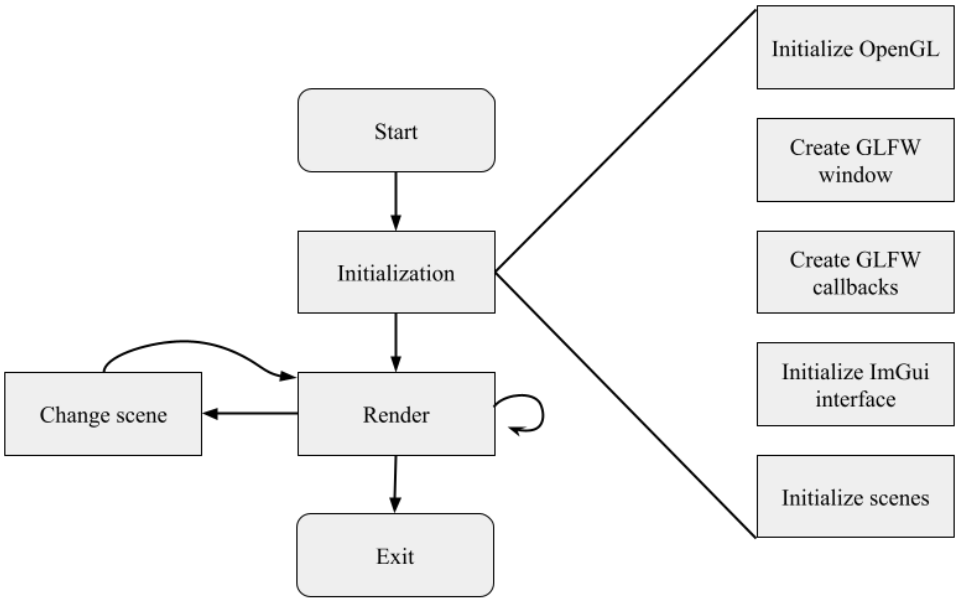
Since OpenGL only focuses on rendering, it was necessary to add the auxiliary library GLFW, which enables input, windowing, and context creation. This is important for a better user experience and will enable mouse and keyboard interaction.

After GLFW is initialized we do the necessary steps to initialize the GUI, ImGui. We need to bind the window created by GLFW and ImGui, this enables user input to change the interface status and, therefore, the state of the scene, while in the render loop.

Finally we then render the scene and the interface. This is the main part of the application and consists of a loop that only finishes when the user closes the window. The user can then change the scene, from the roster of existing scenes, at any time using the given drop-down made for scene selection.

All of these steps can be summarized as pictured in fig. 3.1.

Figure 3.1 – Initialization and rendering process



Source: the authors

3.2.1 Base Technologies

3.2.1.1 OpenGL

OpenGL is a graphics API with a set of routines implemented by graphics card manufacturers that enable developers to control the hardware. OpenGL is cross-platform, meaning it can run on multiple platforms using the same code. It concerns itself with rendering only, it doesn't have a GUI built-in, an audio processing library, windowing nor input capabilities.

3.2.1.2 GLFW

GLFW is a lightweight Open Source library for OpenGL that provides windowing, context, and input capabilities.

3.2.1.3 ImGui

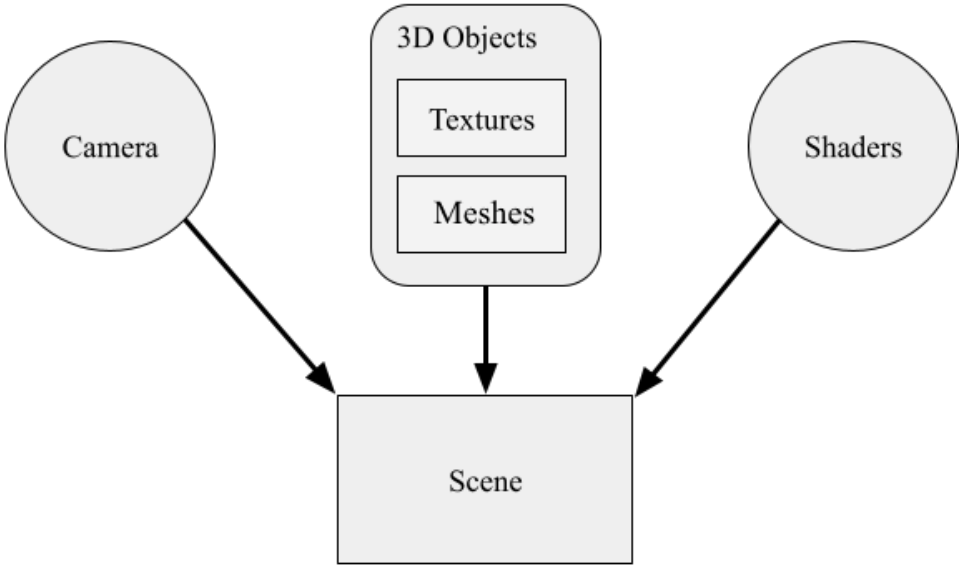
ImGui is a bloat-free graphical user interface library for C++. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline enabled application. It is fast, portable, renderer agnostic, and self-contained (no external dependencies) (CORNUT, 2016).

3.2.2 Scenes

When approaching all design goals, we opted for a very simple way to create scenes. Scenes should stand on their own and not interfere in the functioning of the rest of the scenes nor in the rendering process.

Each scene is composed of three components, **Cameras**, **3D Objects** and **Shaders**, as pictured in fig. 3.2.

Figure 3.2 – A scene object



Source: the authors

- **Cameras** are objects that designate a point of view in a virtual scene. In our scenes, cameras will be either look-at or free. Look-at cameras always face a single point in world space and Free cameras will look at an arbitrary point that is directly in front of the camera, this point is rotated in tandem with the camera rotation, giving the impression of a "free" camera.

- **3D Objects** in turn, are objects comprised of meshes - a collection of vertices, faces, and normal vectors - and textures (which are optional). These objects can be loaded from Wavefront OBJ files or by defining each vertex, face, and normal vector. When projecting textures we also require texture coordinates that can be defined by the user or defined automatically by the model's bounding box[3].
- **Shaders** program the interaction between material and light to compute the color of a pixel. In our project, shaders can be either by vertex or by fragment.

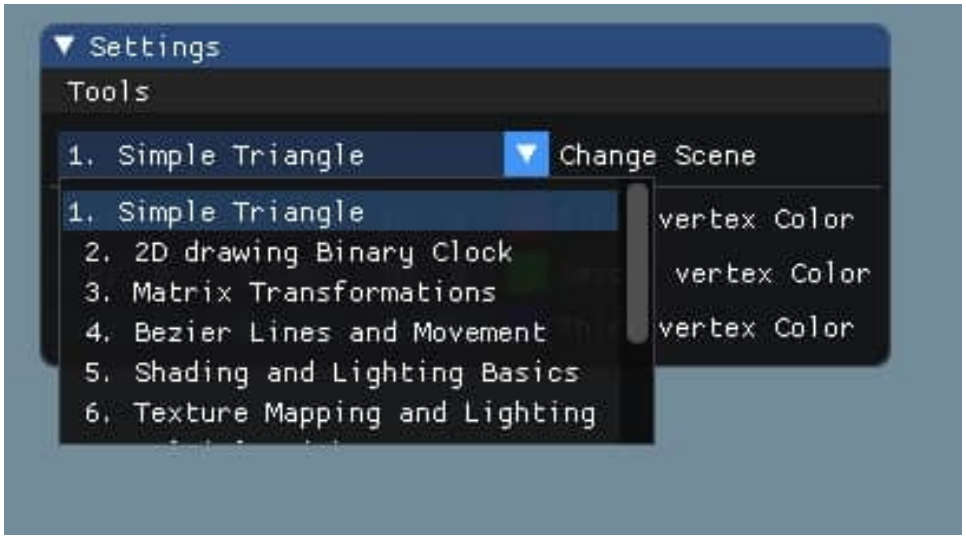
Scenes can have multiple cameras, 3D Objects, and shaders, essentially giving full control of what is being rendered to the scene's author. Having multiple cameras can be useful to show how the rendered scene is behaving when seen from a different perspective. These cameras are implemented in the latter, more complex, scenes of the application.

[3] A 3D box made with the mesh's outermost vertices

4 RESULTS

This chapter discusses how each scene was implemented, what were the motivations for developing them, and the results. Each scene has a focus on one important part of computer graphics. To change the scene the user chooses the desired scene from "Change Scene" drop down, as pictured in fig. 4.1.

Figure 4.1 – Scene changer drop-down



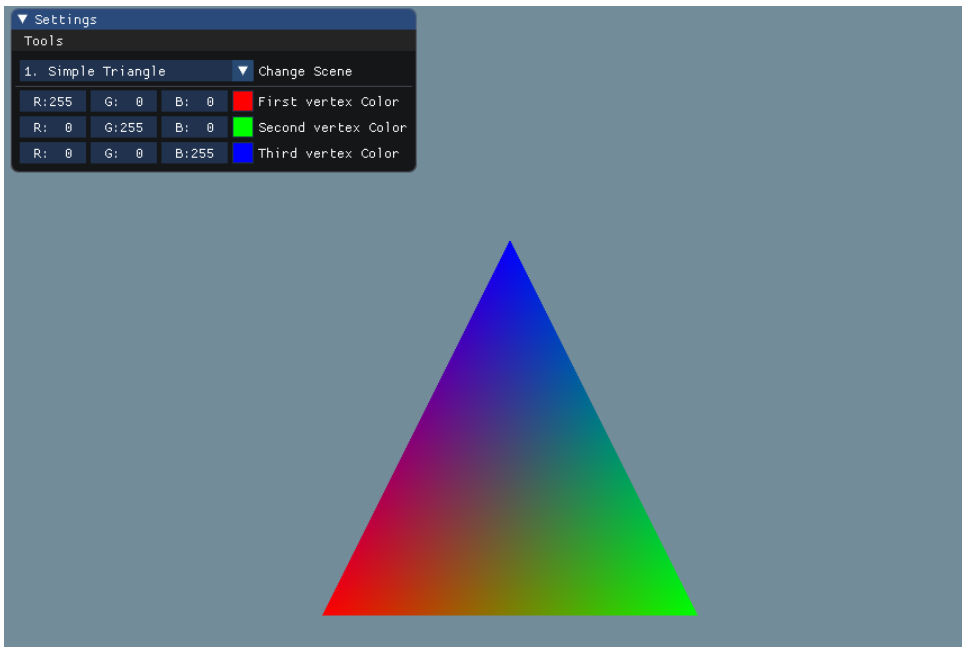
Source: our system

4.1 Drawing a simple triangle

A triangle is a shape made out of three vertices connected in a particular order. A triangle the simplest shape that draws a plane, i.e. a point and a line do not draw a plane, we need at least three points to draw a plane, hence triangles are an efficient way to store a model's mesh data. Because of this most meshes are comprised of complexes of triangles with shared vertices (MARSCHNER; SHIRLEY, 2015).

To specify what way the triangle is facing, a normal vector is used. Normals are vectors that are orthogonal to the triangle's surface. They can face either the front side or the backside. They are important in computer graphics because most models are only drawn in the outermost shell, i.e. the back-face of these triangles are not drawn to save processing power.

Figure 4.2 – Color blending in a red, green, and blue vertices triangle in our system



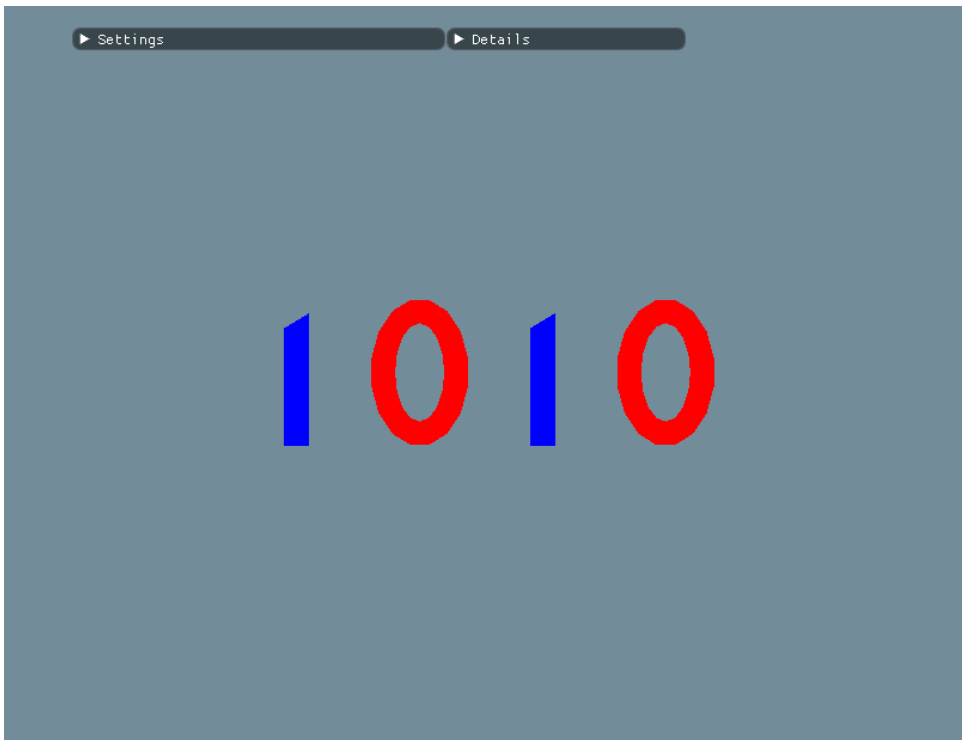
Source: our system

With this in mind, the first scene starts with the simplest possible drawing, a triangle. To achieve this, first, it is necessary to declare its vertices and send it to the graphics card through OpenGL. A Vertex Array Object (VAO) is a data structure that can hold many Vertex Buffer Objects (VBO), which in turn, are structures that hold an arbitrary attribute of the model. These attributes can be either vertex position, texture coordinates, color, normals, or any other attribute usable for the scene being built.

