POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Design and Extension of an Educational Vulkan Framework for Deferred Shading and Collision Detection Systems

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Riccardo Speroni**

Student ID: 10671842
Advisor: Prof. Marco Gribaudo
Co-advisors: Marco Domenico Buttiglione
Academic Year: 2024-2025

Abstract

The transition from high-level graphics APIs to explicit standards like Vulkan has introduced a steep learning curve in computer graphics education. While offering unprecedented control over hardware resources, the complexity of Vulkan hinders students from focusing on high-level rendering concepts. This thesis addresses this challenge by extending a custom educational Vulkan framework used at Politecnico di Milano, enhancing its capabilities to support advanced lighting via Deferred Shading and a robust Collision Detection system.

The work is structured around three main contributions. First, the rendering pipeline was re-engineered to support Deferred Shading. This involved modifying the core architecture to handle Multiple Render Targets (MRT) and implementing a two-pass workflow: a Geometry Pass for G-Buffer generation (storing Position, Normal, and Albedo) and a Lighting Pass for efficient screen-space illumination. Second, the thesis focuses on the stabilization and completion of the Colliders Module. The existing collision detection system was redesigned to correct mathematical instabilities in Oriented Bounding Box (OOBB) interactions using the Separating Axis Theorem (SAT) and to refine Bounding Volume Hierarchy (BVH) traversals. Crucially, a real-time Visual Debugger was implemented, allowing students to verify spatial interactions through dynamic wireframe rendering. Finally, these systems were unified under a Data-Driven Architecture. The scene management system was extended to parse spatial attributes directly from JSON configuration files, decoupling the logic from the C++ code. The resulting framework was validated through a comprehensive demo application, proving its capability to handle dynamic scenes with multiple light sources and collisions, thereby providing a solid platform for future academic projects.

Keywords: Computer Graphics, Vulkan, Deferred Shading, Collision Detection, Educational Framework, Framework Integration.

Abstract in lingua italiana

Il passaggio da API grafiche di alto livello a standard espliciti come Vulkan ha introdotto una ripida curva di apprendimento nella didattica della computer grafica. Sebbene Vulkan offra un elevato controllo sulle risorse hardware, la sua complessità ostacola gli studenti nell'apprendere i concetti di rendering di alto livello. Questa tesi affronta tale sfida estendendo un framework didattico Vulkan in uso presso il Politecnico di Milano, arricchendolo con il supporto all'illuminazione avanzata tramite Deferred Shading e con un robusto sistema di rilevamento delle collisioni.

Il lavoro si articola in tre contributi principali. In primo luogo, la pipeline di rendering è stata riprogettata per supportare il Deferred Shading. Ciò ha comportato modifiche architetturali per gestire Multiple Render Targets (MRT) e l'implementazione di un flusso di lavoro a due passaggi: un Geometry Pass per la generazione del G-Buffer (memorizzando Posizione, Normale e Albedo) e un Lighting Pass per un'efficiente illuminazione in screen-space. In secondo luogo, la tesi si concentra sulla stabilizzazione e sul completamento del modulo Colliders. Il sistema di rilevamento collisioni esistente è stato revisionato per correggere le instabilità matematiche nelle interazioni tra Oriented Bounding Box (OOBB) utilizzando il Teorema dell'Asse di Separazione (SAT) e per raffinare l'attraversamento delle Bounding Volume Hierarchies (BVH). Di fondamentale importanza è stata l'implementazione di un Visual Debugger in tempo reale, il quale consente agli studenti di verificare le interazioni spaziali attraverso la visualizzazione in wireframe dei collider. Infine, questi sistemi sono stati unificati in un'architettura Data-Driven. Il sistema di gestione della scena è stato esteso per supportare la specifica degli attributi spaziali direttamente dai file di configurazione JSON, disaccoppiando la logica dal codice C++. Il framework risultante è stato validato attraverso un'applicazione dimostrativa completa, comprovandone la capacità di gestire scene dinamiche con molteplici sorgenti luminose e collisioni, e offrendo così una solida piattaforma per futuri progetti accademici.

Parole Chiave: Computer Grafica, Vulkan, Deferred Shading, Rilevamento Collisioni, Framework Didattico, Integrazione Framework.

Contents

1 | Introduction

The field of Real-Time Computer Graphics has undergone a significant paradigm shift in recent years. The transition from high-level, driver-managed APIs like OpenGL to low-level, explicit APIs such as Vulkan and DirectX 12 has granted developers unprecedented control over GPU resources [1]. However, this power comes at the cost of complexity. For students and educators, this shift presents a unique challenge: how to teach modern graphics concepts without being overwhelmed by the verbose boilerplate code required to render even a simple primitive.

In an academic environment, the primary goal of a Computer Graphics course is to teach algorithms (e.g., lighting models) rather than the intricacies of graphics API initialization. Yet, ignoring modern explicit APIs is no longer a viable option, as they represent the industry standard for high-performance graphics [2].

Vulkan, developed by the Khronos Group [3], is particularly relevant due to its cross-platform nature and its ability to expose the underlying architecture of modern GPUs (e.g., explicit memory management, command buffer synchronization). However, the learning curve is notoriously steep. Drawing a single triangle in Vulkan can require hundreds of lines of initialization code, whereas OpenGL might require fewer than fifty [4].

To bridge this gap, many technical universities have adopted the strategy of developing custom "educational frameworks" or "wrapper engines." Notable examples include the *Vulkan Launchpad* developed at TU Wien [5] or the *VkCV* framework at the University of Koblenz [6]. These tools aim to abstract the low-level initialization (e.g., swapchain creation, validation layers setup) while leaving the core pipeline definition accessible to students.

At Politecnico di Milano, the Computer Graphics course adopts a similar approach, utilizing a custom C++ framework centered around a core class named `BaseProject` (defined in the `Starter.hpp` module). This architecture allows students to focus on shader programming and pipeline configuration without implementing the entire infrastructure from scratch.

While the current implementation successfully handles basic initialization, it presents limitations that hinder the realization of more ambitious student projects.

Specifically regarding the graphics pipeline, the framework was originally conceived for simple *Forward Rendering*. Consequently, it lacks the high-level infrastructure and the necessary reference implementation required to handle multi-pass techniques such as *Deferred Shading*. This absence prevents students from practically exploring advanced lighting algorithms, including the management of extensive dynamic light sources.

Moreover, although the framework includes the `Colliders.hpp` module for collision detection, it is currently in an experimental and unfinished state, making it insufficient for complex applications. While it defines data structures for five collider types (*Point, Sphere, AABB, OOBB, BVH*), the underlying intersection logic is largely unverified and lacks a corresponding sample application. Furthermore, the module suffers from a non-functional visualization system. Without the ability to render "invisible" collision volumes as wireframes in real-time, students would face significant frustration during debugging, as they could not visually distinguish between logical implementation errors and simple spatial misalignments.

The primary objective of this thesis is to extend the capabilities of the educational Vulkan framework, transforming it into a robust tool. To achieve this and simultaneously provide educational resources for future courses, the work was structured around the progressive development of four distinct sample applications, each targeting a specific architectural improvement.

First, to overcome the forward rendering limitations, a dedicated sample application was developed to validate the implementation of a Deferred Shading pipeline, using the renowned Vulkan examples made by Sascha Willems [7] as a reference. This required extending the core render pass architecture to support G-Buffer generation, handle Multiple Render Targets (MRT), and manage input attachment dependencies for efficient multipass rendering.

Second, to stabilize the collision system, a specific test application was created to isolate and debug the `Colliders.hpp` module. This phase involved fixing the mathematical instabilities and implementing a visual debugger to render collision wireframes, a critical feature previously missing.

Third, to bridge the gap between the Colliders module and the scene management system, a third application was developed to verify the extension of the `Scene.hpp` module. This enhanced the framework with the ability to parse spatial attributes (i.e., collider types,

parameters, visibility) directly from the JSON configuration file, thus aligning the collision workflow with the engine's data-driven philosophy.

Finally, to demonstrate the synergy of the implemented features, a complete demo application was developed. This final project combines the Deferred Shading pipeline with the integrated collision system in a dynamic scene, serving as a comprehensive reference for future student projects.

Consistent with the educational nature of the project, particular attention was paid to code readability and the production of these sample applications, which are intended to serve as educational references for the course.

1.1. Structure of the Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2** provides the theoretical background for the thesis. Beginning with the transition toward explicit graphics APIs, it examines how various universities address this complexity through dedicated didactic tools. Subsequently, it contrasts Forward and Deferred Shading techniques, and concludes by outlining the fundamental concepts required to understand the work done on the collision detection module.
- **Chapter 3** presents the core System Design and Implementation developed for this thesis. After an initial overview of the engine's starting architecture, the chapter details the engineering of the Deferred Shading rendering pipeline. It then describes the revision of the collision detection module, the development of a real-time Visual Debugger, and the integration of these features into a unified data-driven architecture. Finally, the overall framework is validated through a comprehensive demo application.
- **Chapter 4** concludes the thesis by summarizing the achieved contributions and outlining potential directions for future work.

2 | Theoretical Background

This chapter outlines the theoretical foundations required to understand the implementation work presented in this thesis. It begins by discussing the evolution of graphics APIs and contextualizing the framework used in this thesis within the broader landscape of academic educational tools. Subsequently, it details the rendering techniques employed, specifically contrasting Forward and Deferred Shading. Finally, it addresses the mathematical principles of collision detection, covering both intersection tests (SAT) and spatial acceleration structures (BVH).

2.1. The Shift in Graphics APIs

Historically, computer graphics education relied on APIs like OpenGL. Designed as a high-level state machine, OpenGL abstracts complexity via a global context and driver-managed resource synchronization. While lowering the entry barrier, this abstraction introduces significant driver overhead and hides the underlying hardware architecture.

Vulkan, conversely, represents a paradigm shift towards "explicit" control [1]. It exposes the GPU's asynchronous nature, requiring the application to explicitly manage memory heaps, synchronization primitives (semaphores, fences), and command buffer recording. This complexity necessitates robust educational tools, like the framework analyzed in this thesis.

2.1.1. Educational Frameworks in Academia

To mitigate the steep learning curve of explicit APIs, academic institutions have developed custom wrappers and teaching methodologies that abstract boilerplate initialization while exposing core rendering concepts. For instance, the *Vulkan Launchpad* developed at TU Wien [5] focuses on minimizing the setup code for swapchains and device queues, allowing students to immediately write shader code. Similarly, the *VkCV* framework developed at the University of Koblenz [6] provides a modular approach to handle descriptor sets and memory management without sacrificing performance. Furthermore,

pioneering educational resources like the courses developed at Oregon State University [4] have demonstrated the necessity of structuring Vulkan's verbosity into accessible teaching modules.

