BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

HOLOGRAPHIC INJECTION
HOLOGRAFICKÁ INJEKCE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ROMAN DOBIÁŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

Master's Thesis Specification

Student:	**Dobiáš Roman, Bc.**
Programme:	Information Technology
Field of study:	Computer Graphics and Multimedia
Title:	**Holographic Injection**
Category:	Computer Graphics

Assignment:

1. Familiarize yourself with the topic of code injection, hooking and visualization on Looking Glass devices.
2. Design an application allowing conversion of existing 3D applications for holographic displays.
3. Implement the application according to the design.
4. Test the application using several 3D applications.
5. Evaluate, measure, draw conclusions and create a demonstration video.

Recommended literature:

- Juan Lopez, Leonardo Babun, Hidayet Aksu, Selcuk Uluagac: A Survey on Function and System Call Hooking Approaches, Journal of Hardware and Systems Security, 2017

Requirements for the semestral defence:

- Items 1 and 2 and the core of the application.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Milet Tomáš, Ing.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	April 6, 2021

Abstract

The adaptation of upcoming autostereoscopic displays by regular users depends on availability of supported applications. To increase such set, this thesis describes compatibility software which turns (semi)-automatically the output of regular OpenGL 3D applications to display-native output of autostereoscopic displays, which take advantage of true 3D displays capabilities. This is achieved using a conversion layer that intercepts subset of OpenGL API call and translates such API calls to the different ones that produce multiview output of the original application.

The thesis is mostly devoted to the process of incremental design of the conversion layer to support different stages of OpenGL API. The description is focused on explaining decisions and alternative possibilities of available API calls.

In the end, examples of converted applications are shown with identified problems, analyzed performance, and suggestions for further development.

Abstrakt

Táto práca sa zaoberá návrhom a implementáciou nástroja, ktorý umožní používať klasické 3D OpenGL aplikácie na tzv. autostereoskopických displayoch s plným využitím ich hĺbkových možností a s minimálnym zásahom od užívateľa. Nástrojom je konverzná vrstva, ktorá umožní transparentne beh OpenGL aplikácií s interným rozšírením o vykreslenie z viacerých pohľadov vo formáte, vhodnom pre 3D display.

Motiváciou tejto diplomovej práce je potenciálne rozšírenie tzv. autostereskopických displayov, ktoré je v súčasnosti závislé na cene a dostupnosti špecializovaných aplikácií pre tieto displaye.

Text práce sa zaoberá dizajnom takejto vrstvy z pohľadu nutných API volaní, ktoré je potrebné korektne prepísať, aby aplikácie, vytvorené pomocou jednotlivých verzii štandardu OpenGL, pracovali správne, ako aj popisom problémov, ktoré vznikajú použitím rôznych vykreslovacích techník, a ktoré sú motiváciou pre komplexnejšie chovanie nástroja.

Na záver práce sú ukážky konverzie programov, dopad na výkonnosť, ako aj identifikácia nedostatkov konverznej vrstvy s návrhmi možných riešení pre ďalší vývoj.

Keywords

OpenGL, autostereoscopic displays, pipeline injection, single to multiview conversion, automated conversion, API call hooking, Looking Glass, projection extraction

Klíčová slova

OpenGL, autostereoskopické displaye, injekcia pipeliny, konverzia z jedného do mnohých pohľadov, automatizovaná konverzia, odchytávanie API volaní, Looking Glass, extrakcia projekcie

Reference

DOBIÁŠ, Roman. *Holographic Injection.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet

Rozšířený abstrakt

Táto diplomová práca je zameraná na nový typ displayov, ktoré umožňujú vytvárať hĺbkový vnem zobrazovaného obrazu bez nutnosti nosenia špecializovaných zariadení, akými sú 3D okuliare. Tieto tzv. autostereoskopické displaye sa postupne začínajú objavovať na trhu pre široké masy konzumentov. Keďže takéto zariadenia potrebujú odlišný formát vstupných dát, aktuálne aplikácie nedokážu plnohodnotne využiť ich výhody zobrazovania, a preto nie sú až tak atraktívne pre konečného spotrebiteľa. Existujúce aplikácie je, prirodzene, možné upraviť manuálne tak, aby ich výstupný formát zodpovedal požiadavkám displayov, avšak za cenu ľudského úsilia a iných nákladov. Motiváciou diplomovej práce je hľadanie spôsobu, ako takýto prevod čo najviac automatizovať bez nutnosti zásahu do zdrojových kódov programu, teda vytvoriť metódu, ktorá by dokázala univerzálne transformovať ľubovoľnú 3D OpenGL aplikáciu.

V úvodnej kapitole 2 práca poskytuje prehľad dnes známych a často používaných zobrazovacích technológií pre zobrazovanie 3D scén s dôrazom na hĺbkový vnem. Práca teda spomína rozličné metódy v stereografii, teda v oblasti, ktorá je verejnosti populárna pre anaglyfy (červeno-modré obrázky s prislúchajúcimi okuliarmi), či rozličné aktívne a pasívne technológie 3D okuliarov, ktoré sú dnes masovo využívané v 3D kinách. Do tejto oblasti patrí taktiež Virtuálna Realita (VR), ktorá aktuálne dominuje trhu vo forme špeciálnych okuliarí so vstavanou zobrazovacou plochou *(ang. Head-Mounted Display)*.

Cieľom tejto kapitoly je ďalej predstaviť autostereoskopické displaye, ich základne princípy funkčnosti a výhody a nevýhody v porovnaní s existujúcimi stereoskopickými technológiami. Gro kapitoly tvoria takzvané light-field displaye, ktoré sú založené na definícii scény pomocou množiny pohľadov, a následne formovaniu výstupného obrazu na základe interpolácie. Súčasťou kapitoly je aj formálna definícia light fieldov, ktorá poskytuje čitateľovi lepšiu predstavu o formovaní, resp. syntéze obrazu, ktorý bude na koniec zobrazený na displayi.

Taktiež je predstavený konceptu quiltu, obecného obrazového formátu tvoreného mriežkou pohľadov, ktorý predstavuje diskrétnu inštanciu light fieldu, a ktorý je následne možné previesť do formátu, špecifického pre konkrétny display. Ultimátnym cieľom diplomovej práce je teda zabezpečiť, aby existujúce aplikácie produkovali obraz vo forme quiltu.

Kapitola taktiež jemne načŕta alternatívy v podobe volumetrických displayov, ktoré ale pracujú nad odlišnými vstupmi, ktoré nie sú jednoducho prevoditeľné z light fieldov.

V závere kapitoly sú prezentované aktuálne metódy, ktoré slúžia na obecný prevod stereo obrazu na autostereoskopický formát *(ang. stereo to multiview)*, alebo sa dotýkajú oblasti konverzie 3D bežných aplikácii na stereoskopické displaye.

Kapitola 3 uvádza čitateľa do problematiky úpravy existujúcich programov bez zásahu do zdrojových kódov. Táto obecná metóda funguje na princípe vkladania medzivrstiev medzi samotnú aplikáciu a jej závislosti. Novo vložené medzivrstvy majú možnosť „zavesiť sa“ *(ang. hooking)* na komunikáciu medzi vrstvami a túto komunikáciu filtrovať či inak ovplyvňovať. K aplikácii tohoto prístupu je možné buď využiť princípy závadzača programu *(ang. dynamic loader)* alebo napadnúť proces originálnej aplikácie a prepisovať funkcie pomocou zmeny inštrukcií *(ang. code patching)*.

Kapitola 4 tvorí gro práce. Jej cieľom je popísať ako vnikajúca vrstva bude prevádzať komunikáciu aplikácie rozhraním OpenGL tak, aby výsledný obraz aplikácie bol viacpohľadový *(ang. multiview)*. Práca začína popisom konverzie tzv. fixed-pipeline prístupu vo vykreslovaní OpenGL, ktoré predstavuje jednoduchší prípad. Vďaka fixnosti a štandardizácii je postačujú ku prevodu takýchto aplikácií priamo rozšíriť transformačnú maticu

pomocou k tomu určených API volaní, a zároveň odchytávať a agregovať vykreslovacie príkazy pomocou mechanizmu tzv. display listov.

Rozšírenie programovateľnej pipeline-y OpenGL predstavuje komplexnejší problém z dôvodu väčších možností, ktoré programátor aplikácie dostáva k predávaniu, spracovaniu a uchovaniu dát. Práca definuje proces, počas ktorého sú extrahované parametre projekčnej matice pôvodného shader programu za cieľom invertovať už transformovanú pozíciu vrcholu späť do priestoru kamery. Následne je možné aplikovať relatívnu transformáciu a projekciu pre daný pohľad quiltu. To je zabezpečené rozšírením (injekciou) kódu do existujúcich shaderov aplikácie.

Z dôvodu podpory aplikácií, ktoré svoj výsledný obraz skladajú z viacerých vykreslovacích prechodov do pomocných vykreslovacích objektov *(ang. Frame Buffer Object)*, práca skúma možnosti replikácie takýchto FBO v pozadí *(ang. shadowing)* pomocou mechanizmov vrstvených FBO objektov *(ang. Layered FBO)* a možnosti vytvárať textúry, ktoré vnútorne odkazujú na podčásť pamäte už existujúcej textúry *(ang. Texture View)*. Cieľom je zabezpečiť konzistentnosť vzorkovania dát z prechádzajúcich prechodov pri vykreslovaní, kedy vstupnú textúru modelu tvorí objekt, ktorý vznikol vykreslením v predchádzajúcom prechode. Bez ošetrenia tohoto prístupu by dochádzalo ku efektu zrkladenia *(ang. mirroring)* vstupnej textúry.

Posledným konceptom práce je rozšírenie existujúcich shader programov o tzv. instanciaciu Geometry Shaderov, ktorá komfortne umožňuje paralelne zapisovať do vrstiev FBO počas jedného vykreslovacieho príkazu. To potenciálne pomáha znížiť réžiu, potrebnú pre vykreslenie viacerých pohľadov pre každý vykreslovací príkaz originálnej aplikácie. Nasledujúca kapitola 5 sa venuje popisu implementácie z pohľadu navrhnutej architektúry. Popísaný je finálny spôsob hookingu ako aj jednotlivé komponenty, ktoré zabezpečujú správu metainformácií, získaných z pozorovania OpenGL volaní, a ktoré umožňujú riadiť proces konverzie.

Obsahovo posledná kapitola 6 popisuje experimenty nad existujúcimi aplikáciami. Vzhľadom na rozsah OpenGL priestoru API volaní a komplexnosť konverzie sú popísané prípady, pre ktoré konverzia nefunguje alebo ju jednoducho nie je možné plne automatizovať. Kapitola obsahuje analýzu dopadu na výkon vzhľadom na komplexnosť originálnych scén.

Výsledná práca je implementovaná pre operačný systém Linux, a bola vyskúšaná na rozličných dostupných OpenGL programoch. Praktickým výstupom práce je teda proof-of-concept implementácia, ktorej komercializácia, či prosté obecné nasadenie by vyžadovalo mnoho človekohodín v oblasti robustného testovania a odľadenia aplikácie.

Implementácia dokáže niektoré vybrané aplikácie transformovať dokonalo, u niektorých dochádza ku artefaktom v prípade vykreslovania objektov shaderom, ktorý nebolo možné previesť, či samotné vykreslovanie zmizne z dôvodu nekozistentnosti transformačného reťazca po úprave.

Implementovaná práca podporuje zásah užívateľa vložením konfigurácie pre konkrétne shadery, ktorými je možné vynútit výsledok transformácie, či skryť geometriu, vykreslovanú týmto typom shaderu.

Táto diplomová práca bola prezentovaná vo forme článku s názvom *Holographic Injection - Let There Be True 3D* na študentskej konferencii Excel@FIT VUT, ročník 2021, na ktorom získala ocenenie odborným panelom, ako aj cenu partnera priemyslu (Edhouse).

Holographic Injection

Declaration

I hereby declare that this Diploma's thesis was prepared as an original work by the author under the supervision of Mr. Tomáš Milet. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Roman Dobiáš
May 17, 2021

Acknowledgements

I would like to express my gratefullness for a long-term collaboration with supervisor Tomáš Milet, his devotion to his job as an advisor and to many interesting thoughts I had been shown during our consultations.

While talking about this thesis, I must definitely give credits to all of these open-source SW that I've been using. The thesis was written using VIM with multiple plugins (e.g. VimTex for LaTeX). The figures were prepared using GIMP and Inkscape. Both are great tools, except for Inskcape's UI, which I find hard to manipulate with.

Next, I would like to express my acknowledgement to all tax-payers, who contributed to my six-years-long study, with one year spending abroad. I promise I will try to make this money pay back for society.

Speaking of society, I have to mention my two crazy housemates, Juraj and Michal, who are keen players of Bulanci and Worms, and who managed to keep my mind busy with cooking, drinking beer and inviting our friends for Friday's night parties.

Last but not least, I can't forget to say thanks to my supportive girlfriend, who accepts my devotion to finish my thesis with all effort that is available.

Žite tak, aby po vašej smrti nezavládla
rovnaká radosť ako pri vašom narodení.

Stanislav Radič

...but in this world nothing can be said
to be certain, except death and taxes.

Benjamin Franklin

Contents

Chapter 1

Introduction

The world is always changing and new technologies are coming to the scene. In general, an adaptation of a technology depends mostly on cost and usefulness.

In terms of Computer Graphics, Virtual Reality and Head-Mounted Displays have recently started slowly taking a larger portion of market share [33]. However, the true revolution in 3D displaying is expected to come with so-called autostereoscopic displays: LCD-like gadgets with the ability to give a user a different image, based on the angle of looking.

The catch is in the fact that autostereoscopic displays require a different representation of an image than a regular 3D or VR application provides. Therefore, every existing 3D application would have to be reprogrammed to support it in order to get the advantage of such display. In combination with a higher price per unit, the adaptation of such displays might be at risk, just as 3D television has partially failed due to missing exclusive content [25].

To help in solving this problem, this thesis is devoted to a (semi-)automated process, which could turn 3D applications to an autostereoscopic-ready format without changing any line in the application's source code.

Although a more precise definition is incoming in the next chapters, for now, let us assume that the idea of conversion to an autostereoscopic display is based on forcing the application to render the content from many different, but close viewpoints.

In Chapter 2, a brief introduction into 3D displays and their properties is given. Next, the definition, formation, and processing of the 3D image are explained. The proper definitions of inputs for 3D displays are given, and the chapter also describes the steps required by the conversion in more detail. The chapter ends with an overview of existing approaches to the conversion of the representation of a regular flat display images to multi-view ones, which are used by 3D displays.

In Chapter 3, the process of so-called hooking is explained. Hooking is a generic programming method used for modifying the behavior of existing SW components by wrapping the components. The technique will be necessary to implement the injector.

In Chapter 4, parts of OpenGL rendering pipeline are examined, and algorithms and ideas are given how to properly support each part of the rendering so that the converted application would render the expected, duplicated image.

In Chapter 5, the structure of the resulting code of this thesis is described.

In Chapter 6, the injector is applied to existing applications and the method's flaws are described and discussed. In addition, performance issues are discussed and suggestions are given for tuning the performance.

Finally, the achievements and possible future challenges are discussed in Chapter 7.

Chapter 2

Introduction to displaying methods

The aim of this chapter is to give the reader an insight into the complex process of forming and displaying images of 3D scenes. Such knowledge is necessary in order to understand the necessity and motivation for creating a new conversion software, whose design is described in the following chapters.

The chapter begins with a description of displays, with emphasis on true 3D displays. Next, a formal definition of storing and manipulating 3D scenes is given in terms of light field, and its use is explained.

Finally, we discuss software that can already be used to convert traditional flat 3D applications to take advantage of 3D displays.

2.1 Overview of 3D displaying

Depth perception of 3D volume in viewers' eyes is caused by *depth cues* [38]. The most important one is *binocular parallax*, which corresponds to *showing two different images of a scene to viewer's eyes, each corresponding to a shifted view of the 3D scene* [32]. Additional minor cues such as *linear perspective*, *occlusion*, or *shadowing* contribute to keep brain focused on the illusion of perceived depth.

In further text, various types of displays are briefly introduced. The classification of 3D displays in this thesis is divided into two major groups, depending on how the image is perceived by user - stereography and autostereoscopy.

At first, *stereograhical displaying* is introduced as it is contemporarily the most prominent way of consumer 3D displaying. These methods require users to wear additional gadgets such as glasses or Head-Mounted Displays and provide two views into a 3D scene from two viewpoints simultaneously.

As an alternative, the second large group of displays named *autostereocopic* attempts to solve many of the problems by taking a different approach to give a viewer a perception of the changing image when the viewer changes his position, without the necessity to wear any additional gadgets such as Head-Mounted Hardware, allowing multiple viewers to perceive the image.

2.1.1 Stereography

Traditionally, the easiest way of causing depth perception is achieved using *stereography*, thus showing two different images to a viewer. This concept forms a broad area of methods

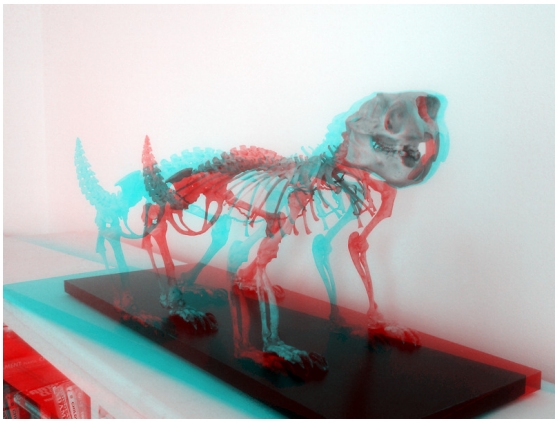



Figure 2.1: An anaglyph of Dino (left) and a Head-Mounted display. Courtesy of Neil Fraser [6] and Hewlett Packard, respectively.

and technologies, which mostly differ in the way how distinctive images are formed in a user's retina and the resulting visual quality.

Historically, the first 3D imaging came in the late 19th century with *stereocopy*. Stereoscopes are plain mechanical devices in which the underlying *stereo photography* is obstructed such that each eye sees its corresponding part. The construction of such a device permits only a single user to perceive the image simultaneously.

Another technology, *anaglyphs*, produces the depth effect by overlaying two images with different colors, which are perceived with specialized color-separating glasses. Typically, red&cyan is used for expressing the left and right image, although any complementary pair of colors could be used. The disadvantage of color-separating glasses is a strongly visible *ghosting effect*, caused by improper separation of colors due to limited display and glasses color response, and a loss of visual quality due to encoding both images as grayscale using shades of two distinctive colors [26]. The advantages include the ability for multiple viewers to perceive the image simultaneously, and the low price of glasses. In addition, anaglyphs can be displayed using common displaying hardware such as CRT, LCD, or plain print, thus making them affordable and widespread.

So far, the presented techniques of displaying do not require any active display such as LCD. The next two technologies utilize 3D glasses while relying on specialized displays. The first displays images in a *time multiplex*, thus by interlacing frames on screen for the left and right eye in time, frame by frame [44, pg. 543]. This requires so-called *active 3D glasses*, which are synchronized with display. They contain a shutter for each of the eyes, which is controlled using the signals provided by the display. They also rely on the *persistence of vision*, an ability of eye, stemming from the limited frequency of the eye as a sensor. Thanks to the property, only a single eye receives the updated image at the same time while the other eye keeps the image from the previous update. Therefore, only a single physical screen is thus required.

The second, *Polarizing displays* produce images that are separated for each of the eyes using polarization filters in 3D glasses, based on the different light polarization of source images. This is achieved by spatially interlacing pixels with different polarizations. The glasses then block the incoming light of the other view, leaving only the correct view to pass to the correct eye [44, pg. 542]. This method employs *passive 3D glasses*, which in comparison to active glasses have plain construction, and thus, are less expensive.

Starting the 21th century, *Head-Mounted Displays* are becoming the most prominent technology of stereography due to their high quality image and immsersing experience. They are constructed so that either each eye has its own display, or portion of a a single display [32]. The price of this technology varies from a few dollars per paper-made frame of lenses [2], which turns a regular smartphone to stereoscopic device, to specialized embedded devices for hundreds of dollars such as Oculus Rift [7], which includes sensors for head position tracking and interactive controllers.

The disadvantages of the aforementioned stereoscopic technologies include problems with keeping sharp images with changing the position of the viewer, causing fatigue and headaches, caused by confusing brain's model of orientation [37], restricting the usable images only for a single user, or requiring additional wearable hardware, used by viewer.

Virtual Reality (VR) and Augmented Reality (AR)

Virtual reality terms an immersive experience for a user of Head-Mounted Displays, in which he perceives the vision of a virtual world as if he was moving and interacting physically in such the world [40]. This is achieved using Head-Mounted Displays, used for providing depth perception, and additional gadgets such as *head-trackers*, which detect the immediate pose of the user's head. In addition, a VR setup can be extended with controllers or glowes, which track the user's interaction with elements of a virtual scene. The rotation of the head together with user's interactions are provided to the underlying rendering system, which then produces an accurate image of the updated world, making the experience more immersive in comparison to the use of traditional stereo displays.

Figure 2.2: An example of VR setup (right). User's vision is completely controlled by displaying stereo image using Head-Mounted Display. To further enhance the experience of immersion, the user can interact via controllers. Pokemon Go (left) is an example of successful AR application. The game provides an overlay of the real world with virtual enemies, placed at real-world coordinates. Courtesy of [7] and `pokemongolive.com`, respectively.

Augmented reality denotes enhanced vision in which a user perceives real-world with additional information in real-time. The term is not limited to any physical display, but rather refers to the ability to extend the reality. For instance, a soldier wearing AR glasses could perceive real-time information about the target's status, displayed as a text overlay next to the target's body [19].

Both VR and AR are illustrated in Figure 2.2.

2.2 Autostereoscopic displays

In contrast to the aforementioned stereoscopic methods, autostereoscopic displays do not require the viewer to wear or use any additional gadgets, and directly separate images into viewer's eyes [30, p. 16].

Although different designs exist, in this thesis we were interested in *single-screen displays* with a specific element, *producing separated images depending on the angle of view.*

The simplest design of such a display provides two views using a *parallax barrier* to display two views by splitting the pixels of the screen into two bitmaps. *Parallax barrier* is a physical mask, which obstructs pixels of the other view, so the actual view seen by a viewer depends on the angle under the display is viewed. This is illustrated in Figure 2.3. Such barrier has the disadvantage of decreasing image brightness and introducing diffraction artifacts due to the wave properties of light.

In practice, different approaches are used in hardware to tackle these problems, such as *micro lenses*, *holographical elements*, or *micropolarizers* [30].

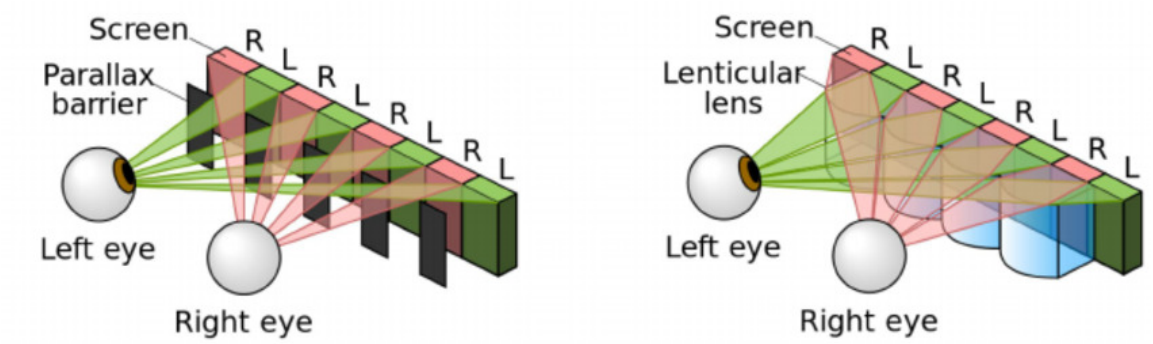


Figure 2.3: Visualization of stereo vision using parallax barrier (left) and lenticular lens (right). The portion of image, perceived at given position of viewer, depends on viewer's angle. Courtesy of C.M.G. Lee[18].