For drawing a simple triangle we used a VAO that held two VBOs, one for the position, and one for the vertex color. The vertices colors can be directly changed by user input, allowing the user to experiment with how each color blends with the other vertices colors at run-time.

After defining the VAO and VBOs we send the vertex data to the graphics pipeline in the GPU. This vertex data is modifiable by the vertex and fragment shader. The vertex shader receives the vertex position and color defined in the VBO, applies the projection and view from the camera, and passes the transformed position along with the color into the pipeline. In the fragment shader, we use the color passed down from the vertex shader and calculate how each pixel will look, this is called color blending, like the mid points between vertices in fig. 4.2.

Figure 4.3 – A binary clock drawn in OpenGL



Source: our system

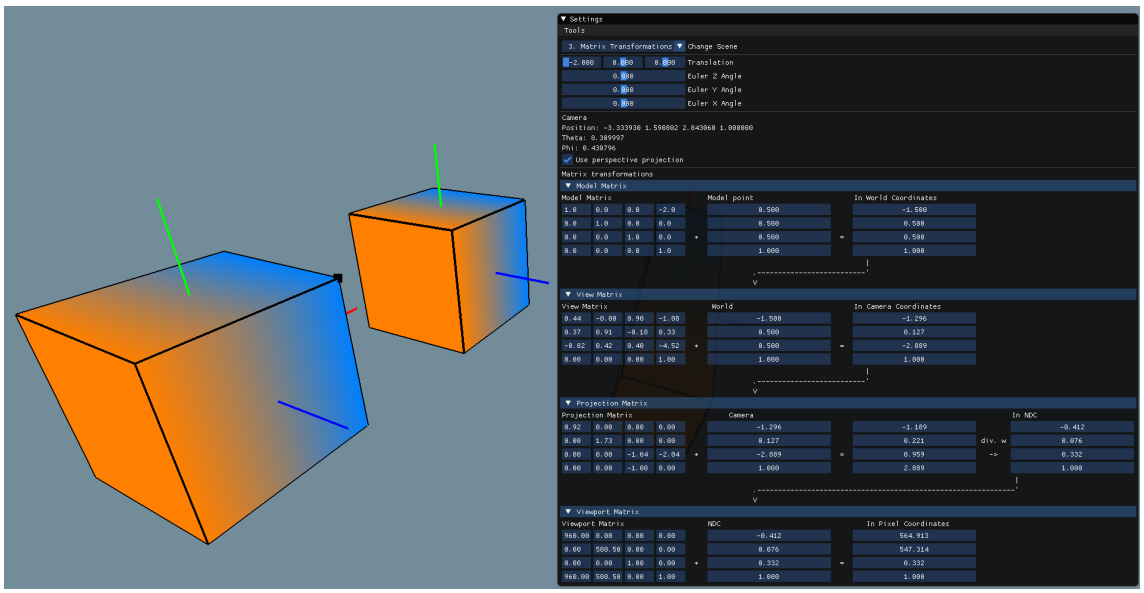
4.2 Animated binary clock and dynamic drawing

To demonstrate how flexible the VBO/VAO structures are we designed a scene capable of handling a dynamic quantity of vertices, opposed to the fixed vertices of the previous scene, section 4.1. This scene shows an animated binary clock with 4 digits and the clock changes digits each second, as expected.

As pictured in fig. 4.3, the digit 1 consists of four vertices organized in a rectangle shape, and the digit 0 consists of two ellipses with an inside radius and an outside radius, which count 32 vertices to achieve a smooth curve. These vertices are connected in triples to form triangles, which are simply determined by their order of creation.

Each digit has its own VBOs both for position and for color, opposed to section 4.1, this structure consists of only one VBO per VAO. While not recommended, this is by design, to show how one could approach using the VBO/VAO structures.

Figure 4.4 – Model-View-Perspective-Viewport matrices effect on a scene



Source: our system

4.3 Matrix transformations

In this scene, we aim to shed light on how the MVPV matrices, introduced in appendix B.3, influence the drawing on the screen of one of the uppermost vertices in one of two cubes. The highlighted vertex is made more evident than the others by being drawn as a black square, seen in the blue part of the left-most cube in fig. 4.4. Interacting with the scene will change the matrices [1]:

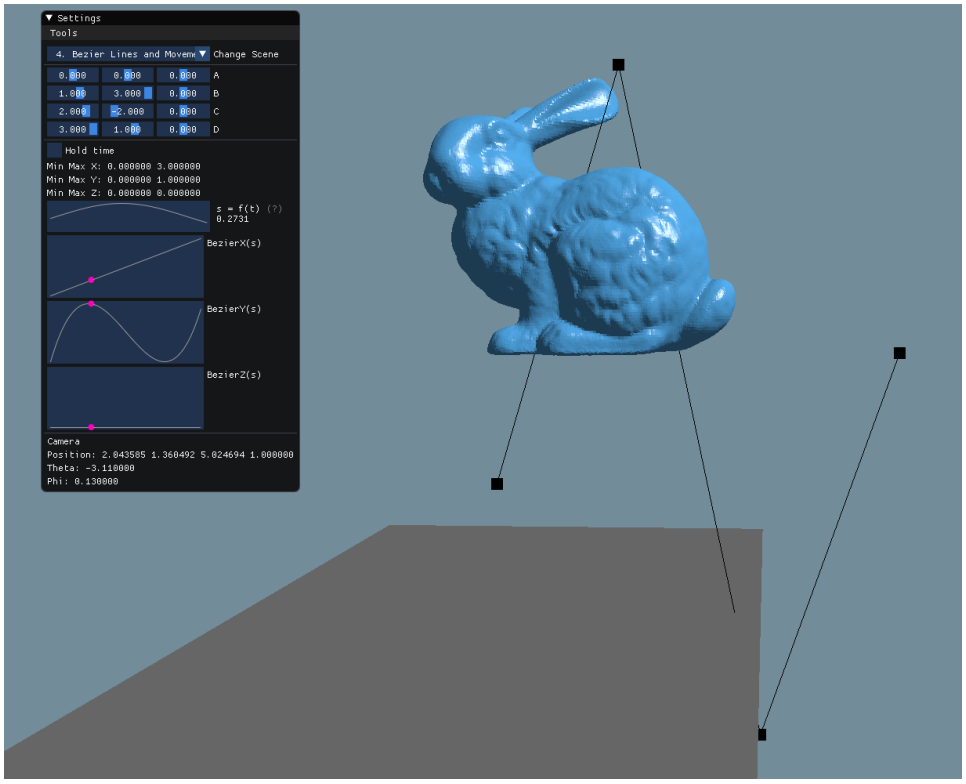
- The cube is translatable and rotatable via the scene's interface. Changing these controls makes very apparent how the model matrix affine transformations change the model attributes by changing the way the cube is drawn, hence influencing how values are shown in the pipeline of matrices of the MVPV.
- The camera is free and is controllable via the keyboard. Keyboard arrows change the position of the camera. Mouse movement changes the rotation of the camera. Changing the camera position or rotation will change the view matrix because the point will be in a different place in the view plane.
- The perspective can be changed to either perspective or orthographic projection with the click of a button in the interface, this will change the projection matrix accordingly.

[1] The matrices fields are not editable, the only way to change them is by interacting with the scene

- Changing the size of the window rendering the application will change the viewport matrix, as it interferes with where the vertex will be drawn as a pixel in the rendering window.

4.4 Bézier lines and 3D movement

Figure 4.5 – Bézier lines movement based on control points



Source: our system

As presented in appendix A.2, Bézier curves determine the point on a trajectory based on the control points of the curve and a ratio. For this scene the ratio s will be calculated based on the t frames that passed in each render cycle, modulated to fit in a $[0, 1]$ range, this can be achieved by using a simple senoid function to define s:

$$s = \frac{\sin(\pi(t - \frac{1}{2}))}{2} + \frac{1}{2}. \tag{4.1}$$

The first graph on the left-side panel of fig. 4.5 will be used to draw eq. (4.1) based on the counts frames rendered. The control points are changeable according to their current $[x, y, z]$ coordinates, and in doing so, the user will have feedback of how the Bézier curves

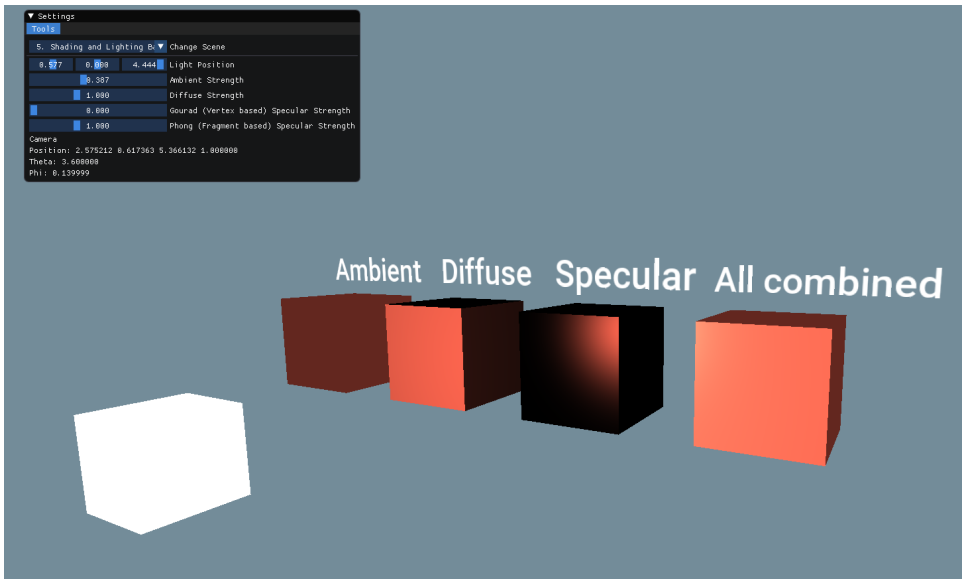
component functions will behave in such configuration, represented by the last 3 graphs in the panel of fig. 4.5. These graphs are separated in the $[x, y, z]$ dimensions and have a pink dot that moves along the graph's line according to the current position of the model of the bunny. To improve usability, the user can also hold the frame counting to freeze the model in place, therefore freezing how the component function graphs behave.

4.5 Shading and lighting basics

As presented on appendix B.4, light can interact in very different ways with a surface. For Phong shading, we can use ambient, diffuse, and specular reflectance. This scene focuses on showing how these different types of reflectances can be separated in the first three cubes, that are later combined in the last right-most cube, as seen in fig. 4.6.

The scene interface controls help the user understand how each combination of reflectance strengths can influence how the cube looks. The position of the light is changeable, as well as the ambient reflectance strength, the diffuse reflectance strength, and the specular Phong and Gouraud reflectance strengths.

Figure 4.6 – Blocks reflecting a point light in different ways

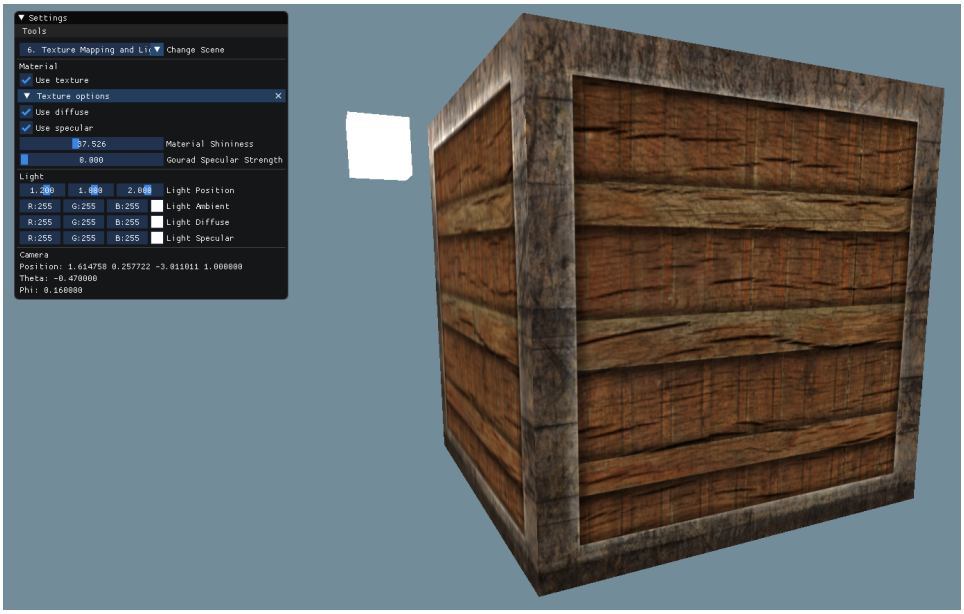


Source: our system

4.6 Simple texture mapping and texture lighting

For this scene, we aim to make the lighting of an object more interesting by using either a diffuse or a specular texture, or the combination of both. As pictured in fig. 4.7, we used a simple UV projection for the texture in this scene, both for the diffuse and the specular texture, as presented on appendix B.5.

Figure 4.7 – A textured block with diffuse and specular texture interacting with a light



Source: our system

The controls enable the user to change characteristics of the light and the cube:

- **For the light** the user can change how each type of emission of a light, presented on appendix B.4, can change when hitting an object. These include the ambient, the diffuse, and the specular color of emission.
- **For the cube** the user can change the color for each type of reflectance and can enable the diffuse and/or the specular texture.

Having the ability to turn on and off the diffuse and specular texture can enable a better understanding of how textures can work when on top of one another:

- For the diffuse texture, pictured in the left-hand side of fig. 4.8, we have the texture itself, with multiple colors. We chose to set it as diffuse to create a more realistic effect, but using the ambient reflectance as a texture would also be possible.

- For the specular texture, pictured in the right-hand side of fig. 4.8, we use a monochromatic texture that allows the shading programs to set different values of specularity for points in a texture. Values closer or equal to white will be completely reflective and values closer or equal to zero will be non-reflective.

