

communicated by:
Lehr- und Forschungsgebiet Informatik 9

Prof Dr. Ulrik Schroeder



The present work was submitted to the Learning Technologies Research Group
Diese Arbeit wurde vorgelegt am Lehr- und Forschungsgebiet Lerntechnologien

Development of an immersive VR simulation for middle school optics lessons
Entwicklung einer immersiven VR-Simulation für den Optikunterricht in der Mittelstufe

Master-Thesis

Masterarbeit

presented by / von

Möhring, Tobias

345522

Examined by / Begutachtet von

Prof. Dr.-Ing. Ulrik Schroeder

Prof. Dr. phil. Bianca Watzka

Supervised by / Betreut von

Dr. Birte Heinemann

Aachen, November 27, 2025

Contents

List of Figures	iii
List of Tables	v
List of Listings	vi
1 Introduction	2
1.1 Motivation	2
1.2 Virtual Reality in Schools	3
1.2.1 Limitations	3
1.2.2 Chances	3
1.3 Structure	4
2 Related Work	6
2.1 RePiX VR	6
2.2 Other Academic works	7
2.3 Simulation Games	7
3 Technology	9
3.1 State of VR Development	9
3.2 Hardware	11
3.2.1 Mobile vs. Stationary XR	11
3.2.2 PICO Neo3 Pro Eye	13
3.3 Development Environment	13
3.3.1 WebXR	14
3.3.2 Game Engines	16
3.3.3 Unity	20
3.4 Version Control	26
4 Design and Implementation	29
4.1 Overview & Hub	29
4.1.1 The UFO	29
4.1.2 Learning Topics	30
4.1.3 Guided Tour	31
4.2 Solar System	34
4.2.1 Outline	34
4.2.2 Orbits and Planet Data	35
4.2.3 Approximating Orbital Motion	39

4.2.4 Planet Data Management and Pre-processing	43
4.2.5 Shader-based Planets	45
4.2.6 Sky Box	51
4.3 Shadow Volumes	54
4.3.1 Outline	55
4.3.2 Limits and Abstractions	57
4.3.3 Shadow Rendering Methods	58
4.3.4 Mesh Preparation	60
4.3.5 Shadow Volume Generation	63
4.3.6 Caveats	68
4.4 Sun Coins	68
4.4.1 Outline	69
4.4.2 Implementation	70
4.4.3 Future Work	72
5 Evaluation	74
6 Conclusion	76
6.1 Challenges and Limitations	76
6.2 Future Work	77
6.3 Conclusion	78
Appendix	78
A Bibliography	79
B Digital Appendix	83

List of Figures

2.1 While there are many educational scenarios in <i>Universe Sandbox</i> , some take on sillier tones such as colliding multiple “spherical cows” into the surface of the moon.	8
3.1 HTC Vive Controllers	14
3.2 Relationships of Unity objects	20
3.3 Screenshot of the Hierarchy window	21
3.4 Screenshot of a component in the inspector window	21
3.5 Simplified implementation of Lambertian diffuse lighting.	25
4.1 A view of the cockpit.	30
4.2 A view of the partially shaded model earth. The white spot marks the current viewing location in europe.	34
4.3 Sky box shader on the inside of an inverted cube around ship. This perspective is usually not visible to users.	35
4.4 Real-world distances compared to hypothetic model scale.	36
4.5 Ellipse geometry: semi-major axis in red, semi-minor in blue	37
4.6 Illustration of eccentric and true anomaly	40
4.7 Basis vectors displayed over user position on globe.	44
4.8 UV Mapping of standard unity sphere mesh (top) compared to custom ray tracing shader (bottom).	46
4.9 Ray intersection, rendering a spherical object onto a cube mesh. Intersection points with the sphere in red and actual fragment position in green	47
4.10 Different components that make up the lighting step.	49
4.11 Direct comparison of base shader and different parameters for the gain function.	50
4.12 Brocken Spectre, Golden Gate Bridge From Brocken Inaglory, <i>Solar glory and Spectre of the Brocken from GGB on 07-05-2011</i> , Wikimedia Commons, link, CC BY-SA 3.0	55
4.13 Sun rays From Kiril Simeonovski <i>Crepuscular rays before sunset</i> , <i>Tikveš</i> , Wikimedia Commons, link CC BY 4.0	55
4.14 Shadow Volume scene setup with height-adjustable lamp to the left and moveable sphere (or other shape) to the right.	55
4.15 Control elements that users can interact with to affect the shadow volume simulation.	56

4.16 Screenshot of real-time demo of Gaussian splatting with dynamic lighting from Andrew Chan's blog Post " <i>Real-Time Lighting with Gaussian Splats</i> ", link	59
4.17 Comparison between a flat-shaded cube (left) and a smooth-shaded one (right).	61
4.18 Flat and smooth icosphere with surface normals as color channel.	62
4.19 Boundary at the silhouette edge of a capsule mesh with smooth shading (left) and wireframe overlay (right) to show underlying geometry. Note the defect in the shadow volume due to a UV seam	62
4.20 One possible UV unwrapping of a cube. Left: Cube mesh with UV seams (in red). Right: corresponding UV unwrapping.	64
4.21 Visualization of silhouette edges (yellow). Additionally shown: lit faces (green normal), unlit faces (red normal), lit edge (blue), unlit edge (red)	66
4.22 Image of sun coins on a paved road, Wikimedia Commons by <i>Quelokee</i> (public domain), link	69
4.23 <i>Crescent Sun shadows</i> , by Harris County Public Library (CC BY-NC-SA 2.0), link	69
4.24 How the scene might look like for a user (left) and how they can affect the simulation (right)	70
4.25 Shadow Volume model, sphere occludes the lamp in the shaded area behind it.	71
4.26 Sun coin model, image of the sphere (which serves as a lamp) being projected through a pinhole	71
4.27 Examples of a normal projected edge (left), a twisted edge, constructed like before but projected through a pinhole, swapping a' and b' (middle) and the same with cleaned up geometry through introduction of a center point (right)	71
6.1 A puzzle in the game <i>Resident Evil 7: Biohazard</i> , Source: Uploaded by ParallelTraveler to <i>Resident Evil Wiki</i> , link	77

List of Tables

3.1 Summary of author impressions of various game engines	19
---	----

List of Listings

language=GLSL, caption=Simplified implementation of Lambertian diffuse lighting	25
4.1 Some lines of what our TOML file might look like	32
4.2 What a hypothetical future extension of the schema may look like	32
csharp	44
4.3 Unit sphere position or normal vector to UV coordinate	46
4.4 Rotation of the hit normal before deriving UV coordinates.	48
4.5 Ray sphere intersection.	48
4.6 Remapping the light value.	50
4.7 Selecting a planet to render based on it's proximity to the camera.	52
4.8 Loop over other planets and check if they could obscure the sun.	53
4.9 Signature of the Face struct type.	65
4.10 Check if face is lit	65
4.11 Helper function that projects a given vertex and adds it to the vertex buffer.	67
4.12 Call projection helper and add new triangles for projected edge	68

Abstract

With the emergence of cost-effective, accessible immersive virtual reality (VR) technologies, opportunities arise to utilize such technologies beyond entertainment and enterprise uses. Utilization in educational contexts is particularly of interest, as a growing niche for serious games and learning applications in virtual environments is being established.

We are motivated by the potential of immersion and the sense of presence that arises from it as a potential amplifying factor for learning effects in educational content. To this end, we design an interactive learning simulation with a guided tour style instructional design to convey concepts of middle school optics to learners. We emphasize modular design and a diegetic user interface, that attempts to anchor users in the virtual world.

To this end, we devised a set of potential experiments in close consultation with an expert in physics didactics, investigated potential technologies to use as a basis for this work and narrowed down our options to Unity 3D, a widely used game engine that offers accessible, robust tools for designing 3D virtual environments, and the PICO Neo 3, a portable VR system that minimizes the setup effort compared to traditional virtual reality systems.

We then implemented a number of experiments, primarily focused on the properties of shadows, both in a solar system context, and their spatial properties. We investigated potential approaches to model these experiments and defined ways in which users were able to affect the simulations to experiment and ideally learn.

Finally, we designed a plain text system that generates the instructional part of our application to facilitate simpler iteration of the didactic content of our application without extensive programming knowledge, and simplifies prototyping new learning content.

We assert the technical viability of the application for use in learning situations and propose evaluation methods in lieu of our own empirical study. We also include a framework to facilitate the collection of empirical user data, particularly positional tracking and interaction with objects in the scene, to aid future studies.

Chapter 1 Introduction

With immersive virtual reality technologies becoming more accessible, flexible and ubiquitous, their use in applications beyond entertainment is increasingly enticing. Immersive technologies have long been a topic in serious games and simulations. While not quite aligned with our modern idea of virtual reality, immersive simulations have been used successfully in simulations aimed to train users in vehicle operation, particularly in situations where training with real life vehicles would prove to be too costly¹ or dangerous, especially early on.

1.1 Motivation

When we talk about immersion or immersive technologies in this work, we mean technologies that attempt to emulate environments or experiences to a believable degree. Immersion may allow users to develop a *sense of presence* [TSR25] within the simulation. Immersion is an important advantage that VR systems possess over other media that users might come into contact with. Research suggests that immersion, particularly if users are actively engaged in the simulation, may prove beneficial for learning situations, although more work is needed to refine instructional design in VR learning applications [CKS24].

We aim to exploit the learning effects of immersive applications to design a learning simulation of our own, aimed at teaching concepts of middle school physics class, primarily the topic of optics commonly taught early on in middle school.

Naturally, Virtual Reality as we think of it today is vastly different from the purpose-built simulators of the past, particularly consumer-grade systems such as the ones we will be discussing in section 3.2.

Head-mounted VR devices offer a large degree of flexibility for the design of learning applications. Both environments and the functional content of applications can be freely designed and a great variety of design concepts for displaying information to users, interacting with the virtual world and locomotion through virtual scenes have been developed and deployed in a wide variety of entertainment and educational software over the years.

¹ Consider this immersive simulator used by the Swiss army in the 1970s to train tank operation: <https://www.youtube.com/watch?v=AcQifPHcMLE>

1.2 Virtual Reality in Schools

While the use of immersive software, particularly virtual reality, is by no means a mainstream topic in schools at this point, it nevertheless becomes an ever more attractive prospect as the underlying technology gets refined over time.

Investigations into the viability of VR devices in school education are currently actively ongoing, such as a pilot project in the German state of North Rhine-Westphalia (NRW), which involves an investment of around five million euro into VR equipment for use in schools (see [Car24], *article in German*).

Notably, the use of VR in educational contexts usually constitutes as an additional learning aide rather than a primary component of the lecture. At least in part, this is owed to the nature of head-mounted devices, particularly the fact that virtual reality devices will necessarily block the user's view of the real world. This usually requires the establishment of a generous free area to ensure safe navigation, although standing or seated VR use is not uncommon.

As users cannot see the world around them, applications that encourage active movement may cause them to bump into obstacles, or even other users, by accident. Broader use in classroom contexts would thus require restraint in the environmental design of a VR application to avoid injury.

1.2.1 Limitations

Users may also experience a number of other barriers to entry when using VR systems that may obstruct a wider adoption until properly addressed. Particularly, cybersickness is a persistent issue of usability within VR applications. While differences in severity of symptoms between users of different sexes are inconclusive, age seems to be especially impactful [OS22]. A multitude of parameters, field of view, frame rate and motion outside the user's control, may cause nausea and discomfort. This drastically limits the duration that users can stay comfortably within an immersive application, making it more suited for bursts of material rather than long-form sessions. Notably, while studies vary in the exact proportion, it is fairly clear, that a non-negligible chunk of users suffers from at least mild symptoms of cybersickness.

Cybersickness limits effective session length

This is especially true for users with visual impairments, where they may struggle or be unable to perceive the virtual environment within the head-mounted display (HMD), are affected by educational content reliant on virtual reality. Users may experience discomfort due to the proximity and brightness of the HMD screens or be unable to use their vision aide properly (although glasses are often fine). The VR world also does not offer much haptic feedback, making usage difficult for users relying on such stimuli.

Visual impairment is a barrier to entry in VR

1.2.2 Chances

Nevertheless, this technology has much potential for school use. In our case, we decide to design our application more along the lines of a learning aide, offering a learning simulation for learners to engage with, including narrated didactic content that is meant to direct the experience and provide valuable information. This content was designed in consultation with experts in school physics education.

Immersive

visits to far-
away worlds

Most straight forward, virtual environments are practically risk-free and often offer an impressive amount of fidelity. This allows applications to feature settings that learners may not otherwise be able to visit, or that are too costly to visit in person. Some applications allow users to navigate out of reach locations such as the International Space Station [The18], the human body, different countries, archeological sites and places that are barred to the public due to conservation concerns.

VRChat dur-
ing COVID-19

A VR application may also offer a semblance of social interaction to socially isolated users, be that due to physical limitations, illness or exceptional societal reasons, such as the COVID-19 pandemic. Virtual reality environments can support social networks that offer a safe community for users to socialize in. One such example is the use of the free, popular VR social media *VRChat* as a tool to facilitate socialization and improve mental health of users during the social isolation of the COVID-19 pandemic [DAO23].

Effects of so-
cial isolation

Prolonged social isolation is a detriment to the mental wellbeing of students, both in universities [TAnZnF⁺25] and schools [RSKE⁺22]. Thus, alleviating symptoms of social isolation in students during exceptional situations such as a pandemic, or students that cannot socialize normally due to conditions outside their control, is a desirable goal to improve the mental wellbeing of learners.

Related Work

1.3 Structure

In chapter 2 we will first examine some other works, both academic and consumer products alike, that influenced this thesis or that we drew inspiration from. This includes a number of projects in the design space of VR learning applications, especially *RePiX VR* [HGS22], as well as a number of playful simulations that take a more implicit approach to learning than we ultimately end up adopting

Technology

In chapter 3 we will then discuss in-depth which technologies we took into consideration and what we ended up deciding to use for this thesis project. This includes the VR hardware that serves as a target platform, namely the PICO Neo 3, as well as the development environment, that is Unity 3D. We will compare these to other alternatives. On the hardware side, we discuss the use of virtual reality in comparison to other extended reality technologies. As for the engines, we compare some of the current state-of-the-art development environments, justify why we went with Unity 3D over the other engines available. We then briefly discuss considerations regarding the version control system, and particularly our chosen schema for adding changes to the code base.

Design

In chapter 4 we delve into the implementation details, starting off with a brief overview of the topics our simulation will cover and how we will cover them. This includes a quick, technical description of the narration system which serves as the backbone of our instructional design. This is followed by a description of the three primary simulations we implemented:

- the *solar system*, which includes the implementation of a variety of ray tracing shaders to precisely render planets and a description of the underlying orbital model we implemented to derive a consistent, reproducible state for the simulation

- *shadow volumes*, which allow students to visualize the 3D space projected by the occlusion of a light source by an object, for which we adapt a shading methodology to generate the appropriate geometry in the scene
- *sun coins*, which adapt the shadow volume method to reproduce a particular optical effect that is based on the concept of pinhole cameras, but can often be naturally encountered in ones everyday life

In chapter 5 we will then briefly evaluate the suitability of our application and suggest methods and topics we deem interesting for further investigation. We will also delve into how we have taken steps to prepare for more sophisticated empirical evaluation in the future.

Evaluation

Finally, in chapter 6 we will reflect on our work, discuss limitations and challenges encountered during this thesis work and discuss any remaining concepts and ideas that may warrant further investigation in the future, should we have not mentioned these in their respective implementation sections.

Conclusion

Chapter 2 Related Work

Currently, virtual reality is by no means a mainstream technology when it comes to the design of learning applications. As we have established before though, the technology is steadily improving to the point where using it in an educational environment is steadily becoming more appealing.

In this section we will explore a number of other works from the field of educational application design, particular with a focus on natural sciences, that this work has drawn inspiration from or that focus on a similar topic. We will then establish how this work relates to them and how we attempt to differentiate us from these prior works.

2.1 RePiX VR

Perhaps the most pertinent project to compare our own efforts to is RePiX VR[HGS22]. RePiX VR aims to convey learning content from the field of computer graphics, more precisely the concept of the *rendering pipeline*.

Initially, we considered building our project on-top of RePiX VR, especially as this project moved increasingly in the direction of a guided learning simulation in the initial planning phase. While RePiX is flexible enough to accommodate arbitrary learning content, even outside of computer graphics technologies, it also provides a robust framework to convey didactic content. Furthermore, RePiX supplies tracking support for its various input modalities, which is a boon to the acquisition of empiric data later down the line.

While we opted to build out our system from scratch, it may be of interest to reframe this project into RePiX later on. To this end, we endeavored to design our stages in mostly stand-alone modules that should be easy to migrate into different applications, should the need arise.

We also attempted to differentiate our work from RePiX through the integration of a diegetic user interface, which has users interact with buttons and levers in the play space rather than abstract UI widgets. The topic of diegetic vs. non-diegetic interface design in particular is interesting, research suggests that academic applications often opt for a more non-diegetic, task-focused and efficient design while commercial games prefer diegetic interfaces that are designed to be engaging and “fun” [STA⁺21]. Especially in learning applications aimed for younger learners, it might be interesting to compare use of these two design philosophies in future research.

2.2 Other Academic works

Another project that deals with the topic of optics, particularly lens optics, in an immersive simulation is the *Virtual Optical Bench* developed by Bellgardt et. al [?]. It provides an immersive, in-depth simulation of lens optics that has even been used in teaching.

Notably, it makes use of the OptiX ray tracing engine developed by GPU manufacturer nVidia¹ to deliver numerically accurate simulations of optical lenses as one might encounter in physics classes. This even extends to the visual appearance of lenses, realistically inverting, blurring and magnifying the world as perceived through the lenses.

To this end, the application makes use of the aforementioned ray tracing engine is used to model the properties of lenses. As we will be utilizing limited, mobile hardware as discussed in chapter 3, this is not particularly viable for our project. However, a simpler version, omitting the visual lens effects, or simplifying it through screen space effects (similar to *screen-space reflections*) may be a suitable addition to broaden the topics covered by our application. This could be focused entirely on visualizing the mathematical model that informs the ways lasers (or “beams” of light representing photon paths) are bent by lenses of different shapes.

In their paper *Improving efficiency in optics education with immersive virtual reality*[FBLC24], Fabre et al. conclude that both immersion and tracking of motion data are important criteria in the design of educational virtual reality applications. They are furthermore critical of approaches involving arousal theory, and subsequently the inclusion of extraneous, attention grabbing material, particularly if it doesn’t contribute to the overall didactics of the application, as well as an overemphasis of “hands-on” practice over written and spoken learning material.

While we will implement a stringent instructional design approach for the learning content, the critique of arousal theory may reasonably apply to our application, particularly the background elements of the solar system discussed in section 4.2 may be guilty of this exact concept. Particularly, if they were to be allowed to move at the user-configured speed or magnification even when we move on to additional stages later on.

For their application, Fabre et al. implement a lab-like setup involving reflection of a laser beam. This is another classic experiment, and also a very common concept for puzzles in video games, and could thus be interesting for future gamification.

2.3 Simulation Games

Many video games delve into the realm of physics simulation, sometimes as a system of puzzle solving, other times for educational purposes or as a way to make scientific simulations more palatable to the broader public. While these are very much still products of entertainment, rather than dedicated serious games, they nevertheless are an interesting item of investigation.

The application *Universe Sandbox* for instance advertises itself as a “physics-based space simulator” [Gia25] which allows users to manipulate model universes by altering the properties of celestial bodies, adding new objects to the simulation, apply a

¹ Website: <https://developer.nvidia.com/rtx/ray-tracing/optix>



Figure 2.1: While there are many educational scenarios in *Universe Sandbox*, some take on sillier tones such as colliding multiple “spherical cows” into the surface of the moon.

variety of visualizations for physical characteristics, applying forces or even shooting bodies at one another to simulate impacts. This provides an open-ended simulation that has very explicit ties to educational content, particularly through pre-configured experiments and informative writing throughout the application.

The space agency simulation *Kerbal Space Program* [Squ15] and its many spiritual successors are more along the lines of a game. While initially entirely a vehicle construction and space faring simulation, more traditional game design elements such as objectives to pursue and an in-game economy were introduced later down the line. However, while it reads squarely like a game, the educational underpinnings are not to be underestimated. During the early stages of the game, a player may be able to get away with an intuitive understanding of Newtonian physics to get their rocket up into the sky, and maybe even into orbit around their planet, but intuition quickly falls short once orbital navigation enters the fray. As the theoretical requirements mount, particularly when trying to reach far-away planets and thus requiring an understanding of interplanetary transfers, players are bound to give up, or seek outside help. A vibrant community of content creators have spawned around this genre, providing a wide range of educational content that users motivated to progress may find themselves looking up.

While we will pursue something more along the line of serious game, or rather an educational simulation, with a much more linear, guided learning experience, we nevertheless hope to capture some of the fascination that physics simulations can create in users.

Chapter 3 Technology

In this chapter we will explore the current state of virtual reality (VR) experiences. We will then talk about how virtual reality fits into the wider extended reality (or XR for short) technology and why we choose to utilize VR for our application. We will also explore the current state of the art of VR development and some different tools that can be used to develop VR applications in an ergonomic, efficient manner (as opposed to a “from scratch” approach).

We will then go on to look at different kinds of VR user devices, often called head-mounted displays as they are commonly worn on the users head as a sort of “helmet” or “goggles”.

Finally, we will elaborate on our choice to work with the Unity3D game engine for development, while we pick the PICO Neo3 as our target system of choice.

3.1 State of VR Development

Virtual reality applications have been available to the wider consumer market for some time now. Although the idea of immersing users in an artificial environment may seem rather modern, the underlying idea of a virtual reality reaches back several decades. Long before it was ever viable from a technological standpoint it was already subject in science fiction writing. One may, for instance, consider the Holodeck¹ depicted in popular sci-fi series Star Trek to be an early entry into the canon of virtual reality depictions.

One of the earliest implementation of immersive technologies akin to what we might today consider as virtual reality was the Sensorama, a prototype multimedia system patented by Morton Heilig in 1962 [Hei].

Japanese game system manufacturer Nintendo even made a first foray into the home consumer market with their doomed *Virtual Boy*² in 1995 that, while being far removed from the conveniences of modern HMDs, nevertheless resembles modern VR systems in form and function. While the Virtual Boy would be a commercial flop [Edw], perhaps due to technical hurdles as the weight of the device would require it to be propped up on a table, forcing users to assume an awkward posture while playing.

While the true starting point of the current wave of VR consumer hardware is hard to pinpoint, it likely occurred around the mid 2010s. On the back of the rapid development of small, high resolution screens as well as advances in motion tracking and

*Popular-
ization of
consumer VR
tech*

*early VR
systems,
including
consumer
grade, pre
2000*

*Beginning of
the modern
VR era in the
2010s*

¹ Wikipedia: <https://en.wikipedia.org/wiki/Holodeck>

² https://de.wikipedia.org/wiki/Virtual_Boy

efficient rendering techniques, the time seemed ripe for new immersive technologies to emerge.

Among others, Google began experimenting with utilizing smartphones to drive virtual reality experiences; first with *Cardboard*, a simple and cheap HMD made entirely of cardboard. This approach was later refined into *Daydream* while other companies made their own smartphone-driven HMDs such as Samsung's *Gear VR* family of devices. While these devices seem simple compared to modern virtual reality, offering at most three degrees of freedom (that is: head tracking in three rotational axes), they would offer an affordable and accessible entry point not just for committed enthusiasts of the technology, but especially curious onlookers that may have otherwise been unwilling to buy into an emerging niche technology trend.

PC VR and
6DoF systems

Another company entering the VR market around the same time was Oculus. Their *Oculus Rift*, initially funded through the crowdfunding service *Kickstarter* is often considered to have reignited public and industry interest in immersive technologies at a larger scale by presenting a consumer-grade device that could deliver a credible sense of immersion. Many other devices would follow suit shortly after including the *Vive* and *Index* family of devices by manufacturer HTC. Oculus would later be acquired by Meta where they went on to produce the *Oculus Quest*, later *Meta Quest*, a device that omits the need of a desktop PC to drive the experience, instead opting for a built-in mobile processor, combining the portability of a mobile standalone system with spatial tracking and input fidelity that could match their rival desktop systems.