The custom framework used at Politecnico di Milano shares a similar didactic philosophy. Crucially, its objective is to serve as an educational vehicle to facilitate the understanding of core computer graphics concepts using Vulkan. It is designed to strike a precise pedagogical balance: it hides the initial API verbosity (via the `BaseProject` class), yet intentionally requires students to explicitly configure the graphics pipeline, manage render passes, and handle shader bindings. This ensures that learners actively engage with modern GPU architectures and rendering algorithms, rather than simply interacting with high-level abstractions.

2.2. Rendering Techniques

Modern real-time graphics engines rely on rasterization pipelines to convert 3D geometry into 2D images. The choice of the rendering architecture fundamentally determines the engine's performance characteristics, particularly regarding how it handles lighting and memory bandwidth. This section analyzes and contrasts the two dominant approaches: Forward Rendering and Deferred Shading.

2.2.1. Forward Rendering

In a traditional Forward Rendering pipeline, geometry rasterization and lighting calculations occur in a single pass. For every object, the GPU calculates the lighting contribution of every active light source. The computational complexity of this approach is approximately $O(N \cdot L)$, where N is the total number of rasterized fragments (including overdraw) and L is the number of light sources. This linear dependency makes it computationally prohibitive for scenes populated by numerous dynamic lights, as the cost of shading grows proportionally with scene complexity.

2.2.2. Deferred Shading and Subpasses

Deferred Shading decouples geometry processing from lighting calculations [8]. The rendering process is split into two distinct passes:

1. **Geometry Pass:** Scene objects are rendered without lighting. Instead of writing a final color to the framebuffer, the fragment shader outputs geometric attributes (Position, Normal, Albedo) to a set of *Multiple Render Targets (MRT)*, collectively

known as the *G-Buffer* (Geometry Buffer).

2. **Lighting Pass:** A full-screen quad is rendered. For each pixel, the fragment shader reads the geometric data from the G-Buffer textures and calculates the lighting contribution.

A visual representation of this process, showcasing the individual G-Buffer attachments alongside the final shaded composition, is illustrated in Figure 2.1.

This approach reduces the lighting complexity to $O(S \cdot L)$, where S is the number of screen pixels and L is the number of lights. Since lighting is calculated only for visible pixels (ignoring occluded geometry), the technique scales efficiently with the number of light sources.

However, this decoupling introduces specific trade-offs. The heavy reliance on multiple large textures requires significant memory bandwidth (VRAM I/O). Furthermore, since geometry is resolved before shading, standard hardware anti-aliasing (MSAA) and transparency handling become non-trivial to implement.

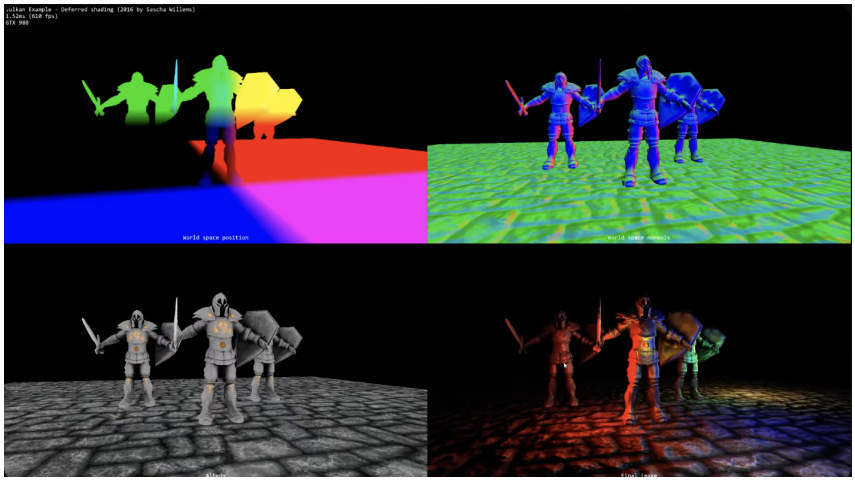


Figure 2.1: Visual breakdown of a Deferred Shading pipeline based on Sascha Willems's Vulkan examples [7]. The image displays the three G-Buffer attachments (Position, Normal, Albedo) alongside the final composition.

2.3. Collision Detection Fundamentals

A collision system relies on simplified geometric representations, known as *Bounding Volumes* or *Primitives*, to perform efficient intersection tests before (or instead of) evaluating complex meshes. The system presented in this thesis implements the following types:

- **Point:** A dimensionless primitive defined by a single position in space.

- **Sphere:** A rotation-invariant volume defined by a center (x, y, z) and a radius r.
- **AABB (Axis-Aligned Bounding Box):** A rectangular volume defined by a minimum corner P_{min} and a maximum corner P_{max}. Its faces are strictly parallel to the global coordinate system axes. While computationally efficient, it often provides a loose fit for rotating objects. Since the axes remain fixed, the box dimensions must expand to accommodate the object's rotation, resulting in a volume significantly larger than the object itself.
- **OOBB (Oriented Bounding Box):** Unlike the AABB, this volume inherits the object's orientation via the World Matrix transformation. This allows the box to rotate rigidly with the mesh, providing a significantly tighter fit at the cost of more complex intersection tests. The visual difference between these two bounding box strategies during rotation is illustrated in Figure 2.2.

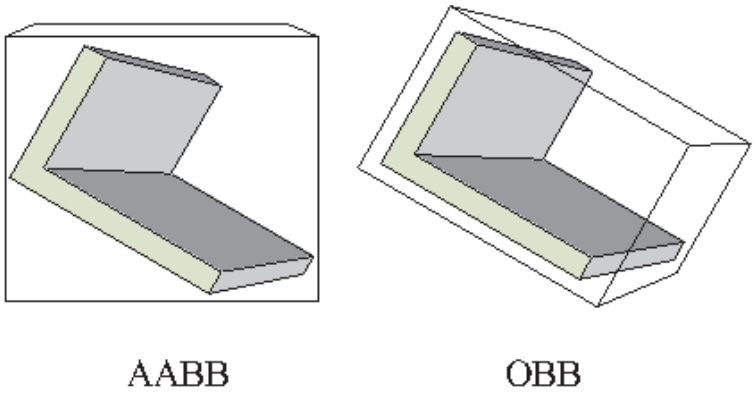


Figure 2.2: Comparison of Bounding Volumes: an Axis-Aligned Bounding Box (AABB) dynamically resizes to encapsulate the rotating mesh, whereas an Oriented Bounding Box (OOBB) rotates rigidly with the object, providing a tighter fit.

2.3.1. The Separating Axis Theorem (SAT)

For Oriented Bounding Boxes (OOBBs), standard axis-aligned comparisons are inapplicable due to the object's arbitrary rotation. Instead, the intersection test relies on the *Separating Axis Theorem (SAT)* [9]. A detailed mathematical formulation of this theorem is provided in Appendix A.

Theorem 2.1. *Two convex polyhedra A and B do not overlap if there exists at least one axis onto which the projections of the two shapes are disjoint.*

For two OOBBs, up to 15 potential separating axes must be tested: the 3 face normals

of box A, the 3 face normals of box B, and the 9 cross-products of their edges. To optimize this process, the test is typically performed by transforming one box into the local coordinate space of the other. This aligns the axes of the reference box with the coordinate system, significantly simplifying the projection calculations.

2.3.2. Spatial Acceleration: BVH

The naive approach of checking every object against every other object results in a computational complexity of $O(N^2)$. To mitigate this, the system implements a *Bounding Volume Hierarchy (BVH)* as a Broad Phase optimization [9].

A BVH is a tree-based acceleration structure where:

- **Leaf Nodes** contain the actual geometric primitives (e.g., Spheres, OOBBs).
- **Root and Internal Nodes** contain an AABB that strictly encloses the union of all its children's volumes.

During a collision query, the algorithm checks against the root node's AABB. If there is no intersection, the entire tree branch is discarded (*Pruning*). If there is an intersection, the check proceeds recursively to the children. This hierarchical strategy reduces the average complexity of the collision detection step to $O(N \log N)$.

The conceptual structure of this hierarchy, demonstrating how space is progressively subdivided, is illustrated in Figure 2.3.

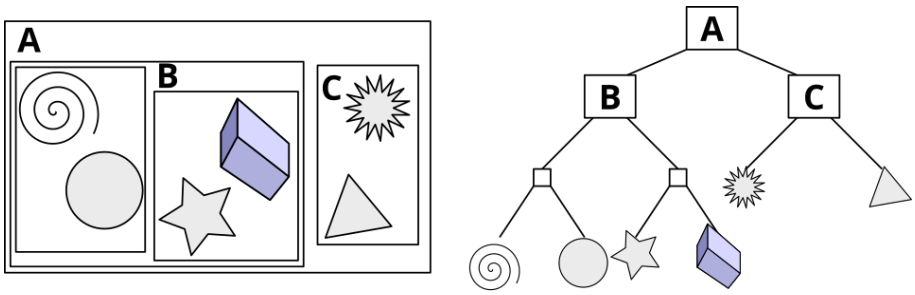


Figure 2.3: A 2D representation of a Bounding Volume Hierarchy (BVH). The root node encapsulates the entire scene, while internal nodes recursively divide the space until reaching the geometric leaf nodes.

3 | System Design and Implementation

This chapter presents the core engineering work and practical contributions developed for this thesis. It details the progressive evolution of the educational Vulkan engine, enhancing its capabilities to handle complex, interactive 3D applications.

The discussion is logically organized into five main sections. Section 3.1 begins with an analysis of the framework's starting architecture, highlighting its abstraction layers and limitations. Section 3.2 then details the engineering of the Deferred Shading rendering pipeline, focusing on the G-Buffer architecture and the management of Multiple Render Targets (MRT). Subsequently, Section 3.3 covers the comprehensive revision of the collision detection module, with particular attention to algorithmic corrections and the development of a real-time Visual Debugger. Section 3.4 explains how these new capabilities were integrated into a unified data-driven architecture using JSON configurations. Finally, Section 3.5 validates the entire framework through the presentation of a comprehensive demo application, demonstrating the synergy between the multi-pass rendering pipeline and the dynamic collision system.

3.1. Analysis of the Existing Framework

Before detailing the implementation of the new features, it is necessary to analyze the software architecture as it stood at the beginning of the thesis. Designed as a pedagogical tool, the framework aims to abstract the verbosity of Vulkan initialization (e.g., instance creation, swapchain management) while explicitly exposing the core rendering pipeline components to the student. This balance allows learners to focus on high-level graphics concepts without being overwhelmed by the complexity of Vulkan.

3.1.1. Architecture Overview

The codebase is constructed around a monolithic header file, `Starter.hpp`, which facilitates the interaction with the Vulkan API. The architecture relies on a central base class, `BaseProject`, which manages the application lifecycle. Surrounding this core, a set of helper classes encapsulates specific Vulkan resources to minimize boilerplate code.