Figure 4.8 – Diffuse and specular textures used for texturing a cube, on the left-hand side and the right-hand side, respectively

Source: Extracted from LearnOpenGL.com (VRIES, 2015)

4.7 Light attenuation and different ways to light a scene

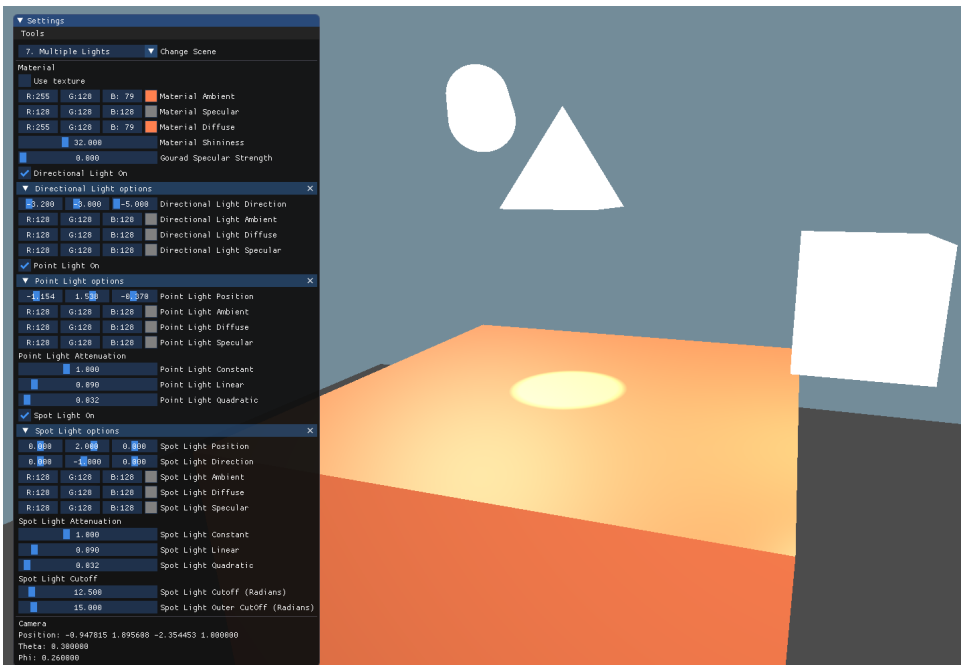
As pictured in fig. 4.9, in this scene we wanted to show the different kinds of lighting seen in appendix B.4.3, so the user can enable a directional, a point, and a spotlight to observe how they interact with an object.

The directional light is represented by a cylinder, seen on the left-hand side of the triangle in fig. 4.9, the user can change its direction and also its emitting properties.

The point light is represented by a cube and uses light attenuation as seen in appendix B.4.3, so the farther the light is from the object, the dimmer the effect it will have on the object. The user can change the constant, the linear, and the quadratic attenuation factors to see how it affects its light falloff.

The spotlight is represented by a pyramid, the base of the pyramid is where the light is coming from and the user can change the direction it points towards and also the cutoff inner and outer angles.

Figure 4.9 – A demonstration of directional, point and spotlight



Source: our system

4.8 Frustum and perspective basics

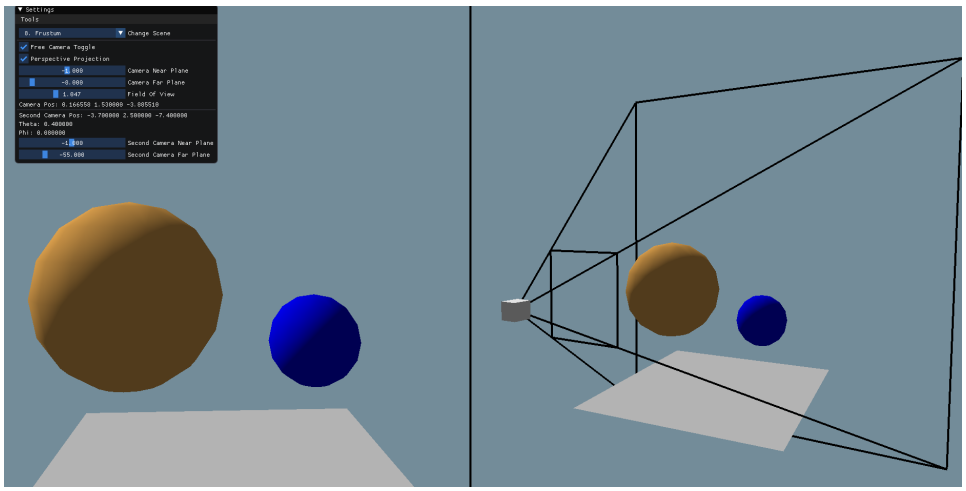
To better understand how a view frustum, seen in appendix B.3.3.1, affects the perspective matrix, we developed a "third-person view" of what it would look like to see a camera and its view frustum. As pictured in fig. 4.10 the left-hand side is the rendered view of the right-hand side gray cube, representing the camera, together with lines representing the view frustum and its near and far planes.

The user can change the perspective to be either orthographic or perspective, and this will change the frustum accordingly, as seen in fig. 4.11. Added to this the user can change the near and far plane distance from the camera, and also the field of view, seen in appendix B.3.3.2. Changing the field of view makes the frustum narrower or wider, depending on its setting, and this changes the camera view accordingly.

4.9 Projection matrix simulation

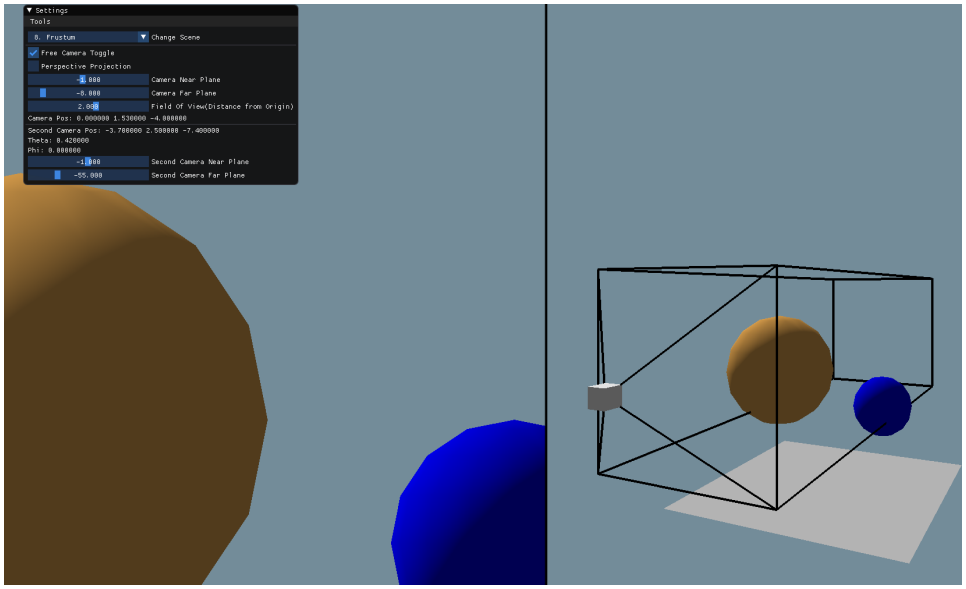
This scene illustrates the NDC space and the perspective deformation defined by a perspective projection matrix, both seen in appendix B.3.

Figure 4.10 – A camera frustum view and its representation in a scene



Source: our system

Figure 4.11 – A camera frustum view with orthographic projection and its representation in a scene

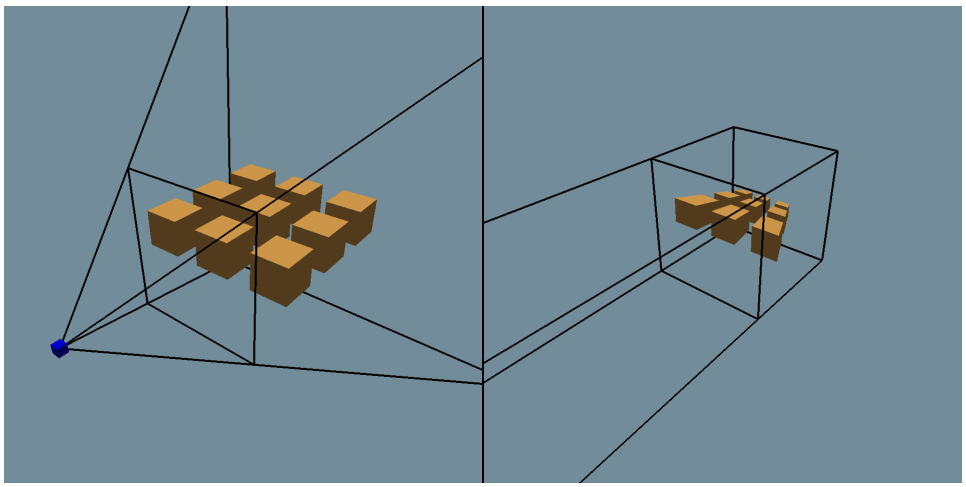


Source: our system

Figure 4.12 shows how the perspective matrix affects objects in a scene [2]. Perspective deformation, seen on the right, is discussed in-depth in appendix B.3.3.1.

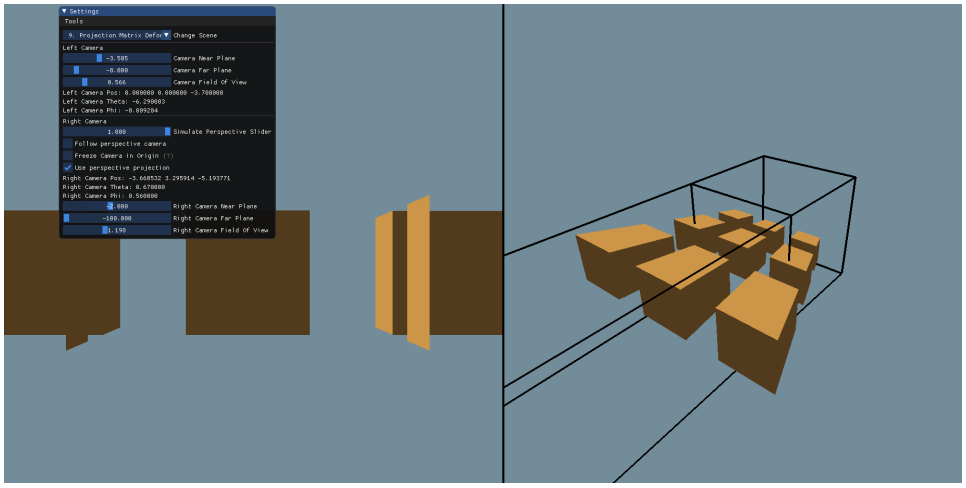
[2] Animation available in CG Guide repository

Figure 4.12 – A camera frustum view with perspective projection before and after the perspective deformation



Source: our system

Figure 4.13 – A camera view in the left-hand side and its respective perspective matrix deformation on the right-hand side



Source: our system

The user can interpolated between the cubes before and after the perspective deformation, but this effect is made only for educational purposes and doesn't reflect how OpenGL does the deformation, i.e. the cubes are either deformed or not. The application of the perspective matrix transforms the frustum into a cube that represents the NDC space. Everything that falls outside of this cube will not be rendered by the observer, as pictured in fig. 4.13.

The user can interact with the scene by moving the near and far plane and the field of view to see how this affects the objects being rendered.

Another setting is the ability to follow the perspective camera (seen on the left-hand side of fig. 4.13), this enables the user to put the second camera (on the right-hand side of fig. 4.13) in the same place as the perspective camera. Enabling the orthographic projection for the second camera makes both of the cameras render exactly the same, but on the second camera, the perspective is actually being simulated by geometric distortions to the scene objects.

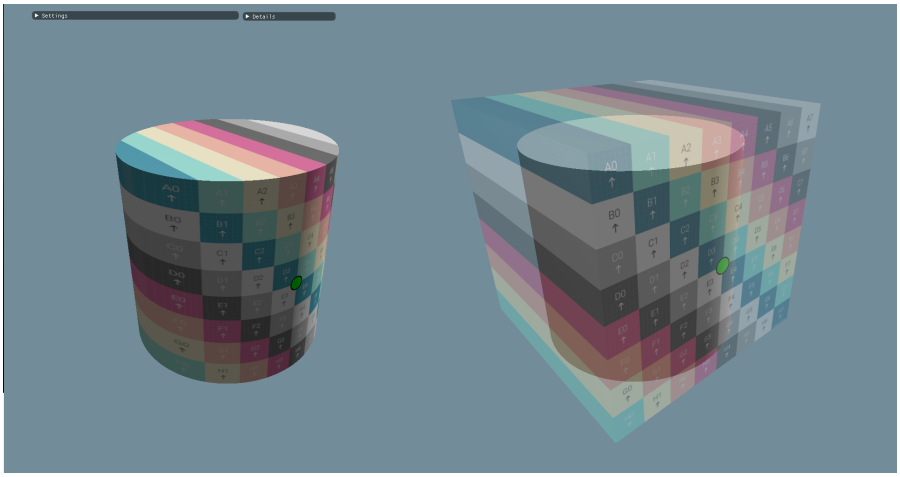
4.10 Advanced texture mapping

This scene focuses on giving an experience very close to what is seen on fig. B.16, a texture that is projected upon a surface and an auxiliary arrow that indicates where the direction it points is projected onto the object. In this section, we will refer to each point in an object by its texture coordinate $(u, v) = \phi(x, y, z)$.

4.10.1 Axis Aligned Bounding Box Texture Projection

Axis Aligned Bounding Box Texture Projection is also known as *Planar Projection*, and as the name suggests, projects a texture plane onto an object, as pictured in fig. 4.14. This is the simplest mapping from 3D to 2D parallel projection, the same as used on appendix B.3.3.2, and just as so, it is done by multiplying a matrix (with no rotation) and discarding the z component.