Mobile VR is
becoming a
popular niche

This approach continues to be popular, since devices can follow a “best-of-both-worlds” approach where they can still connect to the user's desktop system if more performance is required, mostly for more complex applications with a higher graphical fidelity. Such “hybrid operation modes” have now become commonplace with standalone VR systems. While desktop systems historically use tracking stations scattered around the room, mobile VR makes use of *inside-out* tracking, i.e. a set of cameras on the device track the user's position in their current environment as well as the position of their controllers. This approach trades some tracking precision (e.g. behind the user, where HMD sensors can't reach) for a vast simplification of the overall setup that enables users to set up and use their system on the fly wherever they want.

In parallel, another technology has been steadily evolving: *augmented reality*. Rather than producing a virtual world that the player navigates, augmented reality involves adding elements to the real world, typically as perceived through a phone screen, car windshield, helmet visor, or, of course, an HMD (usually this is called pass-through mode, but specialized AR HMDs exist as well).

The appeal of augmented reality is, that it merely needs a camera (and a sufficiently powerful device to handle the necessary computer vision and rendering tasks) in order to work. As smartphones grow more powerful, AR is ever more accessible and finds various uses in entertainment and serious applications such as for navigation and education.

Mixed reality
combines VR
with the real
world

There is also *mixed reality* or MR for short, which is easily confused for augmented reality. We can think of mixed reality as an in-between of AR and VR where elements of the real and virtual world can influence one another (although not necessarily in the physical sense). It requires the applications to have a certain degree of understanding of a real scene, perhaps a spatial mapping or semantic understand of the scene, to be

able to perform certain actions such as hiding behind real objects. Virtual objects are, essentially, integrated into the real world as perceived by the user. Common examples of primarily mixed reality systems are the Microsoft HoloLens and Apple Vision Pro the latter of which, while expensive, was especially successful at drawing attention to the mixed reality niche.

Together, these three technologies are often referred to as extended reality (or XR). Support of and development for XR applications has drastically improved over the years as they slowly make their way into the commercial mainstream.

*Standardiza-
tion of VR
technologies*

Today, a wide selection of XR devices exist, and the whole development pipeline has been streamlined in no small part due to the convergence of hardware capabilities, input modalities and software interfaces of modern XR systems. The wild-west era of XR has mostly departed, making way for uniform control schemas, popular libraries, abstraction away from vendor-specific implementations and standards that are either actively considered or have already been widely implemented (such as OpenXR³). As these systems mature, the ecosystem surrounding them similarly stabilizes and experiences both on the development and the user side of the topic constantly improve with the accumulated knowledge and technology.

3.2 Hardware

As previously mentioned, the selection of XR hardware has grown considerably over the years. There are many different device families and ways to engage with XR applications. The hardware platforms are rapidly diversifying with niches such as cheap, portable systems to high-end systems with advanced tracking features to enterprise mixed reality devices such as the aforementioned Microsoft HoloLens.

3.2.1 Mobile vs. Stationary XR

As we have seen, one of the main differences between VR systems is whether they can function on their own or require a companion device, usually a fully fledged computer capable of running applications that utilize modern high-fidelity graphics.

*Desktop
VR offers
nice visuals
and high
precision*

Off-loading their work on a companion device is especially useful for high-fidelity applications such as games or particularly complex environments and simulations. Evidence suggests that higher framerates and resolutions can improve the user experience and reduce the effects of simulation sickness [WSX⁺22] and that visual fidelity may affect the user's sense of immersion [SGS⁺24]. Desktop-driven systems are often more open, giving greater access to both developers and users. Even if they somehow don't support general VR applications, they will likely support services such as *SteamVR* which initially served to bridge differences in device APIs and is still commonly used today.

³ <https://www.khronos.org/OpenXR>

3.2. Hardware

On the other hand, a desktop introduces additional points of failure, as it increases the amount of devices that can malfunction and require knowledge to use and maintain. The sensor stations required for input tracking by many desktop VR systems are also often a liability. They need to be set up correctly, which may take time as proper mounting locations must be determined first. They can malfunction or may not connect properly sometimes, and they take up physical space that needs to be considered when partitioning the play area. The hardware necessary to run high-fidelity VR applications is also a large cost factor, whereas desktop VR devices themselves are rarely cheaper than their portable counterparts, if not downright more expensive due to the accessories required to use them.

Mobile VR is (often) cheap and convenient

Cost and maintenance may both make the use of desktop VR less attractive depending on the circumstance. It may be required to dedicate a fixed space for a sort of “VR pool” as such an environment cannot be easily set up and torn down for every session. It may also require additional training for teachers to successfully deal with issues in an educational setting if VR were to be used on a class-wide level rather than as an extracurricular novelty.

Mobile VR devices are usually cheaper and simpler to use but also more limited in their hardware capabilities. Inside-out tracking requires barely any preparation, it can often be restricted to a small area (stationary mode) and can be set up and altered on the fly. This makes mobile VR devices fairly flexible, which is unsurprising as they are built to be transported and used in various different circumstances. Applications can be managed on-device, usually through an application store. This can be a downside, as installing software that is not officially offered on the device’s store, such as custom research applications, may be obstructed by design. Portable devices are also battery powered, which users need to keep in mind, but also makes it possible to deploy the device in a much wider array of situations, such as outdoors, in exhibition halls, where power supplies may not be readily available, or in classrooms where the amount of power outlets may be limited.

Simple setup, high flexibility, but performance considerations needed

They can also be easily loaned out and can have all the necessary software pre-installed so that a learner may get to work with it basically immediately after booting it up. Furthermore, as mentioned above, they can easily double as a desktop VR system through vendor-provided bridge software.

Even though mobile systems steadily get more powerful, some care is still needed to design applications in a way that doesn’t overexert the device’s hardware resources, as this can quickly lead to reduced frame rates which in turn may cause the user to experience signs of simulation sickness.

While it is too limiting in its immersive qualities, we do not want to entirely discount historic phone-based VR applications. Even if they are not the right choice for us due to their limits in interactivity (e.g. cardboard was limited to a single button and the user’s look direction) and the lacking ergonomics three degrees of freedom, it can nevertheless offer a cheap and accessible gateway into immersive technologies even today. The most likely way that users may stumble across an application like this nowadays is in their browser.

3.2.2 PICO Neo3 Pro Eye

While the device selection does not matter as much as it used to and mostly serves as a sort of lower bound to what devices we want to potentially run on, we nevertheless need to decide on some target platform to test on.

For this, we opt to use a standalone VR system, specifically the *PICO Neo3 Pro Eye*⁴, an all-in-one (that is, standalone mobile) HMD.

It does not require any sort of companion device to run, although it can transcend platform boundaries and work as a desktop HMD through PICO's *Streaming-Assistent* or, on later devices in the product family, PICO Connect. The Pro Eye version also features a built-in eye tracker that may prove interesting for future research endeavors.

It uses a controller layout that has by now become commonplace for the current wave of VR devices. The controller interface can be separated into three areas:

- the “face” that is operated by the thumb and consists of two face buttons (labeled A and B on the right, and X and Y on the left controller), a joystick and a menu and home button
- a “trigger” located on the back of the controllers that acts as a natural resting place for the index finger
- a “grip button” located on the side of the button and usually operated by the middle or ring finger

The Neo3 furthermore features two high-resolution screens (one per eye) that render images at 1832 x 1920 pixels each.

It was initially released in 2021 and has since been superseded by the *PICO 4 Ultra* and other devices such as the *HTC Vive XR Elite*, but is still a popular device with clear research utility due to its eye tracking capabilities.

The PICO Neo3 OS is built on Android, which allows it to run applications built for Android systems.

3.3 Development Environment

As mentioned before, the ecosystem for XR development has become much more streamlined than only a short while ago. Previously, devices would feature quite different input modalities and each bring their own unique API, making multi-device support a huge hassle.

Consider, for instance, the HTC Vive controllers depicted in figure 3.1: while they possess the trigger and grip buttons common in other devices, they lack face buttons entirely. Instead, they utilize a single large trackpad that also differentiates between touch and click actions.

*VR ecosystem
has become
streamlined,
interfaces
standardized*



Figure 3.1: Photograph of the HTC Vive controllers. Source: C. Steber, “HTC Vive Controllers”, Wikimedia Commons, CC BY-SA 4.0.

The US game development and publishing house *Valve*, which would go on to make their own HMD in collaboration with manufacturer HTC, the *Valve Index*, offered their own solution to the problem: the *SteamVR* client, powered by the open source OpenVR API.

Bridge API
to abstract
away inter-
face and
implementa-
tion details
(*SteamVR*,
followed
by *OpenXR*
standard)

Developers could simply call abstractions from the OpenVR API, mostly so-called Actions that would define platform-independent inputs, without interfacing with the hardware itself. On the other side, device vendors would simply need to implement the OpenVR API bindings for their respective devices.

This would make it much easier for developers to support the vastly different devices that were on the market, allowing their users to avoid getting locked into a specific device’s ecosystem. Perhaps the biggest caveat is, that this would still necessitate the *SteamVR* client which, while free, is a proprietary piece of software available through the software distribution platform *Steam*.

Eventually, OpenVR would be phased out in favor of the fully open source *OpenXR* standard. OpenXR is stewarded by the Khronos Group, an industry consortium that is also known for other open-source technologies in the graphics computing sphere, such as *OpenGL*, *WebGL*, *OpenCL*, *Vulkan* and many more. In a similar vein to OpenVR, it would serve as an abstraction layer over hardware details and encompasses support for a wide variety of XR features from locomotion and input events to advanced tracking features like hand and eye tracking.

OpenXR
replaced
SteamVR

3.3.1 WebXR

Although we will not directly interface with it, the emergence of the *WebXR* API standard, which would allow VR devices to run immersive applications directly from the browser, is fairly exciting. While it is still not fully matured, it is already implemented in some browsers [moz] and works through the ever improving access to native graphics hardware through *WebGL* and more recently *WebGPU*.

Web apps
are easy to
distribute
and well
supported

The main advantage of browser-based VR is its accessibility, especially on mobile hardware where access is usually restricted by the vendor-provided application store. Whereas native applications may need to be installed through development pathways or require publishing to the official stores, websites can be easily shared and accessed, should the device’s browser implement the *WebXR* standard.

⁴ <https://www.picoxr.com/global/products/neo3-pro-eye>

One way we can utilize this technology, even through non-web approaches, is through the various engines' web client features. This leaves options for distributing this application to different devices in the future, including perhaps non-VR systems. The degree to which this would impact the current implementation needs to first be determined, however with trends in web-based graphics rapidly improving render performance this may just become the most viable way to distribute learning applications in the future.

Other than the fully integrated engines we will discuss below, the tool chain for WebXR applications is somewhat more loosely defined. Native web applications will likely utilize a library such as *Three.js* as a baseline for managing a modern render pipeline. However, anything that handles 3D graphics in WebGL goes and an entirely custom workflow is thinkable.

Yet, while this approach to tooling is not uncommon in web development and often offers a flexible experience where developers can adjust their tooling to their whims, it is also a major downside over more dedicated game engines as we will encounter them below. In fully fledged game engines, all the underlying tools, renderers, physics engines, state management etc. are tuned to work well with one another. While this does not always work out, it is nevertheless an important aspect of the developer experience that cannot be easily emulated with an assembly of different tools that may have been developed with vastly different teams and different opinions, even if they base their work around the same base libraries (e.g. *Three.js*). There will likely be friction that must be handled by the application developers and may introduce obscure or difficult to solve issues.

WebXR development affords a lot of flexibility, perhaps excessively so

Furthermore, where a web developer working on an XR application may need to search for solutions to different problems, the engines will already provide some sort of solution in most cases. As an application evolves, it will perhaps need to handle animations, or need some sort of graphical level editor. While an engine may integrate all these in one bundle, a pure WebXR project may need to find a variety of tools, ensure that they work well with the assets they intend to use and resolve any inconsistencies themselves.

Engines offer more direction and bundle useful APIs directly

Conveniently, tools that are commonly used together may align over time and gradually approach a more integrated development experience, but with no governing body to direct this alignment this will require time and effort.

Another downside of splitting up tooling into a variety of libraries and frameworks is, that extra work is required to learn the technologies involved. Documentation may be of varying quality and may be scattered across many different locations and organizations. This becomes even more pressing if new developers need to be introduced to the application and onboarded.

Fragmented tooling may raise learning curve

While there is a learning curve with integrated tools as well, most engines also happen to come with training programs and tutorials, and usually keep their documentation in one easily navigable space.

3.3.2 Game Engines

A game engine is a fully fledged framework, potentially spanning a multitude of different tools, with the primary goal of assisting development of interactive applications such as games and simulations.

In the realm of 3D game development, engines usually provide solutions for a multitude of common problems:

- graphics rendering (including light and shadows, material handling, custom shaders and customizable render pipelines)
- animation (keyframe handling, mesh deformation and dynamic animation rigs, animation state handling)
- real-time physics simulation (collision detection, gravity, rigid body, and possibly soft body simulation)
- game state management (entry points, update hooks, lifecycle handling)
- input handling (across multiple input devices)
- audio (handling encoding, spatialization, dynamic volume management)
- deployment (creating an executable for a variety of platforms)

Engines are collections of tools, UI is optional but appreciated

Engines are often conflated with their user interface, but some game engines do not feature a user interface at all. A classic example would be *XNA* and more recently the data-driven Rust game engine *Bevy Engine*. Some developers may even utilize a popular engine such as *Unity* while completely avoiding its editor interface as much as possible.

The user interfaces in engines tend to serve as a level editor and a convenient way to render access to the development more accessible and convenient. Many allow extensive customization of the user interface to deploy custom tools that may fit the projects needs. But even at a simpler level they offer useful abstractions, such as working directly on serialized data within the scene or changing aspects of the application during runtime for testing and debugging purposes.

Unity

Unity is a staple of independent game development

The Unity game engine was released in 2005 and has thus been around for two decades at this point. It is considered a popular tool in the industry, driving projects of all scales. It is often used for small to medium scale independently developed and published games such as *Kerbal Space Program (2011)* to complex multiplayer experiences such as *Rust (2013)*. But it has also found use in a variety of major studio games (especially in the mobile game sector) and is a mainstay in serious games, film and education.

The Unity engine is proprietary, but it offers free personal and educational use versions and doesn't impose restrictive licenses on projects built within the engine.

Development involves the writing of scripts using the *C#* language, compiling either for *.NET* or the open source *Mono* runtime. Thus, it supports builds for a variety of systems, primarily: Microsoft Windows, Apple's MacOS and iOS, Android and Linux.

Architecturally, Unity makes use of an object-oriented approach: projects consist of one or more scenes that contain a number of game objects. Each game object may have a hierarchy of child objects, each a game object themselves, as well as a number of components that control how the game object behaves.

Through the composition of game objects and writing suitable custom components or utilizing the many pre-built components provided by Unity, developers can build up their application as desired.

The maturity of Unity as a game development tool provides a mostly stable, predictable API. Furthermore, the maturity of the engine also shows in the multitude of courses and learning resources available for most if not all its minute features, officially provided through the engine's parent company *Unity Technologies*⁵.

Developers may also receive help to any inquiries on a community-maintained page on the popular website *Reddit*, an official server on the real-time messaging service *Discord*, and the *Unity Discussions* forum which replaced the question-and-answer format forum *UnityAnswers* some time ago.

There are also countless community tutorials on *Youtube* or strewn across blog posts all over the internet.

Furthermore, while this is not a unique feature in modern engines, Unity offers both a prolific asset store with a number of free and commercial assets and tools, as well as an easy-to-use package manager that gives access to a number of optional and experimental features.

The package manager can also be used to include in-offical packages from repository services such as *GitHub* and can be extended with custom package sources.

Asset store packages may be unsuited to an open source package such as this due to their dubious licensing, but are nevertheless a helpful feature to many aspiring developers.

Unreal Engine

The oldest, perhaps even the most established of the engines we will consider is *Unreal Engine*. It has been conceived as an in-house engine for the software developer Epic Games before being released to the public as a commercial development tool. Similar to Unity it is thus a proprietary game engine.

Architecturally, it uses an actor-based model. While this may appear similar to Unity's game object and component model, it follows a different underlying philosophy. In Unity, the game logic of an object lives entirely in the components it is composed of. In contrast, Unreal favors an inheritance-heavy approach, where the core of the logic lives in the Actor, the Unreal Engine equivalent of *GameObject*, which then orchestrates its components as needed, although compositional patterns are still possible in Unreal's approach.

Gameplay logic can both be defined through a visual scripting system called *Blueprints* and *C++* code that calls to the engines core API. The advantage of the visual scripting system is, that it avoids the learning curve associated with *C++*. While slightly slower, it is suitable for narrative systems or level scripting, that involves non-technical parts of a development team such as designers and artists that

Heavily utilizes object composition as its main design pattern

Large community, dedicated forum

Package manager and Asset store

Professional game development tool

Visual scripting or C++, hierarchical class composition

⁵ Overview of learning materials: <https://learn.unity.com/>

may have a hard time learning C++ to fulfill their tasks.

On the other hand, the strong performance of C++ makes it well suited to run the core logic of complex applications, and it also is better suited to remain maintainable even if a program approaches tens of thousands of cumulative lines. This comes at the cost of some pitfalls of manual memory management.

Steep learning curve, professional, powerful tools

Since Unreal is geared more towards experienced, professional development teams, it is notoriously harder to use initially compared to other solutions in the game engine space.

This is in part due to C++ simply being a less forgiving programming language compared to many other languages, since mistakes and inconsistencies can quickly build up tech debt that will only get harder to resolve as development continues.

High fidelity rendering tools also come at a cost, as they require higher quality assets to truly shine and rendering logic is more complicated and involved than in the smaller-scale engines.

As such, we avoided using Unreal, development for low-powered devices such as mobile VR HMDs doesn't benefit from the high-fidelity graphics pipeline and performance may in fact suffer as Unreal is not necessarily designed with such devices in mind.

Godot

Promising newcomer engine

Among the engines we discussed, *Godot* stands out as the only fully open-source solution (MIT license). It is also the youngest of the three being available to the public only since 2014.

It has recently benefitted from a surge of popularity and funding [Obe], including interest from technology giants such as Apple [Bec].

Everything is a scene

Godot utilizes a scene graph as its main architectural philosophy. This means that unlike Unity and Unreal, components and game objects are one and the same. Logic lives in so-called nodes, which implement a number of different behaviors. Users can then alter the node behavior through class inheritance, write custom components that can be directly added to the existing scene graph or reference other components for a more compositional approach. Hierarchies of nodes are called scenes, a concept similar to what would be considered a *Prefab* in Unity. This concept also extends to the editor itself, which is also a collection of nodes that can be interfaced with.

Custom python-like language, C# and bindings for language adoption

The primary language for Godot scripts is a language called *GDScript* that is syntactically similar to the *Python* language. Like *Python*, *GDScript* is also dynamically typed and offers some syntactic sugar around common patterns such as addressing objects in the scene graph.

More conventionally, Godot has an alternative build that also supports C# scripting. This may be attractive to projects that require additional performance, as *GDScript* is generally slower. It can also help Unity converts to find their footing.

While somewhat more involved compared to the other solutions, developers can also utilize C++ bindings, which happens to be the language that Godot was implemented in.

Finally, through the *GDEXTension* system, other languages can also be set up to access Godot bindings. One notable project that utilizes this is *godot-rust*⁶, which aims to bring Rust development into the Godot ecosystem and.

Custom bindings to other languages, e.g. Rust

Documentation sources and official tutorials for the Godot engine also exist and are considerably well written, including a crowdsourced translation effort into a variety of different languages.

Unfortunately, there are some aspects of Godot that serve to make it unfit for our purposes. For one, Godot lacks some of the abstractions that make Unity development pleasantly simple.

Primarily, handling transformations requires some extra effort and many other systems feel less ergonomic compared to the Unity API.

The package system around Godot is also less mature, making it hard to find certain solutions that are readily available from official sources in Unity. This is perhaps not improved by the focus on an engine-specific language that makes porting over packages from other ecosystems a more involved process.

Lacking maturity compared to commercial competitors

Finally, vendor support is not as abundant as for Unity, which makes it harder to find help when running into device specific issues and also means that vendors may not care to supply plugins for the Godot ecosystem, which shifts the burden of implementing and maintaining these to the community.

Many vendor plugins are community maintained, which may result in inconsistent updates as maintainers move on to different projects and neglect the plugin a project may rely on, which may break at any moment due to frequent system updates on mobile HMDs with no one left to fix it.

Unity	Godot
• mature development environment • broad platform and vendor support • somewhat sluggish dev experience	• open source (MIT) • flexible and extensible • support steadily improving but still lacking
Unreal Engine	WebXR
• strong graphic API and physics engine • professional development tools • harder to learn and use	• easily accessible on most devices • very flexible, but few "standard" solutions • technologies still unstable

Table 3.1: Summary of author impressions of various game engines

⁶ Project website: <https://godot-rust.github.io/>

3.3.3 Unity

All that being said, Unity is ultimately ideal for our use case. It is a well-supported and easy to use game engine with a broad list of target devices it can reliably produce builds for. It is a notably developer friendly ecosystem that can be navigated with little prior knowledge, and developers can easily acquire whatever skills they may need from a variety of detailed online tutorials, both community-driven and provided by Unity Technologies.

Wide variety of build targets, easy to change

The extensive build support the engine provides for other platforms allows us, to easily reconfigure the application for other HMDs and (with some additional effort designing traditional control schemes) even non-VR systems. Even deployment to a web application is not unfeasible and could avoid issues with vendor-locked application stores as are often present on mobile HMD systems.

Ultimately, engine selection is a matter of preference

However, it should be made clear that, while we decided on Unity here for the reasons above, other engines are also more than viable for the design of learning applications, and we expect especially the Godot Engine to thrive in the near future due to its open-source nature, developer friendly and modular design and a growing influx of new users and funding into the ecosystem.

We will use this section to elaborate a bit more on how Unity applications are designed and how this may facilitate, or impede, our work.

Scenes

Let us begin by revisiting the basic model underlying Scene composition in Unity. As mentioned above, and as displayed in figure 3.2, Unity programs consist of so-called Scenes at the top-level.

Scenes are at the top of Unity's hierarchy

An unity application must always contain at least one scene that is loaded right at the start. It is not an uncommon pattern to have one main scene that deals with initialization of all relevant systems that will be needed during the application session, be that connection services, telemetry or a centralized game manager that supervises the game state and many other details.

Scenes allow developers to section of parts of the application that can be statically loaded together. They can be loaded during an active session both synchronously and asynchronously. Scenes can be loaded and exist in parallel, but exactly one scene must always be active. The active scene serves as the target location for any newly instantiated items.

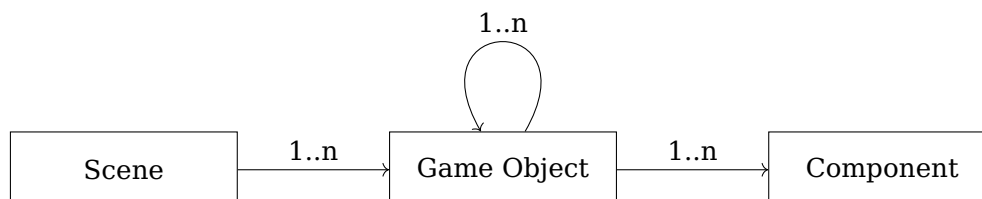


Figure 3.2: Relationships of Unity objects

Game Objects

A scene may contain one or more game objects, which serve as mounting points for game logic. Game Objects store some basic information, such as the name of the object in the scene hierarchy, any associated tag and physics layer groups, a reference to the transform (which is a mandatory component on every game object) and the active state of an object.