The architecture is organized into the following logical layers:

- **Core Layer:** The `BaseProject` class, responsible for Windowing, Device selection, Swapchain creation, and Command Pool management.
- **Resource Abstraction:** Wrapper structures such as `Model`, `Texture`, `Pipeline`, and `RenderPass`. These components manage the underlying Vulkan handles and memory allocation.
- **Data Layer:** The `Scene` class (defined in `Scene.hpp`), which parses JSON configuration files to load meshes, textures, and instantiate the scene graph.
- **Collision Layer (Experimental):** The `Colliders.hpp` module, providing definitions for various bounding volumes.
- **Auxiliary Modules:** Utilities such as `TextMaker.hpp` for overlay text rendering and `Animations.hpp` for skeletal bone transformations.

3.1.2. The BaseProject Class

The `BaseProject` class serves as the foundational backbone of the application's lifecycle, encapsulating the complexity required to bootstrap a valid Vulkan context and effectively hiding the initialization verbosity from the user.

Its responsibilities begin with the hardware setup, where it handles the selection of the most suitable Physical Device (GPU), validates support for essential extensions (e.g., `VK_KHR_swapchain`), and creates the Logical Device alongside the necessary Graphics and Presentation Queues. Following this, it manages the creation of the presentation surface and the swapchain images, automatically accommodating window resizing events via GLFW callbacks to dynamically recreate the swapchain when dimensions change.

Beyond initial allocation, the class orchestrates the core rendering loop. It owns the global `VkCommandPool` and abstracts the command submission logic by exposing a callback mechanism (typically `populateCommandBuffer`). This architectural choice enforces a strict separation between command *submission*, handled internally by the engine, and

command *recording*, which is assigned to the student.

Finally, to ensure robust execution, the class manages the allocation and signaling of all synchronization primitives, specifically Semaphores and Fences, coordinating CPU-GPU execution and maintaining consistent frame pacing throughout the application's runtime.

3.1.3. Resource Management Wrappers

To mitigate the verbosity associated with raw Vulkan resource management, the framework provides a set of high-level wrapper structures defined within `Starter.hpp`. These classes abstract the complex allocation and binding procedures required by the API:

- **Model:** Encapsulates the management of Vertex and Index Buffers. It integrates external libraries (specifically `tinyobjloader` and `tinygltf`) to parse 3D geometry from standard file formats and handles the synchronization required to upload vertex data to device-local GPU memory.
- **Texture:** Abstracts the creation and bundling of the three essential components required for texturing: the `VkImage` (pixel data), the `VkImageView` (format interpretation), and the `VkSampler` (filtering modes). It simplifies the loading of image assets from disk, handling format conversion automatically.
- **VertexDescriptor:** Acts as a bridge between C++ data structures and the GPU vertex shader input. It automates the creation of the `VkVertexInputAttributeDescription` and `VkVertexInputBindingDescription`, allowing the user to specify which components (Position, Normal, UV, etc.) are present in the mesh without manually calculating byte offsets and strides.
- **DescriptorSetLayout:** Encapsulates the `VkDescriptorSetLayout`, serving as the "blueprint" for shader resources. It defines the contract between the shader stages and the application, specifying the types of resources (e.g., Uniform Buffers, Combined Image Samplers) and their binding points, effectively abstracting the pipeline resource interface configuration.
- **DescriptorSet:** Manages the allocation of `VkDescriptorSets` and the underlying Uniform Buffers. It provides a simplified interface for mapping CPU memory to GPU-visible memory, allowing the student to update shader uniforms (e.g., transformation matrices, light parameters) frame-by-frame without managing raw memory mapping or buffer synchronization manually.
- **Pipeline:** Wraps the `VkPipeline` state object and its `VkPipelineLayout`. Its pri-

mary role is to simplify the configuration of the graphics pipeline stages, allowing the user to define programmable shaders, vertex input layouts, and fixed-function states (e.g., rasterization, depth testing, and color blending) without manually populating the extensive `VkGraphicsPipelineCreateInfo` structure.

- **RenderPass:** Encapsulates the definition of the `VkRenderPass` object and the creation of the associated `VkFramebuffers`. It manages the configuration of Input and Output Attachments (Color, Depth, Resolve) and defines Subpass Dependencies. This class is particularly crucial as it abstracts the linkage between the rendering pipeline and the swapchain (or offscreen) images.

3.1.4. Data-Driven Design: The Scene Class

The framework implements a data-driven architecture for scene composition through the `Scene` class. Rather than hardcoding object instantiation and positioning in C++, the module parses a JSON configuration file (typically `scene.json`) to dynamically load resources and construct the render graph at runtime.

The architecture decouples raw resource loading from logical scene placement using three primary data arrays. First, the `Assets` array defines the raw 3D files (such as GLTF, OBJ, or the custom MGCG format) to be loaded from disk. Second, the `Models` array specifies the geometry resources that the renderer will use. A model entry can point to a single file or extract a specific mesh node from a shared asset, a distinction that optimizes memory usage by allowing multiple logical models to reuse geometry from a single heavy GLTF container. Finally, the `Textures` array maps image files to internal identifiers, enabling the user to specify usage flags, such as whether a texture contains RGB color data or linear non-color data.

The core of the scene definition resides in the `Instances` array, which organizes renderable objects hierarchically by grouping them according to their `Technique` (i.e., the specific Shader Pipeline they utilize). This design choice is not merely organizational, but acts as a crucial rendering optimization: by grouping objects that share the same pipeline, expensive state changes during command buffer recording are minimized. For each instance, the JSON defines its *Resource Binding* (the identifiers of the associated model and textures) and its spatial *Transformation*, configured either as an explicit 4×4 World Matrix or via individual Translation, Rotation, and Scale components.

Crucially, in its initial version, this JSON parser was strictly limited to graphical rendering properties. The `Scene` class contained no logic to parse spatial attributes, such as collider types or bounding box dimensions, meaning that any collider definition had to be

hardcoded in C++.

Note on the MGCG Format. The framework supports a custom file format with the extension `.mgcg`. Technically, this is a container that wraps standard GLTF data using AES encryption and Deflate compression (implemented via the `plusaes` and `sinfl` libraries included in `Starter.hpp`). This mechanism was designed to protect high-quality commercial assets used in the course from unauthorized extraction and distribution.

3.1.5. The Collision Detection Module: Colliders.hpp

The initial codebase included a header file named `Colliders.hpp`, designed to provide collision detection capabilities. Unlike the rendering components, which were largely functional, this module was in a prototypal state.

The core of the module was the `Collider` class. It utilized a `colliderType` enumeration to determine the active shape and a series of generic data fields (e.g., `x1, y1, z1` for positions, `r` for radius) to store geometric parameters.

The supported bounding volumes, defined in the enumeration, included:

- `CLD_POINT`: A dimensionless point in 3D space.
- `CLD_SPHERE`: Defined by a center and a radius.
- `CLD_AABB`: An Axis-Aligned Bounding Box.
- `CLD_OOBB`: An Oriented Bounding Box, intended to support rotation.
- `CLD_BVH`: A node structure for Bounding Volume Hierarchies.

A preliminary inspection of the code confirmed the prototypal nature of the module. While the data structures were defined, essential debugging tools were missing, and the reliability of the collision algorithms was unproven. Specifically regarding the visualization infrastructure, although the `ColliderShow` class was present, it lacked the implementation for essential rendering logic. Methods required to generate debug geometry, such as `MakeBox`, were declared but empty, making it impossible to visually validate the position and orientation of the bounding volumes.

Furthermore, the underlying intersection logic was unverified. The collision handling relied on a double-dispatch mechanism via nested `switch` statements, but since the module was provided as an experimental draft without guarantees on functionality, the mathematical correctness of these algorithms was unknown. Consequently, a primary requirement

was to subject these primitives to rigorous testing to determine whether the existing implementation was usable or required rewriting.

Finally, these collision features were completely disconnected from the engine's data-driven pipeline. The `Scene` class lacked the logic to parse collider definitions from the JSON configuration files, forcing the user to hardcode collision properties in C++ and negating the flexibility of the asset management architecture.

3.2. Implementation of Deferred Shading

This section details the technical work undertaken to implement a Deferred Shading technique starting from the professor's educational engine. This process required modifications to the core `Starter.hpp` module to correctly support Multiple Render Targets (MRT) and the creation of a specialized pipeline for G-Buffer generation.

3.2.1. G-Buffer Architecture

The foundational step was defining the structure of the Geometry Buffer (G-Buffer). A new configuration, `AT_GBUFFER`, was added to the `StockAttachmentsConfiguration` enumeration within the framework. The `RenderPass::getStandardAttchmentsProperties` method was updated to strictly define the format and usage of the four required attachments:

- **Position Attachment:** `VK_FORMAT_R16G16B16A16_SFLOAT`. Stores the world-space coordinates.
- **Normal Attachment:** `VK_FORMAT_R16G16B16A16_SFLOAT`. Stores the surface normals.
- **Albedo Attachment:** `VK_FORMAT_R8G8B8A8_UNORM`. Stores the base color of the material.
- **Depth Attachment**: The depth format is dynamically selected via `findDepthFormat` (typically `VK_FORMAT_D32_SFLOAT`). This serves as the standard depth buffer for occlusion testing.

The configuration prioritizes high-precision floating-point formats for geometric data to ensure accurate lighting calculations in the subsequent pass. For the sake of brevity, the detailed C++ configuration struct (including specific usage flags, load/store operations, and image layout transitions) is reported in Appendix B.

3.2.2. Modifications for MRT Support

To support the G-Buffer, the engine's pipeline creation logic required structural updates, as the original architecture was implicitly designed for Forward Rendering and assumed a single color output (the Swapchain image).

A critical technical challenge encountered during this transition was the management of Vulkan's color blending states. The `VkPipelineColorBlendStateCreateInfo` structure requires an array of `VkPipelineColorBlendAttachmentState` structures that must match exactly the number of color outputs defined in the subpass.

The original `Pipeline::create` method in `Starter.hpp` hardcoded a single attachment. Failing to provide a matching array for the G-Buffer (which requires three distinct color targets: Position, Normal, and Albedo) results in validation layer errors and undefined behavior, typically with only the first attachment being written.