Figure 4.14 – Axis Aligned Bounding Box Texture Projection, the projected texture on the left-hand side, and unwrapped on the right-hand side



Source: Our system

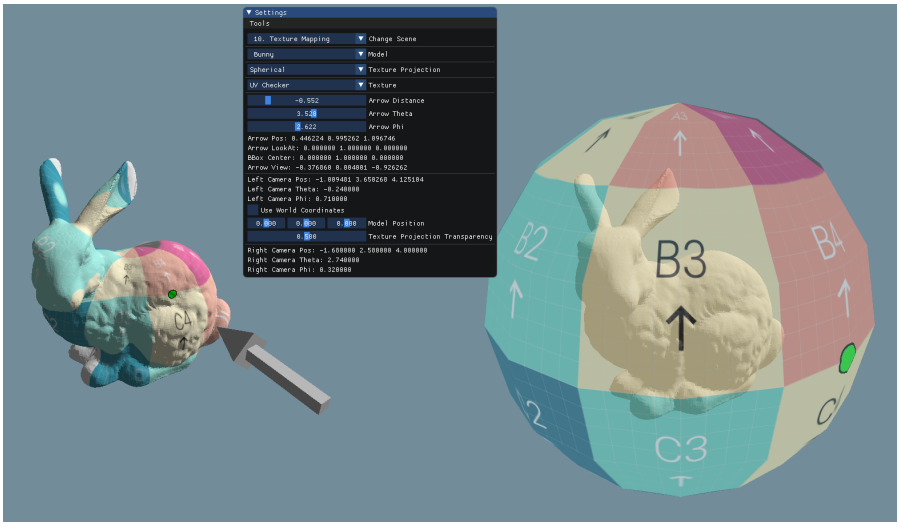
$$\phi(x, y, z) = (u, v) \quad where \begin{bmatrix} u \\ v \\ * \\ 1 \end{bmatrix} = M_t \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{4.2}$$

The texturing matrix M_t in eq. (4.2) represents an affine transformation (with rotation discarded), and the asterisk $*$ indicates that we don't care what ends up in the z coordinate. (MARSCHNER; SHIRLEY, 2015)

4.10.2 Spherical

We can parametrize a point on the surface of an object by mapping it to the point on a sphere through radial projection: take a line from the center of the sphere through the point on the surface, and find the intersection with the sphere, like in fig. B.16, and as shown in our application in fig. 4.15.

Figure 4.15 – Spherical Texture Projection, the projected texture on the left-hand side, and the sphere projecting the texture on the right-hand side



Source: Our system

This can be done by expressing the surface point in spherical coordinates (ρ, θ, ϕ) and then discarding the ρ coordinate and mapping θ and ϕ to the range $[0, 1]$ (MARSCHNER;

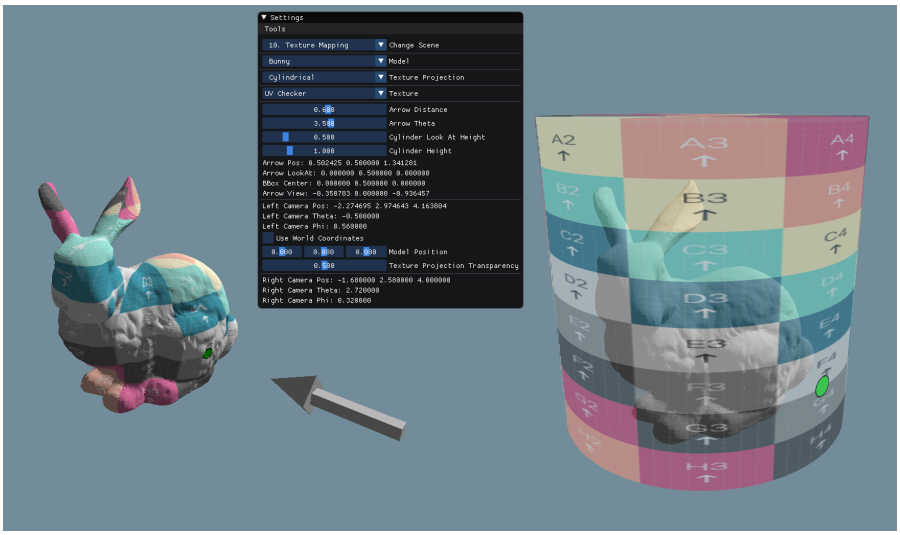
SHIRLEY, 2015):

$$\phi(x, y, z) = ([\pi + atan2(y, x)]/2\pi \ , \ [\pi - acos(\frac{z}{||x||})]/\pi). \tag{4.3}$$

4.10.3 Cylindrical

Cylindrical projection is made for models that are more columnar than spherical, for example, a vase. Figure 4.16 demonstrates how our application demonstrates this texture projection.

Figure 4.16 – Cylindrical Texture Projection, the projected texture on the left-hand side, and the cylinder projecting the texture on the right



Source: Our system

Analogous to spherical texture projection, this can be simply done by converting coordinates to cylindrical coordinates and discarding the radius (MARSCHNER; SHIRLEY, 2015). Hence:

$$\phi(x, y, z) = (\frac{1}{2\pi}[\pi + atan2(y, x)]/2\pi \ , \ \frac{1}{2}[1 + z]). \tag{4.4}$$

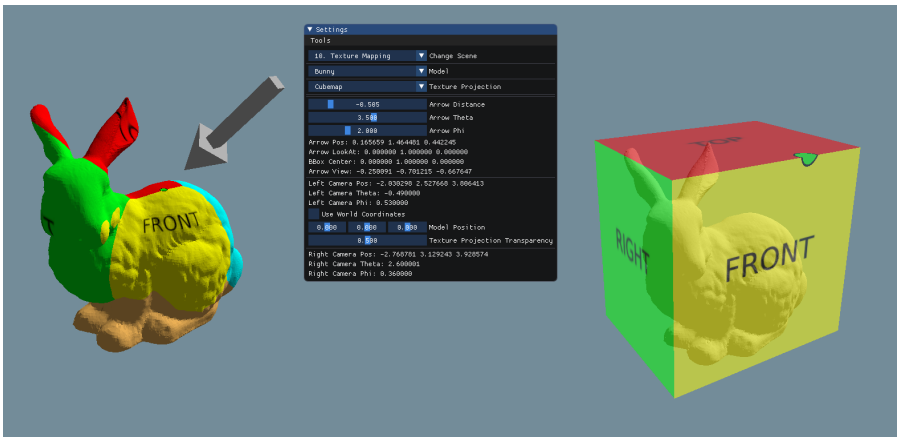
4.10.4 Cube map

Using spherical coordinates to parameterize a planar shape (the 2D texture) leads to a high distortion of shape and area near the poles. A popular alternative is more uniform at the cost of having more discontinuities. The idea is to project onto a cube, rather than a sphere, and then use six separate square textures for the six faces of the cube. The collection of six square textures is called a *cube map*. This introduces discontinuities along all the cube edges, but it keeps distortion of shape and area low (MARSCHNER; SHIRLEY, 2015).

Cube map projection, as pictured in fig. 4.17, is made by having one texture for each face of the cube of projection, it is cheaper than spherical coordinates because projecting onto a plane just requires a division. The point that projects onto the $+z$ face of the cube is defined by:

$$(x, y, z) \implies (\frac{x}{z}, \frac{y}{z}). \tag{4.5}$$

Figure 4.17 – Cube map Texture Projection, the projected texture on the left-hand side, and cube projecting the texture on the right-hand side



Source: Our system

We can tell which face a point projects by looking at the coordinate of the largest absolute value: for example, if $|x| > |y|$ and $|x| > |z|$, the point projects to the $+x$ or $-x$ face, depending on the sign of x (MARSCHNER; SHIRLEY, 2015).

5 CONCLUSION AND FUTURE WORK

We presented an application capable of helping teachers teach and students to better understand concepts of computer graphics. It allows for a highly interactive environment that portrays computer graphics principles in a more student-friendly way. The need for an application such as CG Guide became very clear once we searched for applications that had the same purpose and only could find a few, and mostly legacy, applications. We aimed to make it as easy to understand and extend as possible, covering a bit of each introductory topic in computer graphics.

This system can easily be built upon in the future, it only touches the tip of the computer graphics iceberg. Making it available to be open source makes this even more exciting because anyone can make their own CG Guide and use it as they please.

A logical follow-up would be creating more scenes that cover more advanced topics. Although challenging and hard to implement this could help many in understanding computer graphics.

Another simple, yet effective, way would be to put explanations of each scene inside of the program itself, this can be done quite easily and has already been experimented with when developing this application. The interface supports markdown syntax and showing images, diagrams, and graphs.

REFERENCES

BALREIRA, D. G.; WALTER, M.; FELLNER, D. W. What we are teaching in Introduction to Computer Graphics. In: BOURDIN, J.-J.; SHESH, A. (Ed.). **EG 2017 - Education Papers**. [S.l.]: The Eurographics Association, 2017. ISSN 1017-4656.

BATTAIOLA, A. L.; ELIAS, N. C.; DOMINGUES, R. de G. Edugraph: software to teach computer graphics concepts. In: **Proceedings. XV Brazilian Symposium on Computer Graphics and Image Processing**. [S.l.]: IEEE Comput. Soc, 2003.

BUSS, S. R. **3D computer graphics: A mathematical introduction with OpenGL**. Cambridge, England: Cambridge University Press, 2003.

CORNUT, O. **ImGui**. 2016. Available in https://github.com/ocornut/imgui.

EBER, D.; WOLFE, R. Teaching computer graphics visual literacy to art and computer science students: Advantages, resources and opportunities. **Comput. Graph. (ACM)**, v. 34, n. 2, p. 22–24, 2000.

GOMES, G.; MANSSOUR, I. Siecg - an interactive tool to teach computer graphics. 01 2003.

HE, Y.; ZHAO, Y. Reform and exploration of the computer graphics. In: . [S.l.: s.n.], 2012. p. 403–405. ISBN 978-1-4673-4825-6.

JOHNSON, C. **Deriving the View Matrix**. 2020. Available in https://twodee.org/blog/17560.

KHRONOS, O. R. **Coordinate Transformations**. 2012. Available in https://www.khronos.org/opengl/wiki/Coordinate_Transformations.

MARSCHNER, S.; SHIRLEY, P. **Fundamentals of computer graphics**. 4. ed. Oakville, MO: Apple Academic Press, 2015.

MAYA. **Documentation**. 2018. Available in https://help.autodesk.com/view/MAYAUL.

MORTENSON, M. E. **Mathematics for Computer Graphics Applications**. 2. ed. New York, NY: Industrial Press, 1999.

NETO, V. D. A. An analysis of failure in the computer science course at ufrgs from the point of view of students. In: . [S.l.: s.n.], 2021.

OWEN, G. S. **HyperGraph**. 2005. Available in https://education.siggraph.org/static/HyperGraph.

PENNA, M. A.; PATTERSON, R. R. **Projective geometry and its applications to computer graphics**. Old Tappan, NJ: Prentice Hall, 1986.

PETERNIER, A.; THALMANN, D.; VEXO, F. Mental vision: A computer graphics teaching platform. In: **Technologies for E-Learning and Digital Entertainment**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 223–232.

POWER, K. **Glasnost, staff web server at the Institute of Technology, Carlow, Ireland**. 2011. Available in http://glasnost.itcarlow.ie/ powerk/GeneralGraphicsNotes/.

SONG, S. et al. Real-time shape estimation for wire-driven flexible robots with multiple bending sections based on quadratic bezier curves. **IEEE Sensors Journal**, 07 2015.

SUNG, K.; SHIRLEY, P. A top-down approach to teaching introductory computer graphics. In: . [S.l.: s.n.], 2003. v. 28.

SUSELO, T.; WüNSCHE, B.; LUXTON-REILLY, A. The journey to improve teaching computer graphics: A systematic review. In: . [S.l.: s.n.], 2017.

TOWLE, T.; DEFANTI, T. GAIN. **ACM SIGGRAPH Computer Graphics**, v. 12, n. 3, p. 54–59, 1978.

VITSAS, N. et al. Rayground: An online educational tool for ray tracing. In: . [S.l.: s.n.], 2020.

VRIES, J. D. **LearnOpenGL.com**. 2015. Available in learnopengl.com/.

WOLFE, R. **Surface Mapping**. 1997. Available in https://education.siggraph.org/static/HyperGraph/mapping/surface0.htm.

WOLFE, R.; SEARS, A. An effective tool for learning the visual effects of rendering algorithms. **Comput. Graph. (ACM)**, v. 30, n. 3, p. 54–55, 1996.

WUNSCHE, B. C. et al. CodeRunnerGL - an interactive web-based tool for computer graphics teaching and assessment. In: **2019 International Conference on Electronics, Information, and Communication (ICEIC)**. [S.l.]: IEEE, 2019.

YANG, l.; SANVER, M. Web-based interactive 3d visualization for computer graphics education. **IJDET**, v. 1, p. 69–77, 07 2003.

APPENDIX A — MATHEMATICAL CONCEPTS

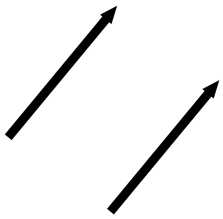
This section presents fundamental mathematical that are used in our application.

A.1 Points and Vectors

Points are the simplest of geometric objects. They can be defined as a set of numbers representing a coordinate in a space and can be used to define vertices, origins or simply a reference to somewhere in a space.

Vectors represent a length and a direction, and are usually represented by an arrow. They can be used to change the coordinates of points in a space and also to represent directions such as the direction in which a surface faces or the direction from an object to a light source. A *unit vector* is a vector whose length is one. Vectors can be added, subtracted and multiplied (through dot and cross products). Two vectors are equal if they have the same length and direction, as illustrated in fig. A.1. (MARSCHNER; SHIRLEY, 2015).

Figure A.1 – Two vectors that are the same, because they have the same length and direction

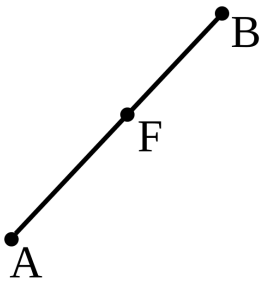


Source: the authors

A.2 Bézier Curves

According to Mortenson (1999), Bézier curves are parametric curves used in computer graphics and other related fields. They are used in vector graphics and on the time domain, as in animation and on smoothness of trajectories. The curves are defined by a collection of control points, and the number of points will determine which order the

Figure A.2 – A linear Bézier curve



Source: the authors

curve belongs to, from a minimum of two points.