Game objects can be active or inactive in a scene. If they are inactive, none of their associated components will be considered in the component lifecycle updates (but they still may be affected by other components and called by events they subscribed to).

The `UnityEngine.GameObject` class also provides some tools to navigate the scene hierarchy, mainly by searching for specific names or tags. Tags are textual identifiers that may be shared among a selection of objects in the scene for arbitrary purposes.

The physics layers determine which objects can interface with one another through the collision system or other physics processes such as ray casts.

Hierarchies of game objects can also be saved as a sort of blueprint called a *Prefab*. Prefabs will share any changes that are committed to the main prefab object (either through opening it from the editor menu or by overriding existing options through a dropdown in the inspector element).

Basic instance, can have child game objects and any number of components

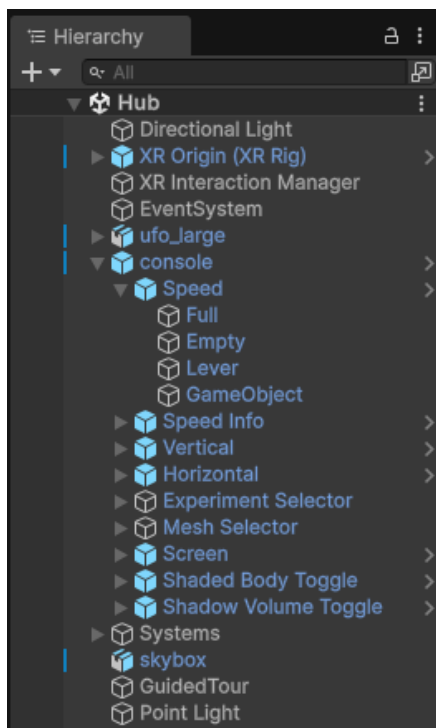


Figure 3.3: Screenshot of the Hierarchy window

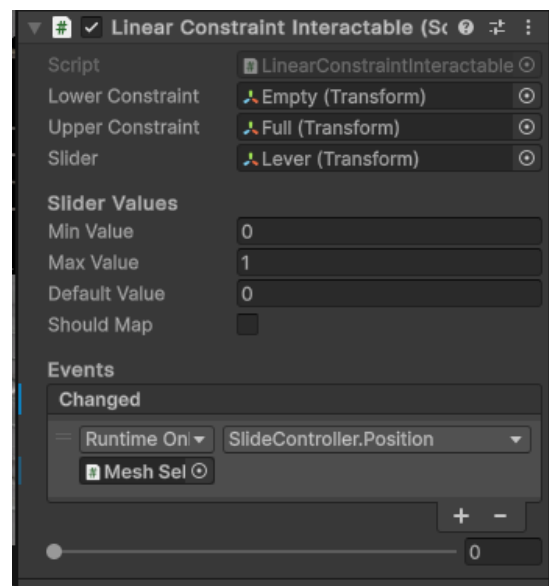


Figure 3.4: Screenshot of a component in the inspector window

Prefabs are an important and powerful design element. They can be easily dropped into a scene, duplicate objects with complex behaviors and, should changes be made, automatically update all related objects. Locally, prefab instances in the scene or

Prefabs are reusable game objects with serialized state

during runtime can also make alterations to their properties *without* affecting the underlying prefab or other instances.

One common use case for prefabs is to serve as a sort of template object that gets instantiated dynamically during runtime, for instance to serve as “enemies” or “bullets” in a game. They can also be used to provide sensible defaults or smart duplicates for objects that would otherwise be bothersome to update manually.

Figure 3.3 show how game objects are displayed in the Unity Editor’s hierarchy window. At the top we see the currently active scene called *Hub*. Scenes can have an arbitrary amount of root objects. We can also notice, that objects in the scene are marked with different icons. A fully blue box shows that an object is the root of a prefab, while the empty box represents basic game objects. The special icon on the *ufo_large* and *skybox* objects shows, that these are imported models rather than true prefabs. Updates to the models mesh files will automatically be reflected in the scene. All objects that are part of a prefab display with a blue label, even if they are not prefabs themselves.

Scene hierarchy shows all objects in a scene, active or not

Components

The actual game logic, as previously established, is located in so-called *components*. Custom components extend the `UnityEngine.MonoBehavior` class and possess a variety of *magic methods* or *messages* that get called by the runtime independently of their actual scope or implementation. Some commonly used messages are [Tec25b]:

- Awake: called when instance of the script component is first loaded
- Start: called when component was enabled, but before update
- Update: called every frame if the component is enabled
- OnEnable and OnDisable: called when component is enabled/disabled

There are, of course, many other messages that Unity can call on any given component. Beside our custom components, Unity also additionally provides a large variety of sophisticated built-in components, from cameras and lights to a slew of physics components (rigid body systems, couplings like springs and ropes), colliders, animators, interaction controllers, render components, input handlers and many more.

While the game objects in a scene are displayed in the Hierarchy window, components of any given game object can be found in the *Inspector* window. Figure 3.4 shows how the inspector displays for a custom component used in this project.

Components are attached to specific game objects and define its behavior

Components expose any fields that are serializable (i.e. simple types, serializable structs and classes, and some reference types that are handled by Unity serialization such as game objects and other components). Only fields under the public scope show up by default, although private and protected fields can be explicitly exposed through the `[SerializeField]` attribute if needed and public fields can be hidden through the `[HideInInspector]` attribute.

The topmost three inspector fields are references to the transform component of other game objects. Below, we see some simple floating point values (trailing zeroes and decimal points are omitted by default) followed by a checkbox that maps to a boolean.

Inspector fields allow us to customize the behavior of components without touching any code. This can be useful for designers, but also allows us to reuse components and game objects in many different contexts and situations. The shown components is part of a slider, so the component UI allows us to redefine the value range depending on our needs if we choose to reuse it in a different part of the application.

Below the simple value fields we can find an `UnityEvent` called *changed*. We can pass different game objects to this field and choose a component and method to subscribe to the underlying event. Events can also receive values to pass on to the subscribers, in this case the current position, either within the specified range or as a value between 0 and 1 depending on the boolean field above it.

Events and observables are, of course, a useful abstraction with a well documented history. The main advantage of the `UnityEvent` class is, that it displays in the inspector view and can thus be dynamically subscribed to without orchestrating the necessary connections from code.

This is useful for a slew of general composition tasks, but it can also be a great tool to help with decoupling code. Rather than appear in the code directly, necessitating another dependency to our local modules assembly reference, an event can be used to call methods across module boundaries as an indirection, avoiding any explicit coupling. This also creates a more general interaction surface to swap out later.

As an example, we use this trick to interface with the `TMPro.TextMeshPro` component without polluting the respective assembly with a reference to its namespace. Text meshes are used to render text content dynamically, in our case into the world space visible to users in VR. Text Mesh Pro is an implementation with noticeably smoother font rendering and support for rich text display, which is not present in Unity's default text mesh. Since all we want to do with it, is to change the text content of an existing text mesh, introducing a reference of it to our component and subsequently making our module dependent on the `TMPro` package is undesirable.

Instead, we expose an event through the component's inspector that is invoked whenever a text change occurs. Instead of referencing the text mesh from our code, we can now simply have it subscribe to the event instead, feeding the text into the text meshes text attribute and thus achieving the desired text change.

This avoids an unwanted coupling to a library we barely interface with, but it also makes it possible to easily pipe the text into other parts of the application if we so desire, as long as these expose a fitting property or method to subscribe with.

Notably, Unity's event system exists in parallel to the system implemented in C#. While these events cannot be serialized and won't appear in the editor, they also have less overhead and are thus preferable for performance heavy tasks. There are also some additional attributes that influence the inspector view, such as `Header`, which adds a bolded label with some additional space above it to delineate sections in the inspector fields, and `Tooltip` which adds some information that is displayed if a field is hovered over for a short time.

Components can expose values for editing in editor, even during play mode

Exposing events is a useful composition pattern to reduce coupling

Use events to trigger behavior or pass data without explicit dependencies

Basic events are faster, but do not serialize

Finally, we can extend the inspectors default view through a custom Editor class. The slider at the bottom of the screenshot is an example of how this may look. This can be useful for debugging purposes, but it also allows us to write more sophisticated serialization solutions or add functionality that manipulates the object or scene content permanently.

We can add additional functionality to the inspector and editor

While this is useful on individual components, Unity also supports an API to design in-editor tools, windows, debug views and whatever else may be needed using editor GUI components and wizards. As this is editor tooling, it lives outside the scene ecosystem.

Universal Render Pipeline

While we mostly interface with Unity's render system on the material level in our work, as discussed below, we do not want to neglect to briefly discuss an important shift in Unity's render pipeline. In 2020 version 2019.3 of the engine released, bringing the new *universal render pipeline*, as well as many other improvements and features such as the new event-based input system [Tec20] to developers.

Composable, customizable render pipeline

The universal render pipeline (URP for short), based on the prior *Scriptable Render Pipeline*, aimed to unify the workflow for different levels of graphical fidelity and allow developers to support a wide range of devices with relative ease by making customization of the render pipeline much easier than previously possible in what is now called the *Legacy renderer*. This new system brought a multitude of changes, and overhauled the material system (which would previously suffer from instantiation-related bottlenecks, necessitating manual oversight to make proper use of efficient batching through techniques such as the now obsolete *material property blocks*).

Drops many old, now obsolete techniques

Materials

Like most engines, Unity utilizes a material system to determine how objects are rendered on screen. Materials reference their underlying shader program as well as any properties that need to be passed to their associated shader (as long as they have been exposed to the material inspector). Every material is a concrete instance that can change during runtime.

Beside using an existing shader, either built into Unity or imported from a package or asset bundle, we can naturally define our own shaders. Currently, there are two separate workflows to define shader programs in Unity:

Custom shaders are defined either through code or as graphs

- *ShaderGraph*, a visual scripting approach that expresses shaders as a flowchart combining into a material output node
- *ShaderLab*, a programming language based on the C-like shader language *HLSL*

The ShaderGraph language brings some ergonomic features with it, such as intermediate outputs that display how the material will look like at various points in the pipeline. It also offers custom function nodes that wrap around shader lab functions and are defined either directly in the visual scripting environment or can be loaded conveniently from a shader file in the filesystem.

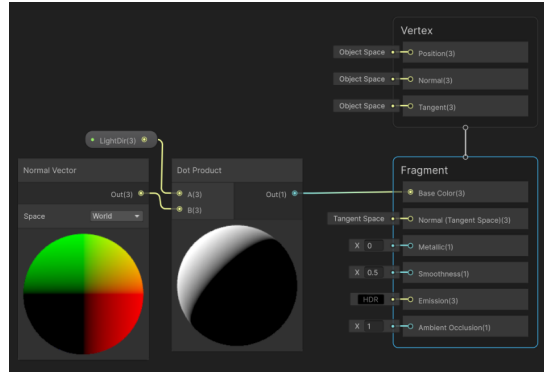


Figure 3.5: Simplified implementation of Lambertian diffuse lighting.

In many use cases, ShaderGraph is preferable due to its simpler semantics, better feedback for intermittent steps, and refined sensible defaults that come with its output nodes. An example of how a (simple) graph may look like can be found in figure 3.5. ShaderGraph can be interfaced with through scripts as one would do with a raw shader, although properties need to explicitly be exposed and named to make this work.

For math-heavy applications such as ours, ShaderGraph quickly becomes convoluted due to the atomic nature of operations implemented through the nodes. This makes expressions much harder to parse visually compared to textual programming languages. To mitigate this, ShaderGraph supports sub-graphs and the aforementioned custom function nodes that allow developers to move out functionality into reusable components.

Shader graph shows intermediate results but gets cumbersome for complex logic

On the other hand, ShaderLab is the “classic” shader workflow that powered advanced materials for some time before its graph-based successor made its debut, and it finds its uses even today. While they require a higher implementation effort compared to shader graphs and are harder to debug than conventional scripts (by the nature of running on the GPU, away from most common debugging tools), ShaderLab is a powerful tool and a versatile one at that.

Clean separation of shader stages

Shaders in URP use the vertex and fragment shader stages we may recognize from GLSL, HLSL and so on. The legacy renderer additionally supported a surface shader stage, that would automatically be expanded into fragment and vertex stages, but has since been discontinued.

While it fulfills general rendering needs, ShaderLab also powers the compute shader system that gives developers access to their GPU as a computation device.

Compute shaders for high performance code

We can also utilize this technology for graphical shaders, most notably to help transfer data to the GPU in a clean structured way through concepts like structured buffers, that enable us to pass sequences of structs for use in our shader.

The following code snippet is roughly equivalent (omitting the boilerplate code for brevity) to the shader graph in figure 3.5:

```

1  v2f vert (appdata v)
2  {
3      v2f o;
4      o.vertex = UnityObjectToClipPos(v.vertex);
5      o.normal = v.normal;

```

```
6     return o;  
7 }  
8  
9 fixed4 frag (v2f i) : SV_Target  
10 {  
11     float3 normal = mul(unity_ObjectToWorld, i.normal);  
12     return dot(normal, _LightDir);  
13 }
```

The scaffolding we omitted takes care of a variety of things related to the execution of our custom shader:

- declares uniforms, variables that can be set from outside the shader
- defines what data gets passed to the shader and between stages (appdata and v2f respectively)
- passes (i.e. different stages of the render process)
- sub shaders (i.e. variations of a shader for different systems)
- properties (i.e. uniforms that are exposed to the material inspector UI)
- various details like tags (used to filter passes), level of detail, depth test types etc.

Many of these aspects can be tinkered with depending on the development needs, but can also serve as pitfalls when alterations break the shader in obscure ways.

3.4 Version Control

Finally, before we get into the concrete design and implementation of this application, we should briefly discuss our versioning software, and the choices we made around how the repository should be interacted with.

Perhaps unsurprisingly, we are using *Git* as our version control tool of choice. It is widely adopted and well understood in modern software engineering circles and by far the most popular versioning tool to date with a multitude of openly available (or easily self-hosted) repository management services. Most such services also offer a variety of useful project management, planning, continuous integration and automated testing tools to sanitize code flowing into the codebase and to enforce agreed-upon standards⁷.

Our commits follow the *Conventional Commit*⁸ schema, which intends to produce commits which are parsable by both human readers and machines. To this end, commits must adhere to the following schema:

⁷ The usefulness of Git is well documented, but as an aside, consider this video essay on how Git helps facilitate consistent and robust project management practices through plain text versioning: <https://www.youtube.com/watch?v=WgV6M1LyfNY>

⁸ Specification: <https://www.conventionalcommits.org/en/v1.0.0/>

Git is by no means novel technology, but is nevertheless a valuable project management aide

```

1 <type>[optional scope]: <description>
2
3 [optional body]
4
5 [optional footer(s)]

```

Where type can be a number of short keywords, primarily feat for new features, and fix for bugfixes. However, others are also allowed, such as chore for code cleanup, refactoring and the like that doesn't affect the overall API in a meaningful way or docs for documentation work. These become particularly important as the API and design of a project harden and changes become more granular.

The body can be omitted entirely or contain whatever textual description may be appropriate for the given commit. As described below, in our case it will be mostly generated as a summary of performed merges.

The footer is commonly used to relegate information to the project's tooling, e.g. letting the repository service know which issues the commit relates to or possibly that a change breaks the existing API.

Conventional commits primarily define the header of commits

While we do not do this here yet, conventional commits can be used to automatically derive the application version. Generally, this happens by traversing the branch history (at least from the last known version) and counting breaking changes, features and patches (i.e. small changes such as bug fixes). If our version number is some M.m.p, then M, or the major version, is the number of breaking changes (usually this is ignored until a first proper release version, or 1.0 version, is specified, as APIs tend to be particularly volatile up to that point). Breaking changes are either identified by a BREAKING_CHANGE: <desc> in the commit footer, or more briefly by an exclamation mark after the type in the commit header. The minor version, or m, specifies the number of feature commits since the last breaking change, and the patch, or p, specifies the amount of bug fixes (and sometimes small changes like typos, documentation changes and minor refactors) since the last feature release.

This can be added at any point later on however, as the version can always be perfectly derived from the commit history.

Can automatically derive version number from past commit history

We choose to implement this schema via the following workflow: initially, a developer opens an issue describing their intended feature or documenting a bug. When they intend to work on this, they open a merge request with respective branch for this issue, which then remains drafted until they are ready to merge.

While working on this feature branch, they can (and probably should) ignore the conventional commit schema. Instead, they should try to make granular, descriptive commits (ideally including a descriptive commit body if appropriate for the changes). Then, when they are done developing and ready to merge, they should either ensure that the merge header fits the conventional commit scheme (since the header will automatically be set to serve as the merge commit's header, and the description as the body), or ensure that they alter the merge commit appropriately. The merge template is set up such that all commits will be automatically appended to the body of the merge commit.

Only apply commit schema to merge commits, branch commits as changelog

We opt to squash the commits involved in a merge. This reduces the pressure to commit “correctly”, as any undesired commit messages can be eliminated from the final merge and repeated changes of the same file, e.g. to eliminate typos end up getting expunged by the squashing process, leaving only the results of the merge in the overall history.

*Commits are
squashed on
merge to sim-
plify merge
commits*

This also helps with merges later on, as frequent changes to the same file are a common source of merge conflicts, even if they will be rolled back later on in the same branch. Thus, eliminating these transient changes upfront avoids having to deal with them later on.

For merges, we choose to prefer rebasing over fast-forward merges. If conflicts arise that can’t be simply resolved in the GitLab interface, developers should instead rebase to the current state of the master branch and eliminate conflicts along the way. Copying a branch and moving only particular commits into a new branch using interactive rebasing can also be a strategy to this end, similar to cherry-picking. Particularly when a branch might be better off being split into multiple features.

Chapter 4 Design and Implementation

In this section we will discuss the various stages of the learning simulation in detail. We will highlight aspects of the implementation and discuss limits, problems and improvement opportunities.

We will start out with a short outline of the application. The design philosophy and how we choose to convey learning information to users. We will then go on to discuss the various simulations we implemented for users to experiment with, starting with a simple orbital model that drives the sky above the users' heads. We will then discuss our implementation of shadow volumes, and finally look at a particular case of the classic pinhole camera model.

4.1 Overview & Hub

Before we get into the respective experiments, that users will get to engage with in this application, we will take a brief look at the overall concept. We will talk about the learning concepts, lay out our intentions as to how users interact with this application, and finally we will introduce the hub and discuss our spin on the guided tour, which serves as a red thread through the learning experience.

Initially, we considered different approaches for this application, including a more gamified version which would see users learn on their own terms through objection-based puzzles. We ultimately opted for a more simulation-based approach, in which learners will engage with a number of experiments in a controlled, directed way.

In each experiment, there are a number of variables they can control to see how different actions may affect the systems in front of them. To keep this engaging, users interact with the world through physical buttons and levers that require navigation through the VR world.

4.1.1 The UFO

At the beginning of a session, users find themselves on board of a grungy space station of sort. They are situated in a near-earth orbit, with a clear view of the starry sky all around them. Once they have acclimatized to the VR world, they can begin a guided tour, which provides them with helpful information and prompts them with suggestions to nudge them into learning about the simulations they are engaging with and the physics behind these simulations.

*Spaceship as
setting*

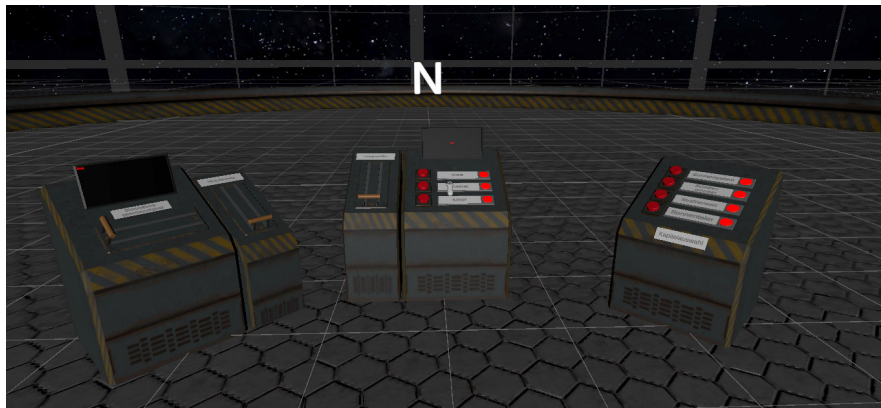


Figure 4.1: A view of the cockpit.

*Simple, dis-
tractionless
hub*

The hub serves as a neutral environment to stage the simulations that users get to engage with. It is minimal in design, mostly empty except for the elements that users may engage with. In particular, this involves the currently displayed experiment as well as a set of control consoles, which users can interact with to interface with the simulations. Along the walls, we furthermore have indicators for the cardinal directions, which can be used as an orientation aide, especially in the solar system section of the simulation.

*Prefer
diegetic,
active user
interface*

When implementing controls, we wanted to emphasize diegetic, physical interaction. Users slide around levers or press buttons. As indicated in section 2.1, we hope that this provides a more engaging and immersive experience for users.

4.1.2 Learning Topics

As this application aims to convey knowledge about the topic of middle-school optics, we had a number of topics to choose from. Particularly, we decided to focus on shadows in this application.

- We start off with section 4.2, where learners can experiment with how moon phases come to be, or view how light and shadows affect earth from the moon's perspective. To enforce the understanding of heliocentric orbital motion, we also include the sun as a static reference frame and simulate orbital motion with a simple, but realistic approximation.
- In section 4.3 we then continue with a more detailed investigation of shadows, primarily how they shouldn't be thought of as some sort of 2D object, but instead the 3D space they occupy behind an occluding object.
- In section 4.4 we then go on to look at pinhole cameras, specifically a special case of the pinhole camera that we call sun coin (after the German word "Son-nentaler")

4.1.3 Guided Tour

The main didactic element, that leads learners through the experiment, is a guided tour experience, similar to applications such as RePiX VR [HGS22]. A machine-generated voice plays, reading bits of informative texts to the learner as they progress. Meanwhile, a screen in the scene shows relevant keywords, tasks, short descriptions etc. depending on the spoken text.

Provide spoken instructions and information

On the technical side, the text for the guided tour is kept in a simple TOML file, a serialized file format similar to JSON or YAML, which has some ergonomic features that we can make use of.

Reliable local speech synthesis

Serializing our speech data means it is easy to feed into any given text-to-speech solution. Currently, we use a free, locally run speech synthesis framework called Coqui TTS¹. This produces reasonably human sounding (albeit noticeably artificial) voices with some small caveats (primarily, issues with umlauts in the German language model we are using) that need to be worked around.

If, at some point in the future, this isn't sufficient anymore, and we want a more sophisticated model, or we want to offload generation to some cloud provider such as OpenAI or Google Cloud, which offer fairly affordable price models for the amount of sound data we would require, we can simply alter the responsible Python script accordingly.

Unity only cares about TOML file and audio data so we can switch generation methods whenever we want

Should we want to move generation entirely into the Unity codebase to skip pre-generating the audio files with a separate Python script we can also do that later on as long as we keep to the designated structure. In Particular, `Assets/StreamingAssets` must contain a folder `guided_tour`, which contains a `tour.toml` and sound files `audio/[section]/[id].wav`, where `section` corresponds to the various sections in the `tour.toml` and `id` is some identifier. The relative paths to each sound file (from the audio directory) must be designated in the TOML file via a `path` option. This option will be automatically generated by the current Python script.

As we only ascribe a specific structure for the output files, it doesn't matter how this structure is generated, which gives us sufficient flexibility to change implementation details later down the line.

Plain text data is robust and simple

The TOML data format offers some conveniences over other data formats, particularly JSON, YAML and XML, which are often used for similar tasks. In particular, it is important to keep in mind that these formats may introduce technological hurdles that make it harder for parts of the development team that *do not* have suitable editing tools or experience with such tooling to work with the data. Interdisciplinary fields like learning technologies are sure to benefit from keeping tools simple and inclusive.