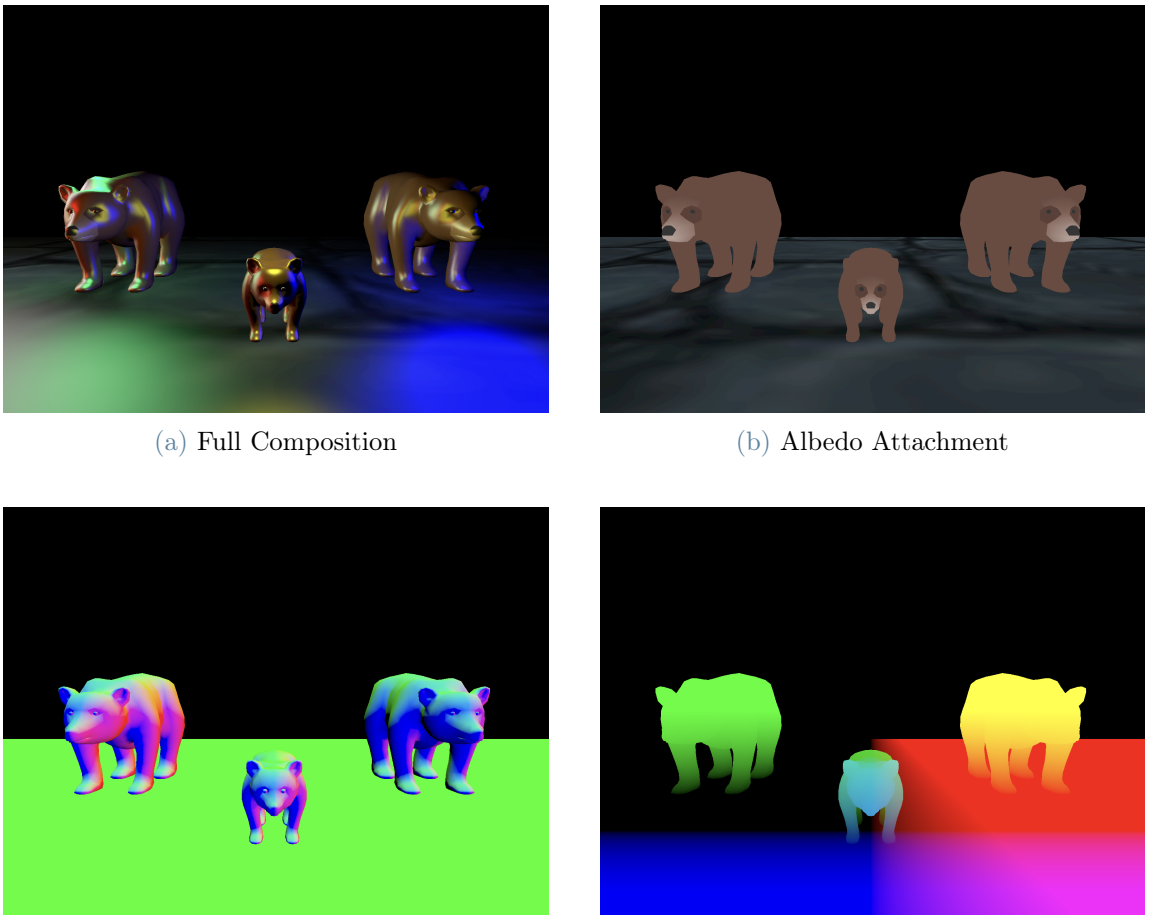
To resolve this, the pipeline creation logic was modified to dynamically resize the blend attachment vector based on the active render pass configuration. This structural update ensures that all G-Buffer targets are written simultaneously and correctly.

3.2.3. The Reference Application

To validate the architectural changes and provide students with a practical reference, a complete example application named *Deferred Shading* was developed. This project, conceptually based on the educational samples by Sascha Willems [7], serves as a blueprint for implementing multi-pass rendering.

The core of the example lies in the `populateCommandBuffer` method, which orchestrates the two-pass rendering strategy and demonstrates how to synchronize resource usage between passes. The workflow begins with the Geometry Pass, during which the application binds the `RP_GBuffer` render pass and renders scene objects using the specialized `P_GBuffer` pipeline. Crucially, strictly geometric data is processed during this phase, while lighting calculations are completely omitted to ensure optimal performance. The output is stored in the off-screen framebuffers defined in Section 3.2.1. Subsequently, the application transitions to the Lighting Pass, targeting the swapchain via the `RP_Lighting` render pass. Here, a descriptor set exposing the G-Buffer attachments as `sampler2D` resources is bound, and the scene lighting is computed by rendering a single full-screen quad. The fragment shader samples the G-Buffer textures for every pixel, applies the Phong model using the light data provided by the Global Uniform Buffer, and outputs the final color.

A critical aspect of understanding Deferred Shading is the ability to visualize the intermediate data stored in the G-Buffer. To facilitate this, the example includes an interactive debug feature controlled via keyboard input. Within the `updateUniformBuffer` method, the application listens for specific keystrokes (keys `1` through `4`) to toggle a `debugMode` flag. As illustrated in Figure 3.1, this allows the user to dynamically inspect individual attachments directly on the screen, switching between the final full composition (Figure 3.1a), the unlit base color (Figure 3.1b), and the RGB representations of both surface normals (Figure 3.1c) and world-space coordinates (Figure 3.1d). This interactive switching allows students to verify the correctness of each attachment independently, making the debugging of rendering artifacts significantly more intuitive.



(a) Full Composition (b) Albedo Attachment

(c) Normal Attachment (d) Position Attachment

Figure 3.1: Visual output of the Deferred Shading application, allowing inspection of the final composition and individual G-Buffer attachments.

Design Choices and Limitations. A deliberate simplification was adopted regarding the material model to prioritize the implementation of the core Deferred Shading tech-

nique. Specifically, specular highlights are currently calculated using a global intensity factor within the Lighting Pass, as the G-Buffer layout does not dedicate a specific render target for per-pixel specular maps.

However, since the Albedo attachment utilizes a standard 4-component format (RGBA), the alpha channel remains currently unused. While the omission of specular maps limits the current visual fidelity, this unallocated channel effectively reserves storage space within the existing G-Buffer structure. This layout allows for the future introduction of spatially varying specularity (via *channel packing*) without requiring additional memory allocation or increasing bandwidth usage.

3.3. Completion and Validation of the Collision System

As highlighted in the architectural analysis (Section 3.1), the existing collision module was experimental and unverified. This section details the engineering work undertaken to transform the `Colliders.hpp` prototype into a robust and fully functional system.

The development efforts focused on three key areas, executed in a progressive workflow. First, a comprehensive *visual debugging infrastructure* was implemented. By developing the necessary rendering logic to display colliders in wireframe mode, it became possible to verify their spatial configurations visually. This visual foundation served as an essential prerequisite for the subsequent *algorithmic corrections*, where the underlying intersection logic was systematically analyzed and rectified to establish mathematical stability, with particular attention paid to Oriented Bounding Boxes (OOBBs) and Bounding Volume Hierarchies (BVH). Finally, to guarantee the reliability of the revised module, a dedicated *validation test suite* was designed to systematically verify the correctness of all interactions between colliders.

3.3.1. Implementing Visual Debugging

The initial step involved finalizing the implementation of the `ColliderShow` class to enable the visual verification of bounding volumes. Given the complexity of spatial intersection algorithms, this visual debugging tool proved indispensable for identifying and diagnosing errors.

The geometry generation logic is encapsulated within dedicated helper methods tailored to each primitive type, as illustrated in Figure 3.2. To keep the main text concise, the

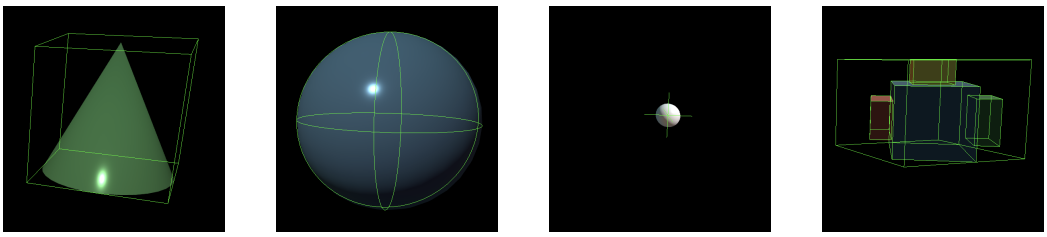
full C++ implementation of these algorithms is provided in Appendix C.

For box-like structures, the `MakeBox` method generates the twelve edges of a cuboid, dynamically adapting its behavior based on the specific type. For Oriented Bounding Boxes (OOBBs, Figure 3.2a), it transforms the eight local corners using the object's World Matrix to correctly visualize rotation; conversely, for Axis-Aligned Bounding Boxes (AABBs), it computes the axis-aligned extents of the transformed shape and rebuilds the corners to remain strictly aligned with the global axes.

Regarding spherical colliders, rendering a full 3D wireframe is computationally expensive and visually cluttered. Therefore, the `MakeSphere` method implements a lightweight representation consisting of three orthogonal circles on the local XY, XZ, and YZ planes (Figure 3.2b). These circles are generated parametrically based on a specific `SPHERE_RESOLUTION`.

For point colliders, since a single pixel is often insufficient to visualize a dimensionless geometric entity in 3D space, the `MakePoint` method renders a small crosshair. This representation consists of three short segments aligned with the cardinal axes, ensuring that the spatial location is clearly visible against the background (Figure 3.2c).

Finally, Bounding Volume Hierarchies (BVH, Figure 3.2d) are visualized recursively. The system renders the AABB of the current node and then traverses its children to display their respective underlying geometries, allowing for a comprehensive visual inspection of the entire tree structure.



(a) OOBB (b) Sphere (c) Point (d) BVH

Figure 3.2: Visual output of the `ColliderShow` class, displaying wireframes for different primitive types.

Beyond geometry generation, the rendering pipeline was updated to provide dynamic visual feedback, making it possible to distinguish between colliding and non-colliding objects. Vulkan Push Constants were utilized to pass a unique `colliderIndex` to the vertex shader for each draw call. This mechanism allows the application to dynamically change the color of a specific collider (e.g., turning it red upon impact, as shown in Figure 3.3) via methods such as `setStrokeByIndex`.

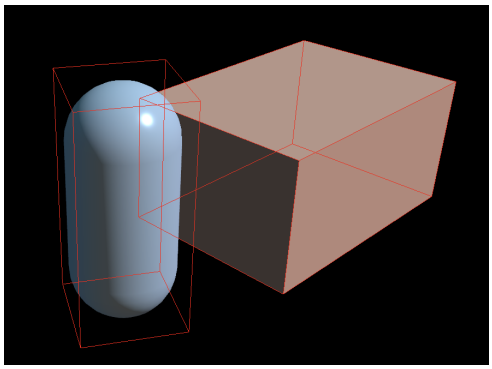


Figure 3.3: Visual debugging in action: collider wireframes turn red when an intersection is detected.