According to Mortenson (1999):

- **Linear**, or first-degree, Bézier curves only have two control points, a starting and and ending point, as pictured in fig. A.2. Both points lie on the curve, which is a straight line between these points. Given A and B as starting and ending points, any Cartesian coordinates (X, Y) for some ratio $v \in [0, 1]$, given that X_F is the X-coordinate of point F and Y_F is the Y-coordinate of point F, any point F belonging to the curve is defined by:

$$\begin{aligned} X_F &= X_A + v(X_B - X_A), \\ Y_F &= Y_A + v(Y_B - Y_A). \end{aligned} \tag{A.1}$$

  Simplifying eq. (A.1), we have:

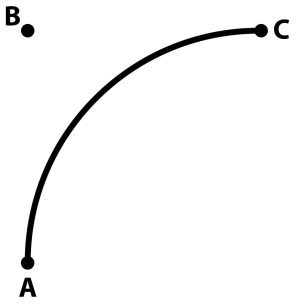
$$\begin{aligned} X_F &= (1 - v)X_A + vX_B, \\ Y_F &= (1 - v)Y_A + vY_B. \end{aligned} \tag{A.2}$$

  Which is the same as a *linear interpolation* equation.

- **Quadratic**, or second-degree, Bézier curves have one intermediate control point, and two control points that lie on the far ends of the curve. Assuming A as the first point, B as the intermediate point, and C as the end point, if an object would move through the curve, it would touch on both the first and the end points A and C, but not the intermediate point B, as the curve suggests in picture fig. A.3.

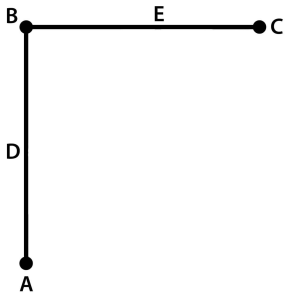
  The curve is constructed using by using auxiliary points D and E, as pictured in fig. A.4 that lie on the lines that connect the control points, lines AB and line BC. These auxiliary points are placed in following the same ratio v, hence, considering $|AD|$ as the length of line AD:

Figure A.3 – A quadratic Bézier curve



Source: the authors

Figure A.4 – A quadratic Bézier curve and its auxiliary points



Source: the authors

$$\frac{|AD|}{|AB|} = \frac{|BE|}{|BC|} = v \tag{A.3}$$

Points E and D's Cartesian coordinates will then be defined by:

$$\begin{aligned} X_D &= (1-v)X_A + vX_B, \\ Y_D &= (1-v)Y_A + vY_B. \end{aligned} \tag{A.4}$$

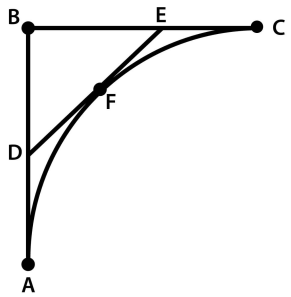
$$\begin{aligned} X_E &= (1-v)X_B + vX_C, \\ Y_E &= (1-v)Y_B + vY_C. \end{aligned} \tag{A.5}$$

As pictured in fig. A.5, point F can then be defined as the interpolation between points D and E, therefore its coordinates are obtained with:

$$\begin{aligned} X_F &= (1-v)X_D + vX_E, \\ Y_F &= (1-v)Y_D + vY_E. \end{aligned} \tag{A.6}$$

To obtain X and Y in terms of points A, B, C for any value of v, we substitute equations in their respective counterparts, that is: equations eq. (A.4) and eq. (A.5)

Figure A.5 – A quadratic Bézier curve with auxiliary points made to define F position



Source: the authors

into eq. (A.6). Which yields:

$$\begin{aligned} X_F &= (1-v)^2 X_A + 2v(1-v)X_B + v^2 X_C, \\ Y_F &= (1-v)^2 Y_A + 2v(1-v)Y_B + v^2 Y_C. \end{aligned} \tag{A.7}$$

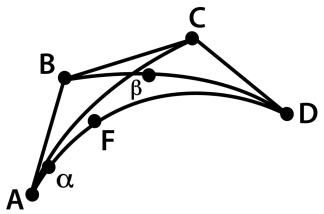
- **Cubic**, or third-degree, Bézier curves can be defined as affine combinations of two quadratic Bézier curves, with points defined as α and β, as pictured in fig. A.6.

$$\begin{aligned} X_F &= (1-v)X_\alpha + vX_\beta, \\ Y_F &= (1-v)Y_\alpha + vY_\beta. \end{aligned} \tag{A.8}$$

Which can be derived with the quadratic equations on A.7:

$$\begin{aligned} X_F &= (1-v)^3 X_A + 3(1-v)^2 v X_B + 3(1-v)v^2 X_C + v^3 X_D, \\ Y_F &= (1-v)^3 Y_A + 3(1-v)^2 v Y_B + 3(1-v)v^2 Y_C + v^3 Y_D. \end{aligned} \tag{A.9}$$

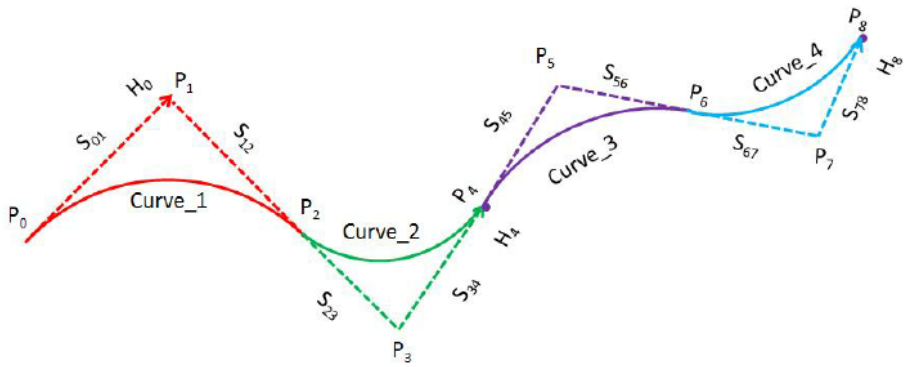
Figure A.6 – A cubic Bézier curve with auxiliary curves and their respective points α and β made to define F position



Source: the authors

- **Piece-wise Bézier curves** Another way to increase the complexity of Bézier curves but without the need of increasing the degree of the curve is by separating the curves in what is called *piece-wise Bézier curves*. The way these curves work is by having control points being shared between curves, generating a more complex Bézier curve. As pictured in fig. A.7, the combined curves can be, but are not limited to, quadratic Bézier curves, so it is possible to combine different degrees of Bézier curves with such combination.

Figure A.7 – A piece-wise Bézier curve, composed of quadratic Bézier curves



Source: Extracted from (SONG et al., 2015)

APPENDIX B — COMPUTER GRAPHICS CONCEPTS

This section presents fundamental concepts from computer graphics that are used in our application.

B.1 Rendering

Rendering is the main focus of computer graphics, and can be defined as drawing on a screen, or, in other words, the synthesis of an image using data that represents that image. It involves considering how each object in a virtual 3D scene contributes to each pixel in the 2D projection of the scene. This can be organized in two general ways:

- In *object-order rendering*, each object is considered in turns, and for each object all the pixels that it influences are found and updated. Rasterization is an example of object-order rendering.
- In *image-order rendering*, each pixel is considered in turn, and for each pixel all the objects that influence it are found and the pixel value is computed. Ray tracing is an example of image-order rendering.

B.1.1 Rasterization

Modern computer displays have a rectangular shape consisting of a grid of pixels. This is the definition of a *raster display* (MARSCHNER; SHIRLEY, 2015). Most computer graphics images are presented to the user on some kind of raster display. Because rasters are so prevalent in devices, raster images are the most common way to store and process images. A raster image is simply a 2D array that stores the pixel value for each pixel—usually a color stored as three numbers, for red, green, and blue. A raster image stored in memory can be displayed by using each pixel in the stored image to control the color of one pixel of the display (MARSCHNER; SHIRLEY, 2015).

Rasterization is considered *object-order rendering* and is the process of finding all the pixels in an image that are occupied by a geometric primitive (e.g. a triangle). For each primitive that comes in, the rasterizer has two jobs: it enumerates the pixels that are covered by the primitive and it interpolates values, called attributes, across the primitive. The output of the rasterizer is a set of *fragments*, one for each pixel covered by

the primitive. Each fragment “lives” at a particular pixel and carries its own set of attribute values (MARSCHNER; SHIRLEY, 2015).

Rendering by rasterization is done through the Graphics Pipeline, presented in appendix B.2

B.1.2 Ray Tracing

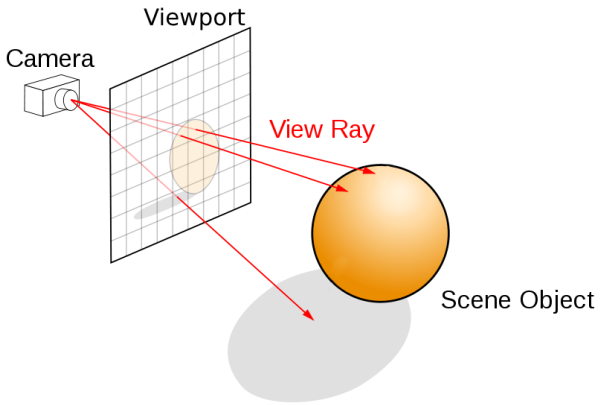
Ray tracing is an image-order algorithm for making renderings of 3D scenes. It works by computing one pixel at a time. For each pixel the basic task is to find the object that is seen at that pixel’s position in the image (MARSCHNER; SHIRLEY, 2015). A *ray* is simply a line that emanates from the viewpoint in the direction that pixel is positioned, as pictured in fig. B.1. The first object hit by the ray, the one nearest the camera is the one that should be shown. The color of the pixel can then be calculated using shading and the object’s *normals* - vectors that are orthogonal to the object’s surface. In recursive ray tracing this ray *bounces* - creates other rays on hit with a scene object - through the scene, before returning a color value. The problem with the recursive call above is that it may never terminate, for example, if a ray starts inside a room, it will bounce forever. This can be fixed by adding a maximum recursion depth (MARSCHNER; SHIRLEY, 2015).

Many effects that take significant work to fit into the object-order rasterization framework, including basics like the shadows and reflections, are simple and elegant in a ray tracer. Although simpler and more elegant, in raytracing shading a scene is more processing intensive because it takes a large number of rays to perform the shading in any nontrivial scene (MARSCHNER; SHIRLEY, 2015).

B.2 Graphics Pipeline

From this section onward we focus on rendering using only rasterization. The graphics pipeline, often referred to as the *rendering pipeline*, is the model that describes all the necessary steps a system must perform to successfully render a 3D scene to a projected 2D scene. It is the graphics pipeline that turns everything in a scene, e.g. models, textures, lighting, etc., and maps it to the computer display via rasterization (MARSCHNER; SHIRLEY, 2015). Most of the pipeline is implemented in hardware, for optimization reasons, and is *not programmable* - these steps are handled by the graphic

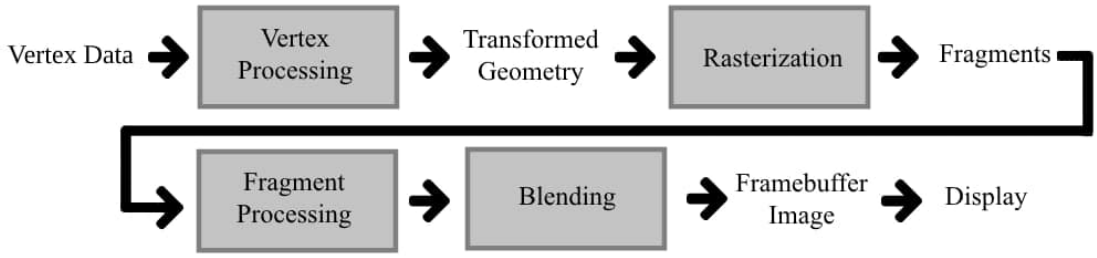
Figure B.1 – A ray emanating from a viewpoint in the direction of the pixel that is going to be rendered



Source: adapted from Wikimedia Commons

APIs (Application Programming Interface) - when using graphic APIs like OpenGL and DirectX.

Figure B.2 – The stages of a graphics pipeline (MARSCHNER; SHIRLEY, 2015)



Source: the authors

The pipeline is divided into multiple steps, each has its own functionality, each step result is carried over the next. Pictured in fig. B.2 are said steps, with highlighted blocks being programmable in the OpenGL rasterization graphics pipeline. To simplify its explanation we will focus our explanation on these highlighted blocks, together with the input and the output. According to Marschner and Shirley (2015):

- **(Input)** Vertex Data: geometric objects are fed into the pipeline as a set of vertices.
- Vertex Processing: vertices are then operated and the primitives using those vertices are fed into the next step of the pipeline. This step is where vertex shaders are executed, as explained in appendix B.2.1.
- Rasterization: conversion of each geometric primitive (e.g. triangles) into a number of *fragments*. *Fragment* is a term that describes the information associated with

a pixel prior to being processed in the final stages of the graphics pipeline. This definition includes much of the data that might be used to calculate the color of the pixel, such as the pixel's scene depth, texture coordinates, or stencil information.

- Fragment Processing: *fragments* are processed in this stage. This step is where fragment shaders are executed, as explained in appendix B.2.1.
- Blending: the combination of fragments generated by different primitives that overlapped each pixel. For opaque surfaces, a common blending approach is to choose the color of the fragment with the smallest depth (closest to the observer).
- **(Output)** Display: processed data is shown to the user.

B.2.1 Shaders

Both the vertex and fragment processing stages can be altered by the user with what is called *shaders*, which are programs executed in the Graphics Processing Unit (GPU) for each vertex or fragment:

- **Vertex shaders** provide control over how vertices are transformed. It can prepare data for later processing in the fragment processing stage.
- **Fragment shaders** provide control over how each fragment is going to be processed (e.g. where the final color of each fragment is defined).