Simplicity

Perhaps the biggest advantage of TOML is the simplicity of its data schema. Where YAML and JSON are concise, especially for nested data, they also bring a slew of syntactic intricacies that may trip up unsuspecting contributors. Particularly JSON has a number of rules that make it easy to create invalid files. Keys must be quoted, trailing commas are forbidden and there is an extensive use of brackets in the specification which can easily cause trouble without an editor that validates the file automatically.

Fewer syntax element that impact usability for non-technical contributors

¹ GitHub: <https://github.com/coqui-ai/TTS>

With YAML files, the biggest issue is perhaps its *significant whitespace* concept, that requires consistent indentation to produce valid nesting, which in turn may easily trip up unsuspecting developers that are not familiar with YAML.

However, TOML is written entirely in a flat, config-like manner, an example of what a single section might look like is provided in listing 4.1.

```
1 [[section]]
2 text = "This is some normal text."
3 display = ""<b>This is</b>
4 text formatted like
5 <i>Unity Rich Text</i>""
```

Listing 4.1: Some lines of what our TOML file might look like

Any given section (here, named simply `section` as an example) consists of an arbitrary number of the above code snippet, one per voice line. Notably, we can easily add fields (like `path`) programmatically to reduce complexity. Fields that are not required can be silently omitted or filled with default values during processing in Unity.

Easy to add
new entries
or extend
schema with-
out breaking
things

Consider for instance the following hypothetical extension of the previous guided tour with a simple multiple-choice selection, perhaps for a quiz feature or customization of the guided tour experience:

```
1 [[section]]
2 type = "multiple-choice"
3 text = "Pick some"
4
5 [[section.option]]
6 text = "option 1"
7 correct = false
8
9 [[section.option]]
10 text = "option 2"
11 correct = true
```

Listing 4.2: What a hypothetical future extension of the schema may look like

We can use the `type` parameter to identify the purpose of a specific field and then specify options as sub-items that we can then process on parsing.

Thus, even without much technical knowledge, contributors may easily extend the script of the guided tour while leaving implementation details to programmers.

Multi-line Strings

Mostly in contrast to JSON, we can also specify multi-line strings in a TOML file. In listing 4.1 we can see this happen for the `display` property, which can then be used as is to display as rich text in Unity's text mesh components.

Splitting
strings
across lines
is more read-
able and
works well
with rich text

In some cases, this is also useful simply to avoid working with outrageously long lines. This especially holds for editors that do not offer line wrapping or for users that might not know how to toggle line wrapping or not want to do so.

Notably, line breaks in multi-line strings *will* be encoded, which may be undesirable in some cases.

While we can't easily call Unity functions from the TOML file's contents (e.g. to dynamically display some value from the game world in our displays), this can be resolved via an indirection by specifying a template syntax that is then later reified when the text is rendered.

Robustness

As mentioned above, TOML is particularly forgiving compared to other data formats that emphasize nesting and structure. But using any form of serialization, or a data file format to handle this kind of data, adds to the overall robustness of the guided tour feature.

For one, our data is safe and easily versioned. Changes can be tracked and reverted without any major issues. If the data in the actual application corrupts or breaks, we can simply re-bake the plain text data and regenerate the data that is serialized in Unity.

Unlike Unity's data, which is kept in `.unity` and `.asset` files, which are either binary or large YAML files with obscure internal IDs depending on project settings, we can easily edit our data with tools of our choice, use common editing tools like `grep` and `sed` (or any tool that works on plain text files, really) to programmatically collate or alter parts of the file and even generate it easily from scripts if we see the need (or alter it, as we do for relative paths already).

Although it is unlikely to ever be needed, we could even add automatic validation tools and test the schema of the guided tour script if we need an additional layer of safety.

*Easy to read
and generate,
no obscure file
ID pointers*

In Unity

On the Unity side, we then build the guided tour script from a slightly post-processed TOML (where we add in relative paths to the audio files) and the generated audio files, which are summarily stored in `Assets/StreamingAssets`.

As this process is mostly data-driven, adding new sections and altering the previous content of the guided tour is as simple as editing the underlying TOML file, which is doable even without extensive technical knowledge as long as the basic underlying structure is respected. This allows non-technical collaborators to extend the guided tour or add new sections entirely. As sections still need to be explicitly connected up with content on the Unity side, this process isn't fully automated, but it helps separate concerns of the narrative design of the application from the implementation details of the corresponding scenes.

Through the addition of new options, behavior can be further extended as long as this is appropriately handled by the development team. For instance, adding options for a multiple-choice quiz at certain parts of the script. Depending on complexity, it may be prudent later on to additionally allow splitting this across multiple files for an easier to organize developer experience.

*We simply
need to
generate
narration
artifacts from
parsed script*

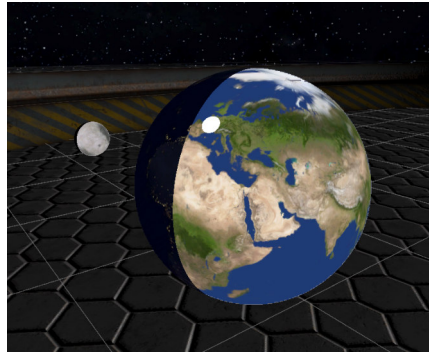


Figure 4.2: A view of the partially shaded model earth. The white spot marks the current viewing location in europe.

4.2 Solar System

Moon appearance depends on its shadow

We are, in our daily lives, surrounded by light and shadow at all times. One of the primary light sources in our life is, inevitably, the sun. Furthermore, one of the most impressive examples of the interplay of light and shadow we perceive in our daily lives is found in the celestial bodies in our sky. Specifically, how the sun casts shadows on this very earth (i.e. the day and night cycle) as well as our only natural satellite (that is visible with the naked eye).

Thus, the solar system was picked as a backdrop for the simulation with the framing of the user space as a spaceship that can travel through space to help learners observe the world from different positions and angles.

4.2.1 Outline

Interactive

model solar system

Within the “ship”, learners can see a model version of the solar system. It is small enough to navigate comfortably to explore from different angles. Furthermore, both the earth and the moon can be interacted with to switch the position of the “spaceship” in the solar system. Thus, learners can experience how days look in the polar circle or experience how different the sky may look from the tidally locked moon.

Solar system hard to display without breaking scale

While the model system is not to scale (primarily due to the vastly greater size of the sun compared to all other bodies in the solar system, and it’s great distance to the earth), it follows an approximation of the correct orbits (albeit scaled to fit the play space).

It is also important to note that in the model view it is evident that the earth revolves around the sun just as the moon revolves around the earth. This is to avoid any misunderstandings when comparing with the large-scale version.

Model solar system reflected in sky

The large-scale version of the solar system, or sky box, appears *outside* the ship. It follows the same motions as the small model within the ship, but adjusts the perspective to whatever position is chosen for the ship by interacting with either the earth or the moon. While this uses the same input data as the small scale solar system, some additional processing was needed to mitigate errors and shift the perspective as needed. However, this also allows us to scale the actual distance between bodies in the system uniformly to “zoom” into the planets to get a better view.

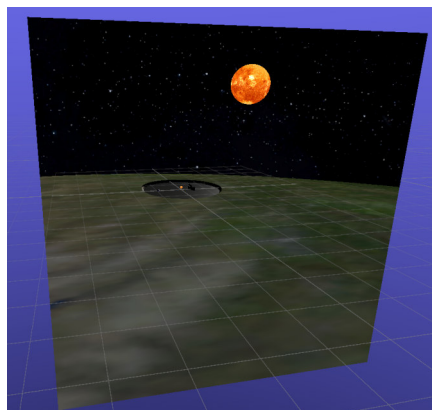


Figure 4.3: Sky box shader on the inside of an inverted cube around ship. This perspective is usually not visible to users.

Where there are no planets to render, learners will be able to see the starry sky that also moves relative to their own orientation as time passes. Finally, the position of the sun in the sky affects the global “sun”, that is the directional light source that serves as the main light source for the scene. At night, the ship will be considerably darker as a consequence.

Learning Goals

While it can be a stunning backdrop, the solar system can also help learners understand a variety of concepts. There are two main tools at the learners disposition to explore this feature. One, a slider, allows them to increase or decrease the speed at which time progresses in the simulation. This serves as an observational aid. Slowed down, learners can inspect the current moon phase and compare their observations to the model view to understand how the current location of sun, moon and earth result in what they are perceiving. Sped up, they can see much easier how the bodies in the sky move, helping them find a specific state they’re looking for or allowing them to observe phenomena such as how their position may affect the position of the sun in the sky.

Time acceleration to quickly observe changes

The other, the model earth and moon, allow them to pick different locations to observe the system from. This allows learners to experiment with positions and situations that they may not encounter easily in their everyday lives, such as how differently the sun moves on the equator or in the Southern Hemisphere, how the earth moves as viewed from the tidally locked moon or how the sun never sets or rises in the arctic regions at certain times of the year.

Perspectives from both earth and the moon

Thus, learners have several avenues to experiment with the simulation at their own leisure. From the perspective of the optics topic of this work, the main factor that interests us is of course how different moon phases occur and potentially how eclipses, both of the sun and the moon, come to be.

4.2.2 Orbits and Planet Data

Before we get into the techniques used to render our solar system, we should first address how we modeled planets for this feature. An initial draft we implemented

Object / Distance	Real (km)	Unity Units
Earth diameter	12 756.00	1.00
Sun diameter	1 392 700.00	109.17
Moon diameter	3474.00	0.27
Sun – Earth (1 AU)	149 000 000.00	11 680.78
Earth – Moon	384 400.00	30.13

Figure 4.4: Real-world distances compared to hypothetical model scale.

simply used Unity’s own Transform components to model the motion of orbital bodies. This worked, but had a number of caveats:

Real orbits are elliptical While this could just be explained away as an abstraction of the model, orbits in reality are not exact circles like they would be if we were to utilize simple 3D transformations (precisely: rotation) to model them.

Consistency For the two systems to be comparable, they must reflect the same state, so we must ensure that they are synced up correctly to be consistently compared.

Scaling issues As we will get into momentarily, it is unfeasible to simply scale down the solar system for our model view. Thus, it is necessary to allow for the model system and the sky box solar system to use different scales. Primarily, the distance of bodies to another is problematic.

Time Control We need some centralized way to control the timing of orbits to ensure that planets do not run out of sync over time. Ideally, this would also be reversible to an extent.

Size and
distance
differences
make scale
models difficult

As mentioned, model scale is a huge pain point here. Distances even within the solar system are fairly monstrous. Figure 4.4 shows how distances work out if we assume the model to be scaled such that the diameter of earth corresponds to one unit within the Unity engine. While the size difference between earth and moon is bearable (and made it into the current version), trouble already arises with the distance of earth and moon. The current version of the ship was designed to accommodate only the moon orbiting the earth at this scale. But with the moon this small and distant it is often already a challenge to quickly locate it within the ship, not to mention identifying how it is currently shaded. Furthermore, without the sun *within* the ship, it becomes hard to identify from which direction light even falls onto the earth and moon leading to ambiguity and muddling the experience.

Things only get worse once the sun is involved. At its default distance and scale, it easily falls outside of Unity’s default render distance *even in the model view*. So clearly it cannot be an option to even attempt to keep the model version of the solar system at a realistic scale.

The answer to this issue is to simply embrace the model abstraction and shrink down distances as needed, especially between the earth system and the sun, to fit things into a more easy to traverse space.

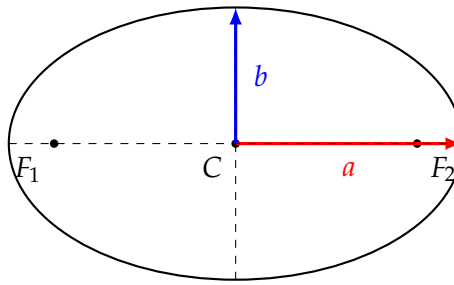


Figure 4.5: Ellipse geometry: semi-major axis in red, semi-minor in blue

Data-driven Orbits

If our system is at model scale now, we run into a different issue: It becomes much harder to undo the model abstractions for “real-scale” solar system in the sky box shader. Previously, we could simply pick a point in the model system, derive a suitable transformation into that point’s local coordinate system and pass that to the shader to apply where needed. Now, since we have different scaling factors for different distances in the system some additional work is needed to cleanly derive our position in space.

Thus, if we need to invest the extra effort either way, it becomes increasingly appealing to switch to an entirely data-driven model of the solar system. In this context, data driven means that we will derive the final positions of all orbital bodies from a mathematical model that we can reify in different ways depending on our needs. Specifically, we can apply some scaling to it when using it to display the model system while keeping the data unaltered for the real-scale system.

There exist many approaches to model systems of orbiting bodies. One approach may be to fully simulate the physics of orbiting bodies, which may sound ambitious. Considering the relatively low amount of bodies in our simulation, this approach is definitely viable to a certain degree. Whether it is exactly what we need is a different question, however. Unity is definitely suited for moderately complex physics simulations. It provides a separate Update lifecycle for fixed-time simulations (which make physics calculations more predictable compared to the oftentimes erratic frame time differences of the default update cycle) and offers suitable tools for computationally taxing tasks through the use of compute shaders. This would definitely help produce realistic looking orbits and if we share the simulation data between the sky box shader and the scale model solar system, we would also have a consistent state to compare.

Time control is a bit more tricky. Scaling the system time inevitably introduces rounding errors that may escalate to some disagreeable state as the simulation continues. Furthermore, this may make it harder to work out specific scenarios in the future. For instance, one day it may be interesting to represent the system at a specific point in time, or to initialize the system to the user’s current time and locale. Finally, it may also require a decent time investment to tweak parameters of the simulation to a degree where it acts as expected.

Instead, we decided to derive the state of the solar system from the orbital characteristics of the modeled planets, mainly using the **semi-major axis**, **eccentricity**, **inclination**, **longitude of ascending node** and **argument of periapsis**.

Problem: undoing model scale is hard

Instead: scale down from consistent model

Modeling gravity too inconsistent and taxing for our purposes

Time scaling exacerbates rounding errors

The **semi-major axis** is the half-diameter of an ellipse along the longest axis crossing the center. That is, it spans from one point on the ellipse to the center while also crossing either of the focal points. We define this to be a distance in kilometers for the sake of our model.

Perpendicular to the semi-major axis is the semi-minor axis (labeled a and b respectively in figure 4.5) which is the shortest half-diameter in the ellipse. It is related to the semi-major axis through its **eccentricity** e as follows:

$$b = a\sqrt{1 - e^2}$$

Eccentricity can be thought of as a number that determines how circular our orbit is, where

- $e = 0$: perfectly circular orbit
- $e = 1$: parabolic orbit (i.e. escape trajectory)
- $e > 1$: hyperbolic orbit

We will only be looking at eccentricities $e \in [0, 1)$ for this application.

The **inclination** is an angular measure of deviation of our ellipse from the reference plane. The points where the ellipse intersects the reference are called ascending and descending nodes, determined by how the orbital body moves in respect to the reference plane.

There are several candidates for such a reference plane, particularly the ecliptic which is the earth's orbital plane and the invariable plane which has its origin in the solar system's barycenter and uses the angular momentum vector as its normal vector. We will be using the invariable reference plane for our purposes.

The **longitude of the ascending node** specifies the angle from the chosen reference direction to the point where the orbit crosses the reference plane from south to north. In our case we work in the ecliptic/J2000 reference frame, where the reference plane is the ecliptic and the reference direction is the First Point of Aries.

Finally, the **argument of periapsis** is the angle of the ascending node to the periapsis, i.e. the point on the orbit closest to the parent body.

Caveats

These parameters are publicly accessible astronomical data. They have been extensively measured and calculated for various orbital bodies, especially the ones in our solar system. Using this data we can now aim to describe the orbital motion mathematically. It is important to note, that these parameters may change over time due to the complicated interplay of gravitational attraction of matter in the local solar system (and beyond). The orbit of a planet may be affected both by its own satellite and even other planets, especially in their neighborhood. Previously, unexpected deviations from the mathematical orbits were successfully used to predict the existence of Neptune through its effects on the orbit of Uranus [Gal46]. This is especially noticeable in the orbit of our moon (or any satellite). The position of the moons ascending node and its orbital eccentricity slowly but noticeably change over time, although

Orbits are
dynamic
systems that
constantly
change

these changes are cyclical in nature. Over time the moon also slowly escapes from the earth's influence, increasing its mean distance by ca. 3.8cm every year [Ste20]. Consequently, events such as solar eclipses may one day stop to occur entirely.

We neglect to implement such effects in our simulation. Some simple approximation could be achieved by simply emulating these gradual shifts through dynamically adjusting our orbital parameters over certain time spans as most orbital shifts happen in cycles of known length. However, this has little effect on our stated learning goals, and if a more sophisticated simulation was required it may be worthwhile to simply implement a more precise and realistic approximation rather than to extend our simple model.

We also omit the concept of barycenters. In reality, rather than orbiting just the center of the heaviest planet, systems of orbiting bodies will rotate around a common barycenter depending on the distribution of mass in the orbital system. This becomes more noticeable as the difference in mass shrinks. Two bodies of equal weight may rotate equidistantly around a shared center while in the system of earth and moon the barycenter is much more biased towards the earth, being located within the much heavier body. Even with discrepancies in mass, rather than sit static this will add a sort of wobble to the heavier of the two bodies on top of the orbital motion of the lighter one.

*Omit
barycen-
ters as orbital
centers*

That is to say, our goal will not be to permanently and consistently predict the state of the solar system at all times, but rather we will choose a practical mathematical model that, while not entirely consistent in its model of the real world, nevertheless fulfills its purpose of modeling a believable night sky for learners to explore.

4.2.3 Approximating Orbital Motion

Now that we outlined how orbits are characterized and discussed what we will and will not be simulating, we can begin detailing our approach to the approximation of orbiting bodies. Our goal is then to derive a function that takes orbital parameters as defined above as well as some time value as inputs and returns a position for the respective planet.

Basis to our calculation is the `OrbitalBody` class. This class consists of the parameters outline above and serves as a data source for orbital motion.

For our approximation, we will follow a simplified version of the two body problem, where we only ever consider the movement of one body around another. For our model, the parent body always remains static. If, like the earth, it is actually in motion, we simply propagate its motion to the orbiting body, effectively shifting its coordinate system. While this abstracts many of the physical realities of orbital motion, it comes at a lower computational cost and is deterministic, which makes scenarios repeatable and gives us greater freedom to manipulate the timeline.

*Model orbits
as two body
problem,
i.e. neglect
outside
influences*

The goal of our function will be, to derive a solution of Kepler's equation [Zec18]

$$M = E - e \sin E$$

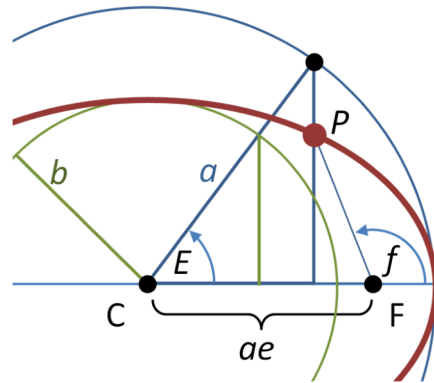


Figure 4.6: Eccentric and true anomaly. From Brews ohare, *Eccentric and True Anomaly*, Wikimedia Commons, link, CC BY-SA 3.0.

Here, e is the *eccentricity* as defined above. It is an intrinsic characteristic of the orbit that is part of our input data.

The *mean anomaly*, designated as M , describes the fraction of an orbital period as an angle past the periapsis, that is, the lowest point in its orbit. Note, that this is *not* the angle spanned by the orbiting body and the periapsis, rather it is the angular position of a fictitious object that moves along a perfectly circular orbit at a constant speed but with the same period as our real orbit. We can think of this as a measure of progression through the orbital period, i.e. how far along we are in the current period.

mean
anomaly
is solely de-
pendent on
time

This is actually fairly simple to compute. For some time value t we have

$$M = 2\pi \cdot t$$

Here, t simply denotes the progress of the orbiting object in the current orbital period. In our simulation all orbiting bodies will share a common time value, a measure of simulated seconds since the start of the simulation. This measure differs from real seconds elapsed by how fast the simulation is set to run by the learner.

For individual orbits, we will then scale this time value according to their respective orbital periods to once again represent the progression through their current orbit. That is, for some period P (measured in seconds) and some shared time value τ we get

$$t = \frac{\tau}{P} \implies M(\tau) = \frac{2\pi\tau}{P}$$

The *eccentric anomaly*, designated as E , is somewhat less straight forward. Geometrically, it is derived fairly simply. Consider figure 4.6, where the eccentric anomaly is designated as the angle E . Like with the mean anomaly, we need to consider a hypothetical circular orbit, this time centered on our ellipse as before but with radius equal to the semi-major axis of the ellipse. If we now draw a line perpendicular to the semi-major axis through the position P of our orbiting body, the angle between the intersection point of this line and the semi-major axis (more concretely at the periapsis) is our eccentric anomaly E .

The eccentric anomaly serves primarily as a mathematical tool to derive the *true anomaly*, which is the actual angular position of the orbiting body, i.e. the angle between the periapsis and the position of the body on the orbital trajectory, as seen from the focus point (where the orbited body resides). In figure 4.6 this corresponds to the angle f .

There is no closed form solution when trying to solve Kepler's equation for the eccentric anomaly. Instead, we will be using an iterative approximation based on the Newton-Raphson method [Gal00, Ló18].

We start by bringing Kepler's equation into a suitable form for the Newton-Raphson method, by rearranging it such that $f(E) = 0$:

$$M = E - e \sin E \quad (4.1)$$

$$\iff 0 = E - e \sin E - M = f(E) \quad (4.2)$$

$$\implies f'(E) = 1 - e \cos E \quad (4.3)$$

From this we can derive the iteration term needed to approximate E . We need an initial guess as a starting point. For this we simply use the value of the mean anomaly we calculated above.

$$E_0 = M \quad (4.4)$$

$$E_{n+1} = E_n - \frac{f(E_n)}{f'(E_n)} \quad (4.5)$$

We can simply keep iterating on this until the change becomes sufficiently small (or until we hit a maximum threshold of iterations in cases where this approximation doesn't quickly converge).

True Anomaly Estimation

Now that we have both the mean and eccentric anomalies, we need to determine the true anomaly ν , which will give us the actual angle to our orbiting body.

The true anomaly is related to the eccentric anomaly as follows:

$$\cos \nu = \frac{\cos E - e}{1 - e \cos E} \quad (4.6)$$

$$\sin \nu = \frac{\sqrt{1 - e^2} \sin E}{1 - e \cos E} \quad (4.7)$$

$$\implies \nu = \arctan 2(\sin \nu, \cos \nu) \quad (4.8)$$

Position Estimation

Finally, we can use the remaining orbital characteristics in conjunction with the freshly estimated true anomaly to determine the actual position of our planet in the space around its parent body.

Our approach will be to determine the position of the orbiting body in two dimensions. This is called the perifocal coordinate system, which is defined such that one unit vector points towards the orbit's periapsis, another points to the 90° position of the

Derive eccentric anomaly E via incremental solver

Determine orbit in parent bodies reference frame

true anomaly and its third unit vector points in the direction of the orbits angular momentum, i.e. perpendicular to the plane of the other two unit vectors.

We begin by determining the radius of the orbit (cf. [At02], p. 4):

$$r = a(1 - e \cos E),$$

where a is the semi-major axis of the orbital ellipse. With this we can determine the coordinates of our orbital body through simple trigonometry from the true anomaly:

$$x = r \cos \nu, \quad y = r \sin \nu,$$

where ν is the true anomaly calculated for the current time value. Finally, we need to construct a rotation of the perifocal plane into the orbital plane of the parent body. This is where our remaining orbital parameters finally come into play.