2.2.1 Single-screen autostereoscopic displays

Multiview displays are autostereoscopic displays, allowing multiple viewers to obtain depth perception simultaneously. An illustration of such a display is shown in Figure 2.5.

A common implementation of such a display provides multiple views of the scene in a horizontal direction and a lenticular lens as a form of *parallax generator*. Dividing the screen into interlaced horizontal views, however, decreases the spatial resolution in the horizontal direction. To keep the same image aspect ratio, a screen with narrower pixels would have to be produced to tackle this problem. Instead, so-called *slanted lenticular lens* are used to distribute contribution of horizontal views to the vertical axis. The lens may are then placed in diagonal direction (*slanted*) as shown in Figure 2.4.

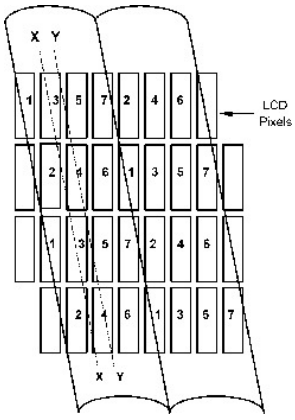


Figure 2.4: Slanted lens spread pixels of horizontal views into vertical direction of LCD. Courtesy of [44].

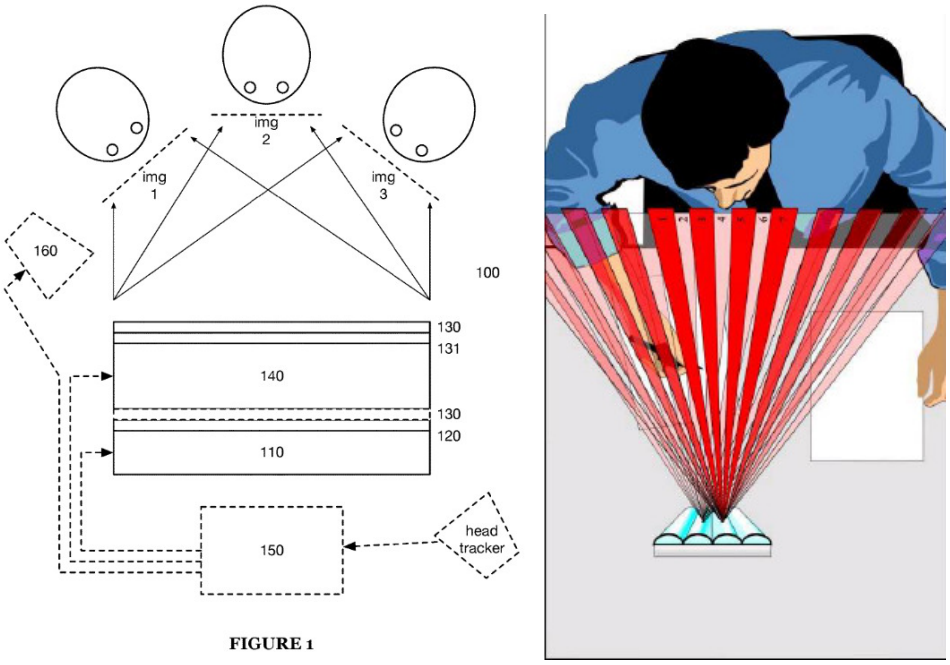


Figure 2.5: A physical overview of patented holographic display [43]. Light coming from light source (110) is reflected into different angles when passing to parallax generator (130). Light source is typically a regular LCD with preprocessed input image in specialized so-called *native display format*.

Figure 2.6: A picture of commercial 3D display by `Looking Glass Inc`. High-index prism together with guiding cues are clearly visible in front of screen's LCD. Courtesy of [29].

2.2.2 Holographic Display by Looking glass Inc.

This section is focused on a particular patented 3D display [43], produced by `Looking Glass Inc.` company, but the principles discussed in this text apply to any device that allows to show multiple views of scenes. An example of a display produced by the company is shown in Figure 2.6.

A generic setup of the display is shown in Figure 2.5. In general, such an autostereoscopic display may consist of the following parts:

1. A light source is typically a flat LCD, showing a preprocessed grid of the scene's views in so-called *native display format*.
2. Parallax generator allows the redirection of incoming light from the specific position of underlying bitmap display to the viewer.
3. High-Index Prism is employed to enhance *off-axis separation*. Refraction of the prism causes the virtual image to lie closer to viewers then the light source, enhancing the illusion of volume.

2.2.3 Volumetric displays

Volumetric displays are autostereoscopic displays that attempts to provide 360 degree vision of the projected 3D world. Typically, the illusion of 3D is achieved by projecting points onto a projection plane using laser. As the plane moves, the brightness of the laser differs as differentent points are projected, corresponding to the 3D point for a given position of plane. Depedining on the design, the plane either rotates or moves vertically. These principles are depicted in Figure 2.7.

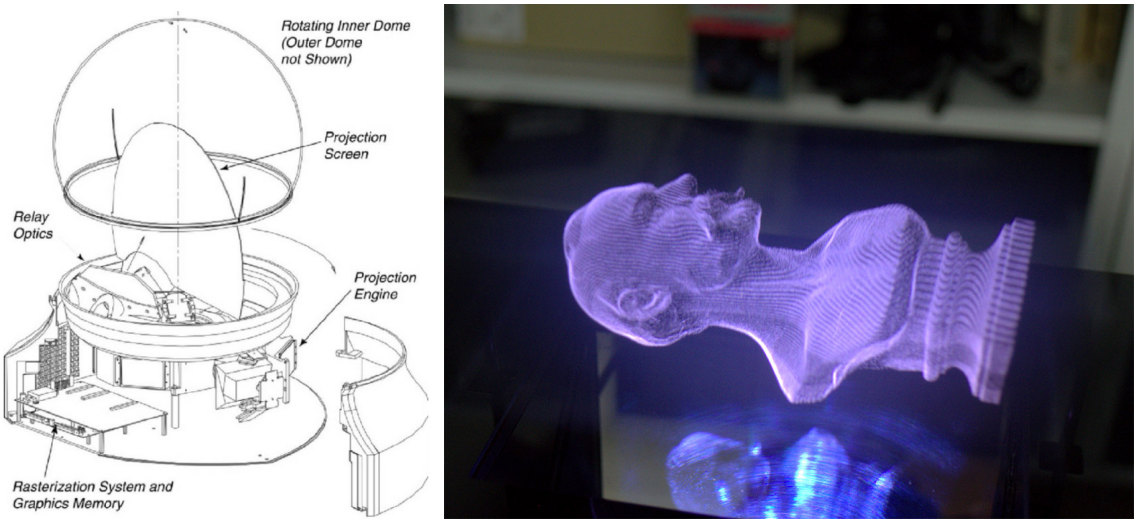


Figure 2.7: Principles of volumetric display (left) - points are projected by optics on planar screen which rotates with high rotates per minute, causing perception of static points. Courtesy of [28]. On right, a display with vertically-moving projection plane and static projector is used. Note visible layers-like artifact along statue's vertical axis due to discrete amount of unique points planes. Courtesy of [13].

Contemporary volumetric displays face challenges in introducing *occlusions*, thus preventing users to see through surface of a solid 3D object, view-dependent light reflection

of the object's surface, and processing of large volumetric data, which defines the resulting 3D model.

These displays are typically employed in medical imaging, mechanical computer-aided design, and military visualization [28].

2.3 Conclusion of 3D displays

In conclusion, all types of presented 3D displays, except for volumetric displays, require images of a scene at input and thus, all of these displays could be used with a system, generating 2 and more views of 3D scenes.

Volumetric displays require a different volumetric representation of 3D scene. Whereas a photography shows a projection of the 3D scene on a plane, the volumetric representation defines presence of volume at different points of space. Typically, this leads to a 3D grid discretisation, in which a finitely large subspace of space (called cell) is represented by a single value.

Although various reconstruction methods for extracting 3D models exist and are later mentioned in Section 2.5.2, converting the projection of 3D scenes to volumetric representation is in general not possible due to missing information about the space outside the view frustum.

2.4 From light fields to Native images of 3D displays

Up to now, the hardware principles of 3D displays were discussed, giving an insight of how different pictures of the scene are separated on a single planar display. This section discusses the formation of these pictures.

2.4.1 Light fields

Definition 2.4.1. *Light field is a function defining the incoming radiance to a given 3D point from a given solid angle. Such a definition is equal to so-called 5D plenoptic function. [34]*

Light fields provide a formal definition of what we see when we take a picture using a camera at a given place in the real world. Provided our camera was ideal (so-called pinhole camera), each pixel would represent the incoming radiance, measured by the camera's sensor. Moreover, each of the camera's pixel uniquely determines angle under the light strikes the sensor. Therefore, a single digital photo represents a discretized subset of the light field at given position. Conversely, if a light field is known for a given space, it is possible to reconstruct any view inside the space in the form of photography.

Figure 2.8: Spherical light field camera rig. Courtesy of [23].

Although a light field is in theory a continuous function, light fields are typically recorded for discrete positions (so-called viewpoints) and a discrete number of samples per angle, termed as *angular resolution*.

One way of capturing the light field is using a so-called *Light-Field Camera*, which contains a microlens layer at a certain posi-

tion in front of the sensor. The resulting image of scene then forms a grid, where each cell corresponds to a different view position, and each subpixel within cell represents a different angle of view. Such a camera, however, requires expensive sensors with high resolution to obtain reasonable angular resolution per view. In addition, as the microlens are tightly positioned in a plane, the camera offers short baselines between each view, thus suiting more for macrophotography than for depicting the light field of distant and large objects.

In practice, *dense camera arrays* are used to capture the light field at different view positions simultaneously. Depending on the application, so-called *camera rigs* are either planar setups of multiple cameras, fixed at well-defined positions, or spherical structures. Such spherical rigs resemble the setups of 360 angle degree cameras, but in comparison, light field rigs are focused on recording from multiple positions. An example is shown in Figure 2.8.

When acquired, the light field can be used for many applications such as synthesizing new views of local scene views with six degrees of freedom (6DoF) using *light field rendering* methods [23], to produce refocused images and videos with arbitrary focus, defined in postproduction, or to reconstruct a 3D depth map from views [46].

Light slabs

In order to the simplify calculations, the light fields are simplified as 4D functions under the assumption of light passing through an unobstructed space. This yields so-called **light slab**, a representation of the 4D light field using points lying on two places - camera and focal plane, respectively. An example of a light slab is shown in Figure 2.9.

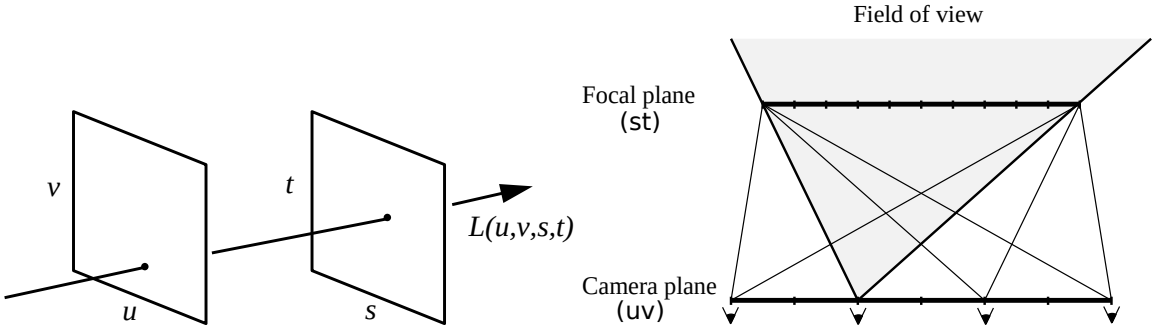


Figure 2.9: Light slab is defined by rays, intersecting two planes, thus yielding 4D function. Left image shows top projection of construction of a slab using multiple views and sheared projection. Courtesy of [34]

The image perceived by the viewer of an autostereoscopic display can also be expressed with a light slabs representation. The viewer is moving his head along the camera plane while watching LCD screen (the focal plane). Because the physical LCD has finite pixels, the number of views which can be uniquely represented, is limited, too. Finite amount of views thus results in a finite amount of discrete positions on the camera plane, under which unique views are perceived at screen. Physical laws, guiding the refraction of light in the lenticual lens, achieve interpolation of source image at positions in between.

Quilt

In practice, light slabs are stored in so-called *quilt*. Quilt is an image format, which defines how the views of a scene are stored inside a single texture. Typically, a quilt is a grid of

views taken at known view positions [8]. An example of a quilt is shown in Figure 2.12. The projection of each view is set so that each view has the same image plane. This is achieved using the *sheared projection*. The view frustum of such projection is also depicted in Figure 2.9.

Quilts can be produced by either resampling light fields, which were previously created by light field cameras, or directly by synthesizing (rendering) 3D scenes. The process of synthesis is shown in Figure 2.10.

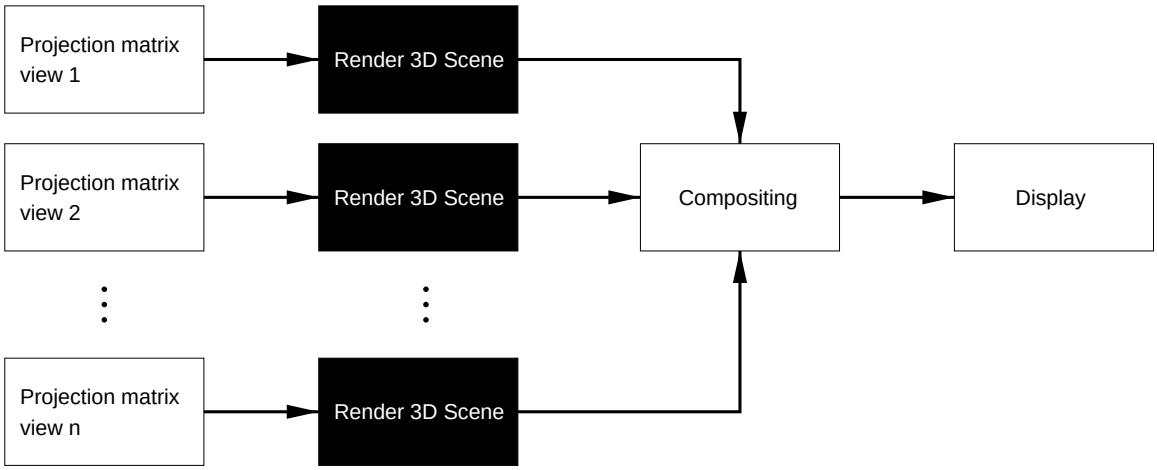


Figure 2.10: In order to display content at 3D display, scene has to rendered from multiple views using different camera transformation matrix, and subsequently composed into output image. Courtesy of [41].

Parameters for creating a Quilt

The quilt format determines how views are placed in memory, but doesnt give any recipe for the precise settings of transformation of each view.

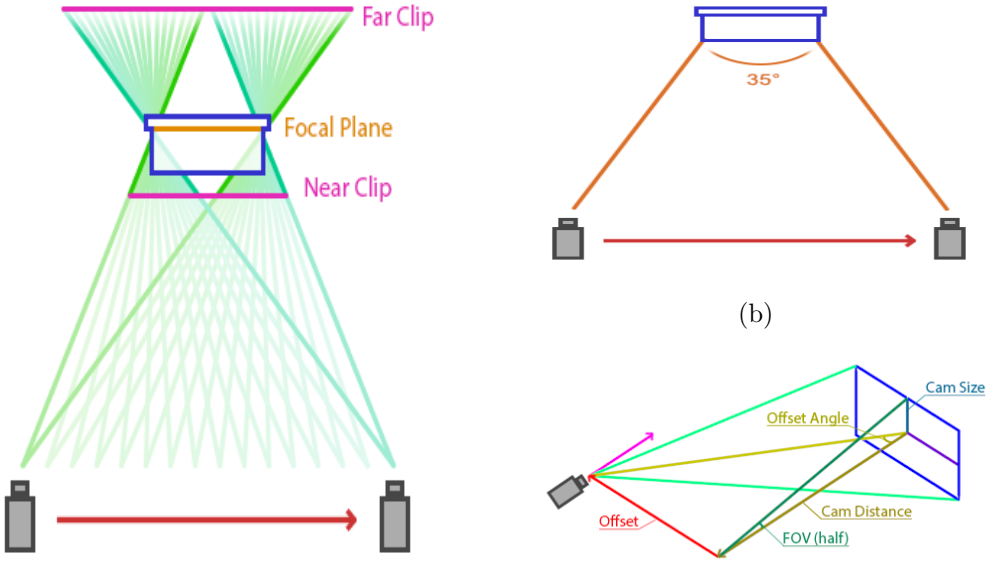
There are two resulting parameters that have to be defined for each camera view: *camera offset* and *shear ratio*. The offset determines how far is the camera placed with respect to the original camera. Shear ratio defines the shear coefficient of the projection matrix which is used to shift the image plane for each camera. In practice, these parameters are only needed for the maximal views, and the rest of the cameras can use parameters, obtained using linear interpolation.

The transformation of each camera depends on the parameters, which stems from the physical restrictions of the display. The most notable parameters are the (vertical) Field of View, camera size, and offset angle.

Field of view defines the angle under which the viewer perceives both the top and bottom sides of display. This depends on the distance of the viewer and the size of camera/display. Offset angle determines the angle under which the leftmost camera views the center of display with respect to axis of display. These parameters are shown in Figure 2.11.

2.4.2 Native image of Looking Glass's LCD

As already mentioned in Section 2.2.2, a typical single-screen autostereoscopic display has a regular flat LCD inside, controlled using common data interfaces such as VGA or HDMI.



(a) (b) (c)

Figure 2.11: Depictions of Looking Glass display setup. (a) shows that apart from traditional near and far clip planes, frustum also contains focal plain, for which zero parallax applies. (b) illustrates maximum suggested angle of frustum cone, under which the user should view the display (c) concludes the parameters. Courtesy of Looking Glass Factory[1]

However, because LGD contains an optical system on top of LCD, which selects which pixels are visible for a given view using the lenticular lens, it is necessary to preprocess the quilts into a *spatially-interlaced multiple view* format, in which the image is divided into groups of pixels, corresponding to the respective lens. In general, displays differ in their parameters, so such mapping depends on multiple display-depending parameters, including *pitch* (angle of the slanted lenticular lens), *scaling* (physical size of lens compared to size of pixel). An example of a remapped quilt is shown in Figure 2.12.

To convert a quilt to a native image, developers can either manually transform the quilt, or a specialized Software Development Kit (SDK) may be provided by the display's producer. For instance, Looking Glass Inc. is providing *HoloPlay Core SDK* [3], which can be integrated by developers and provides the process of identifying the display's parameters, and shaders from transforming quilt to native image.

2.5 Converting 3D applications to multiview displays

In general, there are two solutions for converting 3D applications to utilize real 3D displays:

- **Reprogramming the Application**
  A typical 3D application is running inside so-called *rendering engine*, which provides abstract structures and workflow for creators and artists to create virtual worlds with limited knowledge of Computer Graphics programming.

  Such engines usually decompose a 3D world into elements such as *scenes* and *objects*. Scene is made of visual objects and the view is generated using a specialized type of object that represents a *camera*.

Figure 2.12: An example of transforming views from game The Witcher into Looking Glass display's native LCD output. Note strong diagonal pattern on left, corresponding to slanted lens. Courtesy of Graph@FIT VUT.

In such a case, it is sufficient to reprogram the part of the rendering engine, which renders the scene to an image of scene to repeat this process multiple times with either different cameras or while alternating camera transformations.

The former solution is typically employed when a broad family of applications is powered by same rendering engine. For instance, many of games produced by `Valve` are running on their `Source Engine`.

- **Using additional Conversion Layer**
  Instead of modifying the application, a generic *conversion layer* could be placed between the application and the underlying hardware. The idea is based on the fact that a typical 3D application is providing a local 3D surface of displayed objects to the graphics card, and the final projection and rasterization of the model occurs in hardware.

  The conversion may happen either by re-estimating the surface of the projected scene with notion of depth, or by translating the transformation commands so that the scene would be drawn multiple times with different transformations on behalf of the application.

2.5.1 Existing applications for converting legacy 3D applications

The following section discusses existing software related to enhancing legacy 3D applications to support 3D displaying hardware such as VR head-sets.

ReShade's Depth3D extension

ReShade is an open-source framework, allowing users to introduce post-processing related shaders into an arbitrary application. As the application is renderer-agnostic, it introduces its own shader language and transcompiles user's programs into the native language of the underlying renderer.

Primarily, the framework is used for enhancing the visual quality of legacy 3D applications by adding post-processing effects such as SSAO, bloom, etc. This is achieved by appending additional draw stages at the end of the application's original pipeline. These

stages can then access the application's resources such as the backbuffer with the scene's color and depth information.

Figure 2.13: Binocular stereo view, produced using Depth3D and FallOut4. Presumably original image of scene (left) and reconstructed view of right eye (right). Visible artifacts (red box) are present, even for small camera's disparities.

Thanks to its flexibility, several VR-allowing shaders have been written, notably so-called Depth3D [22]. Such shaders usually make use of depth buffer and *parallax mapping* techniques in the horizontal direction. This allows to obtain pseudo-3D reconstruction with the possibility of filling holes. The aforementioned technique is illustrated in Figure 2.14.

As the implementation of the aforementioned shaders is usually available together with source codes publicly, it should be possible to adapt such piece of software for the needs of holographic display by adding extra views rendered the same way, but with different disparities. On the other hand, VR rendering takes advantage of the short disparity between human eyes.

ReShade is also used by another commercial product *Trinus* [11], which also employs post-processing to convert the video to side-by-side VR output.

vorpX

VorpX [14] is a non-free software for enhancing 3D computer games with the ability to produce output, suitable for VR.

As VorpX is commercially sold, limited information is revealed to users regarding the technique of conversion. The official website mentions terms such as *Geometry Stereo 3D* together with stating that the scene is rendered twice. Additionally, it also provides *Depth Buffer Stereo 3D*, suggesting that a post-processing method is being used for reconstructing geometry from depth buffer.

Due to missing source codes and limited functionality to two views, the outputs of this software can not be used for holographic display.

As the official website provides a curated list of supported games [10], this suggests that Geometry Stereo 3D mode does not work automatically, but instead requires manual tweaking.

Conclusion of existing SW

Most of the existing applications employ reconstruction approaches to display two views of the scene. Such an approach can satisfy needs of VR, but would cause massive vi-

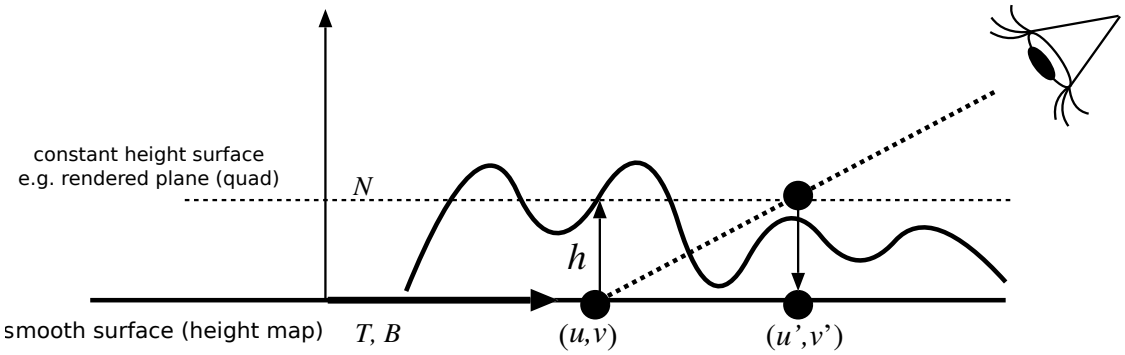


Figure 2.14: Parallax mapping is rendering method, typically used for compute displacement and normals of simplified surfaces, which were precomputed from smoother representation of geometry. for a given height map, the intersection of camera ray and surface is calculated. The variants of this method use different number of steps and approaches to find the intersection, and e.g. steep paralax mapping use fixed count of iterations (e.g. 64) to sample the height map. In case of stereo conversion, this method is applied as a post-processing pass, which takes scene's depth buffer as a height map, and shifts the UV coordinates of scene's colour texture by tracing the height map. This approach is approximative and leads to defects at edges of objects due to depth map discontinuity. Courtesy of [39].

sual artifacts when pushing the disparity to longer distances, due to missing information. Whereas typical human has 50-60mm long baseline [38] between eyes, users of autostereoscopic displays may watch the display from a distance comparable to LCD, resulting in a large horizontal baseline.