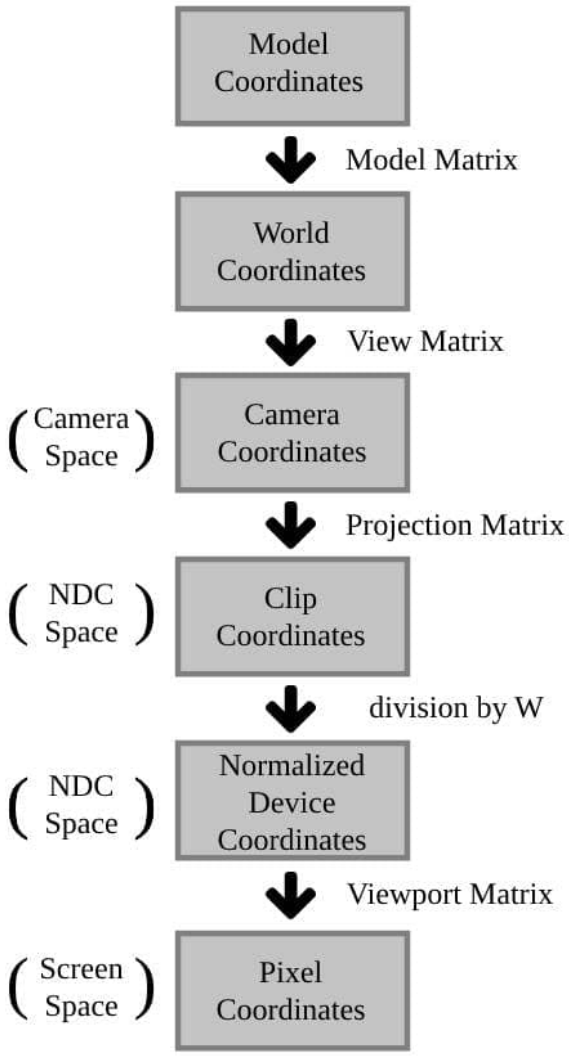
B.3 Model-View-Projection-Viewport Matrices

An observer and an observable object are necessary for the rendering process. For this, we use meshes in a 2D or 3D space, to represent objects, and cameras that represent the observer. In a 2D space, we have objects organized in the X and Y axis only, whereas in a 3D space, we add depth and introduce it as a new dimension, represented by Z.

Organizing where each object is and how each camera observes them in a 2D or 3D space is a challenge that is best approached by using linear algebra. With it, we can transform the coordinates of points and vectors through matrices so that each polygon is in the right position in a given configuration of observable objects and observers. For this purpose, one uses Model-View-Projection-Viewport (MVPV) matrices. These matrices follow the sequence that the name suggests, first the model then the view, the projection,

and finally the viewport matrices transformations. Figure B.3 shows the relation of how coordinates will behave in these transformations and their respective spaces, which will all be discussed in this section.

Figure B.3 – Model-View-Projection-Viewport matrices effects on coordinates



Source: the authors

An important principle of computer graphics is the projection of 3D points of a scene onto the 2D projection plane, divided into discrete pixels, available in the device rendering the scene. It is important to consider how depth will affect this projection so that we know which object will be drawn in front, and which in the back. Just as a ray tracer needs to find the closest surface intersection, a rasterization renderer needs to work out which of the (possibly many) surfaces drawn at any given point is the closest and only display it (MARSCHNER; SHIRLEY, 2015).

While the Model transformations depend only on the model coordinates, View, Projection and Viewport transformations depend on the camera position, the type of projection, the field of view (the resolution of the image and the size of the display also need to be considered if the scene is being rasterized). To simplify this procedure most graphic systems do it by a sequence of three transformations, according to Marschner and Shirley (2015):

- **Viewing transformation (View Matrix)** puts the objects from the scene in a coordinate system where the camera is at the origin in a convenient orientation, the result is what we call ***Camera Space***.
- **Normalized Device Coordinates (NDC) transformation (Projection Matrix)** projects points first to *clip coordinates*, which efficiently determine on an object-by-object basis which portions of the objects will be visible to the viewer. Clip coordinates are then divided by w, in what is called perspective division, creating Normalized Device Coordinates. In NDC all visible points fall in the -1 to 1 range, in a space called ***NDC Space***.
- **Viewport transformation (Viewport Matrix)** maps the unit image rectangle to the desired rectangle in pixel coordinates in what we call ***Screen Space***, the end goal for all transformations.

B.3.1 Model Matrices

Models are nothing more than a collection of vertices organized in a particular way, therefore we can transform them very quickly using matrices. The vertices collection of a model are relative to the origin in what is called *Local Space* or *Local Coordinate System* of the object. But models are usually not in the origin of the world (the coordinate $(0, 0, 0)$ of the 3D world, or $(0, 0)$ for a 2D world) this is where the model matrices come in: The most basic transformations are **scaling, translating, and rotating**. After the desired transformations, the model is considered to be in the *World Space*.

B.3.1.1 Scale

The simplest transform is scaling (MARSCHNER; SHIRLEY, 2015). Considering a 3D space and s_x, s_y and s_z as the scaling factor in the X, Y and Z coordinates

respectively:

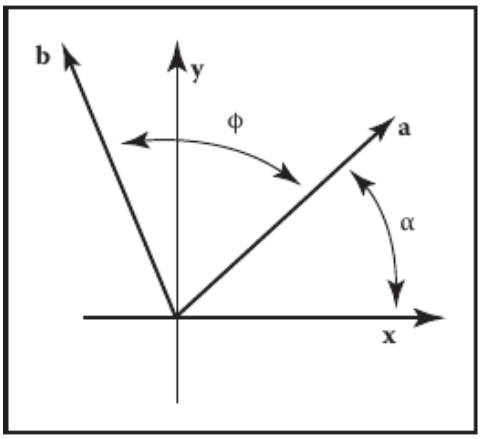
$$scale(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_x & 0 \\ 0 & 0 & s_z \end{bmatrix}. \tag{B.1}$$

Then applying eq. (B.1) to an arbitrary vector (x, y, z), we produce a scaled vector:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix}. \tag{B.2}$$

B.3.1.2 Rotation

Figure B.4 – Example of arbitrary counterclockwise rotation of vector a by ϕ



Source: extracted from (MARSCHNER; SHIRLEY, 2015)

As pictured in fig. B.4, considering a 2D space, supposing that we'd like to rotate a vector a by an arbitrary angle ϕ to get vector b, we first decompose the a vector (assumed to be a unit vector) in the X-axis, given that α is the angle between the vector and the X-axis, with basic trigonometric properties, we have:

$$\begin{aligned} x_a &= \cos\alpha, \\ y_a &= \sin\alpha. \end{aligned} \tag{B.3}$$

Since b is a rotation of a, its rotation makes a $\phi + \alpha$ angle with the X-axis, and we

can expand this sum with eq. (B.3), yielding:

$$\begin{aligned} x_b &= cos(\alpha + \phi) = \cos\alpha\cos\phi - sin\alpha sin\phi = x_a\cos\phi - y_a sin\phi, \\ y_b &= \sin(\alpha + \phi) = \sin\alpha\cos\phi + \cos\alpha\sin\phi = y_a\cos\phi - x_a\sin\phi. \end{aligned} \tag{B.4}$$

As can be seen by eq. (B.3) and eq. (B.4), the operation that transforms vector a into vector b can thus be represented by a matrix:

$$rotate(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}. \tag{B.5}$$

For 3D rotations, there are multiple possible axes of rotation (MARSCHNER; SHIRLEY, 2015), as it is not the focus of this work we will not detail how to derive its formulas. According to Marschner and Shirley (2015) if we want to rotate on the Z-axis:

$$rotateZ(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{B.6}$$

On the X-axis:

$$rotateX(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}. \tag{B.7}$$

And on the Y-axis:

$$rotateY(\phi) = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}. \tag{B.8}$$

B.3.1.3 Translation

Translations can only be done for points, since vectors cannot be translated. Up until this point we have been looking at methods that change a vector attributes using a matrix, but we cannot use such transforms to move points in a model, because the origin always remains fixed under these linear transformations. In order to achieve translation we need to shift all the points of an object by the same amount. For a 2D space, where (x^t, y^t) is the translation amount for x and y-coordinates, and (x', y') the resulting coordinates after the transformation:

$$\begin{aligned} x' &= x + x^t, \\ y' &= y + y^t. \end{aligned} \tag{B.9}$$

It is impossible to multiply x and y by a 2×2 matrix (as it was done in scale and rotation) and get this result. A possible way is to represent the point by using a 3D vector with the third coordinate as one $[x\ y\ 1]^T$, In order to do that it will be necessary to use a 3×3 matrix. According to Marschner and Shirley (2015):

$$\begin{bmatrix} 1 & 0 & x^t \\ 0 & 1 & y^t \\ 0 & 0 & 1 \end{bmatrix}. \tag{B.10}$$

This matrix does a linear transformation followed by a translation. This is called an *affine transformation*, and the way to implement them by adding an extra dimension is called *homogeneous coordinates* (PENNA; PATTERSON, 1986).

A problem with this approach happens when we transform vectors, because they represent directions and should not change when we translate the object. This can be easily fixed by setting the third coordinate to zero $[x\ y\ 0]^T$. Hence, when used together with eq. (B.10):

$$\begin{bmatrix} 1 & 0 & x^t \\ 0 & 1 & y^t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix}. \tag{B.11}$$

So when we refer to homogeneous coordinates we use the third dimension to express the difference between position/points and vectors/directions. For a 2D space, having the third dimension as 1 implies a *position*, when the third dimension is 0 we have a *direction*. In a 3D space, the same technique works, and we add a fourth dimension (MARSCHNER; SHIRLEY, 2015), hence:

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ z + z_t \\ 1 \end{bmatrix}. \tag{B.12}$$

B.3.2 View Matrix

As presented in the opening of the section, the View Matrix is applied to the object after its coordinates are processed by the model matrices transformations and are already in *World Space*. The View Matrix puts the camera at the origin of the world in a

rotation facing the Z-axis. To do this we'll define a matrix V with the following properties, according to Johnson (2020):

$$V \begin{bmatrix} r_x & u_x & f_x & p_x \\ r_y & u_y & f_y & p_y \\ r_z & u_z & f_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.13}$$

Considering r to be the right (positive X) vector of the camera, u to be the up (positive Y) vector of the camera, f the front (positive Z) vector of the camera (all three vectors, r, u, f are orthogonal to each other) and p to be the position of the camera in *World Space* (in this text we opted to use the conventional left-handed system in World Space coordinates, even though OpenGL uses the right-handed system (KHRONOS, 2012)). We have previously touched upon how rotation matrices and translation matrices were built in appendix B.3.1.2 and appendix B.3.1.3, respectively - so we can dismember the dot product matrix in eq. (B.13):

$$T = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{B.14}$$

$$R = \begin{bmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.15}$$

And if we consider the right-hand side of the eq. (B.13) as an identity matrix identified by I, we have:

$$V(TR) = I. \tag{B.16}$$

Solving eq. (B.13) for V yields:

$$\begin{aligned} &V(TR) = I, \\ &V(TR)(TR)^{-1} = I(TR)^{-1}, \\ &V = (TR)^{-1}, \\ &V = R^{-1}T^{-1}. \end{aligned} \tag{B.17}$$

Which then resolves to:

$$
\begin{aligned}
V &= R^{-1}T^{-1}, \\
V &= \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \\
V &= \begin{bmatrix} r_x & r_y & r_z & -p_x\, r_x - p_y\, r_y - p_z\, r_z \\ u_x & u_y & u_z & -p_x\, u_x - p_y\, u_y - p_z\, u_z \\ f_x & f_y & f_z & -p_x\, f_x - p_y\, f_y - p_z\, f_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.
\end{aligned}
\tag{B.18}
$$

B.3.3 Projection Matrix

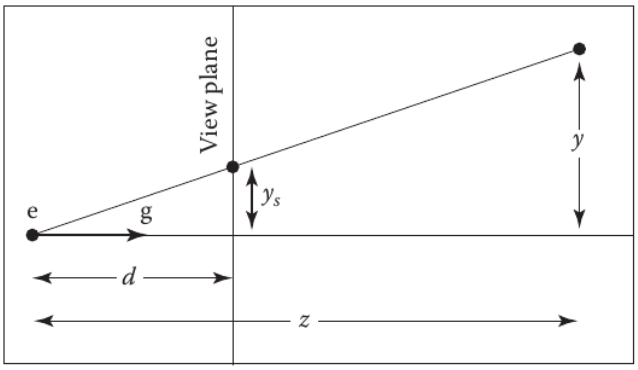
The projection matrix is the second to last matrix that is necessary in the sequence of matrices needed to render a point in the correct portion of the screen. It will produce the NDC space, a space where X, Y, and Z coordinates vary between -1 and 1, and any point that falls outside this range will be discarded further down on the graphics pipeline. Section 4.9 presents and discusses a way to visualize the NDC space.

B.3.3.1 Perspective Projection

As pictured in fig. B.5, considering a camera e facing the direction g in a distance d from a view plane, limited by e's range of vision, determined by y_s and y, so that y_s is the intersection with the view plane and y is the furthest point of the range of vision (distant of z from the camera e), we can extract how y_s depends on the distances d and z properties from fig. B.5 using similar triangles (this will be the basis of the projection of y in the view plane):

$$
\begin{aligned}
\tfrac{y_s}{d} &= \tfrac{y}{z}, \\
y_s &= d\tfrac{y}{z}.
\end{aligned}
\tag{B.19}
$$

Figure B.5 – Geometry involved in how a camera e sees



Source: extracted from (MARSCHNER; SHIRLEY, 2015)

One of the input vectors of eq. (B.19) appears as a denominator, and this can't be achieved with just *affine transformations*, in a similar situation we had in appendix B.3.1.3. Hence it is necessary to use homogeneous coordinates again. With homogeneous coordinates, we agreed that the point (x, y) can be represented with the homogeneous vector $[x\ y\ 1]^T$.