3.3.2. Algorithmic Corrections

To systematically address the reliability issues of the collision module, an iterative "verify-and-fix" workflow was adopted. For each collider type, the development process followed three distinct steps:

1. **Visualization Implementation:** First, the wireframe rendering logic was finalized (as described in Section 3.3.1) to ensure that the bounding volumes could be accurately perceived in 3D space.
2. **Scenario-Based Testing:** Using the test application, specific interaction scenarios were created to stress-test the intersection algorithms.
3. **Algorithmic Corrections:** Any anomaly observed during the dynamic tests (such as missed collisions or false positives) was analyzed and resolved by rewriting the underlying mathematical logic.

This systematic approach exposed structural weaknesses in the handling of Oriented Bounding Boxes (OOBBs) and hierarchical structures (BVHs). The specific mathematical solutions implemented to address these flaws are detailed in the following.

Redesign of the Collision Dispatcher

The initial architecture of the `collidesWith` method relied on an ad-hoc code reuse strategy. Instead of implementing specific intersection tests for Oriented Bounding Boxes (OOBBs), the original code attempted to reduce them to Axis-Aligned Bounding Box (AABB) checks directly within the dispatch switch. This was achieved by creating temporary collider instances and applying inverse World Matrix transformations on-the-fly.

Listing 3.1: Original instability: inline transformations in the dispatcher

```
// ORIGINAL CODE (Snippet)
case CLD_OOBB:
   switch(dest.type) {
     case CLD_SPHERE:
        // Attempt to recycle Sphere-AABB logic inline
        Collider k = dest;
        k.setWorldMatrix(glm::inverse(Wm) * k.Wm);
        return collisionSphereAABB(k, *this); // Unstable result
     // ... other cases using similar inline workarounds
   }
```

During the testing phase, this approach proved to be fragile. The repeated creation of temporary objects and the inline matrix inversions led to spatial misalignments, where collisions were detected with significant offsets or false negatives.

To resolve these issues, the dispatch logic was redesigned to rely on explicit delegation. All inline transformation logic was removed and replaced with calls to dedicated intersection methods. This architectural change necessitated the expansion of the `Collider` class interface, adding explicit support for `collisionPointOOBB`, `collisionSphereOOBB`, and `collisionOOBBOOBB`.

Listing 3.2: Redesigned dispatcher: delegating to specific algorithms

```
// FINAL CODE (Snippet)
case CLD_OOBB:
   switch(dest.type) {
     case CLD_SPHERE:
        // Clean delegation to a dedicated, encapsulated algorithm
        return collisionSphereOOBB(dest, *this);
     // ...
   }
```

Implementation of Dedicated OOBB Algorithms

Following the architectural redesign, robust mathematical logic was implemented for the newly added methods.

For the **OOBB-OOBB** intersection (`collisionOOBBOOBB`), the coordinate transformation approach previously attempted inline was formally structured. Encapsulating this logic in a dedicated function allowed for precise verification of the underlying matrix operations. Using the inverse World Matrix, the algorithm transforms the second OOBB into the local coordinate system of the first. In this local space, the first box effectively acts as an Axis-Aligned Bounding Box (AABB). This transformation allows to delegate the actual intersection test to the `collisionOOBBAABB` method, which implements the *Separating Axis Theorem (SAT)*. This method sequentially evaluates up to 15 potential

separating axes (face normals and edge cross-products), halting the check as soon as a separation is found to ensure both accuracy and optimal performance.

The **Point-OOBB** intersection (`collisionPointOOBB`) follows the same spatial principle. Instead of relying on ad-hoc approximations, the algorithm transforms the query point from world space into the OOBB's local coordinate system via the inverse World Matrix. Once in local space, the problem simplifies to a standard containment check against the box's minimum and maximum extents.

Finally, the **Sphere-OOBB** intersection (`collisionSphereOOBB`) was developed to resolve the inaccuracies of the original code reuse attempt. Rather than approximating the bounding volume, the newly implemented algorithm performs the following steps:

1. Computes the vector from the OOBB's center to the sphere's center, projecting it onto the OOBB's oriented axes using dot products.
2. Clamps these projected distances within the box's half-extents, reconstructing the world-space coordinates of the closest point in or on the OOBB.
3. Verifies the intersection by comparing the squared distance between the sphere's center and this closest point against the sphere's squared radius, thus avoiding computationally expensive square root operations.

Refinement of the BVH Logic

The integration of the Bounding Volume Hierarchy (BVH) required corrections in both the construction and traversal phases.

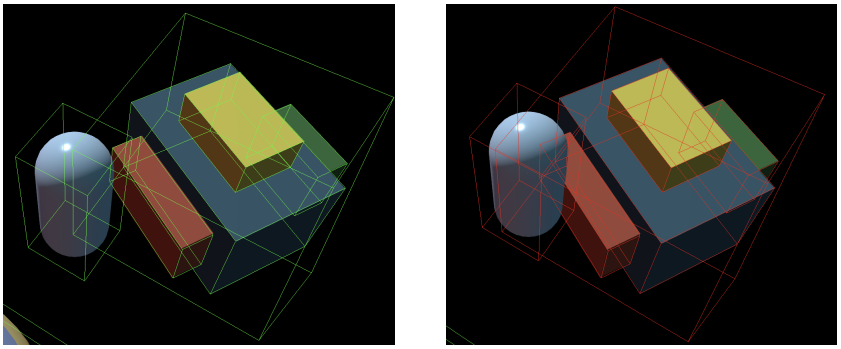
During the construction phase (`initBVH`), the original code failed to correctly compute the bounding volume of parent nodes. To resolve this, a loop was implemented to iterate through all assigned children, computing the union of their bounds via *min* and *max* operations. This ensures that the parent AABB perfectly encapsulates all sub-colliders, which is a fundamental prerequisite for the pruning optimization.

In the traversal phase (`collisionBVHCollider`), the initial recursive implementation contained a critical logical flaw: it failed to handle leaf nodes correctly. If a BVH node had no children (representing actual geometry), the function would return `false` immediately, causing missed collisions. To address this issue, the algorithm was restructured to handle three distinct cases:

- **Pruning:** The target is first checked against the aggregate AABB of the current node. If no intersection is detected, the branch is pruned immediately.

- **Leaf Node:** If the node has no children, it represents a geometric primitive. The algorithm delegates the intersection test to the specific method via the `collidesWith` dispatcher.
- **Internal Node (Recursion):** If the node is internal, `collisionBVHCollider` is called recursively on all its children. This ensures that the AABB pruning optimization is applied at every level of the hierarchy, rather than just at the root.

The correctness of this logic is verified visually in Figure 3.4. Figure 3.4a demonstrates the hierarchical precision: although the player is inside the root node's bounding volume, the system correctly reports no collision (green wireframe) because no specific child primitive is intersected. Conversely, Figure 3.4b shows a confirmed collision (red wireframe) where the player intersects the actual geometry of a leaf node.



(a) BVH before collision (b) BVH during collision

Figure 3.4: BVH Collisions: visualizing the hierarchy structure and the successful detection of intersections.

3.3.3. Validation: The Colliders Test Application

To validate the algorithmic corrections and the visualization system, a dedicated testing application named *Colliders Test* (implemented in the `cTest` class) was developed. Although initially conceived as a development sandbox for debugging wireframes and intersection tests, the final implementation has been extensively documented and structured to serve as an educational reference for students.

Scenario Configuration

The application instantiates twelve distinct 3D models, carefully selected to represent diverse geometric topologies (e.g., Torus, Cone, Cylinder, Monkey Head). The initialization phase (`localInit`) validates the versatility of the system by applying different collider

strategies.

For meshes that geometrically correspond to one of the engine's supported primitive colliders, the specific collider type is manually instantiated to ensure an exact fit. For example, a `Point` collider is assigned to a small point-like mesh (Index 7), and a `Sphere` collider is mapped to the sphere mesh (Index 9), mirroring their intrinsic geometry.

For general geometry lacking a direct topological counterpart (such as the irregular "Suzanne" monkey head or the "Torus"), the application utilizes auto-fitting algorithms via the `fitAABB` or `fitOOBB` methods. This demonstrates the engine's capability to automatically analyze the raw vertex buffer and compute the optimal bounding box approximation.

Finally, to demonstrate the Bounding Volume Hierarchy (BVH) capabilities, a specific composite mesh (Index 1) is included. This object features a large central block flanked by three smaller sub-components. To validate the hierarchy logic, the `setupBVH` method manually constructs a collider tree: it instantiates four child OOBBs (aligned with these sub-components) and groups them under a root BVH node. This configuration confirms that the system correctly aggregates the extents of spatially distributed children into a coherent parent volume.

The Validation Loop

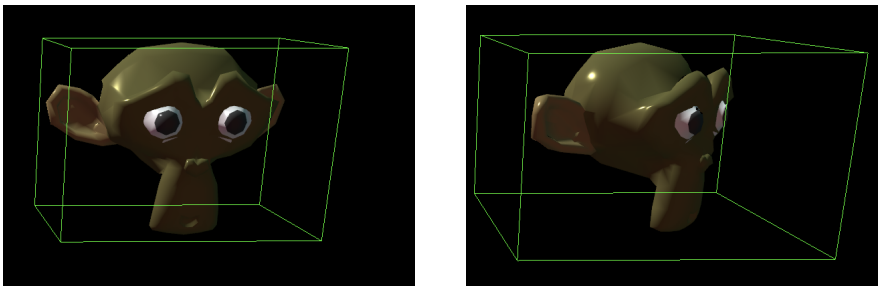
The core logic resides in the `updateUniformBuffer` method, which acts as the main application loop. It implements a continuous validation cycle consisting of five stages:

1. **Input and Movement:** The user controls a "Player" object via keyboard input, allowing for dynamic interaction with the scene elements.
2. **Runtime Reconfiguration:** To verify the robustness of the fitting algorithms, the application allows for runtime switching of collider types using debug keys. Pressing `'1'` forces all auto-generated colliders (thus, excluding the BVH and manually assigned primitive colliders like Point and Sphere) to be recalculated as AABBs, while pressing `'2'` switches them to OOBBs.
3. **Spatial Synchronization:** A critical step involves ensuring that the physical representation aligns perfectly with the visual mesh. The application iterates through all scene objects and updates their colliders via `setWorldMatrix`. This process highlights the fundamental difference between the bounding volumes, which is particularly evident in the rotating "Suzanne" model. In AABB mode (Figure 3.5), the box remains strictly axis-aligned and must dynamically resize (expanding and

shrinking) to continuously encompass the rotating geometry. Conversely, in OOBB mode (Figure 3.6), the bounding box rotates rigidly with the mesh, maintaining a constant, precise fit throughout the animation.