In addition, the reconstruction method used, resembling steep parallax mapping, is bounded by the number of iterations, and the distance of projection of a single point into two cameras depends both on depth and disparity, resulting in an increased number of steps required for the algorithm to find the correct position.

2.5.2 Alternative methods for conversion to autostereoscopic displays

In the following section, a closer look is taken at alternative methods, which are used in similar fields, but may not be fit to converting applications as they are.

Stereo to Multiview Conversion

Alternatively, numerous methods exist in the field of conversion from *stereo to multiview conversion* such as [31]. These methods estimate the depth map from two views using disparity and subsequently reproject or warp the input images based on reconstructed 3D surface.

The state-of-the-art method [31] in this field is able to do novel image interpolation, extrapolation beyond original views, inter-view aliasing and other complex operations. It achieves almost perfect results for stereomovies. The principle of the paper is explain using Figure 2.15.

The only lack of this method is the solving of obstructions, e.g. if a closer object obstructs object behind. In this situation, information about the space behind the obstacle

is missing, and none of the approximate methods will ever be able to correctly predict what lays behind.

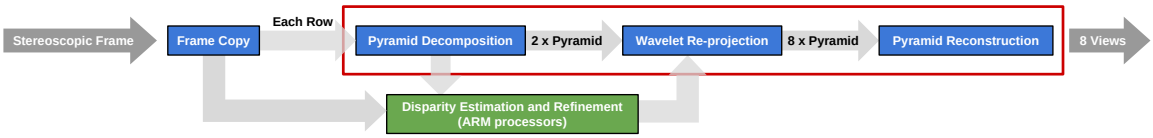


Figure 2.15: A pipeline of the state-of-the-art stereo-to-multiview conversion method. For two input images, the image is decomposed to wavelets, which are correlated, and this way, disparity for each wavelet pair is obtain. By reprojecting shifted wavelets for each pixel according to disparity, novel views are obtained. Use of wavelet suggests interpolation based on local information of each pixel of input images. However, such information can not be used for extrapolating heavily-obstructed scenes satisfyingly. Courtesy of [31].

Structure from Motion and SLAM

In addition to the solutions that synthesize new views from RGBD images as introduced above, there is a category of methods dedicated to the conversion of series of single-view images (such as photos).

Structure from motion (SfM) reconstructs 3D models by estimating the position of each pixels in 3D space using *epipolar geometry* and (possible) known location of cameras with respect to each other using *epipolar geometry*. Traditionally, variations of this method are used for reconstruction of 3D models of the real world structures such as sculptures or buildings to acquire digital models, for instance, for computer games [21]. The use of SfM is illustrated in Figure 2.16.

A superset of this method is **Simultaneous Localization and Mapping**, which in addition also determines position of an camera on-line. This is achieved by matching points of subsequent views, and computing relative changes of the camera transformation or by integrating camera movements from IMU.

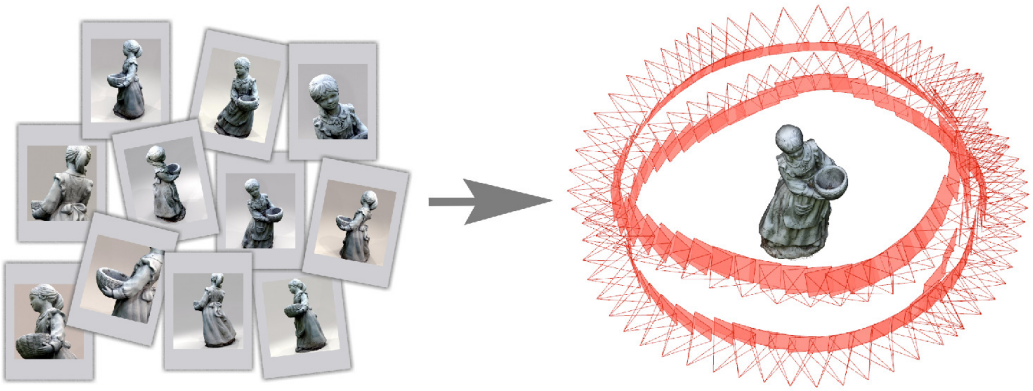


Figure 2.16: Structure from Motion attempts to identify unique positions of views, and then utilize this piece of information to reproject pixels, resulting into a point-cloud representation of model. Courtesy of [21].

Implementations of both methods use commonly *point clouds* for representation of reconstructed 3D points and various techniques for fusing reconstructed points of multiple

views. Resulting point clouds can than be converted to any suitable representation for rendering.

The major disadvantages of these methods include performance requirements, making them until recently the offline-based methods. In addition, the fusion of geometry works under assumption of static geometry and consistency of topology. The fusion of models breaks if the geometry of environment change over time. Multiple methods, attempting to solve this, exists, but mostly focus on deformation in time rather than on complete change of geometry in terms of entering or leaving objects.

In conclusion, both methods could be used to convert output of typical single-view 3D applications into multi view format needed by autostereoscopic display by estimating 3D scene in the form of a point cloud, tracking position of the camera in the reconstructed scene, and reprojecting the point cloud for all views of quilt in each frame.

However, mixing the output of the methods with the application's HUD would be complex to handle in general. In addition, the reconstructed scene can only be rendered with the full quality after it was reconstructed, thus after enough views of the scene were collected, resulting to holes in the geometry during the initial phase of algorithm.

2.6 Conclusion

In this chapter, different technologies of 3D displaying were presented. Notably, Light-field displays were introduced together with the correspoding native image format and the process of forming 3D image for a 3D display.

Additionally, the existing methods for converting regular single view applications were discussed with advantages and disadvantages.

As there is not any software for rendering ground truth output without artifacts, the process of designing such software is discussed in the next chapters.

Chapter 3

Theory of application hooking

In the previous chapter, several existing software applications were introduced, which can manipulate the execution flow of existing programs and thus achieve the desired outputs. This chapter analysis how similar programs can be created and what are the options for monitoring, altering or stopping the functions of existing programs.

3.1 Motivation for hooking

Usually, when trying to adapt an existing software solution to different conditions, source code changes are required. However, lots of legacy applications either hide their source code due to licensing and monetization, or the code has already been lost since the release. In addition, introducing new features at the source code level into most of the software can not be generalized and would require manual changes.

Luckily, most of the software is not reinventing the wheel and instead, they rely on existing components. For instance, in terms of computer graphics, most of the graphical applications use standardized libraries to form the image of a desired scene.

Thus, instead of changing the source code of the application, we could intercept communication between the application and the underlying libraries. Such a process is discussed in this chapter.

3.2 Definition of hooking

In the literature, hooking is defined as „*the interception of specific functions or system calls to monitor and/or alter the execution of the specified call.*“ [36]

As an example, consider Figure 3.1 which depicts a scenario in which an application uses one of functions of *Application Programming Interface* (API), provided by the example library. Originally, such an API call (named *handler*) would be served by the library and the process control would immediately return to the application, called *caller* in terms of hooking.

Instead, the call to API's function (*target*) is redirected to another handler called *detour*, which can in turn change the parameters, change its own internal state, discard the API call or anything else.

In software engineering, hooking techniques are commonly used by software that extend other software capabilities, such as various visual overlays (for instance, Steam Overlay [9]), but also by malware and malicious codes such as keyloggers [45], which try to take control

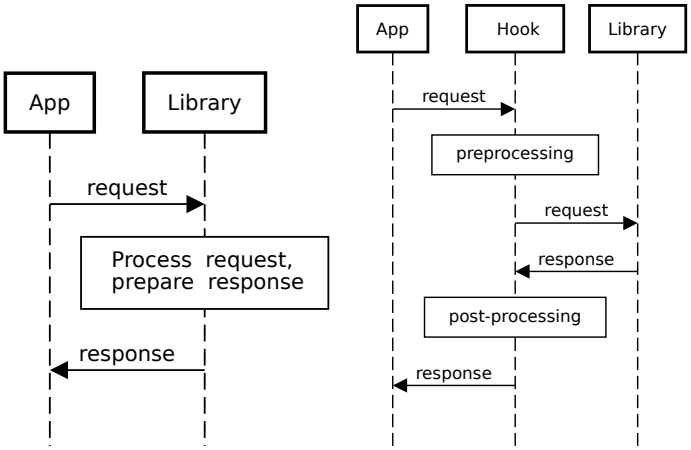


Figure 3.1: A comparison of two scenarios. In the first one, API call is directly sent to library and so is the response. The second scenario shows how the propagation of message is changed when the API call is hooked.

of hosting machine by replacing system's functionality. On the other hand, hooking can also be used in white-hat applications such as antimalvers and antispies, and in system tracing utilities such as *strace* [27] or *valgrind*.

The following section will focus only on the two most commonly used hooking approaches - static and dynamic hooking.

3.2.1 Static hooking - preloading a shared library

Static hooking alters the flow of execution during loading of the application, leading to a permanent change of flow during the whole run of the program, and is almost solely achieved by exploiting the dynamic loader.

Dynamic loader [17] is a part of the operating system, whose purpose is to allow loading parts of code on demand. Instead of putting implementations of all functions inside an executable, applications can be linked to dynamic libraries, which promise that will provide the functionality referred by the application, in terms of functions. When application is compiled, the required functions are declared to be imported and linkage is declared to be resolved during load-time.

A *shared library* is a collection of hidden and visible methods. Visible methods are said to be *exported*. When dynamic loader loads a shared library into process, the application can query addresses of exported functions and use such addresses to call the functions. In case of Linux, querying is achieved by using `dlopen/dlsym`.

During the load-time of executables, the system dynamic loader tries to link the references to unknown (external) functions, required by executable. This is achieved by sequential loading of the shared libraries, specified in the executable's file, and looking for missing symbols.

Static hooking relies on the specific behavior of the dynamic loader which can be enforced by specifying an environmental variable `LD_PRELOAD`. When this variable is set, the dynamic loader loads at first the libraries specified in the variable, and then the rest of dependencies as usual [17]. Hooking is achieved by defining an exported function in preloaded library with the same name as the target function.

As an example, suppose that we would like to trace the usage of `malloc` in application. We could create a library that would export its own `malloc` handler. By pre-loading such library, application's calls to `malloc` would be trapped by the handler.

In addition to hooking, it is also desired to be able to delegate the call to the original function as well. This can be achieved by using a special flag `RTLD_NEXT` when calling `dlsym`, which forces the loader to search for the symbol in the modules, loaded after the caller's module [16].

Symbol look-up chaining

This results in a possibility of *chaining symbol look-up*. Chaining is a method in which multiple shared libraries are preloaded, all declaring the same export function, when each of the function definition delagates the call to the library, preloaded after the current library.

For instance, consider again an application, which use `malloc`, the allocation tracer utility and a simple utility that wraps the allocation by allocating a huge buffer using original `malloc`, and assigns a subset of this buffer to calls. Each of two utilities thus defines and exports their own `malloc`, and uses chaining to find the next handler. When the application is started with both libraries defined in `LD_PRELOAD`, the resulting system depends on the order of library names in the variable. In first scenario, the tracer will catch the original calls to `malloc`. This is illustrated in Figure 3.2. In the second one, the tracer only sees the optimized memory-pool allocations of huge memory chunks, produced by the utility.

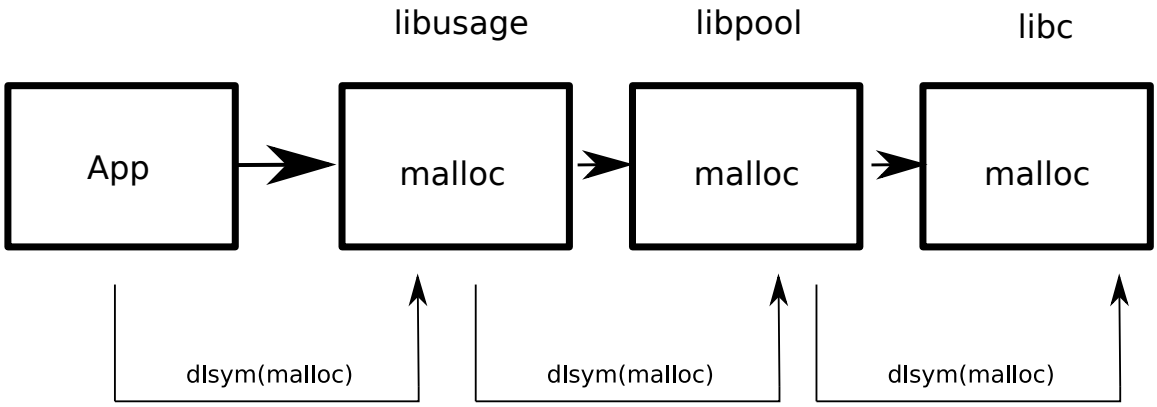


Figure 3.2: An example of Symbol look-up chaining. When the process is started with multiple dynamic libraries, which export the same method, the result of symbol resolution depends on the order of library listing. For instance, consider two libraries, both exporting malloc. The original application will receive the address, defined by the one, which comes first the list. The first library could be a wrapper (e.g. profiler) and it would also want to use the original malloc. In case of two libraries, call to *dlsym* with *RTDL_NEXT* flag will chain the address of the next library. Finally, the chain ends with the address of the last (and thus original) library, which is *libc*.

3.2.2 Dynamic hooking - Runtime code patching

Contrary to static hooking, *dynamic hooking* occurs during the program's run-time and can be used generally to hook any program's functions, not just the exported ones.

The generic idea is that the beginning of the target function, so-called *prolog*, can be replaced (patched) with instructions that call the hook's callback handler. However, as the original function is now permanently redirected to the callback, the handler can not simply call it in order to delegate the default call as it would lead to *infinite recursion*.

This can be solved by *temporarily restoring the original instructions* when calling original function from handler. On the other hand, this solution will cause problems if the hooked method is called by *multiple threads* simultaneously as code patching is not an atomic operation. Even if the code was replaced atomically, there isn't any mechanism to prevent the other threads ending up in the original function instead of being handled by the hook's handler.

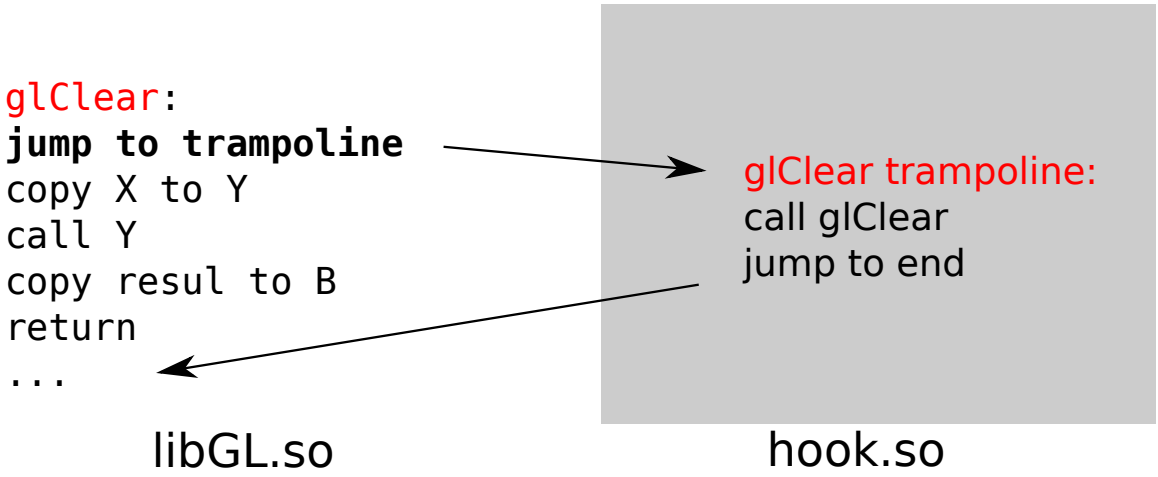


Figure 3.3: Runtime code patching works by replacing first instructions of original function with jump to specialized area (so-called trampoline), that redirects call to hooker-specified handler.

Alternatively, another approach is to create a new code area called *trampoline*, which contains the original instructions that were previously replaced by the jumping sequence [24]. The hook can then call the original function by calling the address of trampoline, which ultimately jumps into the original function's code area right after the patched sequence.

Compared to static hooking, dynamic hooking is more complex to handle. In order to use it, the following problems need to be tackled:

- **platform-dependency**
  The instruction set used by CPU differs per CPU's architecture, requiring a different patch to be generated for each of platform. This problem can be eased by using specialized libraries which provide hook utilities for given instruction set such as *subhook*[1].

- **memory protection**
  Most of the operating systems protect the program's code areas, making them read-only. Luckily, the most common platforms such as x86 allow changing of user-space protection without necessity of elevated administrator rights.

  However, as code patching is commonly used by malware, operating systems may detect and prevent such behavior. For instance, Linux kernels are commonly deployed

[1] https://github.com/Zeex/subhook

with an extension called *SELinux* [42], which detects *program's execution on heap*, which happens when trampoline is called. The only feasible option to prevent such interference is allow such policy by turning on `allow_execheap` flag.

- **lazy evaluation of dynamic loader**
  Typically, dynamic loaders load the imported functions in lazy-evaluation manner, termed as *lazy binding* [20]. After load-time, the address of imported function points to a sequence of instructions, which triggers import table look-up when called for the first time, and only then the final address of function is stored.

  Hooking such a lazy sequence prior to the first call would result into overwriting the look-up mechanism and ultimately leading to crash.

3.3 Hooking OpenGL's API

After giving an introduction to different approaches for hooking, this section gives more specific information on OpenGL API and its implementation.

3.3.1 Overview of OpenGL's API

The implementation of OpenGL is GPU vendor-specific and consists of a set of shared libraries. In the following list, the most relevant libraries are mentioned:

- libGL.so - contains `glABC` functions
- libGLX.so - contains `glXABC` functions - frame swapping, keyboard input messages

The library `libGL` contains all regular functions with prefix `gl`. As OpenGL does not manage its own context, functions for swapping front/back buffers are not available in `libGL`. Context-management is platform-specific. For instance, on Linux distributions, OpenGL is usually available as an extension of X11 graphical server called `GLX`, which has its own functions exported in `libGLX.so`.

Acquiring the API method address

OpenGL API is version-dependent in such a way that newer versions are supersets of older ones. Due to that, applications are never statically linked to `libGL`, but instead the availability of API functions is checked in run-time.

This is achieved either by using platform-dependent method such as `dlsym` (Unix) or `getProcAddress` (Windows). By calling such a function, the caller retrieves either the user-space address of function, or a null pointer. In addition, a special function `glXGetProcAddress` is provided to allow getting methods of both libraries [4].

In practice, the applications are advised to use function loader libraries such as *GLEW*, *GLAD* or *libepoxy*. However, these libraries use the aforementioned functions internally.

Generic algorithm of using OpenGL API in applications

Concluding from the facts given above, a typical OpenGL uses the following algorithm:

1. The application's execute file is loaded into memory.

2. Dynamic loader makes OpenGL's `libGL` and `libGLX` available.
3. Application use `dlopen` and `dlsym` to find `glXGetProcAddress`.
4. Application calls `glXGetProcAddress` with the names of all functions that is interested in to obtain the addresses.
5. Application calls OpenGL API method by calling the obtained address.

3.3.2 Attacking the loading chain

As applications use `dynamic loader` to load OpenGL functionality from `libGL`, there are a few different approaches how to take control of API.

Replacing `libgl` with proxy shared library

This approach follows the idea of defining each exported symbol in a proxy library that was exported in the original `libGL`, and then, when such a handler is called, use internally the mechanism of the `dynamic loader` to load the original library, and reroute the call to the original function. An example is illustrated in Figure 3.4.

The major disadvantage lies in the necessity to define every function of OpenGL API in order to make sure that any application will find all of its required API methods. As OpenGL API contains circa 3000 distinctive API methods [2], such a proxy library can take up several megabytes, causing slower load-time, and possibly performance decrease due to another level of indirection.

This method may be commonly employed by API tracers such as *apitrace*[3], which must detect all calls to OpenGL function in either case.

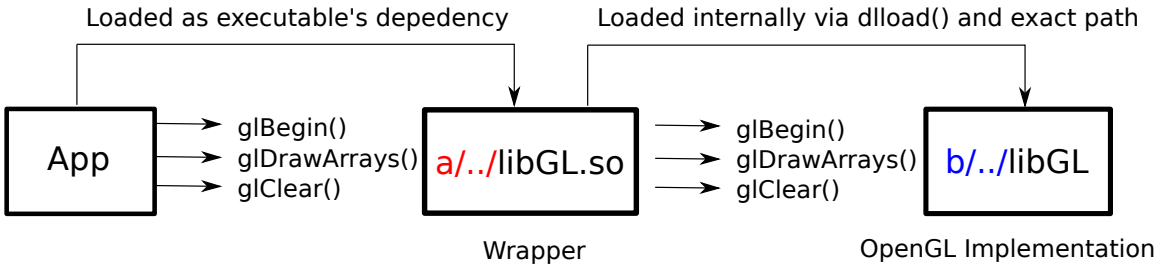


Figure 3.4: Libraries can be wrapped by implementing new library, which defines the same symbol exports and use dynamic loader to delegate calls to the original implementation internally. The new library is then placed in search path, defined by `LD_LIBRARY_PATH`, and internally loads and use the original library to delegate default calls.

Replacing `glXGetProcAddress` with the proxy method

This method requires defining the symbol `glXGetProcAddress` in hook's library and preloading the library.

As many applications use `glXGetProcAddress` to query available addresses, the function could be replaced with a proxy handler that would redirect the functions that we are interested in into our code.

[2] Roughly estimated using `gl.h` and `glext.h`
[3] https://github.com/apitrace/apitrace

Contrary to the previous method, it is possible to **redirect only the necessary methods needed by hook**, instead of redirecting and handling complete API.

The major drawback of this approach is that the function may not be statically linked, and an application can use a combination of `dlopen/dlsym` instead, which bypasses our handler.

Replacing `dlopen/dlsym` with the proxy methods

The most generic approach is to take control over the dynamic loader. By hooking dlopen/dlsym, it is possible to redirect the address of `glXGetProcAddress` to custom handler, which returns the address of callbacks instead of the real OpenGL API mehods. In addition, many other useful functions from both `libGL` or `libGLX` can be simply hooked, which may be queried directly using platform-specific loader.

This method, however, does not handle statically linked symbols (symbols, which are declared to be unkwnown at the time of linkage), and may thus be combined with exporting functions in the hook's shared library.

3.4 Conclusion

This chapter has described two most common approaches to hooking existing applications without knowledge of their source code as well as the details of OpenGL hooking.

The approaches differs in flexibility (static vs dynamic), amount of code needed for minimal product (proxy library vs library injection).

In practice, the method which replaces *dlopen/dlsym* is prefered as it allows to hook or redefine functions of multiple libraries that is not possible when replacing a library file.

Chapter 4

Designing the conversion layer

This chapter analyses the parts of OpenGL API and pipeline, which are used for rendering, and suggestions are given for altering the existing pipeline to such one, which preserves the intention of the creators of the original application, yet provides a multiview output.

4.1 Ultimate goal of conversion

The ultimate goal of the convertor is to provide *a generic conversion of any OpenGL 3D application*. In reality, the state space of all possible OpenGL applications is large, and thus, creating an application, which would correctly handle all possible combinations is hard, if not impossible.