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} d\frac{y}{z} \\ d \\ 1 \end{bmatrix}. \tag{B.20}$$

The third dimension for our homogeneous vector in this 2D space will be called w and will represent the *perspective distortion*. The *perspective distortion* is necessary to compensate the viewer's perspective in a scene. In an orthographic environment, presented in appendix B.3.3.2, parallel lines will never meet, but in a perspective environment this is not true and they meet in infinity. When *perspective distortion* is applied the farther the object is, the more distorted it will become, achieving this effect. This will be more apparent in a 3D space, so for now we will consider $w = z$.

After the projection matrix - which is not an *affine transformation* - has been applied, w can (and should) be != 1, in what is called clipping coordinates. For this reason, the perspective distortion should divide the homogeneous vector:

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dy \\ dz \\ z \end{bmatrix} div.w \rightarrow \begin{bmatrix} d\frac{y}{z} \\ d \\ 1 \end{bmatrix}. \tag{B.21}$$

Which then finally defines our perspective matrix M:

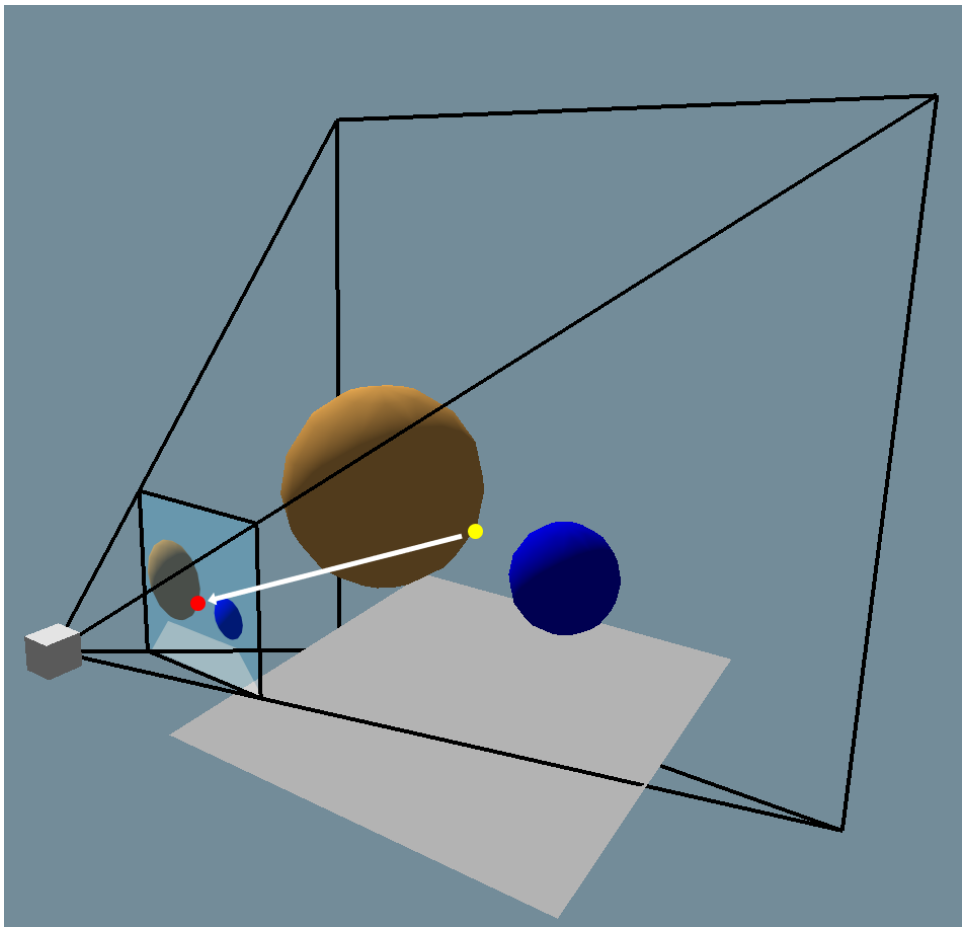
$$M = \begin{bmatrix} d & 0 & 0 \\ 0 & d & 0 \\ 0 & 1 & 0 \end{bmatrix}. \tag{B.22}$$

In a 3D space, adding the x dimension, we have:

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ dz \\ z \end{bmatrix} \; div.w \rightarrow \begin{bmatrix} d\frac{x}{z} \\ d\frac{y}{z} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}. \tag{B.23}$$

Points $[x'\ y'\ z']$ in eq. (B.23) will represent the projection of points $[x\ y\ z]$ in the viewport plane, as pictured in fig. B.6.

Figure B.6 – Projection of $[x\ y\ z]$, pictured in yellow to $[x'\ y'\ z']$, pictured in red, on top of the view plane



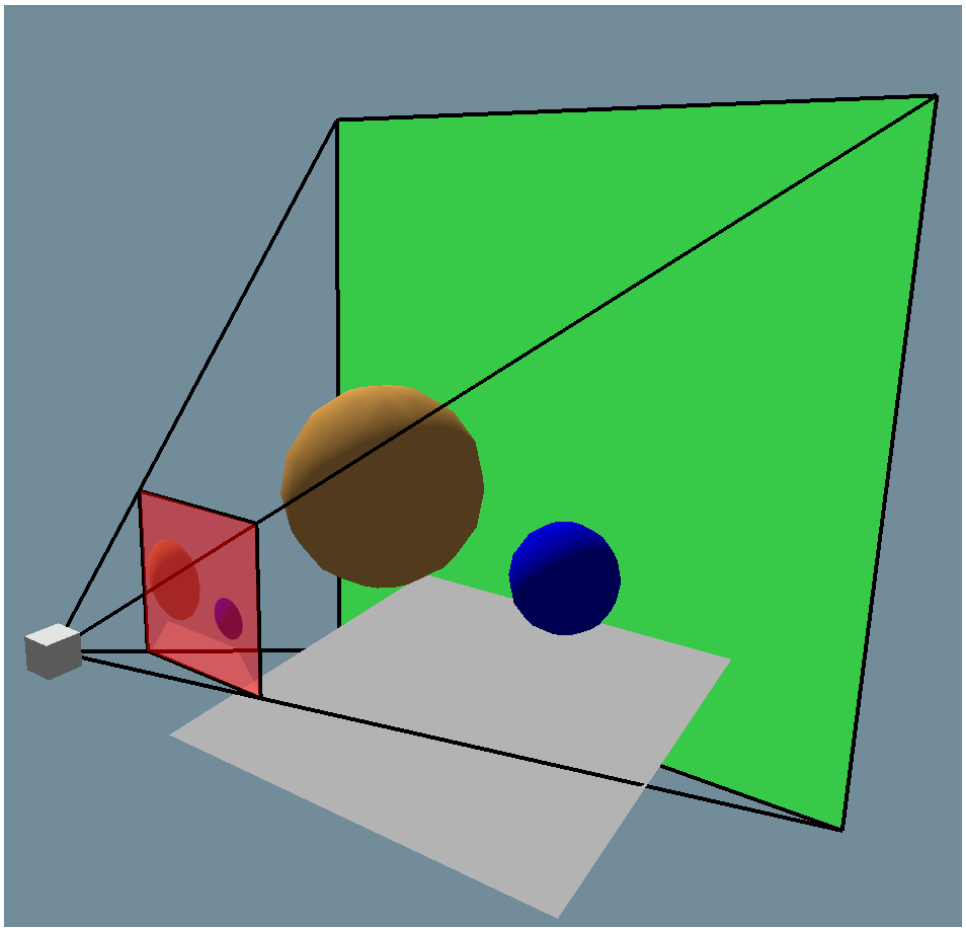
Source: the authors

From eq. (B.23) we can see that z' has a particular characteristic that makes it always sit on top of the view plane and z will always be distant d from the viewer, so depth is lost in this projection, but the ordering of objects can be preserved using the *z-buffer algorithm*.

In simple terms, the *z-buffer algorithm* is a way to know the order of all fragments that project onto the same pixel. At each pixel, in the fragment blending phase of the graphics pipeline, we keep track of the distance to the closest surface that has been drawn so far and discard fragments that are farther away than that distance. The closest distance is stored by allocating an extra value for each pixel, in addition to the red, green, and blue color values. To ensure that the first fragment will pass the depth test the *z-buffer* is initialized to the maximum depth (MARSCHNER; SHIRLEY, 2015).

Since our previous matrix ignored all depth by projecting z in the view plane. We'll define a *view frustum* that will act as the boundaries of what can be seen and will help in organizing the orders of the fragments for the *z-buffer algorithm*.

Figure B.7 – The near plane n, pictured in red color, and the far plane f, pictured in green color



Source: the authors

We define two planes for the view frustum, a near plane n and and a far plane f, pictured in fig. B.7. Both of these planes will be defined by the distance from the viewer. With this information we can define the perspective matrix with depth information:

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} div.w \rightarrow \begin{bmatrix} n\frac{x}{z} \\ n\frac{y}{z} \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}. \tag{B.24}$$

We can test the depth z' using $n + f - \frac{fn}{z}$ with our desired output:

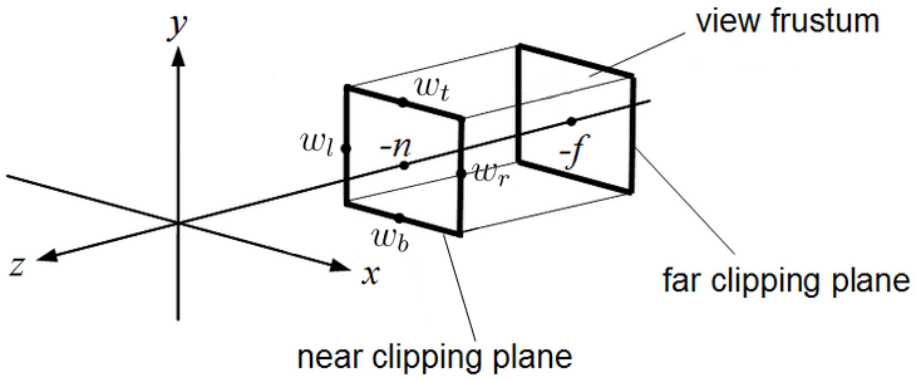
- If $z = n$ we have $n + f - \frac{fn}{z} = n$.
- If $z = f$ we have $n + f - \frac{fn}{z} = f$.
- And $n < z_1 < z_2 < f$ yields $n < z'_1 < z'_2 < f$.

We are not in NDC space yet, the next step is orthographic projection, seen in the next section, which will accommodate the *field of view* and the *aspect ratio* of the view frustum.

B.3.3.2 Orthographic Projection

To consider which portions of the near and far plane will compose the view frustum we need to introduce the concept of *clipping window*, which will work to define a rectangle for the view frustum, this will be the *view volume*.

Figure B.8 – Clipping window and its coordinates (w_t, w_b, w_l, w_r)



Source: adapted from Wikimedia Commons

The word clipping (removing) already suggests that only everything inside it will be considered for the rendering of the final image, saving computer resources. This is done by considering four coordinates: (w_t, w_b, w_l, w_r), top, bottom, left, and right, respectively, that will determine where the frustum view volume will be defined, as pictured in fig. B.8.

After having the view frustum volume defined we need to transform it in a cube to finally reach the NDC space - as explained at the beginning of appendix B.3 - which is centered at the origin, serving the purpose of simplifying the later stages of the graphics pipeline. According to Power (2011), for this to happen we need the following.

- translate the frustum view volume to be centered at the origin,
- scale the frustum view volume to the size of the normalized view volume.

The center of the frustum volume can be defined as $(\frac{w_l+w_r}{2}, \frac{w_t+w_b}{2}, \frac{near+far}{2})$, which enables us to translate it to the origin using a translation matrix:

$$T = \begin{bmatrix} 1 & 0 & 0 & -\frac{w_l+w_r}{2} \\ 0 & 1 & 0 & -\frac{w_t+w_b}{2} \\ 0 & 0 & 1 & -\frac{near+far}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.25}$$

The normalized frustum view volume has a height, width, and depth of 2, since the NDC space is defined by a cube in ranges $[-1, 1]$. To scale a value from one range of values to another, we divide by the size of the original range and multiply by the size of the new range, hence:

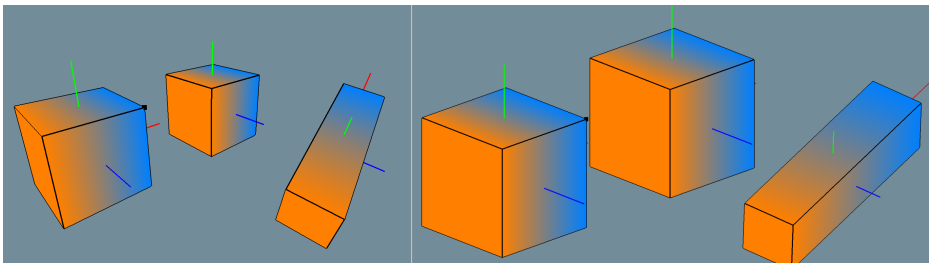
$$S = \begin{bmatrix} \frac{2}{w_r-w_l} & 0 & 0 & 0 \\ 0 & \frac{2}{w_t-w_b} & 0 & 0 \\ 0 & 0 & \frac{2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.26}$$

Therefore our affine transformation matrix for the orthographic projection O is defined by:

$$O = ST = \begin{bmatrix} \frac{2}{w_r-w_l} & 0 & 0 & -\frac{w_l+w_r}{2} \\ 0 & \frac{2}{w_t-w_b} & 0 & -\frac{w_t+w_b}{2} \\ 0 & 0 & \frac{2}{far-near} & -\frac{near+far}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.27}$$

It is important to highlight that the orthographic projection matrix does not require that the perspective projection matrix be applied first, it can be used by itself producing the effect seen on the right-hand side of fig. B.9, opposed to the perspective projection (with orthographic projection applied posterior) in left-hand side.

Figure B.9 – Perspective and orthographic projections differences, left-hand side and right-hand side, respectively



Source: our system

B.3.4 Viewport Matrix

After the Perspective/Orthographic Projection and the division by w we get in the NDC space (as shown in fig. B.3), which means that we all the points coordinates are between -1 and +1, in all three dimensions (x, y, z). To reach Screen Space, we need to map the points from NDC space to the viewport in the screen, i.e. the window that is going to be showing the render. For that to happen we must first scale the NDC cube to the shape of the viewport and then transform the NDC cube to the plane defining the viewport, we can do this with an affine transformation (MARSCHNER; SHIRLEY, 2015), which was presented in appendix B.3.3.1.