We will be using the following definitions:

- i : inclination of the orbital plane
- Ω : longitude of the ascending node
- ω : argument of periapsis

First, we define the rotation matrices R_x and R_z to be euler rotation matrices such that

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}, \quad R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We can then assemble the final rotation matrix as an euler rotation sequence as follows:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = R_z(-\Omega) R_x(-i) R_z(-\omega) \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \quad (4.9)$$

$$= \begin{bmatrix} -\cos \Omega & \sin \Omega & 0 \\ -\sin \Omega & -\cos \Omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\cos i & \sin i \\ 0 & -\sin i & -\cos i \end{bmatrix} \begin{bmatrix} -\cos \omega & \sin \omega & 0 \\ -\sin \omega & -\cos \omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \quad (4.10)$$

$$= \underbrace{\begin{bmatrix} \cos \omega \cos \Omega - \cos i \sin \omega \sin \Omega & \cos i \cos \omega \sin \Omega - \sin \omega \cos \Omega & \sin i \sin \Omega \\ \cos \omega \sin \Omega - \cos i \sin \omega \cos \Omega & -\sin \omega \sin \Omega - \cos i \cos \omega \cos \Omega & -\sin i \cos \Omega \\ \sin i \sin \omega & \sin i \cos \omega & \cos i \end{bmatrix}}_R \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \quad (4.11)$$

We can apply this rotation matrix R to the position of the orbiting body we determined previously, to calculate its position (x', y', z') in its parent planet's inertial reference frame.

This inertial frame corresponds to the global coordinate system used by the engine, where all orbital bodies are positioned relative to their parents.

4.2.4 Planet Data Management and Pre-processing

With this, we have a working approach to modeling orbital movement. We now need to relegate this data to our game objects and the renderer. On the Unity side, the behavior of planets is determined by four classes:

- `OrbitalBody` is a plain class that defines orbital parameters and simulates unperturbed two-body orbital movement as mentioned above
- `PlanetBehavior` is a component class that wraps an orbital body and maps its results (based on the current time step) to the game object's Transform component and shader material
- `SolarSystemController` is a component class that initializes planet bodies and coordinates updating the current time step to all active planets (based on seconds elapsed and a dynamic scaling value affected by the user)
- `CanopyController` is a component class that translates orbital data (via planet behaviors) to the canopy shader, i.e. the skybox

One complication arises right away. Since the scales in the solar system are rather large, one may quickly run into precision issues depending on the chosen model scale. The distance between sun and earth is defined to be roughly one astronomical unit (AU) or $1.5 \cdot 10^8$ kilometers.

If we were to model this at a kilometer scale on a standard IEEE-754 floating point number this would correspond to a conversion error of 4km^2 .

Particularly problematic is the fact that we will be dealing with affine transformations with large translations, which will amplify errors when applying matrix transformations later down the line. We can mitigate this somewhat without impacting performance too much, by doing our model calculations with *double* precision instead of floats. This covers most of the potential precision errors, particularly when we apply transformations later on. We merely need to defer the conversion to floats as much as possible.

To help with this, Unity provides the `Unity.Mathematics` package that implements a variety of operations and data structures that follow shader-like semantics but also support double and fixed-point precision. We use this to supplement the otherwise poor, built-in double math API provided in Unity.

Final positions are then converted into single-precision floating point numbers only where needed. To handle scaling, specifically in the model view, every orbiter supports a scaling parameter that is applied to their distance from their parent body. This consequently also affects their satellites, which themselves can also be scaled.

In the model view we ignore a planet's radius entirely, instead opting to simply utilize the object scale defined by their Transform component.

Potential precision errors with floats

*double precision through Unity.Mathematics package
Convert only at the end*

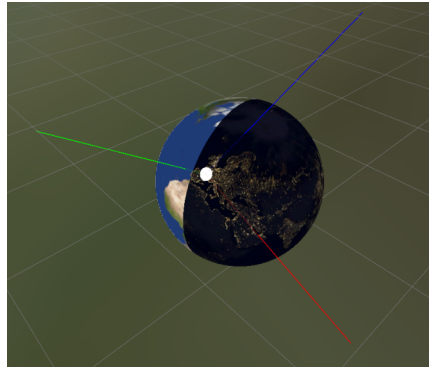


Figure 4.7: Basis vectors displayed over user position on globe.

Canopy Controller

Shift so-
lar system
into viewer
perspective

Before we can actually render our planets, especially the ones in the skybox, we need to do some further adjustments. In particular, we need to shift the perspective in our simulation to the point of view determined by the learner. As the user picks a position in the model view, either on earth or the moon, we need to transform planet positions accordingly such that the skybox matches up with the modeled positions of the planets.

First, we need to determine a reliable orientation, determined by an **up**, **forward** and **right** direction. This corresponds to the basis vectors of the learners local coordinate system.

The **up** vector is easiest of the three, we merely need to determine the normal vector at the users position. This derives from their position on the planet and is trivial. The only caveat here is, that we need to additionally consider axial tilt and the current rotation of the planet.

We can convert this information into a quaternion and apply it to the user-selected position, labeled normal here as follows, given an obliquity (angle of the axial tilt) and the current rotation (degrees from starting position):

```
1 var rotation = Quaternion.AngleAxis(CurrentRotation * 360f, Vector3.up)
2   * Quaternion.AngleAxis(obliquity, Vector3.right);
3
4 var up = rotation * normal;
```

User perspec-
tive's forward
vector points
north

For the **forward** vector, we need to determine a point of reference. Since we place users on a globe, we deem a north-south alignment, where the scene's forward direction points north, to be the most suitable. This is a commonly used reference direction in many maps and navigational tasks, so learners may already be familiar with it. But even if they are not, it should still serve its purpose by keeping a comparable reference direction.

North Pole
aligns with
planet's local
up vector

The directional vector of the North Pole can be determined as the local coordinate system's up vector, although it is subject to the same rotation we applied to the user's up vector above. We can then simply determine the projection of the north poles direction onto the previously determined up direction and subtract it from the north

² Determined using this calculator: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

direction to receive a tangent vector that points towards the North Pole:

$$\mathbf{forward} = \mathbf{north} - \text{dot}(\mathbf{up}, \mathbf{north}) \cdot \mathbf{up}$$

Next, we can derive **right** as the (normalized) cross product of the forward and up vectors. For consistency, we also re-project the forward direction as a normalized cross product of the up and right directions. Now we have a set of orthogonal, normalized vectors that form an orthonormal basis for the user's local coordinate system. A visualization of the vectors overlaid on the model earth can be found in figure 4.7, where the forward, up and right directions are colored blue, green and red respectively. With this we get the following basis transformation and inverse transformation:

$$T = \begin{bmatrix} R_x & U_x & F_x & 0 \\ R_y & U_y & F_y & 0 \\ R_z & U_z & F_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \iff T^{-1} = T^T = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ F_x & F_y & F_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix T encodes the transformation of directions from the user's perspective into the world space. We want the opposite, translating the world-space coordinates from the model into the user's point of view. To this end, we need to invert the transformation.

Since T is orthonormal by construction, as seen above, the inverse can be trivially achieved through simple transposition. Notably, even though we use a 4×4 matrix for this transformation, we handle translation separately to avoid lossy floating point operations.

Even combined, the inverse would be rather trivial:

$$T^{-1} = \begin{bmatrix} R^T & -R^T \cdot t \\ 0 & 1 \end{bmatrix}$$

Where R is the rotation as defined above, and t is the translation vector.

Initially, this choice was made to deal with what seemed like a clear indication of floating point precision issues, likely incurred by the shader during application of the transformation into the observer's inertial frame. While it is not entirely clear in retrospect whether this was as significant as it seemed, this method does make it much easier to relate the position of planets in the sky box to positions in the Unity scene later on as they are now computed in a component that is easily accessible to other scripts in the ecosystem.

What seems like floating point errors may sometimes just be interpolation issues

Finally, we package up the data (after converting positions to floats for the shader) and send it over. The radius (for any planet that isn't the one the player is situated on currently) is also manipulated by an additional scaling value to allow users to "zoom in" on the planets they are observing. This is particularly helpful when trying to find a planet in the first place and also to make features such as moon phases easier to recognize.

Let users zoom in to see surface details better

4.2.5 Shader-based Planets

Early on in development it became clear, that it might be advantageous to pursue a different approach to rendering the planets than the naive method of simply mapping

Unity Spheres have ugly texture mapping

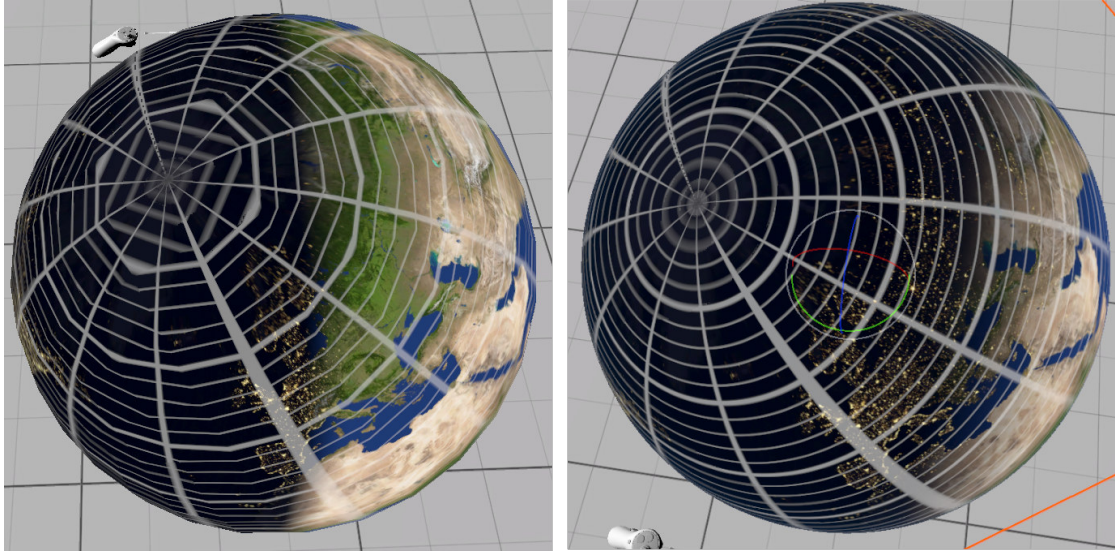


Figure 4.8: UV Mapping of standard unity sphere mesh (top) compared to custom ray tracing shader (bottom).

a texture to some spherical mesh. The default sphere mesh in Unity is particularly prone to unappealing distortions in the polar regions of the sphere geometry. In figure 4.8 we can see a comparison between the UV mapping on a simple sphere mesh compared to the shader-based solution we settled for.

*Remap UVs
based on sur-
face normal*

A first step towards a more appealing mapping is a custom sampling method for texture coordinates rather than standard UVs. However, as 4.8 shows, this can still have noticeable distortions in the polar regions. To implement this, we only need some fairly simple trigonometry and mapping surface normals to a vector $(\text{lat}, \text{lon}) \in [0, 1]^2$, which aligns with the standard UV vector space. The code implementing this transformation can be found in listing 4.3.

While this maps much nicer than the default mapping provided by Unity, and likely looks fairly natural on a more suitable sphere geometry such as an icosphere, it still has some issues. It still comes down to how the normal vector is sampled in the fragment shader stage.

*Normals dis-
torted due to
mesh geome-
try*

By default, Unity interpolates the fragment normal from the normals of the contributing vertices. This works well, but may introduce distortions along the edges which make the underlying mesh structure more noticeable. This can alternatively be alleviated with a higher detailed, more even mesh such as an icosphere. We can also derive the normal by normalizing the fragments position in the sphere's object space. However, this similarly leads to distortions to the degree that the underlying mesh differs from a perfect sphere.

```

1  float2 norm_to_uv(float3 normal)
2  {
3      float long = 0.5 - atan2(normal.z, normal.x) / (2.0f * pi);
4      float lat = 0.5 + asin(normal.y) / pi;
5
6      return float2(1.0 - long, lat);
7  }

```

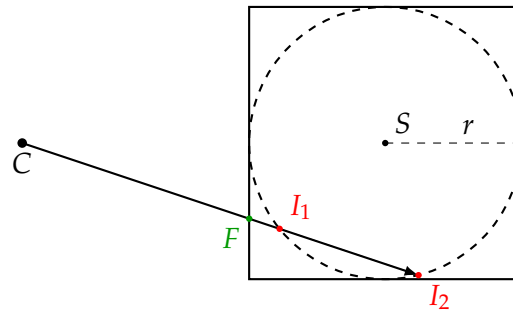


Figure 4.9: Ray intersection, rendering a spherical object onto a cube mesh. Intersection points with the sphere in red and actual fragment position in green

Listing 4.3: Unit sphere position or normal vector to UV coordinate

This brings us to a different approach entirely: ray tracing. Since our issue is that mesh approximations of spheres introduce distortions, we can simply directly describe the mathematical object and render that instead.

Ray Tracing

The concept of ray tracing is used extensively in many applications such as realistic lighting and weather effects [SV15]. While this approach can have a significant impact on performance, a critical flaw when dealing with limited mobile hardware, in our case the burden is more than manageable. To begin with, spheres are rather simple, and ray-sphere intersections are well-defined.

Mathematical sphere instead of mesh for smoother surface

There are two geometric objects we need to figure out for our shader to work:

- the view ray, which emerges from the camera position and passes through the fragment coordinates
- the parameters of the sphere, that is position and radius

In figure 4.9 we can see the general process. We can derive the fragment position, labeled F in the graphic, and the camera position from the data Unity passes to the shader by default. From this we can derive the position to cast our ray from (the camera position C) as well as its direction (by normalizing the vector that spans from the camera to the fragment position).

Mathematical sphere inside of cube mesh

How the parameters for our spheres are determined depends on the situation. For the planets in our model, we simply used a cube mesh as a basis for the shader material. In our planet shader, we then assume that they are simple unit spheres that sit in the center of their own coordinate system. In their shader we can then apply the inverse model transformation (available through `unity_WorldToObject`) to move both the fragment and camera positions into the sphere's object space, where we can proceed as normal.

This has the advantage of working well with Unity's own game object system, so we can simply scale the cube mesh to a size that we think to be adequate for the model planets. This even works if we were to scale the underlying mesh, squishing the rendered sphere accordingly.

```

1  float rot = _Rotation * 2 * pi; // current rotation (range: [0, 1])
2  hit_norm = rotate_around(hit_norm, float3(0, 1, 0), rot);
3
4  float tilt = -_Obliquity / 360.0 * 2 * pi; // axial tilt as radian
5  hit_norm = rotate_around(hit_norm, float3(1, 0, 0), tilt);

```

Listing 4.4: Rotation of the hit normal before deriving UV coordinates.

We apply tilt
and rotation
to normal
vector

While this could also work for rotations, i.e. to model axial tilt or the rotation of the planet itself, we can solve this more consistently (between the model planets and the sky box ones) by rotating the normal vector of the hit point before we use it to calculate the UV coordinates as described in listing 4.3.

The function `rotate_around` found in listing 4.4 is an implementation of Rodrigues' rotation formula, initially described in 1840 [Rod40, Piñ11].

Ray-Sphere Intersection

ray-sphere in-
tersection

Now that we have the needed parameters for our ray cast and the target sphere, the next step is to calculate the intersection points, which are highlighted in red in figure 4.9. Due to the popularity of ray tracing and ray marching, there are plenty of algorithms available for most simple geometric shapes. Our implementation in listing 4.5 was adapted from one such collection of algorithms [Quia]:

```

1  float ray_trace(RayCast r, float3 sphere_pos, float radius)
2  {
3      float3 to_center = r.origin - sphere_pos;
4      float b = dot(to_center, r.direction);
5      float c = dot(to_center, to_center) - radius * radius;
6      float h = b * b - c;
7      return h < 0.0 ? -1 : -b - sqrt(h);
8  }

```

Listing 4.5: Ray sphere intersection.

We begin by determining the vector pointing from the camera (ray origin) to the center of the sphere, which we call `to_center` and project it onto the ray direction (here: `b`).

We then compare the ray origin's square distance to the squared distance of the surface of the sphere to its center (here: `c`). This is a simple distance check to determine if the ray origin is located *within* the sphere. Then we determine `h`, the discriminant of the quadratic equation. If it is below zero, the ray misses the sphere and we can simply discard the fragment. Finally, we get the two intersection points $-b - \sqrt{h}$ and $-b + \sqrt{h}$ of which we return the first one which happens to be the one closer to the ray origin.

We can then calculate the actual intersection point as

$$I_1 = \text{ray_origin} + \text{ray_direction} \cdot (-b - \sqrt{h})$$

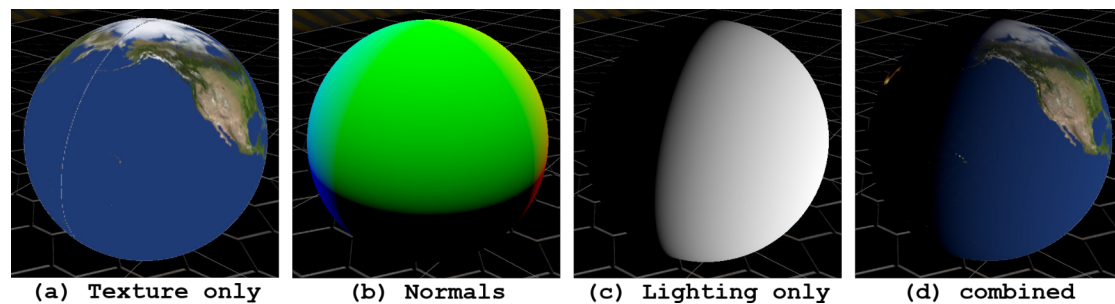


Figure 4.10: Different components that make up the lighting step.

If the return value of `ray_trace` is negative, the camera is located within the sphere. We can generally use this for any similar ray-surface intersection formula, so this process is surprisingly flexible. While not needed in our case, more complex shapes can easily be generated from the combination of multiple primitive intersection functions as long as it is clear which intersection point gets preserved by the output.

Lighting

With the above method we can determine a surface point for each pixel on the screen resulting in a smooth sphere (i.e. no mesh geometry evident whatsoever). We can use the UVs derived by the function in listing 4.3 to sample the correct color from an underlying texture map onto the sphere. In figure 4.10 we can see an example of how this might look like in image (a), which shows the texture-only render of planet earth. Note that due to the mercator projection that underlies the input texture, the polar region still experiences artifacting. While we left it as is, in future work the texture may be replaced with a cube map. Cube maps consist of six different textures that can be sampled from using a normal vector instead of UV coordinates. While a sphere cannot be perfectly mapped onto a sphere, it will result in much less distortion in the extremes than our texture that features heavy distortion as we move away from the equator. Rather than a technical hurdle, the biggest challenge here is to find (or create) a suitable asset.

*directly
sample in-
tersection
points from
texture*

For the lighting term we use simple Lambertian surface scattering. To this end, we begin by determining the position of the sun in the respective coordinate system. Each of the shaded model planets possess a simple component that is tasked with passing the world position of the model sun to the shader. Here, we need to subtract the position of the fragment from it and normalize the result to receive the direction of the incoming light at this specific position.

*cube maps
may provide
nicer texture
mapping still*

Using this direction and the normal position of the fragment (which corresponds to the fragments position in the object space of the planet, normalized) we can then calculate the impact of light on that point using a simple dot product:

*Lambertian
lighting term*

```
dot(light_dir, frag_normal)
```

Notably, we approximate the sun as a point light source. This is primarily a technical limitation as modeling area lights is much more computationally taxing compared to simple point lights. Point lights are characterized by casting hard shadows with distinct edges.

4.2. Solar System

At the same time, the amount of light scattered by the surface of an object is primarily dependent on its orientation to the light source and viewer, or surface normal vector. This is especially true in simplified models as commonly used in computer graphics applications such as this, which do not take material properties such as reflectiveness or environmental effects such as scattering or passive illumination into consideration. This also leads to a more gradual change from day to night side than what we may expect. In reality, light sources usually consist of more area than a single point, and thus provide a wider range of angles that may contribute to the illumination of any given surface point of an object. This means that the surface of objects are usually illuminated slightly brighter (depending on light size) than they would be with a point light source.

We can emulate this behavior by defining a function that exaggerates the brightly lit areas to a degree:

```
1 light = lerp(-1.0, 1.0, gain((1.0 + light) / 2.0, _TerminatorHardness));
```

Listing 4.6: Remapping the light value.

Here, we apply a gain function [Quib], which acts somewhat like a logistic function. In short, the mapping concentrates values at both extremes of the input domain with a rapid transition around the middle. This darkens dark areas of the range while lightening light values, thus raising the contrast between day and night sides while keeping a smooth transition line. This effect can be observed in figure 4.11.

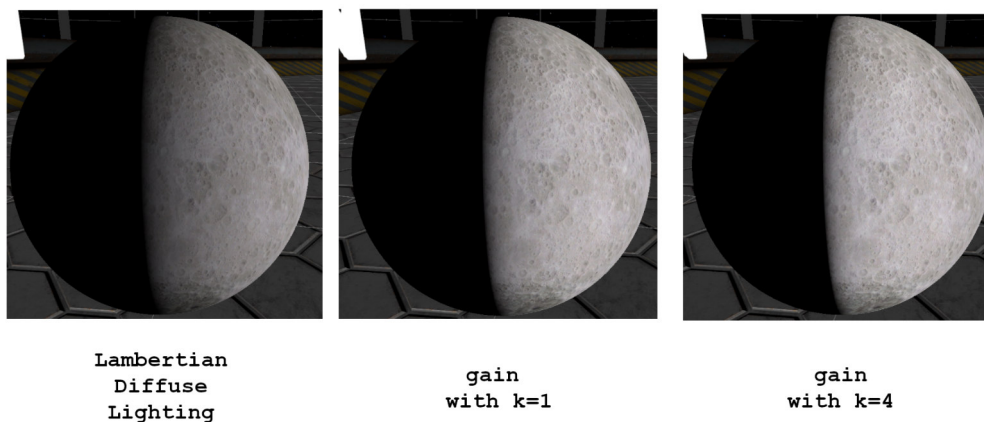


Figure 4.11: Direct comparison of base shader and different parameters for the gain function.

Since $\text{light} \in [-1, 1]$, we must remap it into the $[0, 1]$ range expected by the function. The `_TerminatorHardness` uniform is set directly in the editor representation of the planet's material within the Unity user interface.

The result is then linearly interpolated back into its original range using the `lerp` function resulting in a transition which is sharper than we would expect from Lambertian diffuse lighting.

In figure 4.10 we can see the lighting values for every point on the surface of a ray-traced sphere (c) and the result of applying these lighting values to the previously sampled texture. For the moon (and most planets in the solar system) the night side can simply be left dark since no natural light sources exist on the surface that can be noticed from space. However, earth is different. To accommodate this, we use

the light value to interpolate between a day and night texture, resulting in bright city lights even on the night side of the planet. This may be a mere visual flourish, but it contributes to the overall mood of the setting within the simulation and may also provide some visual context when players are exposed to the night side of the planet under their feed, rather than just seeing a yawning black void beneath them.

4.2.6 Sky Box

The sky box that allows users to immerse themselves in the vacuum of space has some particular challenges to offer. First and foremost, as shown in figure 4.3, we are dealing with a literal sky box. Some issue arose that caused bizarre errors in the resulting ray-traced objects in the sky. In retrospect, this was likely due to premature normalization within the vertex shader, but due to time constraints this was not further investigated, instead opting for a material shader that resides on a large box mesh with inverted normals. Since there is no way to venture beyond the bounds of this box, this solution does not currently pose any design challenges. In the future, if an area larger than the box should be required, it might be necessary though to fix the issue mentioned above.

Literal sky box, in a very old-school approach

Data Management

As the sky box is not confined to a single planet like the model planet shaders are, the issue of data management rears its head. Shaders are not capable of accessing the normal program storage and instead require all necessary data to be explicitly pushed to them. This becomes especially tricky when dealing with sequential data. The naive solution would be to simply pass each planet in its own set of uniforms. This wouldn't be an unreasonable approach since there is (at least currently) no reason to render any further planets. While it is possible to observe some of the other planets in our neighborhood directly with the naked eye, they are mere specs that would be hard to spot.

Shader needs information about planets

Nevertheless, this may become a concern in the future. Even if it doesn't, it would be much more desirable and elegant to be able to pass data in an organized way rather than relying on a number of named global uniforms.