Instead, the development of the layer was planned in an *incremental fashion*. Due to historical reasons, OpenGL evolved through multiple stages of `API` calls, each of them extending the previous ones. Therefore, by incrementally supporting higher stages of API, a larger subset of applications will be supported.

In addition to API evolution, applications themselves may use different *displaying methods*, which need to be correctly handled by the convertor.

This chapter therefore summarizes the different approaches targeted by the resulting application.

The ultimate goal of the convertor should be to display the output of an underlying single-view application in multiview fashion. The role and position of such a convertor is illustrated in Figure 4.1.

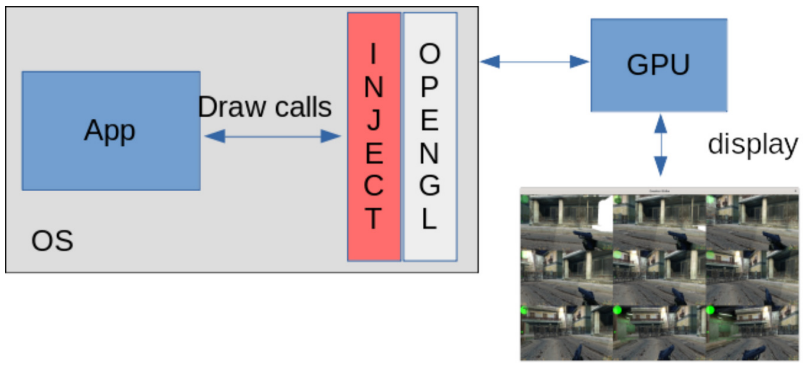


Figure 4.1: In the picture, the role of the conversion layer can be perceived. It is a wrapper over exposed OpenGL API, which internally translates API calls to different API calls.

From the visual point of view, a generic usage of OpenGL for rendering can be decomposed into *frames*, and each frame is made of *draw calls*. Each draw call can draw to a different output buffer, called Frame Buffer Object (FBO). The default FBO is back-buffer.

The fundamental idea of this thesis is that a description of the scene to be rendered is provided by an application to a graphics card in such a form which *preserves the depth of the scene*.

For example, when the application dispatches a draw call of a model of bunny, the model is typically present in a form of 3D mesh at GPU side, and only then, the GPU pipeline transforms the 3D model to a 2D projection. Thus, by hooking and altering the transformation provided by the application in draw calls, the models can be rendered from slightly different views, corresponding to the views of quilt.

Duplicating the draw calls

In order to duplicate views, draw calls have to be recorded and dispatched repeatedly with corresponding settings for each virtual view. Draw calls in a single frame could be recorded all at once, and subsequently dispatched multiple times.

However, such an approach would require a complex state-command tracking of OpenGL API, and could potentially break whenever the application changes OpenGL's object lifetime during the frame. For instance, when unloading a part of 3D scene, the application could delete shaders in the middle of frame.

In addition, some draw calls are products of parameters, calculated in real-time, and such recording would duplicate potentially hardware-demanding calculations (typically implemented via Compute Shaders).

Instead, the convertor intercepts and duplicates **each draw call** separately, wrapping the resulting calls with the required state change calls. This introduces additional overhead, but simplifies the implementation. The duplication of draw calls is illustrated in Figure 4.2.

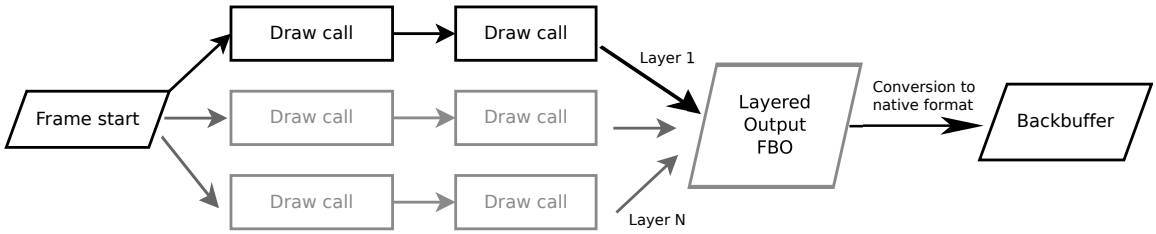


Figure 4.2: Generic approach to extending: during each frame, application's draw calls are intercepted, and internally duplicated for each corresponding virtual view. In this picture, we can see two draw calls being dispatched three times to render 3 views.

4.2 Modifying fixed-pipeline rendering

So-called *fixed-pipeline rendering* is the first and the most elementary part of OpenGL pipeline. Although most of the provided API calls by this stage are being already either obsolete or deprecated due to a change of rendering paradigm, the stage is widely used by legacy applications.

In general, fixed-pipeline applications rely on setting the desired state of the state machine prior to each draw call. This includes material parameters, but more importantly, the transformation of a mesh onto screen.

The transformation consists of separately stored matrices, which are then internally multiplied during the draw call, resulting in a typical *ModelViewProjection matrix*. In addition, each part of transformation has its own **matrix stack**, allowing applications to work more comfortably with object hierarchies. By pushing a matrix to the stack, the current matrix at the top is multiplied with the pushed one, which is useful for applications which store the scene in so-called *scene graph*, in which objects can be relative to the other ones.

The type of matrix being manipulated is then referred as *mode*. For instance, a *projection mode* manipulates the projection matrix stack, and *model-view* mode manipulates model-view matrix.

Technically, applications set the state by choosing the matrix mode via `glMatrixMode` call, and then either by overwriting or pushing matrix at the top of the stack using `glLoadMatrix` and `glPushMatrix`, respectively. Such state change is then done each time a new mesh is drawn (model-view mode) or each time a camera is switched (the projection mode). If the scene only contains a single camera view and camera parameters such as *field of view* does not change over time, projection matrix may be set only once per whole application life time.

4.2.1 Inserting projection into transformation stack

As already mentioned, the transformation matrix, used to process vertices of geometry, is made of submatrices as noted in Equation 4.1.

$$\begin{aligned} Transformation &= Projection \times ModelView \\ V_{cameraspace} &= Transformation \times V_{mesh} \end{aligned} \tag{4.1}$$

As `glMultiplyMatrix` multiplies the matrix at the top of the stack from the left, the typical application starts the frame with *world-to-cameraspace* transformation at the top of model-view stack. The projection matrix stack is then solely used for projection.

Provided that the application is following the pipeline mentioned above, the projection stack always contains a matrix in the form of perspective or orthogonal projection. As we expect a single-view application at the input of the convertor, we can safely assume that the projection matrix is also *symmetrical*, which greatly simplifies the form of matrix.

In order to draw the mesh from the perspective of the other view, two steps are necessary: *horizontal translation* and *optical axis shift* (shear).

The translation is implemented by inserting a translation matrix with a translation along X between the projection and model-view matrices. In terms of fixed-pipeline, this is equal to multiplying the model-view matrix from right. However, as multiplication is limited to

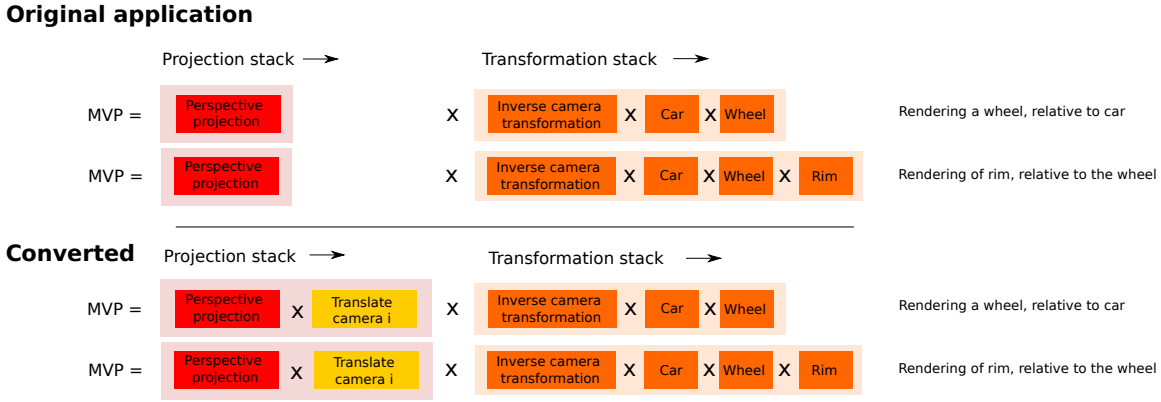


Figure 4.3: In fixed-pipeline OpenGL, applications set the transformation matrix of a draw call using built-in stacks: the transformation (model-view) stack and the projection stack. In order to convert the draw call, a translation matrix is needed to be inserted between the projection and camera translation to translate resulting vertex from the original camera to one of quilt's cameras. As supported operations only provide pushing to the stack from right, the projection matrix can be multiplied with the translation by pusing the translation matrix to the projection stack.

multiplying from the left, the projection matrix is multiplied from the left instead, resulting in the same expression. This is illustrated in Figure 4.3.

Algorithm 1: Intercepted fixed-pipeline draw call

Result: Draw the mesh into multiple views of quilt while implementing intercepted draw call's body

```
oldProj := get current projection at the top of stack ;
glMatrixMode(GL_PROJECTION);
for view in quilt do
    glViewport(view's rectangle);
    newProj := oldProj;
    newProj[2][0] := shear;
    Xtranslation:= calculate translation along X matrix for a given view;
    newProj := newProj * Xtranslation;
    glLoadMatrix(newProj);
    drawCall();
end
glLoadMatrix(oldProj);
glViewport(original);
```

The shift of the optical center is achieved by modifying the projection matrix. In case of perspective projection, the matrix is altered to shift the optical axis of the axis of the original camera. The shift is calculated using Equation 4.2. *Off-axis distance* determines how far is the camera offset from the original view. *Shift multiplier* and *centerOfField* are user-defined variables, determining how far on X axis the camera should be offset and how far is the center point of all views in front of the original camera, respectively. The position of center point is relative to near-plane ($centerOfField = 0$) and far-plane ($centerOfField = 1$) distance.

$$TotalShift = offAxisDistance * shiftMultiplier * (1.0 - centerOfField) \quad (4.2)$$

In the case of orthogonal projection, the shift is not set. Technically, the orthogonal projection matrices can be distinguished from each other by looking at the diagonal of the matrix.

Final approach to duplicate fixed-pipeline draw calls

In conclusion, the convertor can investigate the state of OpenGL's matrix stack either by tracking `glMatrixMode`, `glLoadMatrix/glMultiplyMatrix`, or by querying the value of the current matrix on the top of each corresponding stack.

Subsequently, on each draw call, the draw call is replicated by changing the transformation for each corresponding virtual view by forming a new projection matrix, which results from multiplying the original projection from the left by translation, and the adding shift term to the original projection. This process is expressed by Algorithm 1.

When rendering the view, `glViewport` is used to split the viewport into a grid of views, corresponding to quilt. However, in the final design of convertor, this is not used, and the rendering of fixed-pipeline is unified with programmable pipeline by rendering to Layered Frame Buffer Objects, which will be defined later in Section 4.4.

4.2.2 Replicating geometry using glCallList

As already mentioned, a typical fixed-pipeline application uploads the geometry by successive sequential API calls, vertex by vertex. In theory, in order to replicate such mesh, one would have to record the sequence of geometry uploads, and replicate the sequence each time a different view is rendered.

Fortunately, such mechanism is already implemented in OpenGL under the name `glCallList`. A *call list* (also referred as *display list*) [12, p. 307] is a recorded sequence of OpenGL API calls, supporting a subset of OpenGL API, mostly including geometry, transformation and shading API calls.

The mechanism is used by the convertor for each mesh. When a `glBegin` is called by the application, the convertor inserts call to `glNewList`, which forces OpenGL to start recording. All successive calls are then recorded internally by OpenGL. The convertor waits until `glEnd` is called, and inserts `glEndList`, which finishes the recording of list.

Subsequently, the convertor can use the list to duplicate draw call. In general, a call list can contain multiple draw calls and transformation changes. However, as the convertor creates a list for each pair of `glBegin/glEnd`, the transformation is not changed in-between, and thus, the convertor can change the transformation prior to applying a call list without the risk of interfering with recorded API calls. This is illustrated in Figure 4.4.

In conclusion, the convertor records a list for each draw pair, which is not propagated to underlying OpenGL driver, and subsequently, dispatches multiple list calls, each time with a different transformation and different viewport.

4.3 Inspecting programmable pipeline

The second broad subset of OpenGL applications are applications, that use so-called *programmable pipeline*. Such a pipeline resembles the fixed one with usage of fixed-functionality

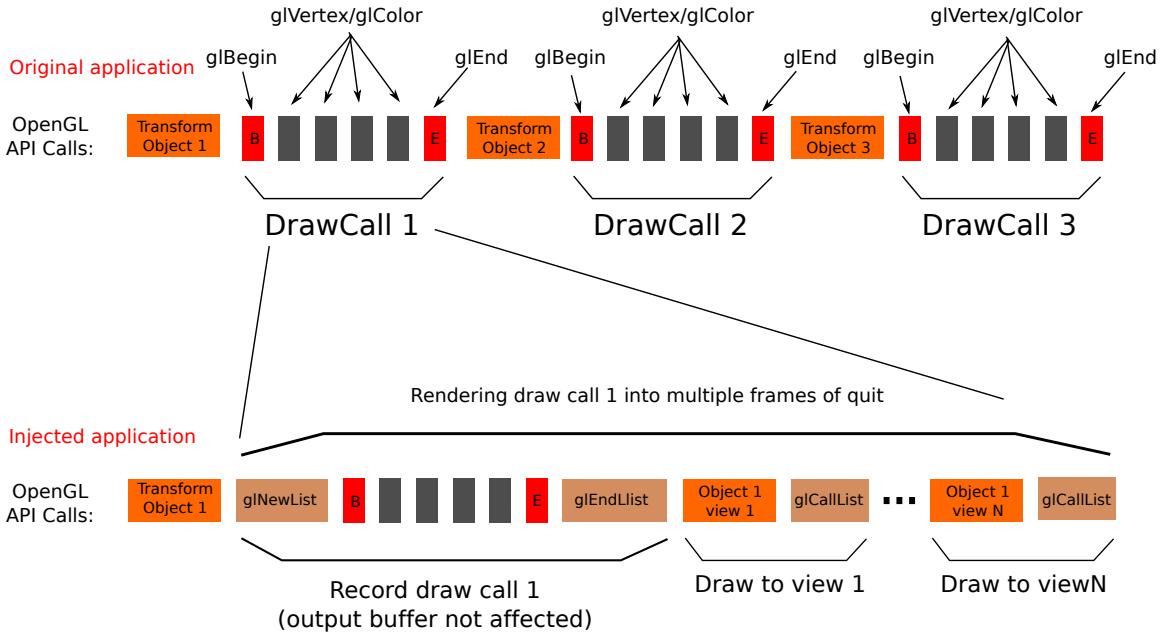


Figure 4.4: Illustration of recording and dispatching a call list. Each draw call is detected and wrapped into `glNewList/glEndlist` pair, recording a new call list. Subsequently, call list with draw call is dispatched for each virtual view with set transformation.

elements such as *rasterizer* or *texture sampler* to obtain and process data. However, certain parts of pipeline, which were previously fixed-functioned and only controlled by parameterization, are now programmable using user-provided programs, called *shaders*. The process of transition to the programmable pipeline started with the introduction of *Vertex* and *Fragment shaders*.

Vertex shader is a program, which receives *vertex data* as it is stored into a buffer by the application. The primary function of the shader is to compute the resulting coordinates of a vertex in *clip-space coordinates*. In addition, `VS` can optionally precompute and pass information such as normals, texturing metadata etc. into further parts of pipeline.

Fragment shader operates over the cells of virtual grid, spreading the screen - *fragments*. By collecting and joining fragments, resulting pixel color is determined. Fragment shader thus receives the optional parts of stages before and computes the resulting RGBA color. This allows programmers to implement various effects and drawing methods independently from the supported hardware. In the simplest scenario, the resulting color is stored in attached FBO (commonly back-buffer). In practice, FS may write into several output buffers using *layered rendering*, which is discussed later in this chapter.

With FS and VS available, programmers define their own transformations and algorithms using a specialized shader programming language `GLSL`, which is a procedural, C-like programming language with built-in types and functions for vector mathematics, texture sampling etc.

Shaders are compiled independently, but they must be linked together to form a pipeline afterwards. Such a collection of linked shaders form so-called *Program*. A typical workflow for programmers is to create a single program for each material/rendering style.

Shaders can communicate using *In/Out variables*, which provide a way to send data down the pipeline to the next stage. In order to parameterize shaders, *Shader Uniforms* are available as a form of global variables to pass data from CPU to shader program.

Goal of programmable pipeline conversion

In general, to convert a draw call which utilizes programmable pipeline the part of pipeline, which computes resulting position of vertices in the original's camera clip space, must be encapsuled and extended so that the draw call would transform the vertex to clip space of chosen camera of quilt. This is illustrated in Figure 4.5.

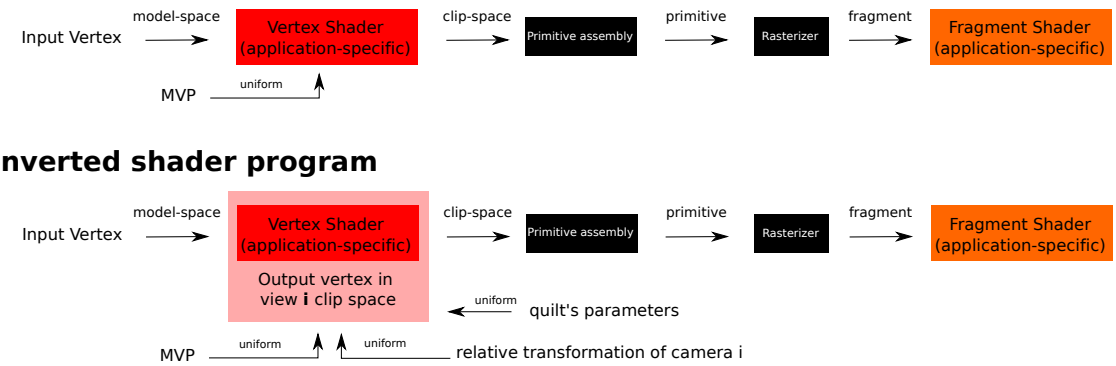


Figure 4.5: In programmable OpenGL rendering, the application affects the transformation pipeline (simplified for the purpose of explanation) by providing a program for transforming vertices (**Vertex Shader**) and for computing the resulting color of pixel (**Fragment Shader**). The idea of conversion is thus to encapsulate the vertex shader with *additional steps* to transform the rendered geometry to specific quilt's view i while keeping the rest of pipeline intact.

4.3.1 Modifying program creation process

Shaders and programs are treated as OpenGL objects. Both of them have to be created prior to using them. A typical *generate/delete* paradigm is used for them.

As already mentioned, applications compile each shader independently, and then link shaders. These actions are done by `glCompileShader` and `glLinkProgram`, respectively. However, before the program can be linked, it has to know which shaders are used for its creation. This is achieved by *attaching a compiled shader to the program* by calling `glAttachShader`.

In general, there are two ways how to change the program. The program can be *changed during creation* by catching call to `glLinkProgram`, reattaching custom shaders and linked.

Alternatively, according to the manual pages, programs are allowed to be relinked. However, in general this brings several drawbacks such as:

- **All values of uniforms are lost and must be reuploaded**
  It is possible to query all existing uniforms for a given program using OpenGL API, and backup uniform values.

- **Positions/indices may change for uniforms**
  If the application caches uniform locations, it is impossible to overcome this problem. In addition, *uniform block bindings* must be altered to reflect new positions.

4.3.2 Introspecting Vertex Shaders

In order to draw multiple views, it is necessary to be able to alter VS so that the transformation of vertices is changed for the corresponding view. As mentioned above, VS computes a position in *clip-space* from an input vertex. Whereas while using fixed-pipeline programmers have to follow a predefined structure of using specific matrices for transformation, OpenGL's standard itself does not define how the transformation should be arranged in shaders. Applications are free to set up an arbitrary sequence of statements, which fulfill their needs.

Luckily, most of the applications use the following variations of vertex shader:

- **identity function**
  Vertex shader simply copies the input vertex data to output. This kind of behavior is used, for instance, for rendering full screen quad geometry for post-processing, or for rendering elements of GUI, which tends to have their position pre-calculated at CPU-side.

- **constant propagation**
  Vertex shader sets the output position to a hard-coded vector, stored inside the shader's code. This is an extreme case, for instance used to generate a full-screen quad.

- **matrix multiplication**
  The most common shader type, which follows *Model-View-Projection* transformation. Typically, it gets either `MVP` or pair *ModelView/Projetion* matrices using uniforms, and then simply multiple input vertex position with these matrices.

- **Far plane rendering**
  A far-plane rendering is a special case of previous methods, in which the resulting output vector has w component set to 1, effectively placing the geometry at the position of far plane. In addition, this maybe accompanied with *model-view matrix* only using rotation and scale to simulate directional rendering.

  This is commonly used when rendering skybox.

By *determining which variant the vertex shader implements*, it is possible to insert custom code so that the original function of the shader is preserved, but additional parameters can be provided to render a different view.

Algorithm of shader analysis

According to the specification, a valid shader always has to provide `main` function, which is called to by the graphic card to trigger the transformation. In theory, for any type of shader, it is possible to parse its code into an *Abstract Syntax Tree*, a graph representation of program.

By tracing the subtrees of such AST, it is possible to find all assignment statements which assign to built-in variable `gl_Position`. Next, by tracing subtrees of such assignment

node, it is possible to determine which variables or uniforms are used to compute the result and thus classify the type of transformation according to the cases defined above.

In the case of transformation matrix, GLSL typically computes resulting `gl_Position` of vector by multiplying a column vector from right. Therefore, by tracing the multiplication expression and searching for the left-most matrix, the name of transformation matrix can be extracted. Depending on the type and implementation of a shader, this could be either a projection, a view-projection or a MVP matrix.

4.3.3 Detecting the transformation uniform in a shader

Once the transformation matrix is identified in the shader program, it is necessary to track its up-to-date position during the run-time of the application.

This can be easily accomplished by hooking and tracing `glUniform4v` function. Next, when the hooked function is called, the used uniform location is searched in convertor's internal shader's metadata and compared against stored uniform name and its location. The value is then marked as transformation matrix if both locations are equal. After acquiring the transformation matrix, the parameters of projection are estimated as described in Section 4.3.4. An example of Vertex Shader is shown in Listing 4.1.

```
1  #version 330 core
2  layout (location = 0) in vec3 aPos;
3  layout (location = 1) in vec3 aNormal;
4  layout (location = 2) in vec2 aTexCoords;
5
6  out vec2 TexCoords;
7
8  uniform mat4 model;
9  uniform mat4 view;
10 uniform mat4 projection;
11
12 void main()
13 {
14     TexCoords = aTexCoords;
15     gl_Position = projection * view * model * vec4(aPos, 1.0);
16 }
```

Listing 4.1: An example of simple vertex shader which computes output vertex position directly by multiplying the matrices inside the code. By tokenizing the assignment statement and taking the first identifier from the left, the resulting matrix uniform name can be detected.

Tracking of Uniform Buffer Objects

More complex shader programs may store matrices in *Uniform Buffer Object* (**UBO**). These are continuous memory buffers with a structure defined similarly as a *struct* in C by a programmer. UBOs are typically used by applications due to ability to share the same data by attaching a single UBO to multiple shader programs. This is illustrated in Figure 4.6. Thanks to this ability, the number of uniform passing API calls can be reduced, and thus, the application's performance can be improved.