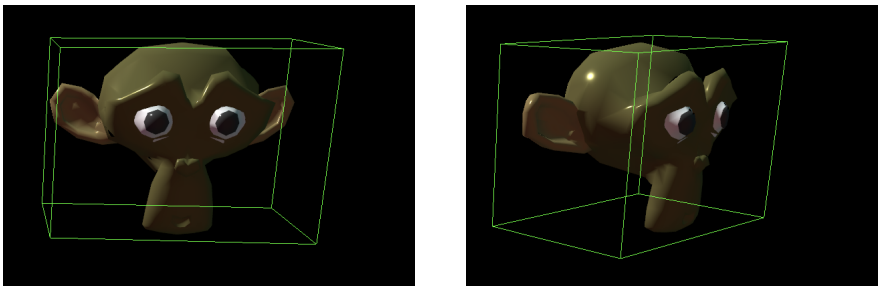
4. **Intersection Testing:** The system performs an $O(N)$ collision check between the Player and all other scene objects, providing immediate visual feedback. Wireframes are rendered in green (`glm::vec4(0,1,0,1)`) when no intersection occurs, and instantly turn red (`glm::vec4(1,0,0,1)`) when the `collidesWith` method detects a collision.

5. **Rendering Optimization:** Finally, the loop handles the visualization update. While the transformation matrices are updated every frame via `updateTransforms` to ensure smooth motion, the computationally expensive mesh reconstruction (triggered by the `refresh` method) is throttled using a timer. This optimization ensures that the Command Buffer re-recording occurs at a controlled rate (i.e., every 10 ms), preventing unnecessary CPU overhead.



(a) Start Position (b) After Rotation (Resized)

Figure 3.5: Dynamic AABB: the box remains axis-aligned but must change dimensions to encapsulate the rotating monkey head.



(a) Start Position (b) After Rotation (Transformed)

Figure 3.6: Dynamic OOBB: the box rotates with the object, maintaining a precise fit.

3.4. Data-Driven Collision Integration

The previous sections detailed the individual implementations of the Deferred Shading technique and the Colliders module. This section focuses on the integration of the collision system into the engine's data-driven architecture. The primary goal was to allow a user to define a scene with its respective collision properties by leveraging the simplicity of a declarative JSON-based configuration.

3.4.1. Extending the Scene Management System

The `Scene.hpp` module is responsible for parsing the JSON configuration file that defines the virtual world. In its original version, this parser only recognized graphical assets (such as meshes and textures) and their spatial placement.

To integrate the collision system, the JSON schema was extended to support a new optional field, `"collider"`, within the `"models"` definition. Defining the collider as an intrinsic property of the model ensures both efficiency and consistency. This approach allows collision properties to be specified once and automatically propagated to all instances of that model within the scene.

The extended schema allows the user to specify several attributes:

- **Type:** The specific bounding volume (`AABB`, `OOBB`, `Sphere`, `Point`, or `BVH`).
- **Visibility:** A boolean flag to enable or disable the wireframe debug rendering for that specific model.
- **Parameters:** An array of floating-point values to specify the collider's geometric dimensions (e.g., the extents for an `AABB`/`OOBB`, or the center and radius for a `Sphere`).
- **Children:** An optional array used specifically for BVHs to define the hierarchy of child colliders.

For `AABB` and `OOBB` types, the system supports a shorthand syntax: the user can simply specify the type string (e.g., `"collider": "AABB"`). In this case, the bounding volume is automatically fitted to the raw mesh vertices, and the wireframe visualization is disabled by default.

3.4.2. Unified Loading Workflow

The integration required a significant update to the scene loading routine. The new workflow proceeds as follows:

1. **Model Parsing:** As the parser iterates through the `"models"` array, it loads the mesh data for each entry into memory while simultaneously checking for the `"collider"` field. If this field is present, the `ParseColliderRecursive` method is invoked to populate a `ColliderDef` structure. Furthermore, for supported auto-fitting types (i.e., `AABB`, `OOBB`), the absence of explicit JSON parameters triggers the automatic computation of optimal bounding volume extents directly from the mesh's vertex buffer, which are then stored in the definition.
2. **Instance Creation and Collider Allocation:** During the parsing of the `"instances"` array, the World Matrix (W_m) is computed for each object. If the corresponding model possesses a collider definition, the `CreateColliderRecursive` method is called. This acts as a factory that:
   - Allocates a new `Collider` object on the heap.
   - Recursively instantiates child colliders if the type is `BVH`.
   - Assigns the instance's initial World Matrix to the collider.
   - Adds the pointer to a `GlobalColliders` list for centralized memory management.
3. **Visualizer Registration:** During instantiation, if the `visible` flag is set in the definition, the new collider pointer is automatically registered with the `ColliderShow` system (the visual debugger described in Section 3.3).
4. **Runtime Synchronization:** The `Instance` struct now holds a pointer to its specific `Collider`. During the rendering loop, any transformation applied to the instance is propagated to the collider via `setWorldMatrix`, ensuring the collision geometry stays perfectly synchronized with the visual representation.

3.4.3. The Scene Integration Example

To validate this new data-driven pipeline, the complex test scenario previously hardcoded in the *Colliders Test* application (Section 3.3.3) was fully reconstructed using the JSON configuration.

The following snippet from the *Scene Integration* example demonstrates this capability,

featuring a mix of auto-generated colliders (e.g., `Suzanne`), manually defined primitives (`Point`, `Sphere`), and the complex hierarchical collider (`BVH`).

Listing 3.3: Scene configuration from the Scene Integration example

```
"models": [
   {
      "id": "Point", "VD": "VDsimp", "model": "assets/models/Point.obj",
      "format":"OBJ",
      "collider": {
         "type": "Point", "visible": true, "params": [0,0,0]
      }
   },
   {
      "id": "Sphere", "VD": "VDsimp", "model": "assets/models/Sphere.obj",
      "format":"OBJ",
      "collider": {
         "type": "Sphere", "visible": true, "params": [0,0,0,1]
      }
   },
   {
      "id": "Suzanne", "VD": "VDsimp", "model": "assets/models/Suzanne.obj",
      "format":"OBJ",
      "collider": {
         "type": "AABB", "visible": true, "params": []
      }
   },
   {
      "id": "BVH", "VD": "VDsimp", "model": "assets/models/BVH.obj",
      "format":"OBJ",
      "collider": {
         "type": "BVH",
         "visible": true,
         "children": [
            { "type": "OOBB", "params": [-0.425, 0.25, -0.53, 0.275, 0.68, 0.53] },
            { "type": "OOBB", "params": [-0.75, -1.0, -1.0, 0.6, 0.25, 1.0] },
            { "type": "OOBB", "params": [0.6, -0.725, -0.725, 1.0, -0.03, 0.725] },
            { "type": "OOBB", "params": [-1.17, -0.775, -0.675, -0.75, 0.025, 0.3] }
         ]
      }
   }
   // ... other models (Cone, Cylinder, etc.) omitted for brevity
],
"textures": [
   {"id": "tex", "texture": "assets/textures/Palette.png", "format": "C"}
],
"instances": [
   {"technique": "CookTorranceNoiseSimp", "elements": [
    {"id": "Point", "model": "Point", "texture": ["tex"], "translate": [6.9,0,4]},
    {"id": "Sphere", "model": "Sphere", "texture": ["tex"], "translate": [-4,0,-6.9]},
    {"id": "Suzanne", "model": "Suzanne", "texture": ["tex"], "translate": [-6.9,0,-4],
    "eulerAngles": [90, 60, 0]},
    {"id": "BVH", "model": "BVH", "texture": ["tex"], "translate": [-4,0,6.9],
    "eulerAngles": [0, -30, 0]}
   ]}
]
```

This data-driven approach effectively decouples the collider definitions from the C++ code. These definitions are parsed during the loading phase to instantiate the corresponding colliders, seamlessly handling both explicit shapes and auto-generated volumes.

With the collision system now fully integrated into the core architecture, the framework is equipped to handle complex dynamic scenes. This sets the stage for the final validation presented in the next section, where the Deferred Shading technique and spatial interactions are combined in a comprehensive demonstration.

3.5. The Final Application

To validate the contributions of this thesis, a final demonstration application named *Complete Example* was developed. This project combines the Deferred Shading technique (Section 3.2) and the integrated collision system (Section 3.4) into a unified, interactive environment. Accordingly, this section details the architectural strategy used to merge the data-driven capabilities of the `Scene` module with the multi-pass rendering requirements of the deferred pipeline.

3.5.1. Scene Setup

The scene reproduces the scenario presented in the *Deferred Shading* application (Section 3.2), enhancing it with a controllable player object and collision properties. The scene content is defined entirely within a JSON configuration file, separating the data from the logic. Specifically, the environment consists of a textured floor acting as the static reference plane, populated by collidable entities (two adult bears and a cub) equipped with auto-fitted `OOBB` colliders. Finally, to validate real-time intersection testing, the user navigates this space using a controllable capsule-shaped mesh, which also relies on an `OOBB` bounding volume.

3.5.2. The Hybrid Rendering Strategy

A key engineering challenge in developing this final application came from the structural limitations of the `Scene.hpp` module. Being strictly designed for instancing 3D meshes from JSON data, the module lacks the underlying infrastructure required to manage global post-processing effects or complex multi-pass dependencies, such as executing a deferred Lighting Pass on a full-screen quad.

To overcome this, a hybrid architecture was adopted:

- **Geometry Pass (Data-Driven):** The generation of the G-Buffer is delegated to the `Scene` object. The scene loader iterates over the JSON instances and issues the draw calls for the 3D models using the injected deferred pipeline.
- **Lighting Pass (Hardcoded):** The composition of the final image is managed explicitly within the C++ application code. This pass does not rely on the JSON scene loader but instead draws a procedurally generated full-screen quad to sample the G-Buffer and compute the final lighting.

To enable this data-driven Geometry Pass, the connection between the JSON definition and the Vulkan pipeline is established during the initialization phase (`localInit`). The application explicitly creates the `P_GBuffer` pipeline and the `RP_GBuffer` render pass. These objects are then wrapped in a `TechniqueRef` structure and passed to the `Scene::init` method.

Listing 3.4: Deferred Pipeline Injection into the Scene Loader

```
// Mapping the C++ Pipeline to the JSON string "Deferred"
PRs[0].init("Deferred", {
    {&P_GBuffer, { /* Descriptor Set Layouts configuration */ }}
}, 1, &VDMesh);

// The Scene object will now use P_GBuffer for any instance
// marked with "technique": "Deferred" in the JSON.
SC.init(this, 1, VDRs, PRs, "assets/models/scene.json");
```

This approach allows the `Scene` to handle the complexity of mesh binding and command buffer recording, while the application retains control over the specific shading technique used.

3.5.3. Implementation Details

The `populateCommandBuffer` method orchestrates the synchronization between the two render passes. First, the command buffer begins the `RP_GBuffer` render pass by delegating the scene drawing to the `SC.populateCommandBuffer` method. During this phase, the `Scene` object binds the `P_GBuffer` pipeline and iterates through all loaded instances, writing their attributes into the G-Buffer attachments (Position, Normal, Albedo).