The viewport is a rectangle defined by the height and the width of the window of the application. Considering v_t and v_b as the y-coordinates of the top and the bottom, respectively, of the window, and considering v_l and v_r as the x-coordinates of the left and the right, respectively of the same window. We divide each fraction by the x and y coordinates of the NDC space, which are in the $(-1, -1, -1)$ to the $(1, 1, 1)$ range:

$$S = \begin{bmatrix} \frac{v_r - v_l}{1-(-1)} & 0 & 0 & 0 \\ 0 & \frac{v_t - v_b}{1-(-1)} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{v_r - v_l}{2} & 0 & 0 & 0 \\ 0 & \frac{v_t - v_b}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.28}$$

Equation B.28 covers the scaling of the points to the viewport's size. We then need to translate them with an affine transformation:

$$T = \begin{bmatrix} 1 & 0 & 0 & \frac{v_r - v_l}{2} \\ 0 & 1 & 0 & \frac{v_t - v_b}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.29}$$

Joining eq. (B.28) and eq. (B.29) into viewport matrix V we have:

$$V = ST = \begin{bmatrix} \frac{v_r - v_l}{2} & 0 & 0 & \frac{v_r - v_l}{2} \\ 0 & \frac{v_r - v_l}{2} & 0 & \frac{v_t - v_b}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.30}$$

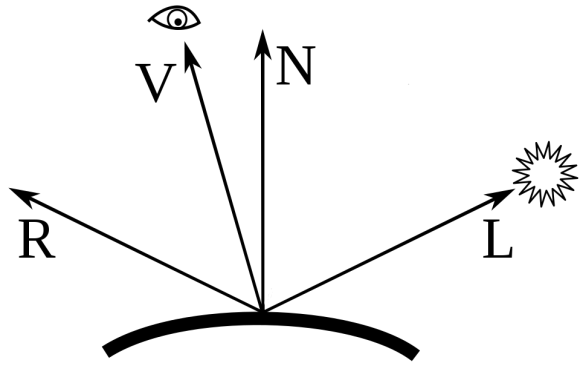
B.4 Lighting and Shading

Lighting in the real world happens when photons interact with matter, bounce, and are absorbed by the retina, where the information they carry is transcoded to electric signals that are processed by the brain (MARSCHNER; SHIRLEY, 2015). In computer graphics, a similar effect can be achieved using ray-tracing, though it is very hard and expensive to achieve it in real-time. A simpler and less expensive way to do it is to use local illumination models (or local shading models), which are easily implementable in GPU programmable shaders.

With shaders, we can approximate what lighting would look like using attributes such as the light position and the viewer position. As previously explained, light can be perceived as photons bouncing off surfaces to reach the retina, with vertex and fragment shaders we can use math to estimate this effect. In the following subsections, we will use the following vectors to illustrate how each shading algorithm works.

- L is the direction from the surface to the light point.
- N is the surface normal vector.
- V is the direction pointing towards the viewer.
- R is the direction L but mirrored in relation to the normal vector N.

Figure B.10 – Vectors involved in a reflecting surface



Source: Extracted from Wikipedia

B.4.1 Phong Illumination Model

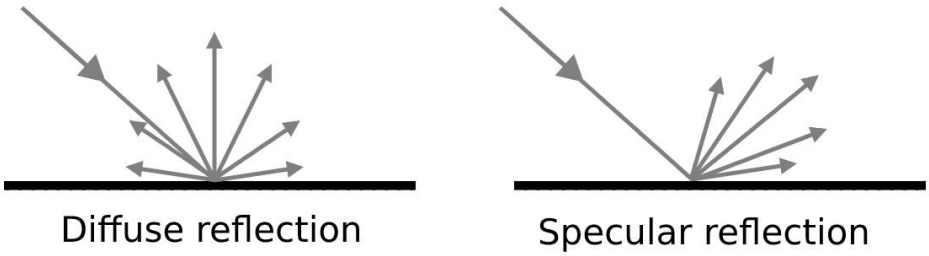
Phong Illumination Model is the simplest and the most popular lighting for 3D graphics because it is flexible to achieve a different range of visual effects and its implementation is efficient in the hardware. According to Buss (2003), there are two types of Phong reflections:

- **Diffuse reflection** reflects the light evenly in all directions away from the surface, as pictured in the left-hand side of fig. B.11. Predominant in non-shiny surfaces.
- **Specular reflection** reflects the light in a mirror-like way, which means that the reflected light is scattered in a determined angle, as pictured in the right-hand side of fig. B.11
- **Ambient Light reflection** is light that arrives equally from all directions rather than from a light source. Ambient reflection is intended to model light that has spread around the environment through multiple reflections.

Considering $Light_t$ as all the lights in the scene, K_d as our diffuse reflectance, K_s as our specular reflectance, K_a as our ambient reflectance, considering $\wedge$ as the symbol indicating normalized vectors and $\bullet$ as the dot product:

$$V = K_a + max(K_d(\hat{N} \bullet \hat{L}),\ 0) + max(K_s(\hat{R} \bullet \hat{N}),\ 0)\ \forall\ Light\ in\ Light_t. \quad \text{(B.31)}$$

Figure B.11 – Diffuse reflection in the left-hand side and specular reflection in the right-hand side



Source: the authors

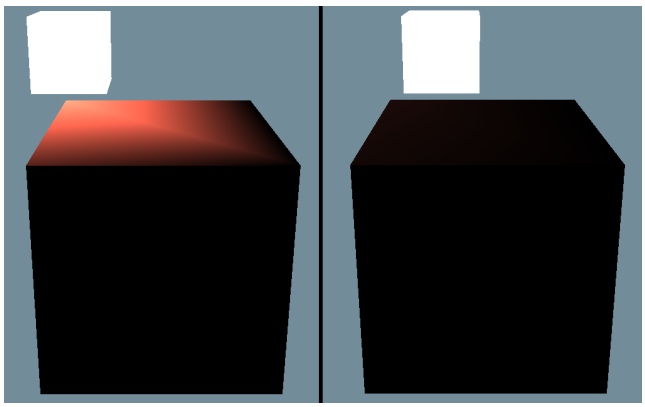
If we consider that the material reflecting the light can have different reflectance q, and also consider that each light source I has different attributes for each type of reflection, so we have I_d for diffuse light reflectance, I_s for specular light reflectance, I_a for ambient light reflectance we can make the equation more interesting and also more flexible:

$$V = K_a I_a + max(K_d I_d(\hat{N} \bullet \hat{L}),\ 0) + max(K_s I_s(\hat{R} \bullet \hat{N})^q,\ 0). \tag{B.32}$$

B.4.2 Gouraud Shading

The Gouraud reflection model uses an estimation of the surface normal on each vertex of a 3D model by averaging the surface normals of the triangles that meet at each vertex. This estimation is then used to produce different color intensities at the vertex, which in turn can be interpolated with the color of the other vertices, creating the effect of reflection. While limited, this technique is very inexpensive to use. Due to its limitation of only reflecting on the vertex and not the fragment, the reflection usually appears faceted, and not having light directly over a vertex in a low poly model will produce incorrect reflections, pictured in fig. B.12.

Figure B.12 – Gouraud shading correctly over a vertex of a cube on the left-hand side, and Gouraud shading not reflecting on the right-hand side



Source: the authors

B.4.3 Types of light sources

Light can interact with the scene's object in different more interesting ways than only the angle of incidence. We are able to use different types of light sources, including directional lights, point lights and spotlights.

B.4.3.1 Directional

A directional light will always have the same direction, meaning that all light rays will be parallel to each other, i.e. in fig. B.10 vector L will always be in the same direction, independent of the position of the directional light. This creates a very similar effect to sunlight.

B.4.3.2 Point

Point lights are equivalent to light bulbs, they emit light in all directions equally. Added to it we can calculate how light dissipates in farther distances from a given point. We call this *light attenuation* and its formula is given by:

$$\sigma = \frac{1}{k_c + k_l d + k_q d^2}. \tag{B.33}$$

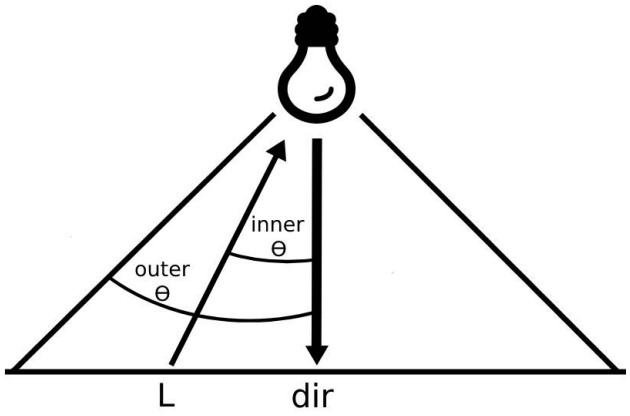
d is the distance from the light and the constant scalars are k_c, k_l and k_q, the constant attenuation factor, the linear attenuation factor, and the quadratic attenuation factor, respectively (BUSS, 2003). All light intensity values, I_d, I_s and I_a are multiplied

by the distance attenuation factor σ before being used in eq. (B.32). Vector L is pointing towards a point in a surface.

B.4.3.3 Spot

A spotlight is characterized by having a direction dir, two cutoff angles θ_{inner} and θ_{outer} which are the inner and outer angles of the cone of light (considering $\theta_{inner} < \theta_{outer}$), as pictured in fig. B.13, and a spotlight exponent ϵ which controls how fast the light intensity decreases from the center of the spotlight. Vector L is pointing towards from a point in the surface.

Figure B.13 – A spotlight and its angles in relation to its direction



Source: the authors

$$\begin{aligned} \theta &= -\hat{L} \bullet dir, \\ \epsilon &= \theta_{inner} - \theta_{outer}. \end{aligned} \tag{B.34}$$

θ returns the cosine of the smallest angle between vector L and the direction of the spotlight, scaled by the vector's norm . The intensity I_{spot} can then be calculated using:

$$I_{spot} = \frac{\theta - \theta_{outer}}{\epsilon}. \tag{B.35}$$

This intensity I_{spot} should only be used in a $[0, 1]$ range, so this result must be clamped to this range and it can be used in conjunction with attenuation seen in eq. (B.33).

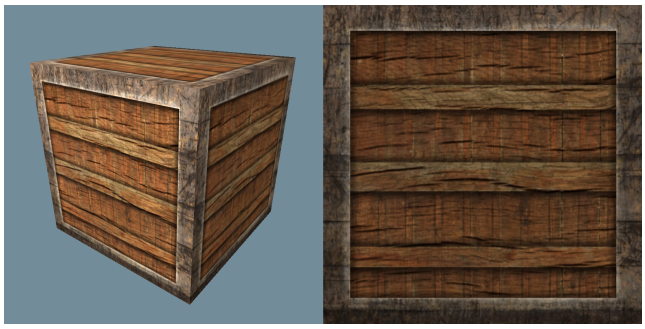
B.5 Texture Mapping

Objects in real life are hardly as uninteresting as a cube with only one color. A cardboard box, for example, has a texture and if you look close enough you can see how this texture affects the light bouncing off of it creates a more interesting look than a simple uniform color. In computer graphics, this can be simulated using textures. Textures can be used to make shadows and reflections, to provide illumination and even surface shape (MARSCHNER; SHIRLEY, 2015). But mapping textures to shapes can be challenging and there are several ways one can go about doing it. In this section, we will look at a few of these approaches.

B.5.1 UV mapping

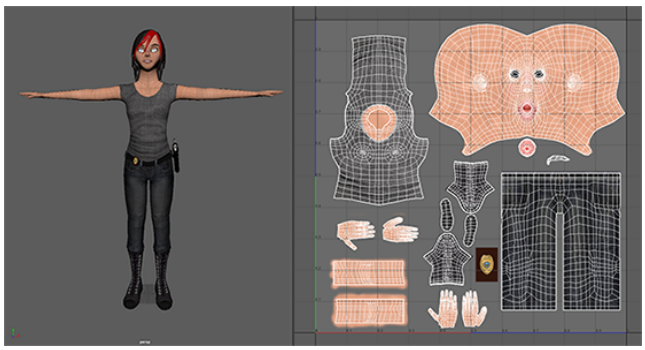
The simplest way to map a texture is using coordinates, this can be as simple as having each vertex of a cube be mapped to all the extremities of an image (fig. B.14), or as complex as having each vertex map to a single point in an unwrapped texture (fig. B.15). The way this is achieved is by using the textures coordinates, called u and v, mapped to the coordinates x and y of a given surface, hence the name *UV mapping*.

Figure B.14 – Simple UV mapping



Source: the authors

Figure B.15 – Complex UV mapping

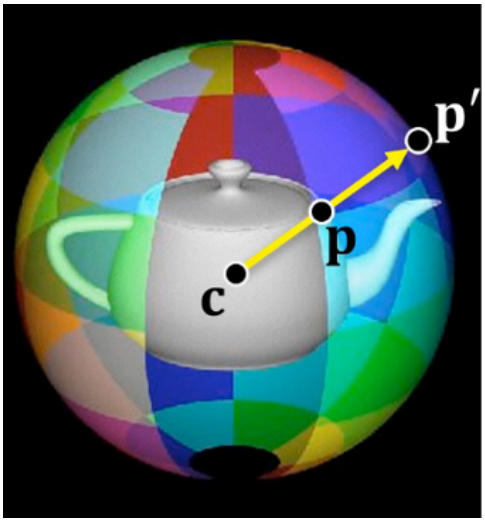


Source: Extracted from Autodesk Maya Documentation, (MAYA, 2018)

B.5.2 Texture projection

Another way to texture an object is to have a textured shape that envelops it and dictates what color each point of the object is going to be. A few examples include cubic, spherical (pictured in fig. B.16), cylindrical, and cube map projection. More on section 4.10.

Figure B.16 – Spherical projection on a teapot model



Source: Extracted from Wolfe (1997)