In case of conversion, the usage of UBO adds additional complexity to tracking. The tracking process boils down into additional steps:

- **Detection of Uniform Block in shader**
  While injecting the shaders, when a matrix uniform is found, it is necessary to identify the instance name of uniform block, which contains the transformation matrix uniform. This is done by parsing GLSL code of the shader.
- **Getting slot location of Uniform Block**
  Each Uniform Block has a unique location with respect to the shader program. This location is needed by convertor in order to match precise UBO with the slot that is being attached. The location can be queried using `glGetUniformBlockIndex`.
- **Locating Uniform Binding Point**
  Each uniform block has a *uniform binding*: a special binding slot that unifies uniform blocks from various programs into a single identifier. The uniform binding ID for given uniform block location of given shader can be detected by sniffing `glUniformBlockBinding` calls.
- **Locating attached UBO's**
  Each uniform binding has at most one UBO binded from which the data are taken during the draw call. For a given uniform binding, the UBO identifier can be detected by tracking `glBindBufferBase` or `glBindBufferRange`.
- **Tracking data changes of UBO**
  Finally, it is necessary to detect when the new data are uploaded to UBO to sniff the current transformation matrix.

  This is done by sniffing `glBufferData`/`glBufferSubData`. However, it is necessary to determine the position of matrix within the buffer at first. For this purpose, OpenGL provides `glGetActiveUniformsiv`, which when used with `GL_UNIFORM_OFFSET` returns the address of uniform within the block.

4.3.4 Estimating projection matrix from transformation

In general, there is not any compulsory format for the multiplication of vertex position in the process of transformation. Applications may use any variation of joined or isolated *ModelViewProjection* matrices. In the worst case, a single matrix, so-called `MVP` matrix, is provided to shader using uniforms.

Although a decomposition of a matrix into arbitrary product of matrices is not possible in general, this section tries to define cases for the transformation matrix when this is possible.

Definition 4.3.1. *Let symmetric projection matrix be matrix* $\begin{pmatrix} F_x & 0 & 0 & 0 \\ 0 & F_y & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix}$

where

$$\begin{aligned} A &= \frac{-(f+n)}{f-n} \\ B &= \frac{-2(fn)}{f-n} \end{aligned} \tag{4.3}$$

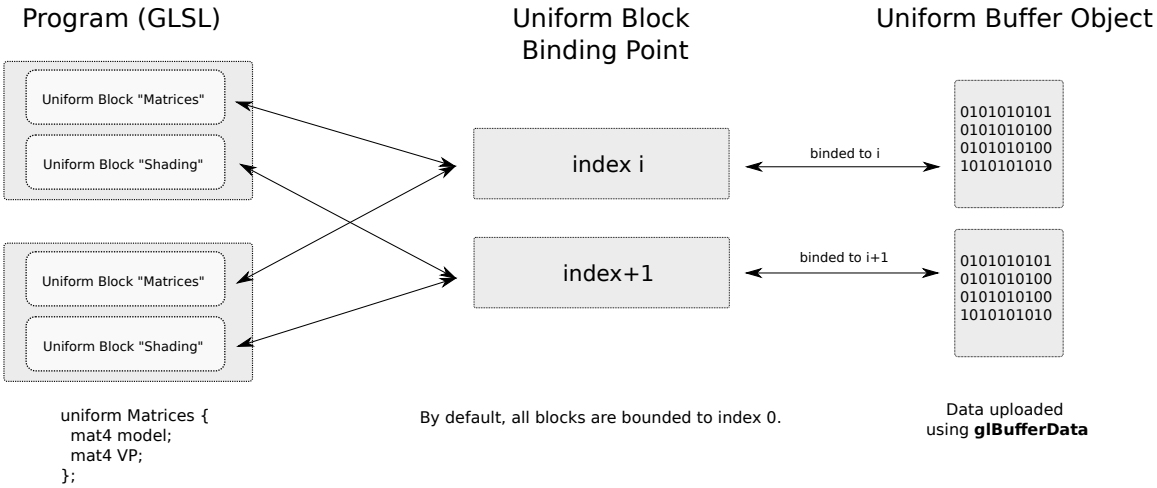


Figure 4.6: Relations between Uniform Block (defined in a shader), Uniform Block Binding points and Uniform Block Objects (UBO). A single UBO can be accessed by multiple shaders under the same name when all uniform blocks over the shaders use the matching uniform block binding point. Thus, to sniff the relations and detect which UBO actually holds data for given uniform block, this levels of indirection must be sniffed from API calls.

Theorem 4.3.2. *Let P be a symmetric perspective projection matrix, MV be a model-view transformation, which is only composed by* ***translation***, ***rotation*** *and* ***scalar scale***, *and $MVP = P \times MV$.*

Then P can be estimated from MVP matrix.

Proof. At first, assume that the transformation matrix consists of a rotation 3x3 matrix R and associated translation 1x3 t vector. We can use the property of $SO3$ group, which ensures that the norm of each 3x3 row or column is equal to 1.

Let us denote the rotation submatrix as in Equation 4.4 and $t = (k, l, m)$.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \tag{4.4}$$

By multiplying MV with P, MVP matrix becomes:

$$MVP = MV \cdot P = \begin{pmatrix} F_x \cdot a & F_x \cdot b & F_x \cdot c & F_x \cdot j \\ F_y \cdot d & F_y \cdot e & F_y \cdot f & F_y \cdot k \\ A \cdot g & A \cdot h & A \cdot i & A \cdot l + B \\ -g & -h & -i & -l \end{pmatrix} \tag{4.5}$$

As $R \in SO3$, norms of (a, b, c), (d, e, f) or (g, h, i) are equal to 1. Based on this fact, we can derive following:

$$\begin{aligned} norm(F_x \cdot a + F_x \cdot b + F_x \cdot c) &\implies \sqrt{(F_x)^2 \cdot a^2 + (F_x)^2 \cdot b^2 + (F_x)^2 \cdot c^2} \\ &\implies \sqrt{(F_x)^2 (a^2 + b^2 + c^2)} \implies \sqrt{(F_x)^2 \cdot norm(a, b, c)} \implies |F_x| \end{aligned} \tag{4.6}$$

However, as F_x is a result of $tan(angle)$ function, it is always positive, leaving out only a unique result positive result of Equation 4.6.

F_y and A can be estimated following the same recipe over vector $(F_y \cdot d, F_y \cdot e, F_y \cdot f)$ and $(A \cdot g, A \cdot h, A \cdot i)$, respectively.

The remaining B can be obtained as shown in Equation 4.7.

$$l = -MPV_{44} \implies A \cdot l + B = MPV_{34} \implies B = MPV_{34} - l \cdot A \tag{4.7}$$

In the second part of the proof, if *model-view* matrix was created using component-symmetric scaling (scalar scaling), given as $S = (s, s, s)$, one can still obtain the correct parameters by inverting scale of matrix's elements.

The resulting MVP matrix for the case of using scale is derived in Equation 4.8.

$$MVP = MV \cdot P = \begin{pmatrix} F_x \cdot a \cdot S & F_x \cdot b \cdot S & F_x \cdot c \cdot S & F_x \cdot j \\ F_y \cdot d \cdot S & F_y \cdot e \cdot S & F_y \cdot f \cdot S & F_y \cdot k \\ A \cdot g \cdot S & A \cdot h \cdot S & A \cdot i \cdot S & A \cdot l + B \\ -g \cdot S & -h \cdot S & -i \cdot S & -l \end{pmatrix} \tag{4.8}$$

Scale S can be estimated by calculating the norm of $(-g \cdot S, -h \cdot S, -i)$.

Then, taking the norm as before results in Equation 4.9.

$$\begin{aligned} norm(F_x \cdot a \cdot S + F_x \cdot b \cdot S + F_x \cdot c \cdot S) &\implies \sqrt{(F_x)^2 \cdot a^2 \cdot S^2 + (F_x)^2 \cdot b^2 \cdot S^2 + (F_x)^2 \cdot c^2 \cdot S^2} \\ &\implies \sqrt{S^2 \cdot (F_x)^2 (a^2 + b^2 + c^2)} \implies \sqrt{S^2 \cdot (F_x)^2 \cdot norm(a, b, c)} \implies |S||F_x| \end{aligned} \tag{4.9}$$

Then, F_x, F_y and A are extracted using norm as before, but multiplied with the inverse of scale S.

□

Note that this proof is suffiecient to always reconstruct the projection in the case of VP, because the transformation of camera is always defined only using translation and rotation.

Estimating Far and Near plane distances

The extract parameters F_x, F_y, A and B are sufficient to recreate the projection matrix. However, it is possible to extract both f and n, which defines the distance to far and near plane, respectively. As $F_x = \frac{l}{n}$ and $F_y = \frac{t}{n}$ in case of symmetric projection, estimation of these two parameters can help us to estimate the distance to left (l) and top (t) clip-planes. By estimating these, the projection could be altered proportionally to the original one.

To derive f and n, we begin with Equation 4.3.

To simplify the form, let us define a substitution $R = f - n$, which is a depth range between near and far plane.

$$\begin{aligned} \frac{-(n + R + n)}{R} = A &\iff \frac{-(2n + R)}{R} = A \\ \frac{-2((n + R)n)}{R} = B &\iff \frac{-2(n^2 + Rn)}{R} = B \end{aligned} \tag{4.10}$$

Next, we will continue by expressing R in one of equations, and plugging into the other one.

$$\frac{-(2n + R)}{R} = A \iff -2n - R - AR = 0 \iff 2n + R(1 + A) = 0 \iff R = \frac{-2n}{1 + A} \tag{4.11}$$

Now, by plugging into the equation, we obtain the parameter n (*near plane distance*) from A and B.

$$\begin{aligned}
&\frac{-2(n^2+Rn)}{R} = B && \iff -2n^2 - 2Rn - RB = 0 \\
&\iff n^2 + Rn + \frac{B}{2}R = 0 && \iff n^2 + n \cdot \frac{-2n}{1+A} + \frac{B}{2} \cdot \frac{-2n}{1+A} = 0 \\
&\iff n^2 + n \cdot \frac{-2n}{1+A} + B \cdot \frac{-n}{1+A} = 0 && \iff (1+A)n^2 + n \cdot -2n + -Bn = 0 \\
&\iff (1+A)n^2 + -2n^2 + -Bn = 0 && \iff (1+A-2)n^2 + -Bn = 0 \\
&\iff n((1+A-2)n - B) = 0 && \iff n = \frac{-B}{1-A}
\end{aligned} \tag{4.12}$$

Parameter f can be obtained by plugging n into the definition of R, derived above.

Estimation of orthogonal matrix

The use of orthogonal matrix as the projection matrix can be detected by comparing the last row of MVP with vector $(0, 0, 0, 1)$

Next, fx and fy can be detected using similar steps as for perspective projection. However, due to shape of the last row of the orthogonal matrix, it is no longer possible to rescale the matrix.

As depth coordinate in view-space does not matter in orthogonal projection, so, intuitively, it can not be estimated from MVP matrix.

4.3.5 Extending pipeline

Now that we can estimate parameters of perspective transformation, it is possible to invert vertex's position from *clip space* to *view space*, apply virtual's view horizontal translation, and multiply corresponding result with adjusted perspective transformation with optical center shift using the same algorithm in fixed-pipeline.

The inversion is expressed in Algorithm 2.

Algorithm 2: Inversion of clip-space to camera-space

Input : Clip-space position of vertex *cs*, resulting from the original shader
Output: Vertex in camera-space coordinates
F_x, F_y := getProjectionParametersFromUniforms();
return $(cs.x/F_x,\ cs.x/F_x,\ -cs.w)$;

A high-level overview of the modified transformation pipeline is shown in Figure 4.7.

4.3.6 Injecting Geometry Shader

Applications which use Geometry Shaders typically compute the transformation in those shaders instead of Vertex Shader. Thus, in order to support applications which use them, it is necessary to define the injection process for them as well.

In general, Geometry Shaders contain the main function, in which the input data from the previous parts of pipeline (typically, from Vertex Shader) are processed, and which outputs transformed vertices of the output primitives.

Each vertex primitive is assembled from the vertices emitted during the run-time of Geometry Shader. Each output vertex is emitted using GLSL's `EmitVertex` function and after all primitive's vertices are emitted, the shader calls `EndPrimitive` to finish the assembling.

In conclusion, from the injection point of view, the process of injection of Geometry Shader is almost identical to Vertex Shader. It is sufficient to find all assignment statements to the shader's built-in variable `gl_Position`, inspect the operation to find the

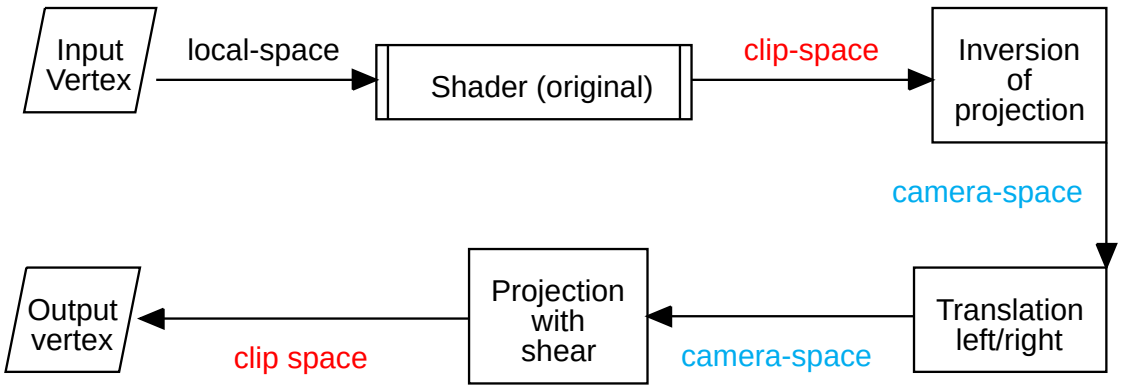


Figure 4.7: A high-level depiction of transformation pipeline of Vertex Shader after injection. The original shader is wrapped and its resulting `gl_Position` is further transformed inversely using estimated projection parameters. The resulting clip-space is retransformed using quilt's view transformation, relative to the original view, and with modified projection, made of estimated parameters and shear.

transformation matrix or determine the type of transformation, and apply the additional steps for achieving the right multi-view position of the output vertex. This processed is expressed in Algorithm 3.

Algorithm 3: Injection to Geometry Shader

Input : Geometry Shader
Output: Extended Geometry Shader that transforms to quilt i camera
for *assignment to gl_Position in GLSL code* **do**
    insertInversionOfProjectionCodeAfterAssignment();
    insertQuiltCameraTransformation();
    insertShearedProjection();
end

4.4 Layered framebuffers

The aforementioned conversion works fine provided the application renders directly to the Backbuffer. As complex applications use artificial *Frame Buffer Objects* (referred as FBO), it is necessary to define the flow for these. Rendering into non-default FBO is a part of many rendering techniques such as post-processing effects (e.g. blur, depth of field), or dereffered shading. The techniques are commonly referred as *multi-pass rendering*, because the rendering process is split into multiple layers (for instance, FBOs), where each layer may use the resulting texture of the previous one.

4.4.1 Mirroring artifact

To illustrate why it is necessary to devote extra time to support multi pass rendering, consider a simple rendering pipeline made of two stages. In the first stage, the scene is rendered as usual and the second stage applies a blur over the image from the first stage. Without taking any extra care to detect and alter the principle of previous duplication

methods, the duplication of draw calls will be dispatched in both stages. In the first stage, the output image will look correctly. However, the second stage will access the first stage's image, which is internally fragmented into a quilt. If the quilt has 3x3 cells in result, the final image would contain 3x3 quilt views, each of them fragmented into 3x3 cells, resulting in structure equal to a 9x9 quilt. This artifact is illustrated at Figure 4.8.

Figure 4.8: Example of the Mirroring artifact. The left image depicts result of the first stage - a regular render of 3x3 quilt from the scene which contains a single backpack. The right image shows the result of applying blur stage on the result of the first one. Clearly, rendering of each cell of the resulting quilt samples whole color texture of the previous stage. However, the texture is internally split into the grid due to quilt rendering. Therefore, the result is a *mirroring effect*, which is incorrect. When the second quilt is used in display, the user will perceive a flat image of 9 bluredbackpacks instead of desired blured 3D backpack.

Note that this issue can not be simply resolved by preventing the duplication. In practice, it is desired to produce a quilt in all stages of multi-pass rendering so that the 3D effect would be preserved.

In specific cases, the quilt rendering can be disabled when the application draws to non-default FBO. For instance, in a scene of a prison, the application may render multiple views to simulate CCTV cameras by rendering subviews to a custom FBO and finally, rendering the color buffer using planar geometry to the final image. In such a case, the output camera image should be flat, so no internal quilt production is needed, and therefore, the quilt rendering can be disabled for such draw calls.

Unfortunately, detecting whether the draw call should be duplicated given FBO is **ambiguous** as it depends on the effect, desired by artist.

Proposed heuristic

In theory, a simple heuristic could be used to expect the desire. Note that post-processing effects typically render simple geometries such as full screen quads using identity transformation, whereas rendering of CCTV cameras uses a regular transformations pipeline to place the screen's image into the space.

4.4.2 Using Layered FBOs to overcome Mirroring artifact

One approach is to support multi-pass rendering is to split the texture of FBO to a grid directly as before using `glViewport` and *override behavior of sampling operations* transparently in the stages. However, this requires tracking to detect if the texture is sampled during the next draw calls, and intercept & alter sampling code in shader so that the proper subregion of the texture is sampled instead. If shader recompilation is not possible, this is even harder to implement correctly if the used program is reused for drawing multiple types of geometries, not only for sampling the previous stage.

In addition, storing all views inside a single texture disallows the use of mip mapping as the generated mip maps would contain color bleeding through the edges between views subregions.

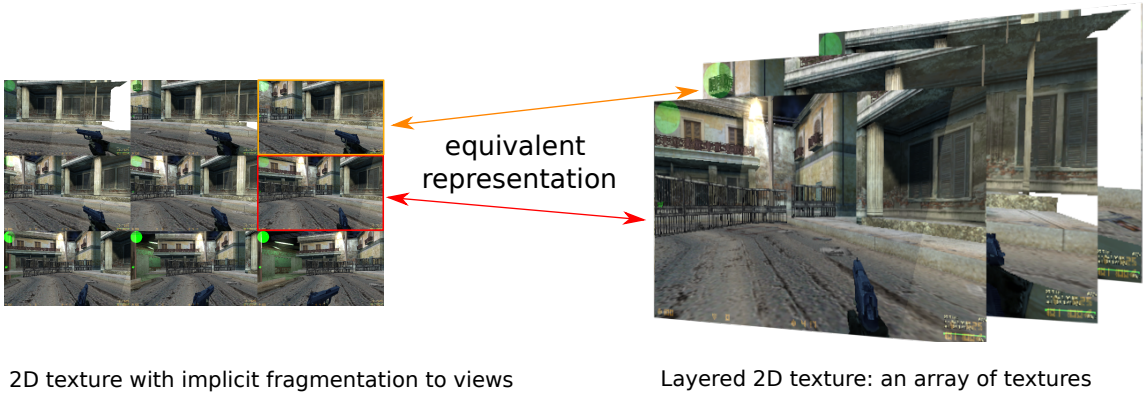


Figure 4.9: Two approaches of rendering to framebuffer when rendering multiview. The framebuffer's textures can either virtually split into a grid of views (left), or created with multiple internal images, named layers (right). The major advantage of layering over splitting is the possibility of simultaneous rendering into each layer in a single draw call using instanced Geometry Shader. This can not be achieved using splitted framebuffers due required complex clipping, which is typically not supported by GPUs. In theory, this could be implemented in Geometry Shader by clipping the output triangle manually at the expense of additional complexity. In addition, layered FBO can be easily accessed as a regular, single view FBO using Proxy FBO, made of texture views into layered textures of the FBO.

4.4.3 Shadowing of Frame Buffer Objects

Alternatively, it is possible to replace (shadow) an FBO with an internally created layered FBO. A *Layered FBO* is a framebuffred, created by attaching layered textures (named *Array Textures* [15, p. 178]). The number of layers matches the count of views in quilt. In comparison to a regular 2D texture, Arrayed Textures can be used in Layered Rendering, in which the draw call affects more than just a single output texture. For instance, Cube Maps are similar to arrayed textures as their internal storage includes 6 times more texture buffers than a regular 2D texture. However, the array textures are a more generalized and less restricted form of cube map texture.

In addition to Layered Rendering, an array texture can be sampled as a regular 2D texture by creating a so-called Texture View *Texture View* [15, p. 270]. A texture view is a proxy texture object which may point to a sublayer, or a sublevel of an existing texture.

Both approaches of storing the quilt are compared in Figure 4.9.

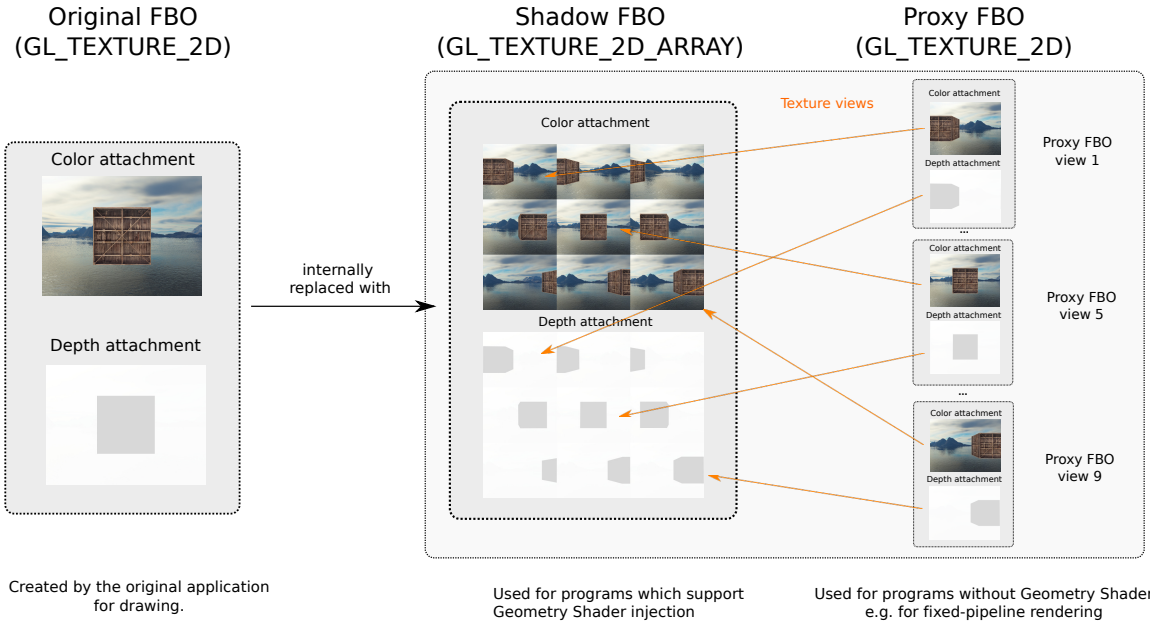


Figure 4.10: Each Frame Buffer Object (FBO) created by the application is internally replaced with a shadow Frame Buffer Object, which consist of layered texture attachments. Whenever the original frame buffer is bound during the call, the convertor use the shadow FBO to draw and to sample from. As some shader programs or draw calls from fixed-pipeline does not support layered rendering, which is only done in Geometry Shader, proxy FBOs are created to allow rendering of the draw call per view. Each proxy FBO is created using textures, resulting from Texture View. Therefore, proxy FBO are memory lightweight indirection of the shadowing FBO.

General idea of Shadowing

This approach requires tracking of all FBO used by the application and internally recreating a compatible FBO with the same attachments, but created using layers. Additionally, it is needed to replace the sampling operations of the layered FBO's texture in all affected shaders and choose the correct layer, corresponding to the layered of currently drawn cell of the quilt.

Algorithm of the Shadowing Process

In conclusion, during each draw call to a non-default FBO, a layered FBO is re-bound instead of the current FBO, and the draw call is drawn using *Geometry Shader* with instancing. If such drawing is possible only using injected `VS`, a temporary proxy FBO is created for each view of the quilt of given layered FBO. For a quilt which contains 45 views, this results to additional 45 proxy Frame Buffer Objects, each made of Texture Views to particular layer of the layered FBO.