Once the Geometry Pass is concluded, the application begins the `RP_Lighting` render pass to perform the image composition. This stage involves binding the `P_Lighting` pipeline along with the global lighting data (`DSGlobalLighting`) and the G-Buffer input textures (`DSLightingInput`). The lighting calculation is then triggered by an explicit draw call on a full-screen quad (`MQuad`), which samples the previously generated buffers

to produce the final result.

The collision logic is handled within the `updateUniformBuffer` method, which effectively acts as the application's "Game Loop". Since the `Scene` module centrally manages collider memory, the application can iterate through the technique instances list (`SC.TI`) to update and check the status of all collidable entities.

Listing 3.5: Synchronization, Detection and Feedback Loop

```
// Iterate through all techniques and instances
for (techniqueId = 0; techniqueId < SC.TechniqueInstanceCount; techniqueId++) {
    for (instanceId = 0; instanceId < SC.TI[techniqueId].InstanceCount; instanceId++) {
        if (SC.TI[techniqueId].I[instanceId].C != nullptr) {
            Collider* currentCollider = SC.TI[techniqueId].I[instanceId].C;

            // Update Collider position to match the visual mesh
            currentCollider->setWorldMatrix(SC.TI[techniqueId].I[instanceId].Wm);

            // Check Collision against the Player
            if (currentCollider != playerCollider) {
                if (playerCollider->collidesWith(*currentCollider)) {
                    isColliding = true;
                    // Visual Feedback: Turn Wireframes Red
                    SC.setColliderStroke(playerCollider, glm::vec4(1, 0, 0, 1));
                    SC.setColliderStroke(currentCollider, glm::vec4(1, 0, 0, 1));
                } else {
                    // No Collision: Reset to Green
                    SC.setColliderStroke(currentCollider, glm::vec4(0, 1, 0, 1));
                }
            }
        }
        // No Collision: Reset to Green (Player)
        if (!isColliding) {
            SC.setColliderStroke(playerCollider, glm::vec4(0, 1, 0, 1));
        }
    }
}
```

This code snippet illustrates the real-time interaction loop, which orchestrates synchronization, detection, and feedback.

First, the `setWorldMatrix` method is invoked for every active instance, updating the collider with the current World Matrix of the visual mesh. Although in this specific example only the Player is animated (via `GameLogic`), this design ensures that any dynamic object in the scene (such as moving obstacles) would automatically maintain perfect alignment between its visual and physical representations.

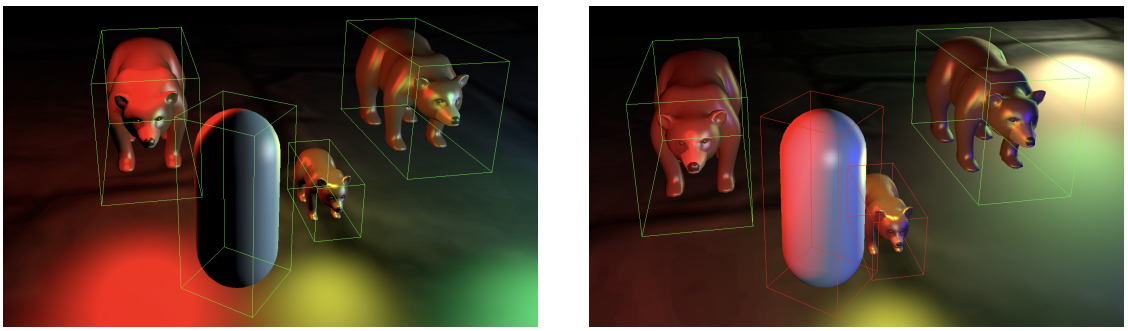
Subsequently, the collision test is executed via the `collidesWith` method. Relying on the double-dispatch mechanism implemented in the Colliders module, this call dynamically resolves to the specific algorithm required for the interacting shapes (employing the

Separating Axis Theorem (SAT) to handle the OOBB-OOBB intersections).

Finally, the loop concludes by providing visual feedback, switching the debug wireframes' color from green to red whenever an overlap is detected.

3.5.4. Results

The resulting application successfully demonstrates the synergy between all implemented features. The scene is correctly illuminated by multiple dynamic point lights without performance degradation, validating the Deferred Shading technique. Simultaneously, as the user navigates the environment, the `OOBB` wireframes update smoothly. Upon contact with an obstacle, the real-time visual feedback confirms the accuracy of the intersection detection (as shown in Figure 3.7).



(a) Before collision (green wireframes) (b) During collision (red wireframes)

Figure 3.7: The Complete Example running.

This seamless integration serves as a comprehensive validation of the proposed architecture. By combining the multi-pass rendering pipeline with the dynamic collision system, this final application confirms that the framework has evolved into a robust engine, fully satisfying the primary objectives outlined at the beginning of this work.

4 | Conclusions and Future Work

4.1. Conclusions

This thesis presented the design and implementation of significant extensions to an educational Vulkan framework, with the primary objective of bridging the gap between basic API initialization and advanced real-time graphics techniques. Starting from an initial architecture that was structurally limited to forward rendering and included an experimental collision module, this work has successfully transformed the project into a robust, multi-feature platform, enriched by well-commented sample applications designed to facilitate the learning process.

The contributions of this thesis can be summarized in three key areas.

Advanced Rendering Architecture. The restrictions of the original single-pass pipeline were overcome by implementing a complete Deferred Shading system. By extending the core `Starter.hpp` module to support Multiple Render Targets (MRT) and subpass dependencies, the framework is now capable of generating a G-Buffer and decoupling geometry processing from lighting. This allows students to experiment with complex lighting scenarios that were previously computationally prohibitive.

Collision System Stabilization. The `Colliders.hpp` module was evolved from a prototype into a reliable collision system. The algorithmic correction of the intersection logic has resolved previous instabilities, ensuring consistent collision detection. Furthermore, the introduction of a real-time Visual Debugger addresses a critical pedagogical need, allowing students to visually inspect bounding volumes and diagnose spatial interactions immediately.

Data-Driven Collision Integration. A crucial achievement was the incorporation of the collision module into the `Scene` management system. By extending the JSON parsing logic, spatial attributes are now recognized as intrinsic properties of the scene entities, coexisting with their graphical definitions. This approach effectively decouples the scene configuration from the C++ application code, enabling a streamlined workflow

where collision parameters and mesh instances are defined contextually within the same configuration file.

The validity of these implementations was demonstrated through the development of the *Complete Example* application. This final demo serves as concrete proof that the framework can now support complex scenarios requiring the simultaneous execution of a multi-pass rendering pipeline and a robust collision detection system.

In conclusion, the framework has evolved from a basic initialization wrapper into a comprehensive didactic tool, providing future Computer Graphics courses at Politecnico di Milano with a solid foundation for exploring advanced rendering techniques and real-time interactive applications.

4.2. Future Work

While the current implementation successfully achieves the primary objectives, there is still room for functional expansion and improvement; in this regard, the following opportunities have been identified.

From an architectural perspective, future developments should aim to fully integrate the Deferred Shading technique within the `Scene` management system. Currently, while geometry instancing is data-driven, the high-level rendering flow (specifically the Lighting Pass) remains hardcoded in the C++ application. Abstracting these definitions into the scene configuration file would decouple the rendering logic from the application code, streamlining the creation of complex custom pipelines.

On the rendering front, the Deferred Shading implementation establishes a solid foundation for several sophisticated optimizations. First, as anticipated in Section 3.2.3, the material model could be enhanced by implementing *channel packing*: utilizing the currently unused alpha channel of the Albedo attachment to store per-pixel specular intensity would improve visual fidelity without increasing memory bandwidth. Furthermore, the Lighting Pass could be optimized using *Light Volumes*, rendering spherical geometry for point lights to process pixels only within their effective radius. Additionally, visual depth could be significantly increased by integrating Screen Space Ambient Occlusion (SSAO) on top of the existing G-Buffer.

Finally, the collision module presents substantial opportunities for expansion. While the current detection system is robust for basic primitives, extending support to cylinders, capsules (essential for character controllers), and convex hulls would greatly widen the variety of possible simulations. Moreover, because the current system is limited to spatial

intersection, the implementation of rigid body dynamics (specifically impulse-based resolution) would effectively equip the framework with a complete physics engine capable of simulating mass, forces, bouncing, and friction.

Bibliography

[1] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*. A K Peters/CRC Press, 2018.

[2] J. Gregory, *Game Engine Architecture, Third Edition*. CRC Press, 2018.

[3] The Khronos Group Inc., *Vulkan 1.3 - A Specification (with all registered extensions)*, 2022.

[4] M. Bailey, "Vulkan: All the Things!," in *SIGGRAPH '16: ACM SIGGRAPH 2016 Courses*, (New York, NY, USA), Association for Computing Machinery, 2016.

[5] J. Unterguggenberger, B. Kerbl, and M. Wimmer, "Vulkan all the way: Transitioning to a modern low-level graphics API in academia," *Computers & Graphics*, vol. 111, pp. 155–165, 2023.

[6] M. Fink, T. Hollander, T. Hädrich, and S. Müller, "VkCV: A Vulkan Convenience Framework for Research and Education," in *The 26th International Conference on 3D Web Technology (Web3D '21)*, (New York, NY, USA), Association for Computing Machinery, 2021.

[7] S. Willems, "Vulkan C++ Examples and Demos." `https://github.com/SaschaWillems/Vulkan`, 2016.

[8] J. de Vries, "LearnOpenGL - Advanced Lighting: Deferred Shading." `https://learnopengl.com/Advanced-Lighting/Deferred-Shading`, 2014.

[9] C. Ericson, *Real-Time Collision Detection*. The Morgan Kaufmann Series in Interactive 3D Technology, Morgan Kaufmann, 2004.

A | Mathematical Background

A.1. The Separating Axis Theorem (SAT)

For robust detection of collisions between Oriented Bounding Boxes (OOBBs), the collision detection system relies on the *Separating Axis Theorem (SAT)*, a fundamental principle in computational geometry described extensively by Ericson [9].

To optimize the evaluation of the test between two arbitrary oriented boxes, a coordinate space transformation is typically applied. By transforming one OOBB into the local coordinate system of the other (using an inverse transformation matrix), the reference OOBB effectively becomes an Axis-Aligned Bounding Box (AABB) relative to the first. This mathematical reduction simplifies the calculation of the cross products and projections, aligning one set of axes with the canonical basis vectors, while strictly preserving the geometric validity of the theorem.

For completeness, the mathematical formulation of the SAT algorithm is described below.

Theorem Definition

The theorem states that two convex polyhedra A and B do not overlap if there exists at least one axis onto which the projections of the two shapes do not overlap. This axis is called a *separating axis*. If no such axis exists, the objects are intersecting.