Fortunately, the demand for data processing in shaders, so called compute shaders, has brought many useful abstractions. Using the `ComputeBuffer`³ class we can reserve space for our data and transmit it for use in our shader.

Here, it is mapped to a `StructuredBuffer<T>` that contains elements of a matching struct type `T` we have to define within the shader. We can then address these elements as one would in any normal array. We furthermore need to pass the number of planets to the shader as a uniform. With this we are almost capable of supporting an arbitrary number of planets in our shader.

Pass info as compute shader buffer

Combine
planet textures
into array
for shader use

Finally, we still need to handle the textures. These need to be bundled and loaded into the shader summarily as a texture array. This is primarily work on the Unity side rather than the shader and involves bundling the surface textures into one package using the Texture2DArray class. We need to ensure that all textures are sampled at the same size (we utilize a set of 8k textures). The process is as simple as determining the dimensions of one sample image, allocating enough space in the texture buffer to accommodate for all the textures and then writing them once at the start of the application into the array object. Since we never change the textures this suffices and doesn't eat away at our runtime budget from here on out.

Depth Test

Since we are dealing with multiple objects that may overlap, we need to decide which object is responsible for the current screen pixel.

```

1      float max_distance = 1.0e38; // arbitrary large float
2      int planet_idx = -1;
3
4      [loop]
5      for (int j = 0; j < _PlanetCount; j++)
6      {
7          Planet p = _PlanetDataBuffer[j];
8          float3 pos = mul(_POVTransform, p.position); // position in user pov
9          float intersect = ray_trace(ray, pos, p.radius);
10
11         if (intersect < 0.0 || intersect > max_distance) continue;
12         max_distance = intersect;
13         planet_idx = j;
14     }

```

Listing 4.7: Selecting a planet to render based on it's proximity to the camera.

Only render
top-most
planet

We keep the selection algorithm simple, as shown in listing 4.7. Iterating over the set of planets passed to the shader in every render frame, we select the one that is closest for the current pixel, saving its index for later use.

Since we cannot guarantee that any given planet contributes to the current pixel, we simply check against an arbitrarily large max_distance. While this solution somewhat inelegantly uses a magic number, it is fairly reliable still as our input numbers can never approach values of this magnitude without completely breaking down due to floating point imprecision anyway.

main loop
cannot be
unrolled

It may be interesting to note that we need to affix this loop with the [loop] attribute to ensure that it is not unrolled. While this would be preferable for performance reasons, the planet count is dynamically determined at the start of the application (as is the input struct, although that doesn't matter for loop optimization). Allowing the loop to be unrolled will thus result in runtime shader errors.

After determining the planet responsible for the current pixel, we can simply advance through the shader as we would for the model planets, rotating the hit vector, sampling from the texture and applying lighting.

³ Documentation: <https://docs.unity3d.com/ScriptReference/ComputeBuffer.html>

Eclipse Test

Since we have infos available about all the planets in our system, we can utilize these for other interesting purposes. Primarily, we can add a test to our shader to check if the current fragment is shaded by another planet:

```

1  [loop]
2  for (int k = 0; k < _PlanetCount; k++)
3  {
4      // planets can't be eclipsed by the sun or themselves
5      if (k == planet_idx || k == _SunIndex) continue;
6
7      // set up ray cast from sun to fragment
8      RayCast r;
9      r.origin = sun_pos;
10     r.direction = light_dir;
11
12     // determine additional raycast hit and distance
13     float3 planet_pos = mul(_POVTransform, _PlanetDataBuffer[k].position);
14     float other = ray_trace(r, planet_pos, _PlanetDataBuffer[k].radius);
15     float actual_hit = length(sun_pos - frag_position);
16
17     if (other > 0 && other < actual_hit) light = 0;
18 }

```

Listing 4.8: Loop over other planets and check if they could obscure the sun.

In listing 4.8 we iterate over all planets that are neither the sun nor the planet we actually want to render for the fragment, as planets cannot meaningfully eclipse themselves. We then test if there are any intersections with this other planet inbetween the current fragment and the sun.

For this, we need to first skip over any undesired planets and then cast our ray from the sun to the current screen pixel using the direction of the light `light_dir` established in the normal lighting step.

Then, if an intersection point exists that lies between the sun and our fragment, that is if the value of `other` is positive, we need to compare if it is also closer to the sun than our fragment.

If an intersection point exists and is also closer, our fragment is shaded by this other planet, and we thus set the light component to be 0.

This method has some caveats. First, a technological hurdle: since we approximate the sun as a point light and simply generate linear ray casts from it, the resulting eclipsing does not reflect some real life phenomena such as the “blood moon” that occurs when the moon is fully engulfed by the earth’s core shadow. The notorious red coloration of the moon is caused by Rayleigh scattering of light passing through the earth’s atmosphere before dimly illuminating the moon (similar to the red coloration of dawn and dusk), which we simply do not model since it would require more elaborate global illumination techniques. It may not be viable for our target hardware at all, although that requires further investigation if there is a demand for this in the future.

Eclipse is simplified, some phenomena are not modeled

In a similar vein (although unrelated to the eclipse check) we currently don't model the corona effect around the moon during a full solar eclipse. One approach to amend this might be to simply compare the similarity of the direction vector to both the sun and moon and conditionally show the effect if the difference is below some threshold. But this goes beyond the scope of this work.

More complex planetary systems may need performance optimization

Another caveat is that the current solution doesn't scale well. If we were to add other planets, i.e. Jupiter which can be observed fairly well with the naked eye and has a multitude of moons, we would quickly feel the performance impact of this loop. Worse yet, there is absolutely no chance or reason to model eclipsing between distant objects like this since this simply does not occur.

To mitigate this however, we need not stray far from the current solution. Rather than process all of our hypothetical planets in the same pass, we could simply split up our shader into several passes that each deal with their own system of planets. Thus, the above loop would only iterate over the planets it *truly needs* to consider, and we can also swap out the passes for distant systems with simpler versions (e.g. by omitting any satellites) as a means to save frame time if necessary.

4.3 Shadow Volumes

Shadows are created by absence of light

If an object occludes a light source, this results in its silhouette being projected onto any surface behind it. We call this a shadow. As shadows are an absence of light, it is commonly only noticeable when a surface exists that can reflect the light (or lack thereof) into our eyes. However, while we may consider these darkened silhouettes shadows, the occluding object actually spans a volume of three-dimensional space where none of the light falls to be reflected at us.

Shadows are 3D

We consider this space to be a *Shadow Volume*. In reality, this concept is much rarer to observe than the shadows we are so familiar with, although not impossible. To begin with, light is not generally reflected by thin air. Some sufficiently opaque particulate such as dust or water vapor such as in fog must be suspended in the air for light to reflect off of.

Natural occurrences of volumetric light and shadows

It is not uncommon to see the volume of a light ray breaking through the tree canopy while hiking in the woods, where light can reflect off the moist forest air. Similarly, a flashlight often clearly illuminates not just the lit surface, but also the tiny droplets of water suspended in the evening air or the dust in a basement or attic.

So called crepuscular rays, colloquially also called god rays, can sometimes be seen as pillars of light breaking through the clouds. This phenomenon, shown in figure 4.13 occurs particularly commonly at dawn or dusk, although they can also be seen during foggy weather or when light scatters on distant rain showers.

We might not notice volumetric shadows even when they occur
Brocken specters as an example of a shadow volume

These phenomena are usually noticeable in environments where sufficient contrast between the light volume and surrounding area exists. Contrary, shadow volumes are harder to notice as conditions where they can be observed neatly are less common. In our daily life, light is abundant, as during the day sunlight is scattered all around us, dimming otherwise sharp contrasts between lit and unlit areas.

However, there are still plenty of occurrences where one may encounter shadow volumes in reality. One particularly striking example are Brocken specters such as the one shown in figure 4.12. In opaque, foggy conditions, a hiker may cast a shadow on



Figure 4.12: Brocken Spectre, Golden Gate Bridge From Brocken Inaglory, *Solar glory and Spectre of the Brocken from GGB on 07-05-2011*, Wikimedia Commons, link, CC BY-SA 3.0



Figure 4.13: Sun rays From Kiril Simonovski *Crepuscular rays before sunset, Tikveš*, Wikimedia Commons, link CC BY 4.0

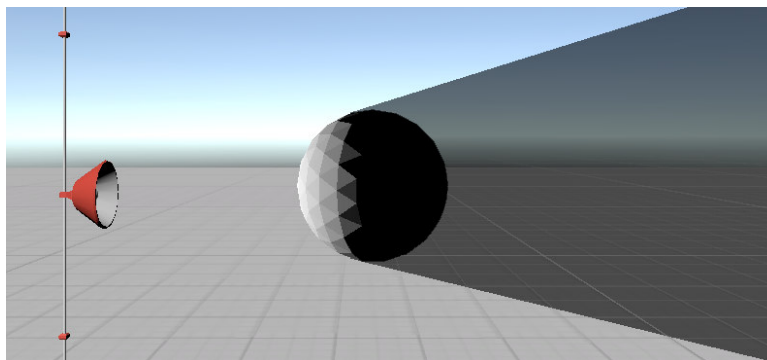


Figure 4.14: Shadow Volume scene setup with height-adjustable lamp to the left and moveable sphere (or other shape) to the right.

the cloud layer below, which then may be perceived as a ghostly silhouette. However, even under these conditions it is hard to actually gauge the geometry of the shadow volume anywhere near as well as one may be able to by, for instance, simply shining a flashlight into the dark. Airborne particles are not opaque enough to occlude the light reflected around it, so the effect is greatly diminished watching from the side.

Thankfully, in virtual reality we have the luxury of being in charge of the graphical environment. If we can derive the geometry of a shadow, we can simply make it as opaque as we wish. With this in mind, in this chapter we set out to investigate how this may be possible and how we chose to implement this.

VR allows us to visualize shadows

4.3.1 Outline

Scene Layout

Learners are presented with a simple setup in which they can see both a lamp and an object. This setup is similar to the one seen in figure 4.14. Using a control console, as seen in figure 4.15, they can affect this setup in a few ways.

4.3. Shadow Volumes



Figure 4.15: Control elements that users can interact with to affect the shadow volume simulation.

Interactions

available to
users

To begin with, two levers allow them to change the height of the lamp (where the range is delimited visually with two red stoppers on the lamp's pole, as can be observed in figure 4.14). This will dynamically change the angle at which the object is shaded, changing the shape of the shadow volume to conform to the objects changing silhouette.

Different
objects cast
different
shadows

Furthermore, the distance of the object can be adjusted. This changes the angle of the shadow volume, causing it to become wide when close to the lamp and narrower as the object moves away.

There are also several objects that learners can choose. Figure 4.14 shows a sphere, or, to be more precise, an icosphere (or geodesic polyhedron). As icospheres are made up entirely of equilateral triangles, they avoid many of the geometry issues that UV spheres⁴, which many sphere geometries default to, are prone to. Their geometry looks exactly the same from every angle.

We also include a cube and a grid box, which is a cube with holes along all three axes. This provides a particularly convoluted shadow volume as the volume now also contains unshadowed holes.

To make observation easier, learners can additionally toggle the visibility of the shadow casting object and the shadow volume.

Reduced
dimension-
ality of user
interactions
for didactic
clarity

In a prior draft of these simulations, users were able to freely move the lamp as they please. However, we were concerned that this added too many degrees of variation to the experiments, motivating the restriction to two axes.

After some optimization, the underlying algorithm now runs at an adequate speed that allows for more complicated objects should this ever be desired. As long as the conditions outlined in section 4.3.4 below are met. For now, we decided against the inclusion of particularly complex shapes to not make the simulation needlessly confusing. In the future, this could however serve to add flavor to more, elaborate directed tasks.

Learning Goals

As discussed above, it is rare to actively notice the existence of a shadow volume in our daily lives. We tend to think of shadows mostly as something two-dimensional, as the projection of a lit object onto a flat surface. Yet, shadows are in fact three-dimensional, effectively the section of space that *is not* lit by a light source.

With this simulation, we intend to allow learners to build a stronger intuition for the geometric properties of shadows: the geometry behind a shadow's shape and how this is affected by the objects' relation to a light source.

The main variables that learners will be able to change are the lamp position (or rather, angle in comparison to the object) and distance between object and light source. They can use this to experiment in a controlled way to infer how the shadow volume is affected by their interactions.

Strengthen user intuitions of shadow volumes

Users can reposition lamp and object along one axis respectively

4.3.2 Limits and Abstractions

Shadows in reality are complicated. We must limit the scope of our simulation to make sure that we can run it adequately even on mobile VR hardware. In this section we will briefly discuss how we limited our implementation in comparison to reality and why we made those decisions.

Idealized point light source

First and foremost, we assume the light source to be an idealized point light. That means, it emits light evenly from a single point in space and into every possible direction. In reality, such a thing as a perfect point light source does not exist, although particularly small, bright lights may achieve a similar effect.

The size of a light source, or apparent size for distant objects such as the sun, inevitably introduces an element of diffusion. As the area that light is emitted onto an object from reduces in size, its shadow grows more distinct. Area lighting is notoriously still a difficult issue and state-of-the-art methods are reliant on real-time ray tracing [MPC19].

Furthermore, since we are interested in the geometry of our shadows more than realistic rendering, diffusion at the edges is actively getting in the way of our goal. Meanwhile, a point light simplifies the necessary computation to a level that is more than doable on a portable VR HMD. It also leaves sharp and simple to understand geometry, as it simply projects out the shaded faces.

Area lighting is hard, especially for low-powered devices

Point lights enable clean geometry

Surface geometry

The digital world has no way to mimic the almost infinite detail of the real world. There are many qualities an object may possess, that affect its shadow in some way, primarily its opacity and shape. Transparent objects may refract and reflect light in complex ways, projecting light into part of the space it occludes (so-called caustics⁵) while leaving other parts in shadow.

Natural surfaces may interact with light in complex ways

Limited surface detail

⁴ e.g. as defined by the Blender 3D modeling software: <https://docs.blender.org/manual/en/2.80/modeling/meshes/primitives.html#uv-sphere>

⁵ Wikipedia: [https://en.wikipedia.org/wiki/Caustic_\(optics\)](https://en.wikipedia.org/wiki/Caustic_(optics))

Even if we assume that our objects are perfectly opaque, merely obstructing the path of light and scattering it, the surface geometry of the object itself is practically infinitely detailed compared to what a modern graphics simulation can render in real-time. Especially on more limited mobile hardware. While there are some approaches that are promising to represent virtual objects with much higher fidelity, approaching effectively photorealism, these are either computationally heavy, ill-suited for real-time applications or unsuited for dynamic lighting.

In the upcoming section, we will be comparing some different methods to determine what areas of a scene are currently shaded and how we might use that to estimate the shadow volume.

4.3.3 Shadow Rendering Methods

Shaders
traditionally
deal with
surfaces

Especially in our case, one major issue is, that shader programs are generally concerned with surfaces rather than volumes. There are, of course, volumetric shaders which are commonly used for dynamic clouds, fog, smoke and water effects. A volumetric shader, likely through the use of ray marching might be our best bet for a ray tracing based shadow volume shader.

Ray Marching

Ray marching:
sample volume at
certain points
along a ray

Ray marching involves sampling a volume, usually a bounding box for the intended feature, at different points, usually along the view ray, that is the ray from the camera through the point representing the respective screen pixel. In our case, we would simply need to check against the occluding shape to see if the respective step along the ray is shaded or not, similar to the way we checked for an eclipse in listing 4.8.

However, there are some caveats to this method. Mainly, we have so far relied upon simple geometry such as spheres where calculating for intersections is a fairly simple operation. If we were to use more complicated shapes, such as the *Stanford Bunny* or *Utah Teapot*, 3D models that are often used as benchmarks in computer graphics research.

Constructive
geometry is
complex

We could either take a *constructive geometry* approach, where we assemble a more complex shape from simpler primitives such as cubes, spheres, cones and so on, through boolean operations, or we check directly against the mesh geometry. This is somewhat more computationally intensive, but overall more flexible and modern graphics cards feature ray tracing optimizations for cases like this.

Unfortunately, there are some downsides to this method. Particularly, we have no geometric description of the shadow volume. That is, the shadow volume is entirely contained in the users view space, which we cannot work with easily for any evaluation tasks such as gaze tracking.

Ray marching
may get ex-
pensive

It is also computationally expensive, as ray tracing still is mostly the domain of desktop GPUs. While there are ray tracing units in mobile GPUs, particularly in recent iterations of the PICO Neo devices, they are much more limited. Due to the simplicity of the current simulation environment this is likely to suffice, but we must also keep in mind that this ray marching process needs to be repeated every single render frame where the user looks at the shadow's bounding box.

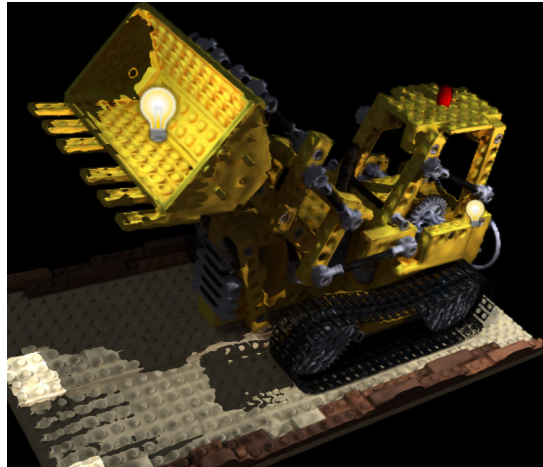


Figure 4.16: Screenshot of real-time demo of Gaussian splatting with dynamic lighting from Andrew Chan’s blog Post “Real-Time Lighting with Gaussian Splats”, link

Gaussian Splatting

An honorable mention goes to a method that has begun to gain traction recently after recent developments have improved its viability in real-time applications [KKLD23]: *Gaussian splatting*.

This method has gotten so viable in fact, that an abundance of in-browser demos are available, including ones that feature real-time dynamic lighting, such as in figure 4.16. While this technology is definitely exciting, it too has some caveats. While the surface lighting and even some of the complex shapes that make up the shown objects surface geometry come out fairly well, it does not take long to spot shading mistakes. Perhaps most noticeable in this figure is the shaded area of the shovel, that should clearly be lit by the adjacent light that also illuminates the remaining shovel.

Surely with time this technique will be refined to be more reliable, at which point it might be equally suited as ray marching.

Gaussian splatting becomes increasingly viable for real-time rendering

Shadow Maps

A common approximation of ray traced lighting traditionally used in many real-time lighting applications, especially in user software such as commercial games and simulations, is *shadow mapping* [Wil78]. In a way, this is fairly similar to the approach taken in the blog post referenced in figure 4.16: We utilize the light source as a camera, which renders a section of the world into a depth-only texture, or depth map.

From the user’s camera perspective, we can now sample any given surface point, project it into the light’s depth texture and compare the distance of that point to the light with the value written to that position of the texture. If we combine this with the ray marching approach above, that is instead of checking against mesh geometry, each of the ray marching probes can simply reference the shadow map instead.

Naturally, shadow mapping has downsides, primarily of the aesthetic variety: depending on how the depth texture is rendered, its resolution and the perspective projection applied to the camera before the render, results can vary drastically in quality. If the texture resolution is low, or the light source camera encompasses too large an area,

Commonly used, well tread approach to real-time shadows

Shadow quality suffers from resolution and perspective defects

the shadow quality (and thus the quality of any volume estimation) suffers, causing shadow edges to become jagged or blurry (depending on how they are sampled).

Shadow Volumes

Another classic method to determine shadows in a scene, that is less popular compared to shadow mapping but particularly suited to our purposes is the *shadow volume* algorithm, first described by Franklin C. Crow in 1977 [Cro77], a primarily geometric approach to determining shaded and unshaded areas of a scene.

*Silhouette
edge separates
lit from
unlit faces*

To this end, we need to first determine the silhouette edges of all the shadow casting objects, that is, exactly the edges that separate the lit and unlit faces of a mesh. We can then extend these out from the light source, resulting in proper mesh geometry for the shadow volumes.

For our purposes, this is where we can stop. However, to briefly summarize the remaining steps, paraphrased here based on the description by Akin-Möller et al. ([AMHH⁺18], pp. 230-233):

*Area is
shaded if
more vol-
umes entered
than left*

For any given view ray v , we want to introduce a counter that is incremented every time the ray encounters a front-facing triangle (that is, for face normal n we get $(n, v) < 0$), and decreases for every back-facing triangle. With this, the final count will indicate that we are indeed in the shadow of an object if the value of the counter is larger than 0.

The stencil buffer can be used to optimize this process. For this, the scene is rendered without any lighting information, and the stencil buffer is updated only for shadow volume faces. It is incremented for every front-facing triangle on the first pass and then decremented for every back-facing triangle on the second. If depth tests are enabled, this will also automatically omit currently occluded parts of the shadow volume. The stencil buffer values then correspond exactly to the counter values we wanted to determine.

*Stencil buffer
trick for per-
formance*

This is a simple approach, but more elaborate versions exist that deal with issues that may arise from this naive implementation.

But as previously mentioned, we are less concerned about the actual shadow. Rather, the volume geometry is all we will need for our purposes.

4.3.4 Mesh Preparation

Since we now have a suitable approach to find our shadow volume, namely the geometric part of the shadow volume shading technique, we can start to think about its implementation.

*Assumption:
input mesh
is closed
manifold*

For our implementation, we will make one important assumption: that the input mesh, which we will calculate the shadow volume for, possesses a closed surface geometry. Specifically, we require every unique edge to be shared by exactly two faces. The mesh itself can possess holes in its topology, and one of our meshes does indeed possess several such holes.

Even if the mesh surface may seem closed at a glance in the editor environment, there are several reasons why this may actually not be the case.

(a) flat-shading

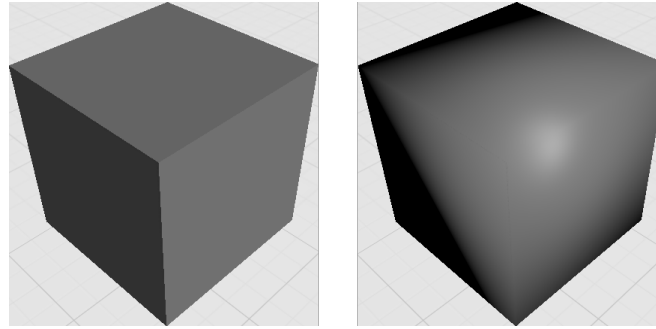


Figure 4.17: Comparison between a flat-shaded cube (left) and a smooth-shaded one (right).

- (b) UV seams
- (c) holes/open mesh geometry
- (d) “bad” geometry, e.g. duplicate faces

Out of the above items (which may not be exhaustive), only case (c) is likely to be an intended single edge, e.g. with flat meshes. These can be easily turned into closed meshes however, by simply extruding them into one direction, while also ensuring that the surface normals are corrected (in the 3D modeling software used) to be outward facing, as they can easily get mixed up and point inwards instead.

For case (d), this occurs most often when edges are accidentally extruded or faces duplicated. There are few cases where a third face being attached to an edge makes geometric sense, especially in a scenario where the mesh is to be used for shadow casting.

As for (a) and (b), these can be mitigated with simple preprocessing steps when initially creating the models for our simulation.

Flat Shading

First, case (a) can be amended by simply duplicating the target mesh and forcing the duplicate to be smooth shaded. Flat shading is often advantageous for aesthetics, especially as the acuity of the angle between faces increases.

Unity uses an indexed vertex buffer representation internally, i.e. it stores a list of unique vertices and represents mesh geometry as a list of vertex triples representing each unique triangle of the mesh. As is fairly common among 3D game engines and editing software, normals are saved on a per-vertex basis and calculated based on the average normal vector of all faces that share the respective vertex.

When rendering the mesh, the normal is then interpolated between the three vertices that make up any given triangle. However, in order to achieve a flat appearance the entire face must share the same normal value. In figure 4.18 this is visualized by rendering the normal vector as the color of the respective screen pixel. Note, that the flat shaded mesh shares the same color, and thus surface normal, across every discrete face of the mesh.