Finally, when the texture of the original FBO is about to be used in rendering, this is detected and the intercepted draw call is dispatched for each view with view's Texture View. During each subdraw call, a texture view is bound instead of the texture of the original FBO, created by the application. The creation of shadow FBO and corresponding proxy FBO is illustrated in Figure 4.10.

Rebinding Texture Units

When the application wants to draw using an attachment of the original FBO, it is necessary to carry on that the original FBO is replaced with the shadowed instance.

If recompilation of shaders was possible, this could be implemented by changing the type of sampler uniform in shaders to `sampler2DArray` or `sampler2DArrayShadow` in case of sampling depth buffer.

However, recompilation is avoided and the use of FBO attachments in samplers of given shader can not be detected in advance, because the shader could also be used for rendering non-shadowed textures, for which the original sampler should be preserved.

Instead, the convertor can track the state of *Texture Units*, which binds OpenGL textures to indexed slots, whose indices are then assigned to uniforms in the shaders. By detecting if Texture Unit is bound to an attachment of FBO, it is possible to create a 2D texture view which is binded instead.

This can be achieved by tracing `glActiveTexture` for detecting which texture unit is currently affected by subsequent calls to `glBindTexture`, or by tracing `glBindTextureUnit`. The state of a texture unit is structured as each texture unit supports multiple targets for binding, which may be bound in parallel.

4.4.4 Tracking of textures and FBO's lifetime

In order to create a shadowed layered FBO for the application's FBO, it is necessary to track the lifetime of attachments of the FBO and to create layered textures for each attachment.

This can be achieved by hooking *glGenTextures*, *glGenFramebuffers*, and *glFramebufferTexture* for attachment.

Shadowing each texture that is created by the application would be memory-demanding and wasteful. Instead, the shadowing process can be done on-demand by book-keeping metadata about existing FBOs and their attachments, and only shadowing the attachments and the FBO when needed.

In addition, it is also possible to cache temporary texture views and proxy FBO between frames.

Preventing shadowing of shadow maps

Some applications may use the shadow mapping technique for rendering direct shadows of dynamic geometry in real-time.

This technique uses an auxiliary FBO for rendering the scene from the light's point of view. The buffer is than accessed to determine if the resulting pixel of rendered geometry should be lit. This includes sampling of the depth buffer attachment of the shadow map.

However, if the shadowing of FBO was turned on when rendering the shadow map, the depth map would be affected by shift and the shadow map would be layered despite rendering from the same position of light from all views. The shift of transformation

would cause a different position of light for each of the quilt's view, leading to shading inconsistencies.

Instead, the shadowing of FBO could be turned off with using a simple heuristic. When the decision about shadowing of FBO is about to be made, the attachments can be queried. If the convertor detects that the FBO has only depth-buffer attachment, this is considered to be a depth map, and the shadowing is turned off for the FBO.

4.5 Instanced Geometry Shader Rendering

After the previous concept, a reasonable implementation of *HoloInjector* would in theory support most of applications.

However, the technique which creates a shadow FBO and dispatches single draw call multi times may use too many redundant API calls to fill the quilt, thus decreasing the FPS.

Instead, a single draw call could be used to draw to multiple layers of FBO. This technique is know as *Layered Rendering* and is achieved using two subconcepts - layer ID and instancing.

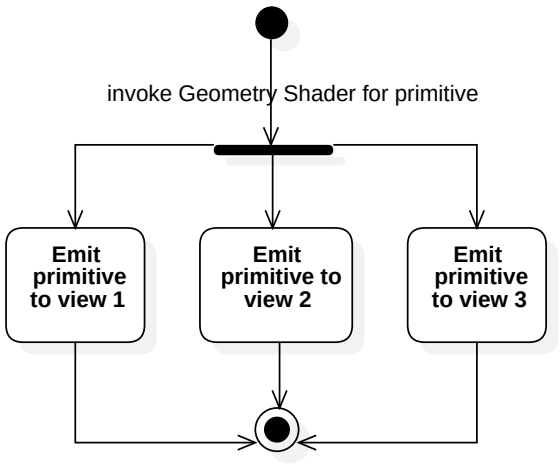


Figure 4.11: Use of instanced Geometry Shader. Upon each invocation of Geometry Shader for a primitive, multiple instances are invoked in parallel. Each instance can emit an output primitive for corresponding view of the quilt. In the figure, 3 invocations are used as an example.

Layer ID is a built-in output variable in Geometry Shader, which is accessed during the emit call to find out the output layer of FBO in which the primitive should be rendered. Typically, this is only done once per primitive as splitting the primitive into multiple views is undefined.

Geometry Shader's Instancing provides the possibility to dispatch the main loop of Geometry Shader multiple times per single draw call. Each instance has its unique ID in the variable `gl_Invocation`.

To implement this technique, two implementations are necessary to support both existing Geometry Shaders and to enhance the existing Vertex Shaders-only's based pipelines with a new stage inserted in between.

4.5.1 Inserting Geometry Shader into the Pipeline

The process of insertion has to tackle with the following processes:

1. Duplicate the geometry in shader correctly
2. Reroute Inputs/Output between Stages

Duplication of Geometry

Due to language requirements, a geometry shader has to define the type of input and output primitives in compile-time by using *layout* keyword [15, p. 430]. For simplicity, consider triangles as both input and output primitives.

Then, it is necessary to determine how many instances of Geometry Shader with how many primitives per instance will be executed. The OpenGL's standard supports at least 32 invocations of Geometry Shader per draw call, defined as `MAX_GEOMETRY_SHADER_INVOCATIONS`. However, a typical autostereoscopic may use more than 32 views, so it is necessary to define this in a configurable way.

In addition to instancing a single Geometry Shader's invocation can yield more than a single primitive. This is determined by setting `max_vertices`'s parameter of output layout directive. By default, OpenGL Specification supports up to 256 vertices, resulting to at least 85 output primitives per invocation.

Together, it is possible to combine both to satisfy the maxima supported by GPU's driver, and to achieve parallelism by using invocations over primitive duplications.

Finally, a single primitive is always emitted with `gl_LayeredID` defined by using Equation 4.13 where l defines the output layer ID, i is the invocation number read from `gl_InvocationID`, D is the count of primitives emitted per invocation and d is the current number of already emitted primitives.

$$l = i * D + d \tag{4.13}$$

Rerouting In/Out variables

After defining the function of the inserted Geometry Shader, it is necessary to ensure that the inserted Geometry Shader will not affect the pipeline by causing linking errors. In a more complex application, simply adding another stage while keeping the same code of the original GLSL shaders is not possible.

In addition to uniforms, which serve as a constant per-primitive memory, the programmable stages of pipelines are allowed to interchange data by using In/Out variables.

Typically, these variables are tightened together by name. The exchange mechanism is based on writing the result in the first stage and reading the value in the next one.

However, as these variables are defined per primitive, a single variable can only be used for two successive shader stages such as VS-GS or GS-FS. Thus, by introducing Geometry Shader into the existing pipeline, the mechanism break.

To support the passing mechanism, it is necessary to add an additional stage of the in/out variable. For each output variable in Vertex Shader, an input variable is defined together with an output variable.

As the variable is tighten to primitive, the Geometry Shader receives an array of an input variable, for each input vertex. Thus, in addition to duplicating, it is necessary to write to the output variable while emitting each vertex.

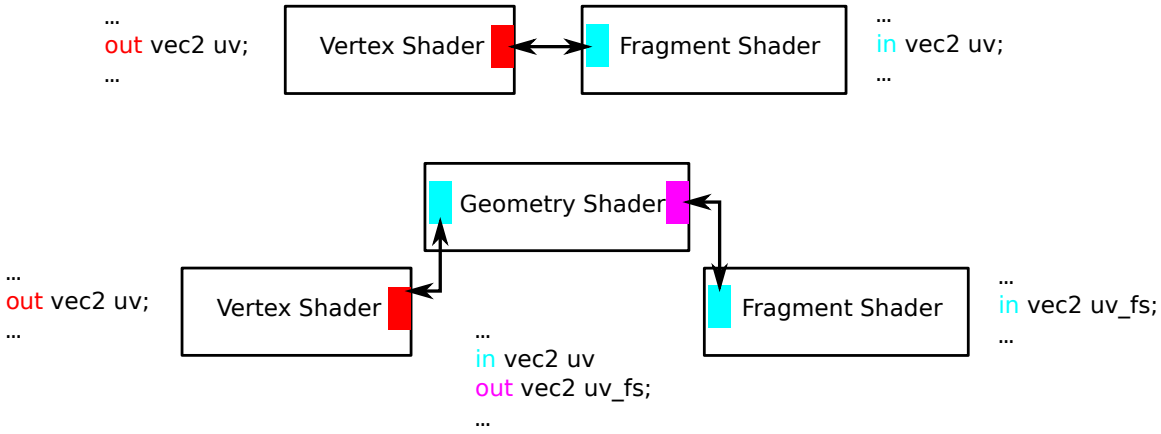


Figure 4.12: Example of In/Out renaming. Originally, both Vertex Shader and Fragment Shader used the same name of in/out variable. However, upon inserting Geometry Shader, a single variable can not be both in and out inside Geometry Shader due to language restriction. Therefore, a temporary variable *uv_fs* is introduced between GS and FS, and the name of original variable must be changed in FS correspondingly.

As two variables with different layout can not be defined with the same name, it is necessary to substitute names in either Fragment or Vertex Shader so that both the input and output variables could be generated.

This is illustrated in Figure 4.12.

Finally, this process is described in Algorithm 4.

Algorithm 4: Rerouting in/out variables

Result: Reroute out variables from Vertex Shader to in variables in Fragment Shader

for *out variable in VS* **do**
  newName:= variableName + „_fs“;
  findAndRenameVariable(fragmentshader, variableName, newName);
  GSdefinitions += „in type variableName[VERTEX_COUNT];“;
  GSdefinitions += „out type newName;“;
  for *assignment to gl_Position in GS* **do**
    vertex := getVertexIDForCurrentAssignment();
    assignmenetCode += „newName = variableName[vertex];“;
  end
end

Type of the input primitive

In a generic case, the type of the input primitive may not be known during the link time, thus the inserted Geometry Shader may cause loss of rendered geometry or unexpected artifacts. The real primitive output type of Vertex Shader always depends on the type chosen in the draw call.

As it may not be possible to alter shaders in run-time, solving this problem would require manual root cause detection and overriding settings for the used program, either to preset a different primitive type, or to disable insertion.

4.5.2 Injection of existing Geometry Shader

In case the application already uses a Geometry Shader, it is still possible to convert such shader to an instanced one by applying following steps:

- Insert `layout` declaration with instancing.
- Multiply the count of `max_vertices` with the count of duplications.
- Substitute *main()* with a different name (e.g.) *oldMain(int duplicationID)*
- Append new *main()* and call *oldMain()* for D times in loop.
- After each *gl_Position* assignment statement, add an assignment to *gl_LayeredID* using Equation 4.13.

4.6 Conclusion

In the previous sections, the major stages and rendering approaches of OpenGL were analyzed and an altered flow for these was suggested. The convered parts of the rendering pipeline were chosen with respect to frequency of use.

4.6.1 Notable missing features

As the state space of all possible OpenGL applications is larger than the aforementioned OpenGL features, many applications remain unsupported. In the following section, a brief list is given with suggestions on possible implementation.

- **Multiple OpenGL Contexts per application**
  Applications may use multiple OpenGL contexts for rendering. Typically, a context represents a visible surface such as a GUI window. The use of multiple contexts is thus typically for editors with multiple preview windows, such as Blender.

  To support such applications, the implementation must explicitly be able to separate between objects of different contexts, to track the selection of the current context, and possibly to support multi-threading.

- **3D enhancement of Ray-tracers/Ray Marchers**
  OpenGL can be used to implement ray tracing by rendering a simple geometry (typically, a full screen quad) and computing the resulting color using ray tracing.

  This approach is used for implementing ray tracers, but also for rendering volumetric scenes such as clouds, flames, or smog.

  Whereas the transformation of geometry is correctly shifted by the injector, the ray tracing can not be altered automatically, because the raytracer may use uniforms for passing the definition of a camera, or generate the camera parameters algoritmically.

  To support such ray tracers, users would manually have to select such shaders and semantically annotate which uniforms are used and what for. Then, in theory, the injection could modify the shader using regular expressions.

- **Precompiled shaders**
  Applications may prefer to compile shaders once, store them in binary format, and load the binary on the next load. In general, this can speed up the load time of demanding applications which contain many materials and thus hundreds of shaders.

  However, the binary format is not standardized, but vendor-specific, thus no systematic support for injection to such binary shaders can be defined. On the other hand, applications are shipped with source codes of shaders with high probability, because architecture of GPUs can change over time, which would break the portability of the application, if it was only equipped with precompiled shaders. Therefore, it is probably sufficient to hook methods which take precompiled shaders, such as `glShaderBinary`, and return an error which should force the application to recompile.

4.6.2 Suggestions on supporting DirectX

The described processes before are valid only for rendering using OpenGL. Luckily, as DirectX's rendering model is similar to OpenGL's, the described method could be ported to this library as well. This section gives a few hints on the additional steps required for successful porting.

Hooking

At Windows, `LD_PRELOAD` can not be used due to system restrictions. Instead, dynamic hooking is typically employed at this platform while using *WinAPI* for creating process and forcing the load of a custom library. This is commonly referred as *DLL injection*.

Analogously to *dlsym*, *getProcAddress* can be hooked and addresses can be overridden.

Differences between OpenGL and DirectX

In contrast to OpenGL's static methods, DirectX's SDK is based on classes with virtual methods. The hooking of a such system would therefore require *virtual table (vtable) hooking*.

The mapping of objects is very similar in both libraries. What differs is the use of different shading languages, but both languages have a similar programming model and similar constructions.

Chapter 5

Implementation

This chapter describes the implementation of ideas introduced in the previous chapter into practice. The ultimate goal of the chapter is to give the reader such knowledge that could be used in extending the implementation with ease.

5.1 Overview of implementation

The conversion layer is implemented in the programming language C++, mostly using *Object-oriented paradigm*. From the architectural point of view, the layer is a collection of classes, which are logically grouped and isolated.

The iterative development of the layer has resulted into a division of classes into the following logical groups:

- **dispatcher.hpp - Dispatcher**
  Dispatcher provides an interface between the application above the convertor and the internals of the convertor. Therefore, all hooked functions of OpenGL and other libraries are catched in the dispatcher, where they are rerouted into internal classes.

- **/hooking/ - Hooking utilities**
  Hooking classes provide a simple framework for adding new functions to be hooked easily, and dealing with a complex process of rerouting and calling the original function.

- **/tracking/ - Resource tracking**
  Resources form a group of classes, tracking the state of each OpenGL object, created by the application. Objects are textures, renderbuffers, Frame Buffer Objects, and shaders. The primary reason is to allow querying of currently bounded resources as well as their properties, which is for instance important for Draw manager in order to choose how to duplicate draw calls.

- **/manager/ - Managers**
  Managers are responsible for implementing a particular responsibility, routed from dispatcher. They do not own any objects or resources.

- **/pipeline/ - Pipeline transformators**
  Pipeline transformators are classes which contain functionality for parsing shaders, extracting metadata, and modifying the shaders.

- **/utils/ - Utilities**
  Utility classes such as loggers, text formatters, and various wrappers around C-style OpenGL calls and object management.
- **/ui/ - Overlay**
  Overlay is an in-application menu, which provides sliders and inputs for controlling the settings of display, but also facitilities for debugging such as inspector. This group constitutes of classes, needed for overlay rendering, input hooking, and glue logic for manipulation of settings using the underlying GUI library.
- **/paralax/ - Paralax-like mapping**
  An attempt to replicate the algorithm, used by VR convertor. This is an experimental code, which is not reachable from the rest of code.

The architecture of HoloInjector is split into several layers and subparts: hooking, state tracking, and managers.

5.1.1 Flow of data

The conversion layer is serving as a translation SW layer between the application and the underlying hardware. Thus, its architecture can be displayed as a black box, which at the input side contains connection points (OpenGL API methods) and output pins at the left side, which are concrete OpenGL API calls, called towards hardware.

In general, the function of the injector boils down to a rerouting of calls with additional side activities such as state tracking.

A typical call to OpenGL by the application thus starts from `Dispatcher`'s virtual method. Next, the call is either directly delegated to OpenGL's driver if it is not important for the injector or the implementation of the call could be as trivial as a single call to state tracker.

More complex API calls are implemented using managers. Managers themselves do not hold any state, but help to reduce the amount of code in the `dispatcher`, and thus to keep the code clean. Subsequently, the method defined in a manager typically leads to multiple OpenGL API calls.

5.1.2 Context

As the name suggests, Context is a class which manages the lifetime of all created objects. The class is thus composed of all trackers and internal classes with a long lifetime.

Currently, the application assumes a single context to exist over during run-time. This could be in future extended to support applications with multiple OpengGL contexts.

Context is implemented using `PIMPL`[1] design pattern to prevent excessive inclusion of header files.

5.1.3 Hooking

The fundamental class for hooking is `OpenglRedirectorBase`, where all functions, whose hooked handlers should be available, are placed there.

There are two ways how a function is hooked. One way is by *exporting the function name* in the compile module of `OpenglRedirectorBase.cpp`. Another way is by redirecting

[1] PIMPL = Pointer to implementation

the function address when the application uses `dlsym` for querying the function's address. Both ways are covered together by using the specialized macro, described in the following section.

In order to introduce a new hooked function, the function is declared as a virtual function in `OpenglRedirectorBase` with the same signature as the original function. Next, an expansion of macro with name `OPENGL_FORWARD` is defined in `OpenglRedirectorBase.cpp` with name and parameters of the hooked function, following the signature. When expanded, the macro automatically generates a function with the same signature, but *C export style*, so that the generated function's address can be passed in hook. The macro also defines the definition of such a function so that the virtual method in the class is used as a hooked function's handler. In addition, the macro generates a definition of the virtual method, which by default calls the original function with arguments serving as an identity. The flow of execution is illustrated in Figure 5.1.

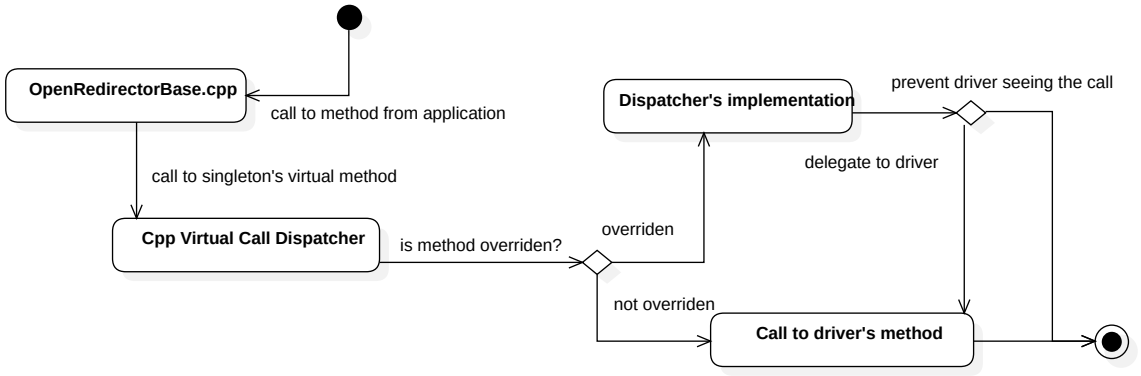


Figure 5.1: A diagram, explaining the flow of execution of the hooked function. When the function is called, the execution ends up in exported function, defined in `OpenglRedirectorBase.cpp`, which delegates the execution to virtual method of `OpenglRedirectorBase`-based class using singleton pattern. Subsequently, depending on whether the method is overridden, either original function is called or user-defined callback. Optionally, the original function can be called from user's callback.

The whole process of making use of the hooked functions is then implemented using *class specialization*. A specialized class then overrides the virtual method with its definition of handler method. The original function is available by calling the base's virtual method.

The process of hooking is done using *singleton pattern*. When a specialized class is constructed, the address is stored in the implementation of module in a static variable. All subsequent calls to hooked functions will result in querying the stored address, and delegating the call to its methods. The overriding handlers are then called via *virtual table mechanism*.

For most of the hooked functions recursion does never happened, thus when a hooked function is called inside of a call to a handler of the other one, the original function is called instead of the hook. This way it is possible to call the original functions by calling their usual interface, and it is not necessary to use to the base virtual methods. This concept is implemented by storing a *thread-local* boolean flag inside `OpenglRedirectorBase`, which determines if currently there is already a hooked function being handled.

5.1.4 Resource tracking

As algorithms used in conversion are context-dependent, resource tracking allows the convertor to keep the state of resources (more precisely, OpenGL objects) without excessive state polling.

Object tracking follows OpenGL's object managing paradigm, where objects may be created and deleted, and also bound to context in order to change.

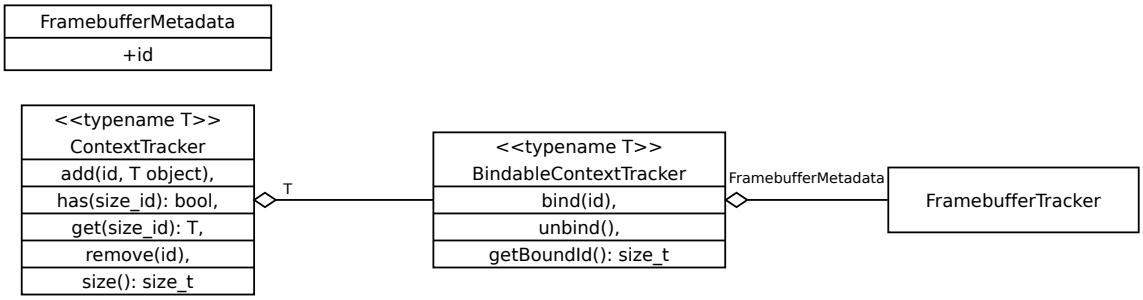


Figure 5.2: All object trackers are derived from `ContextTracker` or `BindableContextTracker`, which implements functionality to mimic OpenGL object management. Trackers store metadata, associated with real OpenGL objects, represented using specialized `XYZMetadata` class.

A special class, `ContextTracker`, is used as a base class for all trackers. It provides the most common methods such as `add(id)` (simulating `glGenerateXYZ`), `remove` (simulating `glDeleteXYZ`). In addition, a derived class `BindableContextTracker` is defined, providing functionality in order to bind specific object to context. Both classes and their members are visualized in Figure 5.2.

These two classes provides enough functionality for the classes such as `ProgramTracker` or `FramebufferTracker` to be pure template instances usings of specific storage type over `BindableContextTracker`.

However, specific objects require additional functionality for tracking so-called *shadowing*.

Shadowing of FBOs

In order to shadow a FBO, the attachments must have a layered version available. This is ensured by implementing method `createShadowedTexture` in `TextureMetadata`. As renderbuffers are a specialization of texture, they also implement the method.

When creating the layered textures, the proper width and height of the original texture must be sniffed together with texture format. In addition, a texture format conversion is implemented to convert from relative size (e.g. GL_RGBA) to fixed size OpenGL data format (e.g. GL_RGBA8).

Shadowing of Frame Buffer Objects (FBOs) is implemented in `FramebufferMetadata` class, in its method `createShadowedFBO()`. The function iterates over attachments, and triggers creation of shadowed attachment by calling method `createShadowedTexture`.

Each `FramebufferMetadata` contains a list of references to `TextureMetadata`, standing for intercepted attachments.

Texture unit remapping

In addition to drawing to shadowed FBO, sampling of such FBO must be defined correctly as well. This is implemented by tracing state of texture units and detecting if any of texture unit is bound to an attachment of FBO. This is implemented as a part of `TextureUnitTracker` (texture_tracker.cpp) which provides methods for detecting if unit is bound to a framebuffer object, and for rebinding to shadow to a texture view of shadow FBO's layer.

5.1.5 Pipeline

Pipeline is a group of classes which are employed to parse and modify shaders. The functionality is split into multiple classes. The top-level class `PipelineInjector` encapsulates the remaining class with a single method call `process()`, which receives the input pipeline as a collection of shaders, and outputs a modified pipeline with extracted metadata.

Internally, the class use `ShaderInspector` over each transformation shader to extract metadata. Some of inspector's methods use internally `ShaderPaser` to tokenize GLSL code.

The remaining classes, such as `ProjectionEstimator` or `OutputFBO`, are used for estimating the parameters of projection matrix and for managing a lifetime of the internal layered backbuffer, respectively.

Rendering for displays different from Looking Glass

Currently, the conversion of quilt to native image is implemeneted as an additional pass using a shader [2], provided by Ing. Tomáš Milet. This is implemented in `pipeline/output_fbo.cpp`, in `OutputFBO` class, in `renderToBackbuffer()` method. The implementation does not support any additional displays, and the parameters of display are entered manually using configuration.

To suppport more displays, a plug-in system could be proposed, which would allow developers of displays to provide their compiled library in specified directory, which would contain *C* methods for registration and for providing conversion shader. Subsequently, the user would be allowed to choose the display using overlay.

5.1.6 Managers

A manager is a class whose responsibility is to implement a subset of hooked OpenGL calls. For instance, `DrawManager` implements dispatching of all draw calls.

Converting draw calls

The implementation of draw calls is available in `DrawManager`. The original draw call is passed as a lambda function to `draw` function. Then, a decision process starts with cheching whether a program is bound and if the program has a detected transformation. The process is then delegated to `drawGeneric` methods, which detects if the program use Geometry Shader or Vertex Shader, or if any program is bound. Depending on these, one of `drawWithGeometryShader`, `drawWithVertexShader` or `drawLegacy` is called.

In both `drawWithGeometryShader` `drawWithVertexShader`, texture units are remapped to use shadowed FBO. In case of Geometry Shader rendering, if texture unit was bound

[2] https://github.com/dormon/3DApps/blob/master/src/quiltToNative.cpp

to an attachment of FBO such as depth map or color texture, the rendering degrates to per-view rendering and remapping the texture init to a texture views of shadow FBO.

Rebinding Frame Buffer Objects

The purpose of `FramebufferManager` clas is to manage binding of FBO. This is governed by a simple rule: draw calls to back buffer should be redirected to the unique instance of OutputFBO, stored in Context, and the rest of draw calls should be redirected to shadow FBO. This logic is implemented in `bindFramebuffer` method. In addition, a correct framebuffer must set when the application calls `glClear`.

The second purpose of class is to trigger rendering of OutputFBO's instance to native image at the end of frame. This is done by hooking `glXSwapBuffers`.

5.1.7 Utils

Utils provide RAII-oriented helpers for communication with OpenGL or processing GLSL. For instance, `glsl_preprocess.hpp` provides the functions for preprocessing GLSL.

Preprocessing is done using a standard C/C++ preprocessor with additional substitution as some of GLSL built-in macros such as `#version` are not defined for C++. Preprocessing is outsourced using external library named *simplecpp*[3]. The reason for using this library is an easy integration into a C++ project in constrast to a standard C preprocessor provided by GNU.

5.2 Overview of platform-depedency

Typically, software engineering strives for platform independency, a state in which software would be compileable and runnable at any platform and hardware. However, as the convertor uses hooking for taking control of the application, this is not possible, because hooking is not standardized and platform-specific details are used to set up hooks. This section tries to give an overview of parts of code, which are platform-dependent and which would need to be reprogrammed when porting the convertor to different platforms than Linux.

5.2.1 Loading the library

Hooking is implemented in the file `startup_injector.cpp` by exploiting the overloading of hidden function `__libc_dlsym`, implementing `dlsym` in the implementation of GNU Dynamic Loader. Thus, this methods assume the use of the aforementioned loader in the system. This trick has been taken from `apitrace`[4] and it is necessary to be able to chain load both the convertor and an additional library such as apitrace for the purposes of debugging.

Alternatively, the implementation using dynamic code patching via library *subhook* is also available in this file. The use of this method requires one to disable *SELinux*'s protection. In practice, both methods are functionally equal.

In addition, the module `startup_main.cpp` contains two methods, `hi_setup` and `hi_cleaner`, which are called before and after passing the context to the application's main function. These methods use GNU Linker's specific attribute `__attribute((constructor))` and `__attribute((destructor))`.

[3] https://github.com/danmar/simplecpp
[4] https://github.com/apitrace/apitrace

5.2.2 Hooking X Window System

As OpenGL does not specify the management of context, this is typically vendor-specific. At Linux, when using *X Window System* (also referred as X11), which is a widespread window manager, the context is provided by an extension *GLX*. Any OpenGL function loader will lead its calls to a subset of this extension. Hooked functions of GLX are part of `hooking/opengl_redirector_base.cpp` and are overloaded in `dispatcher.cpp`. This includes methods such as `glXCreateContext`, `glXMakeCurrent`, `glXSwapBuffers` and most importantly, `glXGetProcAddress`.

In addition to context management, window manager also provides *events from input peripherals*. These are important for implementing a GUI overlay, which serves for adjusting settings of the convertor during run-time or for more comfortable debugging of resources.

X11 provides `XNextEvent` method for developers to poll an incoming event to a window. By hooking this function and sniffing for event types *KeyPress*, *MotionNotify*, *ButtonPress*, events from keyboard, mouse and mouse button can be detected, respectively.

5.3 Code quality

As the purpose of the convertor is to convert OpenGL API calls to different API calls, such a task would be time demanding to implement as it would require to either mock up an OpenGL driver, or use Computer Vision to process the rendered image, and determine consistency.

Instead, parts of the code were tested using *unit testing*. This was, however, only possible for encapsuled and self-standing modules, which does not call any OpenGL API method themselves. Such modules are trackers, projection estimator, pipeline injector, parsers, and string utils. Despite of this list, the overal *code coverage* stays low due to large modules such as dispatcher or hooking classes which can not be simply tested using unit testing.

Chapter 6

Experiments

This chapter describes practical experiments with applying the conversion layer to existing various OpenGL applications. In practice, it is impossible to verify the functionality with all existing OpenGL applications, but instead, testing focuses on verifying that applications which implement a certain rendering method are possible to convert. Although such testing does not necessarily imply that all applications implementing the technique will work flawlessly, it at least increases predictability and reveals limitations.

6.1 Test setup

The programmable pipeline was tested using OpenGL tutorials such as *LearnOpenGL*[1] to verify different rendering techniques. The method has been employed on examples of techniques such as shadow mapping, normal mapping, skybox drawing, drawing to auxiliary framebuffers.

For techniques such as screen-space ambient occlusion, the method failed to provide a consistent multiview image, due to suspected problems in passing data to FS (see Section 6.3.4).

The fixed-pipeline OpenGL was tested on such as *NeHe tutorials*[2] and the more complex application *Nexuiz*. Due to the simplicity of fixed-pipeline transformation, an implementation based on the method above is sufficient to convert most of such applications with minimal visual defects.

6.2 Measuring performance

The impact of injection was measured on selected supported applications. The results are shown in Table 6.1 and in Table 6.2. Each measurement was a recording of the frame period (time per frame) in milliseconds. As most of the applications are static, the recorded sequences consist of the same frame, rendered multiple times. The measurements test the correlation between the frame rate and increasing number of views, and the correlation between increasing resolution and frame period.

Three different applications were used. *Stanford Dragon* features a large mesh, which should solely fill the pipeline. *Cubes* provides trivial geometry with simple shading. The purpose of *Steep Parallax Mapping* is to feature simple geometry, but expensive shading, so

[1] GitHub.com:JoeyDeVries/LearnOpenGL
[2] GitHub.com:gamedev-net/nehe-opengl

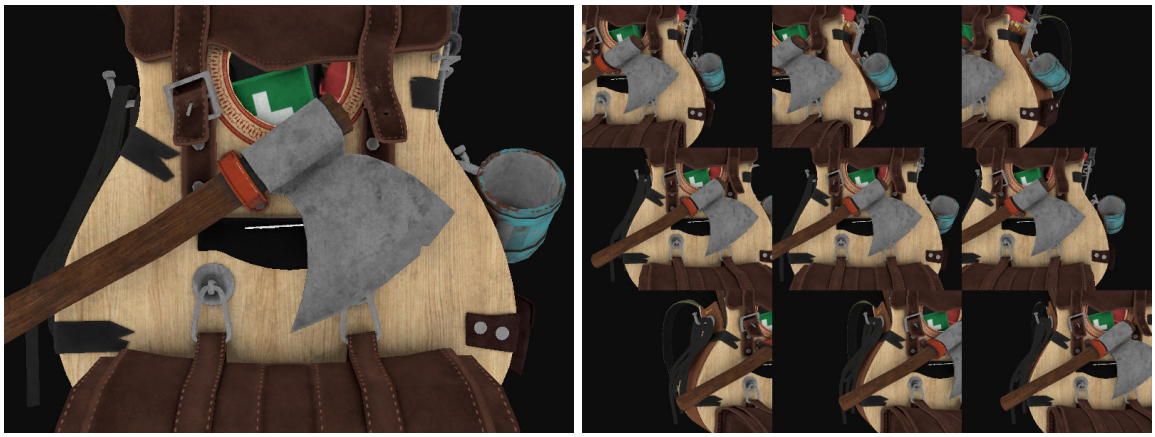


Figure 6.1: Pairs of images, showing original application (left) and resulting quilt (right).

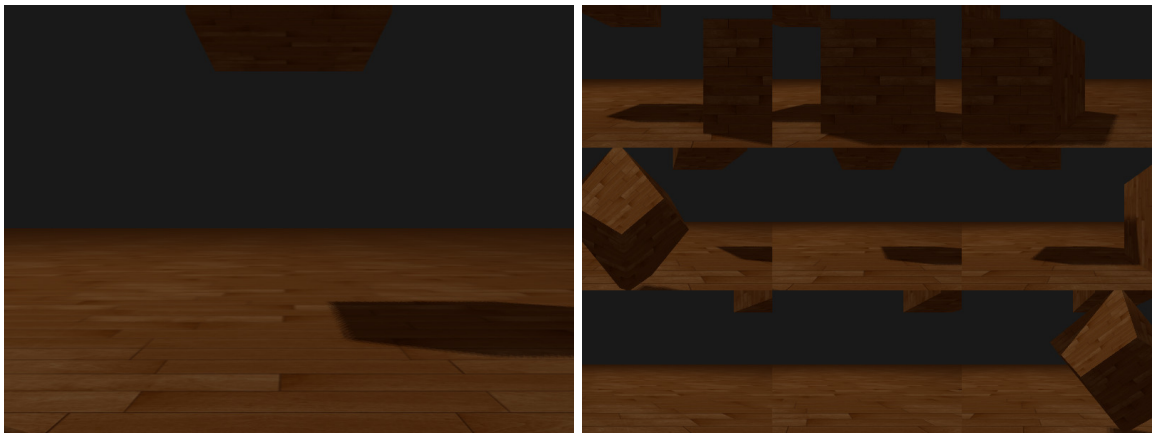


Figure 6.2: Pairs of images, showing original application (left) and resulting quilt (right). This example demonstrates ability to handle the applications, which use shadow mapping

Figure 6.3: Demonstration of skybox rendering.

Quilt/Resolution	128^2	256^2	512^2	1024^2	2048^2	4096^2
1x1	16.5ms	16.5ms	16.5ms	16.5	16.5ms	16.5ms
3x3	63.8ms	70.3ms	71.5ms	68.2ms	79.6ms	80.6ms
5x9	263.9ms	271.0ms	284.5ms	307.0ms	343.6ms	400.5ms

(a) Stanford Dragon (complex geometry)

Quilt/Resolution	128^2	256^2	512^2	1024^2	2048^2	4096^2
1x1	16.5ms	16.5ms	16.5ms	16.5ms	16.5ms	16.5ms
3x3	16.5ms	16.5ms	16.5ms	20.6ms	36.0ms	56.4ms
5x9	19.4ms	32.3ms	50.2ms	69.5ms	93.1ms	166.5ms

(b) Steep Parallax Mapping scene (complex shading)

Quilt/Resolution	128^2	256^2	512^2	1024^2	2048^2	4096^2
1x1	16.1ms	16.2ms	16.1ms	16.5ms	16.4ms	16.4ms
3x3	16.5ms	16.5ms	16.5ms	16.5ms	16.5ms	24.7ms
5x9	16.5ms	16.1ms	24.2ms	37.9ms	65.6ms	115.6ms

(c) Cubes (simple geometry)

Quilt/Resolution	128^2	256^2	512^2	1024^2	2048^2	4096^2
1x1	21.3ms	22.5ms	21.0ms	21.3ms	21.8ms	24.5ms
3x3	23.4ms	22.1ms	26.0ms	25.8ms	32.7ms	44.8ms
5x9	67.3ms	74.5ms	86.0ms	105.3ms	144.3ms	196.4ms

(d) Asteroids (many trivial draw calls)

Table 6.1: Average frame period. Lower is better. Quilt (number of views) vs resolution (width/height of each of view). Test setup: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz and NVIDIA GeForce GT 920M.

that the dependency on resolution could be measured. Finally, *Asteroids* provide hundreds of draw calls with trivial geometry

Clearly, it can be concluded that the injection and rendering into multiple views affect performance. In the case of complex geometry, but cheap shading, the performance does not depend much on the resolution, because the graphics card's pipeline is busy with the transformation of geometry.

Contrary, in case of expensive shading, the period increases less than twice when the resolution is doubled. While keeping same number of views, the frame period is correlated with increasing the resolution.

In more complex geometric scenes, the performance directly depends on the number of views. In some non-optimized scenes, such as Asteroids, which dispatch many draw calls with trivial meshes, performance does not decrease linearly with number of views initially, because instancing of Geometry Shader can help in overcoming the overhead of many draw calls with simple geometry.

In conclusion, the exact effect on performance depends on **scene's complexity** and **complexity of shading**. For simple applications and lower resolutions, the effect can be minimal, in some cases neglectible. Note that contemporary Looking Glass Display use

Quilt/Resolution	256^2	512^2	1024^2	2048^2	4096^2
1x1	0.72ms	0.75ms	0.78ms	0.8ms	1.49ms
3x3	1.67ms	1.76ms	2.46ms	7.5ms	40.6ms
5x9	6.72ms	11.1ms	25.0ms	265.6ms	300.5ms

(a) Stanford Dragon (complex geometry)

Quilt/Resolution	256^2	512^2	1024^2	2048^2	4096^2
1x1	0.17ms	0.21ms	0.21ms	0.21ms	0.4ms
3x3	0.21ms	0.21ms	0.29ms	0.78ms	4.5ms
5x9	0.19ms	0.42ms	1.05ms	10.6ms	40.5ms

(b) Steep Parallax Mapping scene (complex shading)

Quilt/Resolution	256^2	512^2	1024^2	2048^2	4096^2
1x1	0.18ms	0.18ms	0.18ms	0.18ms	0.32ms
3x3	0.17ms	0.19ms	0.24ms	0.64ms	4.71ms
5x9	0.21ms	0.33ms	0.99ms	12.7ms	57.5ms

(c) Cubes (simple geometry)

Quilt/Resolution	256^2	512^2	1024^2	2048^2	4096^2
1x1	10.3ms	10.3ms	10.4ms	10.5ms	10.6ms
3x3	10.2ms	10.3ms	10.4ms	10.3ms	10.3ms
5x9	10.1ms	10.7ms	10.3ms	34.9ms	118.5ms

(d) Asteroids (many trivial draw calls)

Table 6.2: Average frame period. Lower is better. Quilt (number of views) vs resolution (width/height of each of view). Test setup: Threadripper 1920X 3.5GHz (12/24) and NVIDIA 2x2080Ti.

2500 x 1600 LCD, which results in 4MPix. A 5x9 quilt with 512x512 texture size per view exceeds 11 MPix. However, the resolution of displays is expected to increase in order to lift antialiasing problem. For instance, the brand new Looking Glass 8k Display uses 7680 x 4320 LCD, which equals to 33 MPix [5]. To feed the display, 1024x1024 quilt must be used which equals to 47 MPix.

6.3 Limitations

The limitations of the presented method can be divided into two categories:

- **Lack of information**
  Many problems stem from ambiguities and limited knowledge of black box conversion.
- **Robustness of implementation**
  Due to limited time to deliver, implementation trade-offs are causing failures in a few specific use-cases.

6.3.1 Drop of FPS

Due to design of autostereoscopic displays, all views of quilt must be rendered in each frame, resulting to significant performance decrease. The performance of a graphical application is typically addressed in *frames per second* (FPS).

A naïve approach which renders all views to layered FBOs with the same size of texture as the original FBO will take at least N times more time to render, where N equals to count of frame. For example, if the original application was providing smooth 100FPS and the display has 45 unique views, the conversion would achieve 2 to 3 FPS, effectively removing interactivity from the application.

To ease this problem, the resolution of layered FBOs can be decreased as typical displays does not provide much huger internal screens than flat LCDs. The resolution of drawing target affects the count of needed shading operations.

If the original application is not bounded by amount of transformations, the use of instancing in Geometry Shaders may help to reduce the amount of OpenGL calls and fill the pipeline more efficiently.

Tracking the position of viewer's eyes

Alternatively, so-called *View-Dependent Light Field Displays* has started emerging recently [35], which actively track positions of user's eyes and thus only require two frames per frame, one per each eye. Instead of rendering the full quilt, only two images are thus needed. Therefore, the duplicating part of draw calls could be altered to draw only for images, specified by bitmask in each frame.

This would reduce the drop of frames only to half of the original's application frame count. Note that even if these displays render only two images, classical stereoscopic conversion algorithms can not be used, because these two view may have a strong off-axis transformation with respect to the original view.

6.3.2 Flatness of HUD

Any rendering technique which projects 3D positions to screen and pass such position to transformation instead of transformation matrix itself are limited to flat 2D rendering due to missing spatial information. Naturally, this is mostly the case for HUD, but in addition, this may affect billboarding as well.

6.3.3 Frustum culling

The conversion layer works over subspaces of volume, provided in draw calls in current frame. A typical optimized rendering engine will employ techniques to skip drawing of the meshes which are invisible from the application's point of view to prevent pipeline operations which does not affect the resulting image, but consume resources. An example is illustrated in Figure 6.4.

Also, during the era of fixed-pipeline rendering, such optimization was even implemented over parts of mesh as geometry was uploaded to GPU in each draw call, and thus more drastic clipping significantly improved the performance.

Luckily, any mesh, which at least partially affects the resulting image, is typically let to be rendered by modern engines, which use programmable pipeline, and clipping is solved by GPU implicitly, as the geometry is already pre-stored at GPU during draw calls.

Figure 6.4: An example of frustum calling in combination with conversion of application *Counter-Strike 1.6*. The left-most, the original and the right-most images clearly show missing geometry (white areas), removed by CPU-side culling.

This issue may affect side views of quilt due to missing information in draw calls.

6.3.4 Shading transformations

Applications implement shading in fragment shader based on angles or positions of lights and position of view. In order to achieve shading which corresponds to the correct shifted position of view, these must be altered as well. However, as no standard exists for passing such data to fragment shader, more complex analysis is required to understand which outputs of transformation pipeline must be changed, and this typically fails.

This limitation is clearly visible when rendering reflective materials, resulting in improper specular reflections, and can be perceived in Figure 6.5.

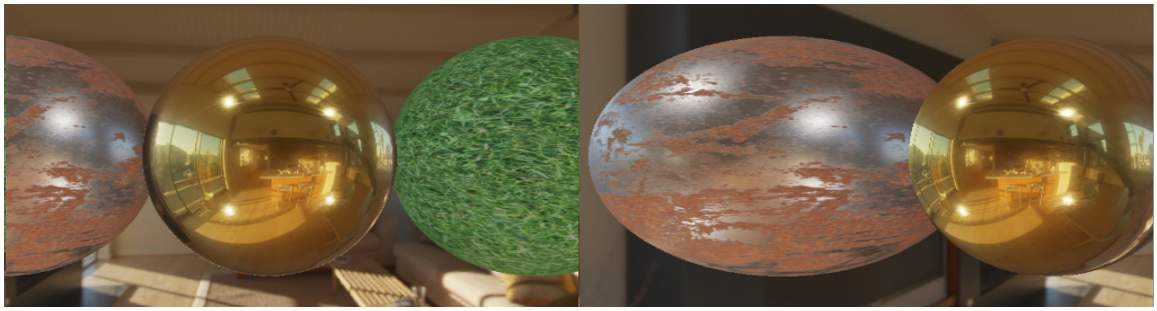


Figure 6.5: The same reflection on golden sphere's surface in side view (right) as in the front view (left), caused by missing propagation of altered transformation matrix to computation of camera-space position and pixel's normal in Fragment Shader.

This could be solved by allowing experienced users to manually edit the injected transformation shader. Changes could be associated permanently with specific shader by hashing content of the original shader.

6.3.5 Complexity of programmable shaders

We currently use *regular expressions* to automatically extract metadata about used uniforms and operations in shaders. This results in failures of detection in complex applications, which may use variable shadowing or if-else branching. We believe this could be improved by using a proper GLSL parser and more complex analysis.

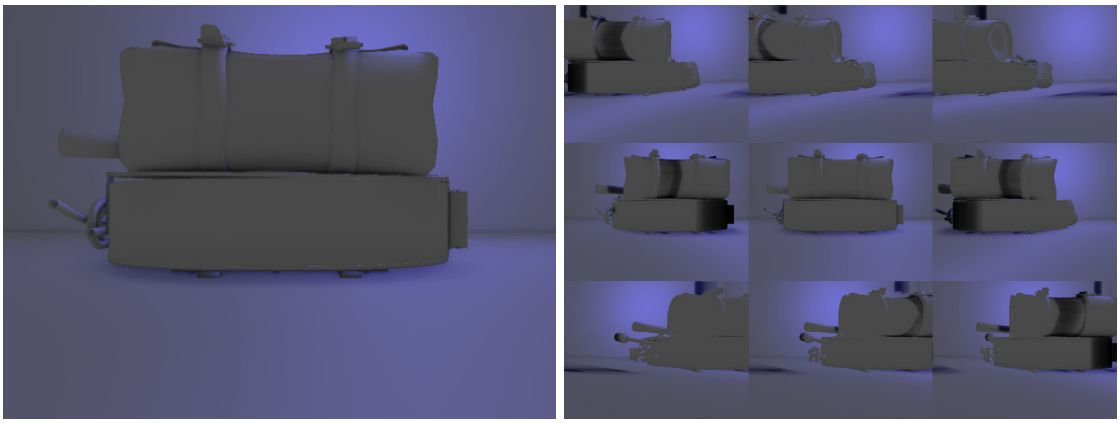


Figure 6.6: Ample shows failure due to technique SSAO, which uses transformations in Fragment Shaders. All examples were converted automatically, and originate from *LearnOpenGL*'s repository.

Chapter 7

Conclusion

This thesis describes the process of designing and implementing software for semi-automated conversion of 3D OpenGL applications to produce the output, suitable for autostereoscopic displays. The major contribution of the thesis is in describing, designing and implementing the process of conversion to multiview.

The resulting convetor is able to automatically convert simple OpenGL applications correctly. This was tested using open-source OpenGL tutorials, which implement various rendering techniques.

The implementation was tested by converting example applications and the performance was measured. In conclusion, the performance varies, and it depends mostly on complexity of the original scene. In the case of suboptimal rendering, the use of Geometry Shader together with instacing can compensate the overhead of the original application, and decrease the performance loss.

This thesis has been succesfully presented at the student conference Excel@FIT VUT in the form of paper. The paper was awarded by scientific commision and the conference's commercial partner. The resulting source codes have been open-sourced at the popular public code repository GitHub.