*Billboards
are usually
kept 2D and
thus not
closed*

*Smooth-
shading to
avoid face
separation*

*Vertex nor-
mals are
mean of ad-
jacent face
normals*

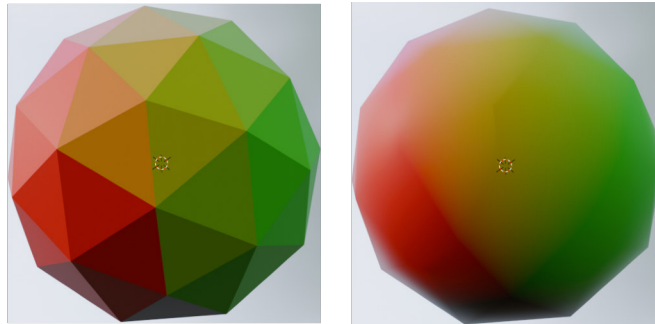


Figure 4.18: Flat and smooth icosphere with surface normals as color channel.

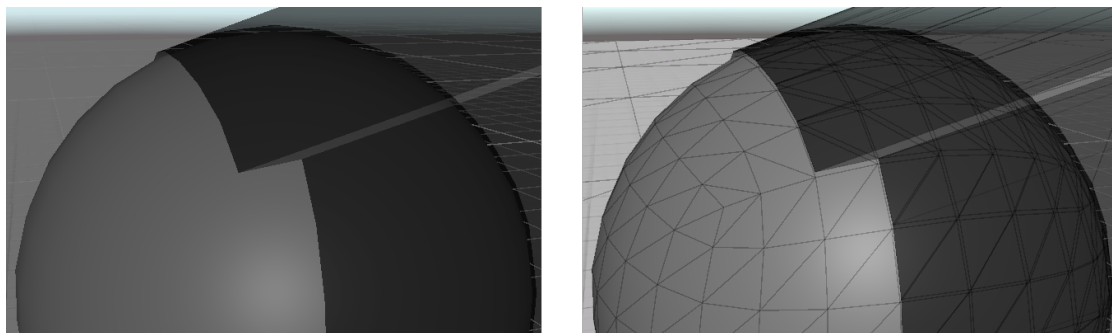


Figure 4.19: Boundary at the silhouette edge of a capsule mesh with smooth shading (left) and wireframe overlay (right) to show underlying geometry. Note the defect in the shadow volume due to a UV seam

To achieve this, the vertices along every creased edge are simply duplicated and the vertex IDs for the respective triangles updated accordingly. Now, the new vertex (but not necessarily the old one) will only be part of a single face and thus the vertex normal will be the face normal exactly.

Vertex duplication, but not face duplication

Notably, while the amount of vertices necessarily increases, the amount of triangles remains the same. The number of edges also increases, although this doesn't allocate any additional memory as edges merely arise from the triangle buffer.

With how we will end up rendering shadows, flat shading also lessens the ambiguity as to how a shadow is shaped in particular edge cases where face angles are near perpendicular to the light direction. In the selection of meshes we are using this doesn't occur generally, but many geometries, particularly among Unity's primitive 3D shapes are prone to this. Particularly, the UV sphere and capsule shapes can result in jagged shadows that only make sense when the underlying geometry is evident. Consider, for instance the extreme case in figure 4.19. We will get into the defects in the shadow volume boundary soon, but it does not play much of a role in the sudden jump of the shadow mesh by what seems to be one rank in the mesh grid.

Shallow angles result in weird shadow geometry

The actual reason for the jump is simply that the face angles at the poles of the capsule happen to hover around the point where they are almost perpendicular to the light source and thus can flip to being lit or unlit with minute changes. With the wireframe version, we can see that this still follows the overall grid structure perfectly, but on the smooth-shaded version the jump looks very arbitrary.

Smooth vs. Flat shading especially

noticeable on acute angles

Now, in figure 4.17 we can clearly see the difference between the two shading methods. Note that the geometry of the corner pointing out is almost indistinguishable from the surrounding surface.

If we duplicate the mesh as suggested above, we can then simply use one for its geometry and the other, flat-shaded version for the aesthetics. The mesh renderer on the smooth mesh can be removed or disabled to avoid it ever being seen.

UV Seams

For case (b), let us look at 4.19 again. The aforementioned defect is exactly caused by a UV seam. In 3D graphics, UV coordinates are commonly used to retrieve the position at which a pixel must sample its color value from a texture. Similarly to the normals above, the UV coordinate of an arbitrary point is interpolated from the UV coordinates of the three vertices that make up a triangle of the mesh.

For a simple example of how this may work out topographically, refer to figure 4.20. Here we see one possible unwrapping for a cube⁶. While the cube unwraps to a single conjoined planar mesh here, this isn't necessarily the case for every UV unwrapping. How seams are positioned may depend on a multitude of factors, and it is not uncommon for the resulting unwrapping to consist of several unconnected sub-meshes.

Trouble arises on anything that is not just a flat 2D structure: mapping a closed 3D mesh into the 2D UV coordinate space is more often than not impractical. To help with this, model artists can add "seams" into the model, that indicate where a model should be cut to help unfold it into a 2D projection. However, just like with the vertex normals above, vertices are assigned a fixed number of UV coordinates. Notably, each vertex has more than one UV slot in the input buffer to the shader to be used at the programmer's digression.

While there are some ways this could be used to somewhat circumvent the UV mapping issue mentioned above, its more trouble than the alternative, which is to simply utilize UV seams. However, just like edge creases in flat shading, we achieve a UV seam by simply duplicating the vertices along the seam, which once again creates unique edges (although normals are copied over as well, thus avoiding edge creases that weren't intended).

To deal with this issue for our algorithm, we can simply delete the UV data from the mesh in our editing software, i.e. Blender. Now, no seams exist that Unity will attempt to split the mesh by, and we should be left with a fully connected, closed mesh to run our algorithm on.

4.3.5 Shadow Volume Generation

With the mesh prepared, we can finally generate the shadow volume. As input to our algorithm serve the mesh we prepared in the previous section as well as the position of the light. Furthermore, while we decide to define a "cutting plane" which serves to define the cut-off point, or maximum extent, of our shadow volume.

⁶ Animations for all 11 unique cube unwrappings by Prof. Mark McClure here: <https://observablehq.com/@mcmclur/the-eleven-unfoldings-of-the-cube>

UV break
closed mesh
in unex-
pected ways

Seams are
necessary for
correct tex-
ture mapping

We simply
delete UVs

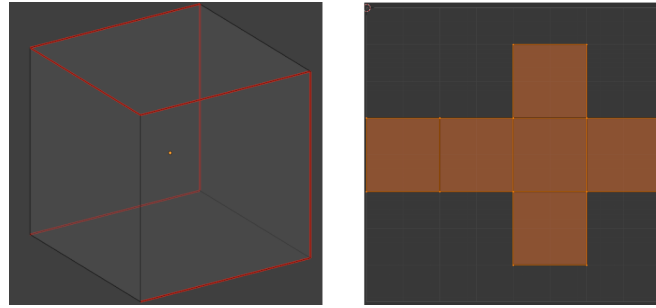


Figure 4.20: One possible UV unwrapping of a cube. Left: Cube mesh with UV seams (in red). Right: corresponding UV unwrapping.

We will utilize two custom components to handle the logic for us: the shadow will be generated and managed by a class `ShadowVolume`, which also contains the main bulk of the code for this part of the application. It is not assigned from the Unity editor. Instead, it is created and attached to the target geometry by its companion class `ShadowCaster`, which handles passing data and cleaning up after the shadow volume if it is not currently needed.

We will mostly be looking at the `ShadowVolume` class here.

There are three things, that `ShadowVolume` needs to know:

- position of the light source
- mesh geometry
- cutting plane

We already know about the light position and mesh geometry, the cutting plane is another simple object that defines the maximum extent of the shadow volume.

*ShadowCaster
generates
ShadowVolume*

When the `ShadowCaster` component is activated, it first creates a new game object and attaches it to the target mesh, which is the smooth-shaded, invisible duplicate of our main mesh. It then creates a new `ShadowVolume` component, attaches it to the game object and passes the relevant data to it.

Now the shadow volume component takes over. Its goal will be to determine the silhouette edges of the mesh it represents relative to the incoming light.

Initialization

*ShadowVolume
must set up
mesh for
dynamic
generation*

Before we can get into our shadow volume implementation, the freshly initialized component must make some preparations. First, it copies the mesh data from its parent's `MeshFilter` component. We call `Mesh.MarkDynamic` to notify the engine that we plan to update the geometry of this new mesh frequently. The API documentation states, that this causes the mesh to use dynamic buffers internally, which are more efficient with frequently changing data [Tec25a].

We increase the size of the vertex buffer of our new mesh to be of size $V' = 2 \cdot V + 1$, where V is the initial vertex buffer size. For the shadow volume we will need to frequently re-project a changing number of vertices depending on the light angle. Thus, instead of recreating the vertex buffer array every time the number of projected vertices changes, we allocate it with enough space for each projected vertex from the start and later determine which ones are actually used through the triangle buffer. The final, additional vertex serves as the “pinhole” that we need later for light coins in section 4.4.

Finally, we can pre-compute the mesh faces. Since we will be doing all of our calculations in the objects referential frame, this is information that never changes as long as the underlying geometry remains the same.

We will generate an array of struct instances of the type Face with the following fields:

```

1  public struct Face
2  {
3      int          Index    { get; } // index of face in sequence
4      (int, int, int) Tri    { get; } // index of face vertices
5      (Vector3, Vector3, Vector3) Verts { get; } // face vertices
6      Vector3      Normal   { get; } // face normal
7      Vector3      Centroid { get; } // center point of face
8  }
```

double ver-
tex buffer
to keep
projected
vertices in
memory

Listing 4.9: Signature of the Face struct type.

Faces are stored as sequential triples in the integer array `Mesh.triangles`. We simply iterate over the entire array (with a stride of 3) and create Face instances for every triple.

This is purely a matter of assignment, except for the face normal, which is simply the cross product of two of the triangle edges:

$\text{face.Normal} := (v_1 - v_0) \times (v_2 - v_0)$, where (v_0, v_1, v_2) are the face vertices

The centroid serves as our arbitrary reference point, which will be used to determine the direction that light hits the face from later on. It is simply defined as:

$\text{face.Centroid} := (v_0 + v_1 + v_2)/3$, where (v_0, v_1, v_2) are the face vertices

With this, we have everything prepared to get into the proper algorithm.

Silhouette Edges

Our next step will be, to find the silhouette edges of our mesh. We can determine these, as previously mentioned, by finding edges where lit to unlit transitions occur. The idea for our algorithm is thus simple: for every edge, we will subtract the number of lit faces from the number of unlit faces. If the result is 0, the edge must be part of the silhouette edge set.

For this, we iterate over our previously defined list of faces, and for every face we begin by checking if it is lit:

```

1  var lightDir = (lightPos - face.Centroid).normalized;
2  _litFaces[face.Index] = Vector3.Dot(face.Normal, lightDir) >= 0f;
```

Find border
edges of lit
and unlit
faces

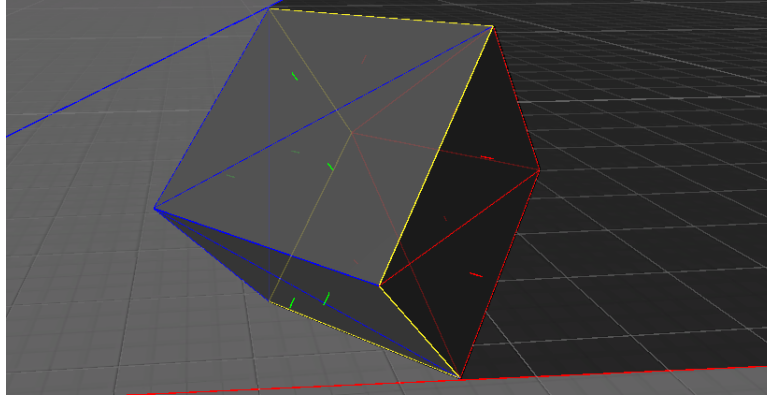


Figure 4.21: Visualization of silhouette edges (yellow). Additionally shown: lit faces (green normal), unlit faces (red normal), lit edge (blue), unlit edge (red)

Listing 4.10: Check if face is lit

*status of face
derives from
orientation
w.r.t light
source*

Simply put, we merely need to check the orientation of the face normal vector and the vector pointing from the face's centroid towards the light source. There is a fairly obvious caveat to this, namely that this does not take self-occlusion into account. This may lead to layered shadow volumes, as the occluder's shadow extends in parallel to the occludee's. In general however, this should be fine.

Now that we know whether the current face is lit, we can enter this information into a hash map we will be using to track how many lit and unlit edges every edge is adjacent to. For this, we will repeat the following code for every edge (a, b) in `Mesh.triangles`:

```
,
1  var key = (Math.Min(a, b), Math.Max(a, b)); // canonize edge direction
2  dict[key] = (isLit ? 1 : -1) + dict.GetValueOrDefault(key, 0);
```

*Face orienta-
tion follows
left-hand rule*

This is a fairly simple operation. First, Unity stores triangles in clockwise winding order [Tec21], that is, the front of a face is determined by the view direction in which vertices connect in a clockwise manner (alternatively, this could be thought of as a left-hand rule, where face normal aligns with the left thumb if the curled fingers follow the connection order). From this, we can reason that two faces sharing the same edge will always contain the edge in two mutually different directions, so we simply canonize the direction by listing vertex indices in ascending order.

We then manipulate the value of the edge dictionary according to the face lighting information. Once we iterated over all edges, the edge dictionary indicates the silhouette edges with a value of 0.

In figure 4.21 we can see a visualization of the different edges, here on a cube. The silhouette edge, here in yellow, separates the lit and unlit sides of the cube.

*Still room for
improvement
(perfor-
mance)*

In retrospect, the hash map is pure excess. It is a relic of a prior implementation and has remained because the algorithm now runs sufficiently fast, rendering further optimization unneeded. However, in the future it might be advisable to seek a more efficient representation, perhaps array-based. This is also the primary obstacle to moving this entire algorithm into a compute shader, where it could make heavy use of

fast GPU operations and parallelization and thus gain further runtime performance. However, for our purposes it has proven to be more than fast enough. Furthermore, we could alter the algorithm to simply ignore unlit faces and consider any edge where the number of adjacent lit faces is 1 to be a silhouette edge. This would allow the algorithm to support some cases in which the mesh topology is not in fact closed, particularly holes in the mesh surface.

Vertex Projection

Now that we know which edges we need to project out, we can generate the appropriate shadow volume geometry. We will utilize a BitArray data structure to track which vertices we have already projected, since every vertex will appear twice in the list of silhouette edges.

We cycle through the dictionary, skipping every non-zero entry. Then, for every edge that is part of the silhouette edge, we project both constituent vertices if they haven't been projected already:

```

1 private void ProjectionHelper(Vector3 lightPos, int a)
2 {
3     if (_wasProjected[a]) return; // returns from projection helper function
4
5     var p = _vertices[a];
6     var d = p - lightPos;
7     var t = -(Vector3.Dot(lightPos, _cuttingPlaneN) + _cuttingPlaneD)
8             / Vector3.Dot(d, _cuttingPlaneN);
9
10    _vertices[_vertCount + a] = lightPos + d * t;
11
12    _wasProjected[a] = true;
13 }
```

Listing 4.11: Helper function that projects a given vertex and adds it to the vertex buffer.

Our goal is to project the vertex v_a with index a onto the cutting plane. The cutting plane is defined as a plane normal vector n and an offset d such that for all points p in the plane:

$$(n, p) + d = 0$$

We calculate n by transforming the *up* vector of the object representing the plane passed to the shadow volume into the components local reference frame. We then determine d by solving the equation for it with p being the position of the cutting plane object in the local reference frame.

This must be done regularly, as the shadow volume's parent can move due to user input, consequently moving the plane relative to the object as well.

We then determine the direction of the light source through the vertex we want to project and calculate its intersection point with the plane, which we promptly save in the vertex buffer array.

For every vertex index a in the mesh, we define its projected point to have the index $a' = |V| + a$, if V is the original vertex set. With this, we can easily correlate vertices with their projections.

Finally, we add the triples to the new triangle list:

*Cutting plane
as projection
endpoint*

*Project ver-
tex based on
light position*

```

1 ProjectionHelper(lightPos, a);
2 var aProj = _vertCount + a;
3
4 ProjectionHelper(lightPos, b);
5 var bProj = _vertCount + b;
6
7 AddTri(backTris, a, b, aProj);
8 AddTri(backTris, b, bProj, aProj);

```

Listing 4.12: Call projection helper and add new triangles for projected edge

unlit faces in
triangle list
for shaded
backside

We omitted it above to keep the code snippet short, but when we calculated the silhouette edges we also added every unlit face to this list. This adds a cap to the object side of the shadow volume.

We need to add two triangles for every edge since we are looking to model a quad between the existing and projected edge.

Finally, we need to assign the data (vertex buffer and triangle list). Now, our shadow mesh should render into the scene as intended.

4.3.6 Caveats

Compute
shader as
performance
enhance-
ment?

This solution has some rough corners and points that could be improved in future iterations. There is, for instance, some additional space for optimization.

Moving to a compute shader would be one way. The main obstacle to this is the edge hash map, as mentioned above. This would be particularly interesting for use with more complex mesh geometry.

Another issue is the order of the edge list. Currently, we sort edges in ascending order by convention. However, this leads to a particular issue: face orientation. Currently, face orientation in our solution is not standardized, instead it arises from the arbitrary index numbers of the edge vertices.

Correct edge
list order
improves
results

We opted to simply utilize a material for the shadow volume that omits back-face culling, so visually this is not an issue at all. It should also not interfere generally with future extensions, e.g. the addition of eye tracking.

However, if it is ever required, the correct order of vertices must follow the order of the unlit side. That is, if (a, b) is the edge in question with the lit face (a, b, c) and unlit face (d, b, a) , then the correctly oriented faces will be (b, a, a') and (b, a', b') if a' is the projection of a and b' is the projection of b .

As stated before, the clockwise vertex order determining the face direction necessitates that no two faces share the same (directed) edge, that is, the same edge vertices in the same order. This must also be true for the projected faces as the lit face as long as the lit face is already in the correct orientation.

4.4 Sun Coins

The name “Sun coin” derives from the german language term “Sonnentaler”, a poetic interpretation of light spots, or dappled light, that can often be observed projected through the gaps of tree canopies. A visual example of this can be seen in figure 4.22.

One might reasonably expect these spots of light to be mere projections of the holes the light falls through, similarly to how we constructed our shadow volumes in section 4.3.5. However, this expectation does not hold up in reality. Even in figure 4.22 one may notice the suspicious regularity and roundness of the spots, a feature that is hard to imagine coming from the chaotic interplay of leaves in a tree canopy. Particularly stunning is the appearance of sun coins during solar eclipses, where they will take on a crescent shape. We can observe this exact spectacle in figure 4.23.



Figure 4.22: Image of sun coins on a paved road, Wikimedia Commons by *Quelokee* (public domain), [link](#)



Figure 4.23: *Crescent Sun shadows*, by Harris County Public Library (CC BY-NC-SA 2.0), [link](#)

What we are looking at is actually a naturally occurring example of a pinhole camera, where the openings between the leaves, in shutters etc. acts as the pinhole and the sun coin is an image projected by the camera, namely of the sun, onto the surface below. While it is hard to find concrete evidence for this, one may even come across claims that minute surface features of the sun, particularly sunspots, can be observed in sun coins.

This effect is, of course, not limited to the sun. It can be replicated with artificial sources of light, although these may need to be dimmed or diffused depending on the conditions. In a classic pinhole camera for instance, the light could be from an object that is passively lit from another bright light source. Often, a light source such as a candle is used in school experiments to show the pinhole camera effect, where we too see the light source projected like in our sun coins. In principle, sun coins can even arise from light sources such as the moon, although observation may be much more difficult due to the significantly dimmer light bouncing off the passively illuminated moon (ref. [Sch94], *German language source*).

4.4.1 Outline

When entering this section, learners are presented with a simple model: a shaped lamp hovering above a perforated table. The lamp casts an image of itself through the hole onto the floor. Learners can adjust the height of the lamp to observe how this affects the size. To help with this, we also outline the outer perimeter of the light and denote the size of the image cast by the light.

We can see this setup in figure 4.24, where the left image displays one configuration the user may encounter. As implied by the control elements in the same figure on the right, users can either shift the height of the lamp or select one of several different lamp shapes.

Furthermore, while the user can read the size of the image, they are also provided with distances of the lamp and image (or rather, the floor) to the pinhole.

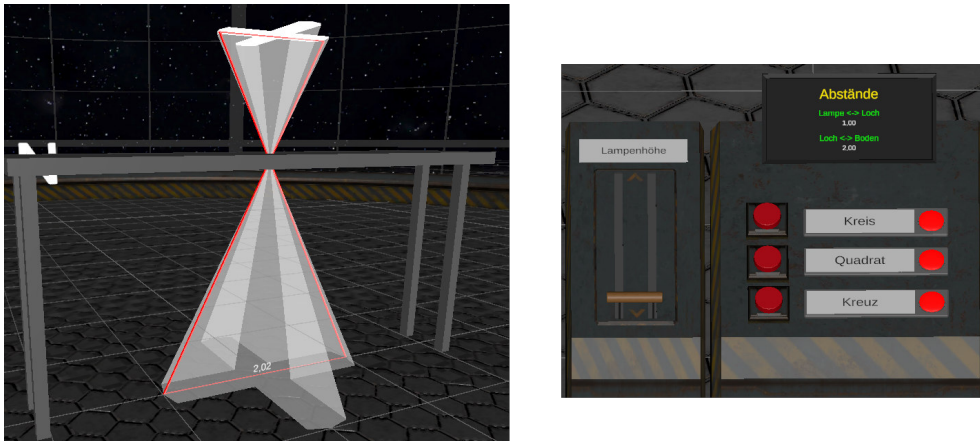


Figure 4.24: How the scene might look like for a user (left) and how they can affect the simulation (right)

Learning Goals

In this section, students can mainly investigate how the position of the lamp is related to the appearance of its image. Intuitively, one may assume that this is mostly determined by the shape of the hole and how distant it is to the floor. However, this does not hold up in reality. Instead, this model is akin to a pinhole camera.

This experiment is built on many of the same principles as the shadow volumes, both conceptually and technologywise. The underlying mathematics are also an application of the intercept theorem⁷, which can be applied to underpin the proportionality of light source and image.

4.4.2 Implementation

In order to implement the above application, we can retool the shadow volume implementation to act as a pinhole camera. While this may at first sound unintuitive, the two concepts are actually remarkably similar. We merely need to reframe the issue somewhat. Our mental model for the shadow volumes, displayed in figure 4.25, requires us to determine the silhouette edge of our geometry and then project that out away from the lamp. This is exactly the area shaded by the object's underlying mesh. For the sun coins we can take a similar approach. Here, we switch the occluder (which we now understand as a lamp, or light source of some sort) and the lamp (which we now understand as a pinhole). Figure 4.26 models this analogously to the shadow volume model.

⁷ Wikipedia: https://en.wikipedia.org/wiki/Intercept_theorem

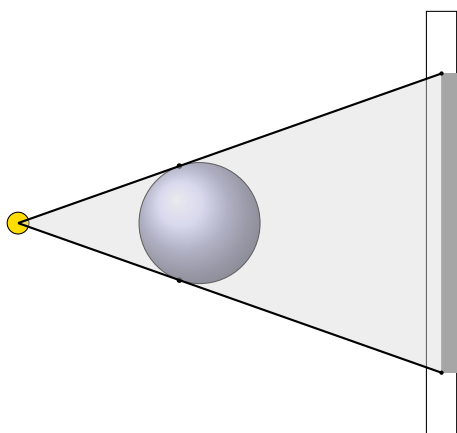


Figure 4.25: Shadow Volume model, sphere occludes the lamp in the shaded area behind it.