Future improvements

The implementation of the thesis is rather a proof of concept than a finished product. More complex applications may lead to crashing or invalid visual output due to missing hooks for less frequently used OpenGL calls and extensions.

In the future, these problems could be overcomed with more thorough testing and man hours invested in verification. A few suggestions are given in Section C.1. The robustness could also be improved by using a proper GLSL parser for extending the pipeline and by supporting more platforms such as Windows or MacOS, porting the convertor to DirectX, or supporting a different window manager from X Window (such as Wayland). In addition, the number of additional OpenGL API calls per a single original API call was not optimized and additional reasonable improvement could be achieved here by *caching state changes* and using *bindless API calls* for named OpenGL objects.

With a few minor changes, the implementation could also be used for converting the applications to stereo for VR at Linux platform. As most of the similar convertors are implemented for Microsoft Windows, this platform could benefit from such a tool.

Bibliography

[1] *Camera - Looking Glass Documentation*. Accessed: 2021-5-07. Available at: https://docs.lookingglassfactory.com/keyconcepts/camera/.

[2] *Google Cardboard*. Google. Accessed: 2021-05-07. Available at: https://arvr.google.com/cardboard/.

[3] *HoloPlay Core SDK - Looking Glass Documentation*. Accessed: 2021-05-07. Available at: https://docs.lookingglassfactory.com/holoplay-core/index.

[4] *Load OpenGL Functions*. Accessed: 2021-4-25. Available at: https://www.khronos.org/opengl/wiki/Load_OpenGL_Functions.

[5] *Looking Glass 8K*. Looking Glass Factory. Accessed: 2021-5-17. Available at: https://lookingglassfactory.com/8k#specs.

[6] *Neil's News*. Accessed: 2021-05-07. Available at: https://neil.fraser.name/news/2010/04/18/.

[7] *Oculus Rift*. Accessed: 2021-05-07. Available at: https://www.oculus.com/rift/.

[8] *Quilts - Looking Glass Documentation*. Accessed: 2021-5-07. Available at: https://docs.lookingglassfactory.com/keyconcepts/quilts/.

[9] *Steam Overlay (Steamworks Documentation)*. Accessed: 2020-04-25. Available at: https://partner.steamgames.com/doc/features/overlay.

[10] *Supported Games - vorpX - VR 3D-Driver for Oculus Rift*. Accessed: 2021-4-25. Available at: https://www.vorpx.com/supported-games/.

[11] *VR Conversion for non-VR games*. Accessed: 2021-4-25. Available at: https://www.trinusvirtualreality.com/vr-conversion-for-non-vr-games/.

[12] *The OpenGL©Graphics System:A Specification - (Version 3.0)*. The Khronos Group Inc., September 2008. Available at: https://www.khronos.org/registry/OpenGL/specs/gl/glspec30.pdf.

[13] *The Voxon VX1, 3D Volumetric Display now commercially available*. Jul 2017. Accessed: 2021-4-25. Available at: https://www.opli.net/opli_magazine/eo/2017/the-voxon-vx1-3d-volumetric-display-now-commercially-available-july-news/.

[14] *Features - vorpX - VR 3D-Driver for Oculus Rift*. Apr 2018. Accessed: 2021-4-25. Available at: https://www.vorpx.com/features/.

[15] *The OpenGL©Graphics System:A Specification - (Version 4.6)*. The Khronos Group Inc., October 2019. Available at: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[16] *Dlsym(3) — Linux manual page*. Kernel.org, November 2020. Accessed: 2020-10-01. Available at: https://man7.org/linux/man-pages/man3/dlsym.3.html.

[17] *Ld.so(8) — Linux manual page*. Kernel.org, November 2020. Accessed: 2020-10-01. Available at: https://man7.org/linux/man-pages/man8/ld.so.8.html.

[18] Algorri, F., Urruchi, V., García Cámara, B. and Sánchez Pena, J. Liquid Crystal Microlenses for Autostereoscopic Displays. *Materials*. january 2016, vol. 9, p. 36. DOI: 10.3390/ma9010036.

[19] Azuma, R. T. A Survey of Augmented Reality. *Presence: Teleoper. Virtual Environ.* Cambridge, MA, USA: MIT Press. august 1997, vol. 6, no. 4, p. 355–385. DOI: 10.1162/pres.1997.6.4.355. ISSN 1054-7460. Available at: https://doi.org/10.1162/pres.1997.6.4.355.

[20] Beazley, D. M., Ward, B. D. and Cooke, I. R. The inside Story on Shared Libraries and Dynamic Loading. *Computing in Science and Engg.* USA: IEEE Educational Activities Department. september 2001, vol. 3, no. 5, p. 90–97. DOI: 10.1109/5992.947112. ISSN 1521-9615. Available at: https://www.dabeaz.com/papers/CiSE/c5090.pdf.

[21] Bianco, S., Ciocca, G. and Marelli, D. Evaluating the Performance of Structure from Motion Pipelines. *Journal of Imaging*. august 2018, vol. 4, p. 98. DOI: 10.3390/jimaging4080098.

[22] BlueSkyDefender. *GitHub: BlueSkyDefender/Depth3D*. Accessed: 2021-02-10. Available at: https://github.com/BlueSkyDefender/Depth3D.

[23] Broxton, M., Flynn, J., Overbeck, R., Erickson, D., Hedman, P. et al. Immersive Light Field Video with a Layered Mesh Representation. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. july 2020, vol. 39, no. 4. DOI: 10.1145/3386569.3392485. ISSN 0730-0301. Available at: https://doi.org/10.1145/3386569.3392485.

[24] Buck, B. and Hollingsworth, J. K. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* USA: Sage Publications, Inc. november 2000, vol. 14, no. 4, p. 317–329. DOI: 10.1177/109434200001400404. ISSN 1094-3420. Available at: https://doi.org/10.1177/109434200001400404.

[25] Cass, S. *3-D TV is Officially Dead (For Now) and This is Why it Failed*. IEEE Spectrum, Jan 2014. Accessed: 2021-04-25. Available at: https://spectrum.ieee.org/tech-talk/consumer-electronics/audiovideo/3d-tv-is-officially-dead-for-now-and-this-is-why-it-failed.

[26] Chang, A., Choi, J. and Yu, K. Ghosting reduction method for color anaglyphs. february 2008, vol. 6803. DOI: 10.1117/12.766422.

[27] Damato, J. *How does strace work?* Feb 2016. Accessed: 2021-04-25. Available at: https://blog.packagecloud.io/eng/2016/02/29/how-does-strace-work/.

[28] Favalora, G. Volumetric 3D Displays and Application Infrastructure. *Computer*. september 2005, vol. 38, p. 37 – 44. DOI: 10.1109/MC.2005.276.

[29] Grunin, L. *Looking Glass holographic display lets you interact with 3D creations like you're in VR – without the headset.* CNET, 24. july 2018. Available at: https://www.cnet.com/reviews/looking-glass-preview/.

[30] Holliman, N. 3D Display Systems. december 2002, vol. 38.

[31] Kellnhofer, P., Didyk, P., Wang, S.-P., Sitthi Amorn, P., Freeman, W. et al. 3DTV at Home: Eulerian-Lagrangian Stereo-to-Multiview Conversion. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. july 2017, vol. 36, no. 4. DOI: 10.1145/3072959.3073617. ISSN 0730-0301. Available at: https://doi.org/10.1145/3072959.3073617.

[32] Kovács, P. and Balogh, T. 3D Visual Experience. In:. January 2010, p. 391–410. DOI: 10.1007/978-3-642-12802-8_17.

[33] Lang, B. *Analysis: Monthly-connected VR Headsets on Steam Pass 2 Million Milestone*. Road to VR, Jan 2021. Accessed: 2021-04-25. Available at: https://www.roadtovr.com/steam-survey-vr-monthly-active-user-2-million-milestone/.

[34] Levoy, M. and Hanrahan, P. Light Field Rendering. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1996, p. 31–42. SIGGRAPH '96. DOI: 10.1145/237170.237199. ISBN 0897917464. Available at: https://doi.org/10.1145/237170.237199.

[35] Li, T., Huang, Q., Alfaro, S., Supikov, A., Ratcliff, J. et al. Light-Field Displays: A View-Dependent Approach. In: *ACM SIGGRAPH 2020 Emerging Technologies*. New York, NY, USA: Association for Computing Machinery, 2020. SIGGRAPH '20. DOI: 10.1145/3388534.3407293. ISBN 9781450379670. Available at: https://doi-org.ezproxy.lib.vutbr.cz/10.1145/3388534.3407293.

[36] Lopez, J., Babun, L., Aksu, H. and Uluagac, S. A Survey on Function and System Call Hooking Approaches. *Journal of Hardware and Systems Security*. september 2017, vol. 1. DOI: 10.1007/s41635-017-0013-2.

[37] Martin S. Banks, R. S. A. and Watt, S. J. Stereoscopy and the Human Visual System. *SMPTE motion imaging*. may 2012. DOI: 10.5594/j18173.

[38] Mehrabi, M., Peek, E. M., Wuensche, B. C. and Lutteroth, C. Making 3D Work: A Classification of Visual Depth Cues, 3D Display Technologies and Their Applications. In: *Proceedings of the Fourteenth Australasian User Interface Conference - Volume 139*. AUS: Australian Computer Society, Inc., 2013, p. 91–100. AUIC '13. ISBN 9781921770241.

[39] Premecz, M. Iterative parallax mapping with slope information. In: *In Central European Seminar on Computer Graphics*. 2006, p. 2006.

[40] Robertson, G., Card, S. and Mackinlay, J. Three views of virtual reality: nonimmersive virtual reality. *Computer*. march 1993, vol. 26, p. 81. DOI: 10.1109/2.192002.

[41] Ruijters, D. Integrating autostereoscopic multi-view lenticular displays in minimally invasive angiography. january 2008.

[42] SELinuxProject. *SELinuxProject/selinux*. Available at: https://github.com/SELinuxProject/selinux.

[43] Shawn Frayne, S. P. L. *Superstereoscopic display with enhanced off-angle separation*. 2019. US10298921B1. Available at: https://patentimages.storage.googleapis.com/c4/16/67/f22fae9bc3a003/US10298921.pdf.

[44] Urey, H., Chellappan, K. V., Erden, E. and Surman, P. State of the Art in Stereoscopic and Autostereoscopic Displays. *Proceedings of the IEEE*. 2011, vol. 99, no. 4, p. 540–555. DOI: 10.1109/JPROC.2010.2098351.

[45] Yin, H., Liang, Z. and Song, D. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In:. January 2008.

[46] Zhou, Y., Guo, H., Fu, R., Liang, G., Wang, C. et al. 3D reconstruction based on light field information. In:. August 2015, p. 976–981. DOI: 10.1109/ICInfA.2015.7279428.

Appendix A

Additional results of conversion

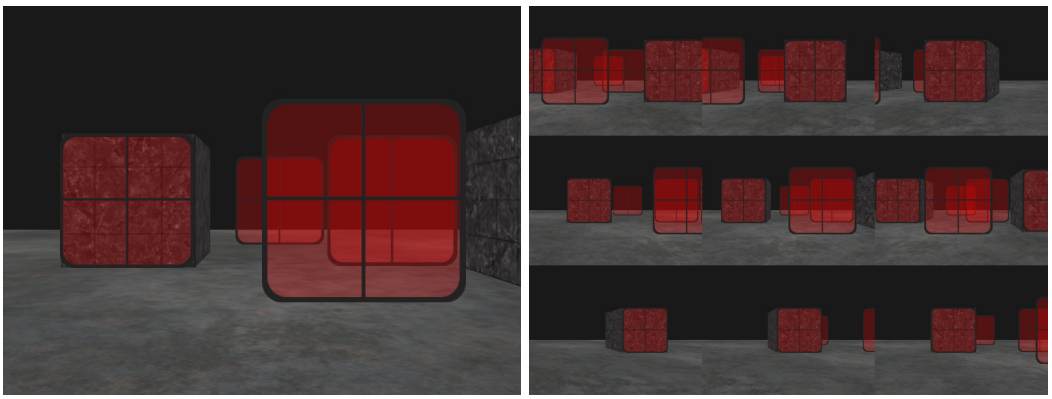


Figure A.1: Blending sort (LearnOpenGL)

Figure A.2: Cubemap environment mapping (LearnOpenGL): note invalid position of camera in reflections.

Figure A.3: Scene Cubes, displayed using Looking Glass display.

Appendix B

HoloInjector's Manual

B.1 Installation

Up-to-day version of HoloInjector can be found at website Romop5/holoinjector@GitHub.com.

Building of **HoloInjector** is based on CMake system. Prior to building, it's necessary to get and build the prerequisite libraries. This is done by `cloneDeps.sh`. The script creates a new directory `3rd` within `holoinjector`, and subdirectory `install` with results of building.

Next, the configuration must be generated in `build` directory with links to prerequisites.

Finally, HoloInjector is built by entering `build` directory, and entering command:

```
$ ccmake ../ && make
```

Upon sucessufull build, the resulting `libHoloInjector.so` is placed in `build` directory.

B.2 Acquiring parameters of display for Looking Glass

The parameters of display could be extracted using two ways. Firstly, a specialized Holo Play SDK is available, which allows querying of device parameters. Secondly, the parameters can be extracted using a USB protocol and by decoding USB packets. An experimental implementation by Ing. Tomáš Milet is available at Github [1].

B.3 Using HoloInjector to obtain 3D image

In order to start the injection, use `LD_PRELOAD` with absolute path to `libHoloInjector.so`.

An example of running arbitrary application from it's directory with the injector:

```
$ LD_PRELOAD=/path/to/libHoloInjector.so ./app <app-specific-arguments>
```

Upon sucessfull loading, the application is in so-called **Normal mode**. In normal mode, the rendering pipeline is not affected by the injector and the output of application should not differ from the original application.

In order to see display-native image, **Duplication mode** has to be reached. This is done by pressing key. See B.3 for specific keys.

[1] https://github.com/dormon/3DApps/blob/master/src/holoCalibration.cpp

Important key bindings

- **F1/F2**
  Increment/decrement horizontal offset.
- **F3/F4**
  Increment/decrement shear coefficient.
- **F5** Reset horizontal offset & shear.
- **F10**
  Sends an event to X11 to hide the upper decoration of the window (needed for applications, running in window).
- **F11**
  In Normal mode, pressing this key toggles **Overlay menu**.
- **F12**
  Toggles between Normal Mode and Duplication mode.

B.3.1 Enviromental variables

To quickly experiment with various parameters, enviromental variables come handy. In the following list, many of variables allow changing of internal parameters of duplication, initiate actions upon start, or allow interesting diagnosis.

- **HI_XMULTIPLIER**
  Initial horizontal shift.
- **HI_DISTANCE**
  Initial distance of focus plane
- **HI_NOW**
  Start the application already in duplication mode.
- **HI_QUILT**
  Display quilt instead of native image in duplication mode.
- **HI_WIDE**
  Set initial shift/distance to some non-zero value.
- **HI_QUILTX**
  Override number of quilt X views (by default: 5).
- **HI_QUILTY**
  Override number of quilt Y views (by default: 9).
- **HI_FBOWIDTH**
  Override width of a single view (e.g. 128/256/512).
- **HI_FBOHEIGHT**
  Override height of a single view (e.g. 128/256/512).

- **HI_EXIT_AFTER**
  Useful for diagnostic: force kill of application after N frames. Screenshot is taken as well. The format of screenshot name can be affected by `HI_SCREENSHOT`.
- **HI_CAMERAID**
  In duplication mode, show output of camera with ID.
- **HI_SCREENSHOT**
  Format of screenshot. See code for more details.
- **HI_NONINTRUSIVE**
  Don't inject shaders/programs. Useful for FPS measuring.
- **HI_RUNINBG**
  Prevent Show() of X11 window. Useful for batch scripts.
- **HI_RECORDFPS**
  After each frame's swap, print frame period time (in microseconds).

B.3.2 Configuration

Apart from enviromental variables, HoloInjector can also accept config-based inputs at predefined directories. It supports a priority loading: the top-level priority has a config in the current working directory, named `holoinjector.cfg`. Next, the path `/.config/holoinjector/holoinjector.cfg` is used. Finally, the settings can be specified in system-wide config in `etc/holoinjector.cfg`.

The list of the YAML config keys is identical to the list of enviromental variables, except for case of letters. Therefore, the list is given without further descriptions: „xmultiplier“, „distance“, „now“, „quilt“, „wide“, „quiltX“, „quiltY“, „fboWidth“, „fboHeight“, „exitAfter“, „cameraID“, „screenshot“, „nonIntrusive“, „runInBg“, „recordFPS“, „vertex“

Overriding pipeline injector

It is possible to overide automatic settings, detected by the pipeline injector when analyzing shader program. This is done by creating a YAML configuration file, named as a hash of the shader. Hashes can be found using Inspector. The path to configuration files is the current working directory by default.

- **transformationMatrixName**
  Overrides the detected transformation matrix. Useful for complex shaders which may use functions to compute `gl_Position`.
- **shouldMakeProgramInvisible**
  Prevents rendering of the geometry using this shader program. Useful if the conversion fails, and one would like to permanently disable the invalid produced geometry.

B.3.3 In-application menu

The purpose of In-application menu is to control various parameters in run-time and to trigger futher actions such as forcing window to turn to a fullscreen or to show an inspector.

Figure B.1: Overlay menu.

Appendix C

For further developers

Debugging an application with complex set of states and inputs can be hard to manage. This section thus attempts to give reader a few tips which were empirically expererienced during creation of this thesis.

C.1 Testing the injector

Typical software projects involve tests at different levels, be it unit tests or system tests, to provide a mechanism for making sure that the quality and stability of project does not decrease over time.

Creating tests, however, requires developer to separate functionality into isolated units with well-defined functionality. This has proven not to be feasibile while developing HoloInjector. In order to test parts of the injector which maps OpenGL calls to different ones, one would have to mock up a whole OpenGL driver with respect to OpenGL specification.

C.1.1 Regression testing

As such task was unbearable and costly for a one-year job, different approach was choosen. In order to test the project, one has to use existing applications such as OpenGL tutorials to feed the injector with. Regression is then achieved as manual comparision of visual results between different commits.

This applies to parts of code which immediately use OpenGL or which call other components. The rest of code consist of algorithms and data structures, which can be isolated and tested separately, so the project also employes limited regression testing using unit tests.

C.1.2 Idea: A concept of automated testing

As such manual regression testing becomes soon repetitive, this text comes up with a concept of automatization, which was not implemented due to time constrains, but the injector contains at least parts of infrastructure, serving as building blocks for further developement of testing.

The idea of automated testing over existing applications is following: most of applications are deterministic themselfs. The stochasticity raises from external sources such as random number generators or time clocks.

In order to test regression, one could possibly identify contemporarily working examples for which injector gives correct visual results. Next, one or multiple screenshots under use of the injector would be created and stored. Afterwards, this process would repeat each time a regression test is run, comparing the most recent screenshot with the reference one. Provided the same rendering machine, graphics card and driver is used for rendering both reference and tested image, the comparison of two same-resolution non-compressed images should give exact match. In practise, a sum of absolute differences would be used with reasonable epsilon for introducing robustness due to possible errors, caused by various defects such as hard-disk errors or non-deterministic outputs of renderer itself.

For applications, which depends on stochascity of external components, a specific dynamic library would be created which would overload such stochastic generators with deterministic ones. For instance, one can replace random number generator with a well defined sequence of numbers, or system clock with a monotonous increasing function.

C.2 Tips for debugging

Due to lack of code coverage via unit tests, debugging of the injector stands for a serious challenge with identifying the smallest example of existing applications.

Even if you manage to find such application, understanding the problem relies on getting a deep insight behind the scenes.

This section provides a few techniques to start with when feeling overwhelmed of the problem's complexity.

C.2.1 Exploiting `git bisect` for fun and profit

When doing many small incremental changes, a mistake or logical error can be simply identified even without having look into code explicitely.

This can be achieved by the tool named `git bisect` which allows us to quickly identify the first commit which introduced the change which broke the code.

This technique is, however, only usable if the error happened after introducing change somewhere in the past.

The usage is pretty straightforwad. At first, mark the current (or the commit you know that does not work) using a command `git bisect bad`. Next, just find the last commit for which the code you are trying to debug was working well, and mark it as a `git bisect good`. The tool will lead you over each commit in logarithmic manner.

The user's experience can be made even smoother with creating a command or script file with command which change directory to build, builds the project, starts the test program, and finaly let you decide whether the current commit is good or bad.

C.2.2 Application overlay

Upon succesfully loading the injector into application, a overlay GUI should be available with possibility of manipulating with parameters used for the injection, as described in user's manual.

In addition, the injector also provides *an inspector* - a graphical widget, which dumps the information tracked by the injector.

Currently, inspector shows following tracked entities:

- **shader programs & shaders**
  The inspector allows you to list intercepted programs and the attached shaders. For each program, the displayed metadata includes state of injection or linkage, giving you insight whether program is correctly drawing. When a problematic program is identified, the attached shaders can be investigated to find out what sort of shaders result into failure of injection into transformation pipeline. The inspector also allows to make shader/program invisible to help you identify the geometry that the program is responsible for being drawn.

- **frame buffer objects**
  The inspector shows tracked FBOs together with their attachements. If attachment is a 2D texture, the texture is rendered, giving you insight whether rendering is correct.

C.2.3 Using different logger's levels

The injector differentiates between several levels of logging.

- **error level**
  This level includes messages of fatal errors which will affect the visual result with high probability. Such errors mostly happen once a time, typically when a resource is started to be tracked or when creating an additional resource (for instance shadowing texture) with improper format.

- **info level**
  Regular message logs and possibly warnings, which does not affect rendering result, but gives insight about the current state of injector on load etc.

- **debug level**
  The purpose of debug level is to give developer more information about the state of injector on events which does not happen regularly. This may include the creation of resource as well, but with more detailed information, including hints on reasons why different paths in decisions of the injector were taken.

- **per-frame debug level**
  This level includes log messages on events that happen repeatedly in every frame, such as draw calls or system message events. The purpose is to give a simplified exhaustive log of important procedure calls, dispatched by the injector. For a complete trace of OpenGL's calls, it's advised to use an utility for API tracing. See section C.2.4 for more details.

C.2.4 API tracing

The most descriptive insight into the state of calls to OpenGL can be achieved by using so-called `API tracer`. API tracers are tool which inserts a wrapping layer over underlying API interface and trace any API call to underlying interface during the run-time.

The trace is recorded and can be analyzed offline using specialized tools, which allows to see parameters of specific API calls. In addition, the OpenGL's state and resources at the time of API call can be analyzed and inspected, and violations of OpenGL's specifications together with programming errors are reported as well.

Such tools are ultimate solution for solving problems such as black screen despite logs and code give you idea of correct resource binding.

The major disadvantage is the count of API calls. For typical application with many meshes, multiple draw passes and the injector working, a single frame can have up to several thousands of API calls. This may significantly increase the time for replayer to look-up the state for given API call.

Even then, the trace will not show the internal state of the injector. This can be partialy relieved with disusing OpenGL's string functions such as `glGetUniformLocation` and providing log message as a string parameter at the cost of false alarms in the log of errors.

Appendix D

Content of enclosed CD

```
├── 3rd
│   ├── glm-master.zip
│   ├── imgui-master.zip
│   └── simplecpp-master.zip
├── holoinjector
│   ├── build
│   ├── cloneDeps.sh
│   ├── cmake
│   ├── CMakeLists.txt
│   ├── LICENSE
│   ├── media
│   ├── README.md
│   ├── src
│   └── tests
├── holoinjector-tests
│   ├── dataset_helpers
│   ├── examplesList.txt
│   ├── fpsGenerator.sh
│   ├── fpsResults
│   ├── generateReadme.sh
│   ├── getDataset.sh
│   ├── README.md
│   ├── results
│   └── runDataset.sh
├── README.md
├── thesis
├── video.mp4
├── video1.mp4
├── video2.mp4
├── video3.mp4
├── video4.mp4
└── xdobia11.pdf
```