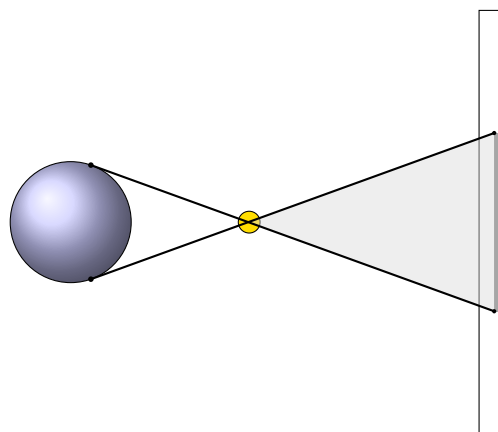


Figure 4.26: Sun coin model, image of the sphere (which serves as a lamp) being projected through a pinhole

The “light source”, now turned pinhole still remains as the point of projection. We can thus proceed as we would for the shadow volumes: determine the silhouette edge based on the face orientation relative to the pinhole, then project it towards the projection plane. Rather than emerging from the “light source”, the rays must now go *through* the pinhole. But this does not actually affect the underlying math, as we only care about the intersection point with the plane that we shade against.

One notable difference is the resulting geometry. We *cannot* build the geometry of the projected light as we would the shadow volumes. This is due to the fact that we project out the silhouette edge as quads and, since the resulting image is an inversion of the light source, rather than converging in the pinhole the resulting mesh geometry will end up twisting incorrectly.

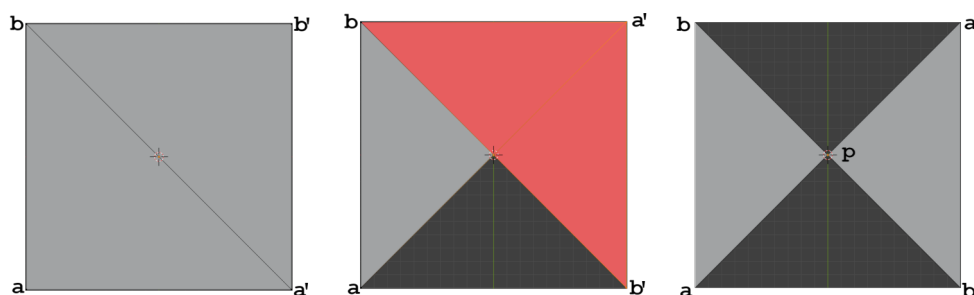


Figure 4.27: Examples of a normal projected edge (left), a twisted edge, constructed like before but projected through a pinhole, swapping a' and b' (middle) and the same with cleaned up geometry through introduction of a center point (right)

Instead, we need to introduce one additional point to our geometry, as alluded to in section 4.3.5. This point represents the pinhole. If we have an edge (a, b) and projected edge (a', b') as well as the new pinhole point p , rather than adding two triangles

$$(a, b, a') \text{ and } (b', a', b),$$

that is, a quad between the two edges, we connect through the intermediary point such that

$$(a, b, p) \quad \text{and} \quad (p, b', a')$$

are the resulting triangles. Now, rather than twist, we instead get a clean convergence in point p . This point can also be thought of as the intersection point of the lines through all points of the silhouette edge and their respective projected points. A clear example of the problem is illustrated in figure 4.27.

Intercept Theorem

As shown in figure 4.24, we also employ a visualizer of the intercept theorem, a common topic in middle school maths and physics. This implementation uses Unity's built in `LineRenderer` component, which (as the name suggests) renders lines into the view space. We need to merely specify the points that make up the line.

We leave this task to the `ShadowVolume` component, which also drives the sun coin demo (toggled through an additional boolean switch to enable the above geometry alteration). Said component defines an event that retrieves the two of the vertices of the silhouette edge and supplies these points and their projected counterparts to use for the line rendering.

To find such points, we need to first decide on a way to order all vertices of the silhouette edge, and then find the *minimal* and *maximal* vertex w.r.t. this order. This is achieved by mapping each candidate vertex to its projection onto an arbitrary axis selected to match the desired orientation. In our case, we want it to be parallel to the projection plane, while also perpendicular to our view direction onto the mesh, such that it orders vertices "left to right" from the viewers default perspective.

For vertices a and b and an arbitrary, normalized axis n we thus get

$$a < b \iff \dot{(a, n)} < \dot{(b, n)}.$$

Determining the minimal and maximal vertices is now a matter of simple iteration. Finally, we project the vertices as we would when generating the mesh geometry and invoke the event.

The visualizer then constructs the line segments and adds a label displaying the distance of the projected points.

4.4.3 Future Work

Perhaps the most needed addition to this simulation is a proper demonstration of sun coins, using the real sun in the sky box. This was omitted due to time constraints, but a simple construction would follow a simple schema:

1. define "holes" that cast the sun coins, e.g. in a potted plant, tree or other hole-filled object
2. render state of the sun in the sky into texture (e.g. using custom shader pass)
3. per hole: approximate size of sun coin, project position into world space

4. use Unity Decal renderer⁸ or custom solution to set up a decal using the sun texture at the appropriate size and position.

This could then go on to exist as part of the scene and be pointed out during the guided tour. The total hole count needs to be chosen depending on performance impact.

⁸ e.g. following this guide https://www.youtube.com/watch?v=5p8cKIu3P_8

Chapter 5 Evaluation

Due to time constraints after some implementation issues and delays, an empirical study of this application became unviable. Nevertheless, we will talk about how we facilitate future studies by implementing a data collection framework. We will also assert the viability of the application for use in the field and propose a potential methodology to assess potential learning effects.

Data collection

*OmiLAXR for
data tracking*

We decided to include the learning framework OmiLAXR [G25], which specializes in collecting learning data in applications that utilize Unity XR. This framework proved easy to integrate, being ready to use practically immediately after including the relevant packages. The framework is based around the use of xAPI, an open source specification designed for data tracking in learning application [Adv13].

*Default track-
ing enabled*

OmiLAXR defines a processing pipeline, that developers can hook into at specific spots to define their own listeners, customize or extend tracking logic, define how statements are composed and even define custom endpoints for the collated data.

It provides some tracking by default, particularly relating to the application state and motion tracking. Data is then passed off to either a learning record store, which can be connected to via an authentication token or name/password pair. Alternatively (or additionally) the data can be saved locally in different file formats, JSON and CSV.

*Pipeline can
be extended
to match
needs*

Later down the line we can extend the pipeline as needed. This goes hand in hand with the generally event-based data flow in our application. All UI elements and some of the other subsystems mostly expose events for interesting interactions that can be subscribed to easily. Thus, the components merely need to be grabbed from the active scene, and suitable tracking behaviors and composers need to be written to produce the desired xAPI statements. This data can then be aggregated from a learning record store for further processing.

Evaluation concepts

One way to evaluate this application would be to set up an expert evaluation, interviewing experts in physics education and applying adapting the application accordingly. Informally, we have adhered to this strategy by presenting changes in bi-weekly meetings and adapting the application based on the feedback. This process also informed the instructional content discussed in section 4.1.3.

Another method that would be viable for evaluation is the classic user study. Ideal subject to such a study would be a student population of the appropriate age group, as this is the most relevant demographic to this work's design goals.

As we want to ideally measure if the application offers any learning benefit, subjects should be split into two groups: one that is exposed to the learning simulation, and one that serves as a control group. The groups should be randomly sampled, ideally from the same class to ensure that all students are roughly on the same level academically.

After the VR session, students should be provided with a short written test designed with the help of physics educators to assess the understanding of the featured topics. Results are then compared with the control group to see if any learning effects can be observed.

Perhaps the best timing to perform such a study is alongside the topic being covered in physics class, as this would provide the same starting point for all subjects and would ensure that we merely evaluate additional learning benefits as we merely seek to enhance existing educational contents. Alternatively, both groups could be introduced to prior instructive material, perhaps a video or short lecture on the material covered by the simulation, to ensure level ground for comparison.

In addition to this (and the incidental collection of empirical data through OmiLAXR), subjects can also be prompted to partake in a survey to ascertain qualitative factors, such as engagement and usability.

Availability

Currently, the application is tested to work with desktop VR via Oculus Quest Link as well as portably on PICO Neo 3. Builds run cleanly for Android (mobile VR) and Windows. For different VR systems, the respective toolkits must be chosen in the Open XR section of the project settings. Depending on the vendor, either the OpenXR system works out of the box, or a vendor toolkit must be first installed.

Unfortunately, a WebXR build does not work out of the box, among other reasons due to conflicts with PICO XR vendor toolkit scripts that need to be properly addressed. As stated before in chapter 3, this would be advantageous for distribution as, rather than installing from an APK file or navigating various store restrictions to upload applications for download, content can simply be hosted via website and accessed directly from the VR system's browser.

Chapter 6 Conclusion

As we are coming to the end of this thesis, we will briefly reflect on this work. We will discuss challenges and limitations, take a brief look at what might be in the future for this project and then conclude our work.

6.1 Challenges and Limitations

Botched time management

Perhaps the biggest detriment to this work was misguided time management, mostly due to an unfocused vision of the project during implementation leading to occasional meandering. In part this is owed to overambition, leading to time investment into some features that may have not warranted such focus. Particularly, the solar system simulation evolved from a novelty to a main feature even though its contribution to the learning content does not match the effort expended to implement it.

Shader programming developer experience

Additionally, some technical challenges presented themselves. Shader programming is notoriously harder to debug than other systems, as debuggers and console output are not trivially available. Most commonly, debugging is done by simply encoding information as texture output to at least enable developers to visually inspect results to see where issues may occur.

However, this method is limited in its efficacy. Particularly when transformations are involved, issues quickly arise. Debugging the skybox shader for the solar system simulation was a significant time sink.

Mobile VR development could be smoother

Development for the PICO HMD was also somewhat error-prone. While the toolkit is easy to install and switching to it is rather simple via the OpenXR section of the project settings menu, it comes with the common pitfalls of Android development. After installing Unity's own distributions of *OpenJDK*, appropriate Android SDKs and NDKs and manually installing *adb* and troubleshooting the device connection, this process still causes trouble occasionally.

WebXR as an alternative?

It may be worthwhile to investigate the viability of WebXR for applications such this, as it should significantly simplify the burden of distribution to user systems and help create more device-agnostic software (although this comes with the caveat of potentially disrupting existing data collection structures, investigation into this topic may be valuable).

In terms of methodology, it would have been prudent to perform a somewhat broader analysis of requirements and potential topics. This could have been achieved by interviewing or surveying a number of teaching professionals about experiments, visualizations or educational content they would have been interested to see. While this

work was produced in constant feedback with experts in physics education and educational VR design, broadening the opinion base in the initial phase of development would have surely added to the academic rigor of this work.

6.2 Future Work

In many ways this simulation could be improved. In the prior sections we have highlighted some of the ways that particular parts of the application could be improved or altered, as well as how we could improve aspects such as availability (e.g. distribution via WebXR), that would add to the convenience of running this application, an aspect that is enticing for broader use.

A reasonable direction this application may go in the future would be to build a more directed, goal-driven experience on top of the current implementation, either through gamification or a more structured guided tour experience, more akin to how applications such as RePiX VR [HGS22] handle this issue.



Figure 6.1: A puzzle in the game *Resident Evil 7: Biohazard*,
Source: Uploaded by ParallelTraveler to *Resident Evil Wiki*, link

Especially the gamification space is fertile soil for optics tasks. Reflection (or mirror) puzzles are a long-time staple in games, and even shadows are frequent targets of puzzle design in games, such as in certain parts of the 2026 game *Resident Evil 7: Biohazard*, which tasks the player with arranging a complex figure to create a silhouette matching the hinted shape in a picture. Similarly, we might see puzzles that task learners with arranging lights to enclose certain points in the shadow volume (or exclude others).

The solar system may task learners with finding specific configurations (such as a solar eclipse) or identify particular places on the surfaces of the earth or moon. Other planets could be added, not for direct observation from earth, but to experience how the altered distance to the sun, or different rotational periods may affect the sky. This could also be a venue to open this project to more topics than just optics. For instance, learners could be given coordinates and, using simulated navigational tools, find the objects described by those coordinates (i.e. distant planets such as Jupiter). Since we opted for a shader-based sky, only minor adjustments should be necessary, to implement optical helpers such as a telescope that gives a zoomed-in view of the sky.

We would also like to once more highlight the plain-text guided tour script as described in section 4.1.3. We see this as a chance to facilitate close collaboration with non-technical contributors by offering an interface into the application logic that does not require extensive knowledge of Unity or even an IDE to work with.

Currently, this system still requires some amount of developer input. Python needs to be set up to run the generator script and the guided tour needs to then be constructed via the appropriate component within the Unity editor. Nevertheless, it still offers an easy way for contributors to the project to alter the content of the guided tour with minimal post-processing required.

We see two ways this could be improved on in the future: For one, the functionality of the script can be enhanced by implementing new node types, e.g. to add quiz sections or by triggering more elaborate events within the simulation. Even before developers actually implement any of these features, alterations to the script can be made as a “mock-up” of the desired functionality.

Additionally, by shifting audio generation into a model that can be run through Unity or remotely generating voice data, the script could be parsed and run live, adapting to changes made locally without the previous processing steps. This means teachers could amend sections as they want or (w.r.t the previous point) add quizzes or other features as needed. This reduces barriers to entry into the development process and reduces the necessity for hand-crafted solutions, broadening the horizon of what is achievable in a learning simulation such as this.

6.3 Conclusion

With this thesis, we set out to design a learning application to serve as supplementary learning material in middle school physics education. To this end, we combined a number of simulations from the field of optics with a guided tour style instructional design, offering diegetic interactions to immerse and engage users. We choose a portable VR system as our target platform and delve into the implementation details of our cast of simulations on offer for learners.

While some limitations impacted the scope of this work and the question remains how VR systems may be incorporated into the broader context of school education, we nevertheless build a solid foundation for future work and empirical studies of the topic. We hope that this work can highlight the potential of virtual reality as a learning aide by combining engaging, immersive elements and didactic instructional design.

Appendix A Bibliography

- [Adv13] Advanced Distributed Learning (ADL) Initiative. xapi specification. <https://github.com/adlnet/xAPI-Spec/tree/master>, 2013. Accessed: 2025-11-27.
- [AMHH⁺18] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering, Fourth Edition*. A K Peters/CRC Press, Boca Raton, FL, 2018.
- [At02] Alain Albouy and H. S. Dumas (translator). Lectures on the two-body problem (extract). Technical Report 77, ASD/IMC-CNRS-UMR8028, Paris, 77 avenue Denfert-Rochereau, F-75014 Paris, 2002. Available at <https://www.math.nthu.edu.tw/~kchen/teaching/NewtonBernoulli.pdf>.
- [Bec] Leo Becker. Quelloffene spiele-engine godot: Apple mit an bord – für vision pro.
- [Car24] Marie-Theres Carl. Nrw setzt auf virtual reality: Großprojekt zur digitalisierung der schulen gestartet, aug 2024. Lehrer News, Schulpolitik.
- [CKS24] Matthias Conrad, David Kablitz, and Stephan Schumann. Learning effectiveness of immersive virtual reality in education and training: A systematic review of findings. *Computers & Education: X Reality*, 4:100053, 2024. Available open access under CC BY-NC-ND license. URN: urn:nbn:de:0111-pedocs-336942.
- [Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '77)*, pages 242–248, San Jose, California, 1977. ACM.
- [DAO23] Mairi Therese Deighan, Amid Ayobi, and Aisling Ann O’Kane. Social virtual reality as a mental health tool: How people use vrchat to support social connectedness and wellbeing. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. ACM, 2023.
- [Edw] Benj Edwards. “virtual boy: The bizarre rise and quick fall of nintendo’s enigmatic red console”. *Ars Technica*.

- [FBLC24] Baptiste Fabre, Julien Bribet, Ludovic Lescieux, and Aude Caussarieu. Improving efficiency in optics education with immersive virtual reality. *Optical Engineering*, 63(7):071411, 2024.
- [G25] Sergej Görzen. Omilaxr – modular learning analytics framework for xr. <https://omilaxr.dev/>, 2025. Accessed: 2025-11-26.
- [Gal46] Johann Gottfried Galle. Account of the discovery of the planet of le verrier at berlin. *Monthly Notices of the Royal Astronomical Society*, 7(9):153–154, 1846.
- [Gal00] A Galántai. The theory of newton’s method. *Mathematics and Computers in Simulation*, 52(1-2):1–10, 2000.
- [Gia25] Giant Army. Universe sandbox. <https://universesandbox.com/>, 2025. Physics-based space simulator.
- [Hei] Morton L. Heilig. Sensorama simulator.
- [HGS22] Birte Heinemann, Sergej Görzen, and Ulrik Schroeder. RePiX VR - Learning environment for the Rendering Pipeline in Virtual Reality. In Jean-Jacques Bourdin and Eric Paquette, editors, *Eurographics 2022 - Education Papers*. The Eurographics Association, 2022.
- [KKLD23] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *CoRR*, abs/2308.04079, 2023.
- [Ló18] R. López. The solution of the generalized kepler’s equation. *Monthly Notices of the Royal Astronomical Society*, 473(2):2583–2591, 2018.
- [moz] mozilla.org. Webvr device api - web apis. https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API#browser_compatibility.
- [MPC19] Pierre Moreau, Matt Pharr, and Petrik Clarberg. Dynamic many-light sampling for real-time ray tracing. In Markus Steinberger and Tim Foley, editors, *High-Performance Graphics - Short Papers*, Goslar, DEU, 2019. The Eurographics Association.
- [Obe] Evgeny Obedkov. Godot doubled its user base in month after unity rolled out its controversial pricing update.
- [OS22] Heeseok Oh and Wookho Son. Cybersickness and its severity arising from virtual reality content: A comprehensive study. *Sensors*, 22(4):1314, 2022.
- [Piñ11] E. Piña. Rotations with rodrigues’ vector. *European Journal of Physics*, 32(5):1171–1178, 2011.
- [Quia] Inigo Quilez. Intersectors: Ray-sphere intersection. <https://iquilezles.org/articles/intersectors/>. Accessed: 2025-10-22.

- [Quib] Inigo Quilez. Useful little functions. <https://iquilezles.org/articles/functions/>. Accessed: 2025-10-22, Gain function.
- [Rod40] Olinde Rodrigues. Des lois géométriques qui régissent les déplacements d'un système solide dans l'espace et de la variation des coordonnées provenant de ces déplacements. *Journal de Mathématiques Pures et Appliquées*, 5:380–440, 1840.
- [RSKE⁺22] Ulrike Ravens-Sieberer, Anke Kaman, Michel Erhart, James Devine, Robert Schlack, and Christian Otto. Psychosocial well-being of schoolchildren during the covid-19 pandemic in berlin, germany, june 2020 to march 2021. *International Journal of Environmental Research and Public Health*, 19(16):10103, 2022.
- [Sch94] H. Joachim Schlichting. Sonnentaler – abbilder der sonne. *Praxis der Naturwissenschaften – Physik*, 43(4), 1994. Nr. 19.
- [SGS⁺24] Saeed Safikhani, Vinzenz Gattringer, Michael Schmied, Johanna Pirker, and Selina Christin Wriessnegger. The influence of realism on the sense of presence in virtual reality: Neurophysiological insights using eeg. *Multimodal Technologies and Interaction*, 8(11):104, 2024.
- [Squ15] Squad. Kerbal space program. <https://www.kerbalspaceprogram.com/>, 2015. Video game; first released out of beta on April 27, 2015. :contentReference[oaicite:0]index=0.
- [STA⁺21] Anthony Steed, Tuukka M. Takala, Daniel Archer, Wallace Lages, and Robert W. Lindeman. Directions for 3d user interface research from consumer vr games. *IEEE Transactions on Visualization and Computer Graphics*, 27(11):4171–4182, 2021.
- [Ste20] William Steigerwald. Laser beams reflected between earth and moon boost science. <https://www.nasa.gov/missions/laser-beams-reflected-between-earth-and-moon-boost-science/>, Aug 2020. Accessed: 2025-10-23.
- [SV15] Andrew Schneider and Nathan Vos. The real-time volumetric cloudscapes of horizon: Zero dawn, 2015. Presented at Advances in Real-Time Rendering, SIGGRAPH 2015.
- [TAnZnF⁺25] Caixin Tian, Roberto Ariel Abeldaño Zuñiga, Moréniké Oluwátóyìn Foláyan, Jarno Tuominen, and Jorma I. Virtanen. Psychological distress and loneliness among european university students during the covid-19 pandemic: a european survey. *BMC Public Health*, 25:3841, 2025.
- [Tec20] Unity Technologies. Unity 2019.3 is now available. Blog post on Unity.com, Jan 2020. Includes mention of the Universal Render Pipeline (URP) replacing LWRP.
- [Tec21] Unity Technologies. Mesh data — unity manual. <https://docs.unity3d.com/2021.2/Documentation/Manual/AnatomyofaMesh.html>, 2021. Accessed: 2025-11-09.

- [Tec25a] Unity Technologies. Mesh.markdynamic. <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Mesh.MarkDynamic.html>, 2025. Accessed: 2025-11-09.
- [Tec25b] Unity Technologies. Monobehaviour — scripting api. <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>, 2025. Accessed: 2025-10-26.
- [The18] The House of Fables. International space station tour vr. Steam application, 2018.
- [TSR25] Stefano Triberti, Caterina Sapone, and Giuseppe Riva. Being there but where? sense of presence theory for virtual reality applications. *Humanities and Social Sciences Communications*, 12(Article number 79), 2025.
- [Wil78] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of SIGGRAPH)*, 12(3):270–274, 1978.
- [WSX⁺22] Jialin Wang, Rongkai Shi, Zehui Xiao, Xueying Qin, and Hai-Ning Liang. Effect of render resolution on gameplay experience, performance, and simulator sickness in virtual reality games. *arXiv preprint arXiv:2203.12294*, 2022. Submitted 23 March 2022.
- [Zec18] Mathias Zechmeister. Solving kepler’s equation cordic-like. *arXiv preprint arXiv:1808.07062*, 2018.

Appendix B Digital Appendix

The digital appendix can be found in the RWTH GitLab server, defined by a repository tag here:

`https://git.rwth-aachen.de/learntech-lufgi9/bama/ma\_tobiasmoehring\_vr/-/tags/ma-hand-in`

Eidesstattliche Versicherung

Declaration of Academic Integrity

Möhring, Tobias

Name, Vorname/Last Name, First Name

345522

Matrikelnummer (freiwillige Angabe)

Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/
Masterarbeit* mit dem Titel

I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis* entitled

Development of an immersive VR simulation for middle school optics lessons

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt; dies umfasst insbesondere auch Software und Dienste zur Sprach-, Text- und Medienproduktion. Ich erkläre, dass für den Fall, dass die Arbeit in unterschiedlichen Formen eingereicht wird (z.B. elektronisch, gedruckt, geplottet, auf einem Datenträger) alle eingereichten Versionen vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without unauthorized assistance from third parties (in particular academic ghostwriting). I have not used any other sources or aids than those indicated; this includes in particular software and services for language, text, and media production. In the event that the work is submitted in different formats (e.g. electronically, printed, plotted, on a data carrier), I declare that all the submitted versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

Aachen, November 27, 2025

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen/Please delete as appropriate

Belehrung:

Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 156 StGB (German Criminal Code): False Unsworn Declarations

Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

§ 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence

(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding one year or to a fine.

(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

I have read and understood the above official notification:

Aachen, November 27, 2025

Ort, Datum/City, Date

Unterschrift/Signature

Acknowledgement

I'd like to offer my gratitude to my thesis supervisors Dr. Birte Heinemann and Prof. Bianca Watzka for their valuable support and advice during the entire process of producing this thesis work. I would also like to thank Prof. Ulrik Schroeder and the research group learning technologies for the opportunity to write this thesis and for the experiences I was allowed to gather during my time as a HiWi, as well as their patience and understanding in the final stretch of my thesis work.