For two OOBBs in 3D space, there are 15 potential separating axes that must be tested to guarantee a correct result:

- The 3 local axes of OOBB A ($\mathbf{u}_A, \mathbf{v}_A, \mathbf{w}_A$).
- The 3 local axes of OOBB B ($\mathbf{u}_B, \mathbf{v}_B, \mathbf{w}_B$).
- The 9 axes formed by the cross products of the edges from A and B (e.g., $\mathbf{u}_A \times \mathbf{u}_B, \mathbf{u}_A \times \mathbf{v}_B, \dots$).

Intersection Test

Mathematically, the intersection test determines whether the projections of the two bounding volumes onto a potential separating axis overlap.

For a given axis **L**, the 8 vertices of each box are projected onto **L** using the dot product to compute their minimum and maximum extents along that axis. Let $[min_A, max_A]$ and $[min_B, max_B]$ be the resulting projection intervals for boxes A and B, respectively.

The intervals overlap if and only if:

$$max_A \geq min_B \quad \wedge \quad min_A \leq max_B \tag{A.1}$$

According to the Separating Axis Theorem, if this overlap condition fails (i.e., the intervals are disjoint) for even a single axis out of the 15 possibilities, that vector is identified as a *separating axis*, mathematically proving that the 3D objects do not intersect. Conversely, if the overlap condition holds true for *all* 15 axes, the convex volumes intersect.

B | Implementation Details

B.1. G-Buffer Configuration Snippet

The following code snippet demonstrates the Vulkan configuration used to initialize the G-Buffer attachments in the `Starter.hpp` module. It details the selection of high-precision floating-point formats for geometric data and the explicit layout transitions required to prepare the buffers for sampling during the subsequent Lighting Pass.

Listing B.1: G-Buffer Attachment Configuration

```
static std::vector <AttachmentProperties> GBuffer = {
    // Position Attachment (RGBA16F)
    { COLOR_AT, VK_FORMAT_R16G16B16A16_SFLOAT,
        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT
            ,
        VK_IMAGE_ASPECT_COLOR_BIT, false, false,
        { .color = {.float32 = {0.0f,0.0f,0.0f,1.0f}} },
        VK_SAMPLE_COUNT_1_BIT,
        VK_ATTACHMENT_LOAD_OP_CLEAR,
        VK_ATTACHMENT_STORE_OP_STORE,
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,
        VK_ATTACHMENT_STORE_OP_DONT_CARE,
        VK_IMAGE_LAYOUT_UNDEFINED,
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL },

    // Normal Attachment (RGBA16F)
    { COLOR_AT, VK_FORMAT_R16G16B16A16_SFLOAT,
        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT
            ,
        VK_IMAGE_ASPECT_COLOR_BIT, false, false,
        { .color = {.float32 = {0.0f,0.0f,0.0f,1.0f}} },
        VK_SAMPLE_COUNT_1_BIT,
        VK_ATTACHMENT_LOAD_OP_CLEAR,
        VK_ATTACHMENT_STORE_OP_STORE,
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,
```

```
        VK_ATTACHMENT_STORE_OP_DONT_CARE,
        VK_IMAGE_LAYOUT_UNDEFINED,
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL },

    // Albedo Attachment (RGBA8)
    { COLOR_AT, VK_FORMAT_R8G8B8A8_UNORM,
        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT
            ,
        VK_IMAGE_ASPECT_COLOR_BIT, false, false,
        { .color = {.float32 = {0.0f,0.0f,0.0f,1.0f}} },
        VK_SAMPLE_COUNT_1_BIT,
        VK_ATTACHMENT_LOAD_OP_CLEAR,
        VK_ATTACHMENT_STORE_OP_STORE,
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,
        VK_ATTACHMENT_STORE_OP_DONT_CARE,
        VK_IMAGE_LAYOUT_UNDEFINED,
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL },

    // Depth Attachment (Dynamic Format)
    { DEPTH_AT, BP->findDepthFormat(),
        VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT |
            VK_IMAGE_USAGE_SAMPLED_BIT,
        VK_IMAGE_ASPECT_DEPTH_BIT, false, false,
        { .depthStencil = {1.0f,0} },
        VK_SAMPLE_COUNT_1_BIT,
        VK_ATTACHMENT_LOAD_OP_CLEAR,
        VK_ATTACHMENT_STORE_OP_DONT_CARE,
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,
        VK_ATTACHMENT_STORE_OP_DONT_CARE,
        VK_IMAGE_LAYOUT_UNDEFINED,
        VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL,
        VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL }
};
```

C | Visualization Algorithms

This appendix contains the core C++ algorithms used to generate the wireframe geometry for the Visual Debugger described in Section 3.3.1.

C.1. Geometry Dispatching

The `MakeBoundingGeometry` function acts as the central dispatcher, selecting the correct visualization method and handling the recursive traversal of BVH structures.

Listing C.1: Dispatcher function

```
void MakeBoundingGeometry(Collider *c, ColliderShowVertex *&V_vertex, int &ib) {
    switch(c->type) {
       case CLD_AABB:
          MakeBox(c, V_vertex, ib, true);
          break;
       case CLD_OOBB:
          MakeBox(c, V_vertex, ib, false);
          break;
       case CLD_SPHERE:
          MakeSphere(c, V_vertex, ib);
          break;
       case CLD_POINT:
          MakePoint(c, V_vertex, ib);
          break;
       case CLD_BVH:
          // Draw the current BVH node's volume (always an AABB)
          MakeBox(c, V_vertex, ib, true);

          // Recursively draw the child nodes
          for (auto child : c->children) {
             MakeBoundingGeometry(child, V_vertex, ib);
          }
          break;
       default:
          MakeBox(c, V_vertex, ib, false);
          break;
    }
}
```

C.2. Box Generation (AABB and OOBB)

The `MakeBox` function generates wireframe geometry for both Axis-Aligned and Oriented Bounding Boxes. If an AABB is requested, it calculates a strictly axis-aligned volume that encloses the oriented points.

Listing C.2: Algorithm for generating Box wireframes

```
void MakeBox(Collider *c, ColliderShowVertex *&V_vertex, int &ib, bool isAxisAligned) {
    // Compute the 8 world space vertices of the box (AABB or OOBB)
    EightPoints P8;
    c->transformAABB(P8, *c);

    // If we want an AABB, recompute the min/max extents
    if (isAxisAligned) {
       // Compute the AABB extents
       AABBextents E;
       c->getExtentsAABB(P8, E);

       // Overwrite P8 with axis-aligned corners
       P8.P[0] = {E.xMin, E.yMin, E.zMin};
       P8.P[1] = {E.xMin, E.yMin, E.zMax};
       P8.P[2] = {E.xMin, E.yMax, E.zMin};
       P8.P[3] = {E.xMin, E.yMax, E.zMax};
       P8.P[4] = {E.xMax, E.yMin, E.zMin};
       P8.P[5] = {E.xMax, E.yMin, E.zMax};
       P8.P[6] = {E.xMax, E.yMax, E.zMin};
       P8.P[7] = {E.xMax, E.yMax, E.zMax};
    }

    // Write the 8 vertices into the vertex buffer
    ColliderShowVertex* baseV = V_vertex;
    for (int i = 0; i < 8; ++i)
       (V_vertex++)->pos = P8.P[i];

    // Retrieve the base vertex index inside the model buffer
    int v0 = int((uint8_t*)baseV - (uint8_t*)&M->vertices[0]) / VD.Bindings[0].stride;

    // Helper lambda that writes two indices (one line)
    auto &I = M->indices;
    auto addEdge = [&](int a, int b){ I[ib++] = a; I[ib++] = b; };

    // Add the 12 edges (line list): 24 indices
    // Bottom Edges
    addEdge(v0+0, v0+1);
    addEdge(v0+0, v0+2);
    addEdge(v0+3, v0+1);
    addEdge(v0+3, v0+2);

    // Top Edges
    addEdge(v0+4, v0+5);
    addEdge(v0+4, v0+6);
    addEdge(v0+7, v0+5);
    addEdge(v0+7, v0+6);
```

```
    // Vertical Edges
    addEdge(v0+0, v0+4);
    addEdge(v0+1, v0+5);
    addEdge(v0+2, v0+6);
    addEdge(v0+3, v0+7);
}
```

C.3. Sphere Generation

The `MakeSphere` function approximates the spherical volume using three orthogonal circles aligned with the XY, XZ, and YZ planes.

Listing C.3: Algorithm for generating Sphere wireframes

```
void MakeSphere(Collider *c, ColliderShowVertex *&V_vertex, int &ib) {
    // Compute sphere center in world space using the model matrix
    glm::vec3 center = glm::vec3(c->Wm * glm::vec4(c->x1, c->y1, c->z1, 1.0f));

    // Compute the radius after world-scale transformation
    float r = c->getTransformedRadius(*c);

    // Calculate base index of this sphere in the vertex buffer
    int baseIndex = (uint8_t*)V_vertex - (uint8_t*)&M->vertices[0];
    baseIndex /= VD.Bindings[0].stride;

    // Helper lambda to draw one circle defined by two perpendicular axes
    auto addCircle = [&](glm::vec3 ax1, glm::vec3 ax2){
       for(int i = 0; i < SPHERE_RESOLUTION; i++){
          float a0 = float(i) / SPHERE_RESOLUTION * 6.2831853f;
          float a1 = float(i + 1) / SPHERE_RESOLUTION * 6.2831853f;

          // Parametric circle equation: P = C + r*(ax1*cos(theta) + ax2*sin(theta))
          glm::vec3 p0 = center + r * (ax1 * cos(a0) + ax2 * sin(a0));
          glm::vec3 p1 = center + r * (ax1 * cos(a1) + ax2 * sin(a1));

          (V_vertex++)->pos = p0;
          (V_vertex++)->pos = p1;

          // Populate the index buffer (Line List)
          M->indices[ib++] = baseIndex + (i * 2);
          M->indices[ib++] = baseIndex + (i * 2 + 1);
       }

       // Update base index for the next circle to avoid overlapping indices
       baseIndex += SPHERE_RESOLUTION * 2;
    };

    // Construct the sphere wireframe using three major circles
    addCircle({1,0,0}, {0,1,0}); // XY Plane
    addCircle({1,0,0}, {0,0,1}); // XZ Plane
    addCircle({0,1,0}, {0,0,1}); // YZ Plane
}
```

C.4. Point Generation

The `MakePoint` function visualizes a point as a 3D crosshair consisting of three short orthogonal line segments.

Listing C.4: Algorithm for generating Point wireframes

```
void MakePoint(Collider *c, ColliderShowVertex *&V_vertex, int &ib) {
    // Transform the point position (local to world space)
    glm::vec3 P = glm::vec3(c->Wm * glm::vec4(c->x1, c->y1, c->z1, 1.0f));

    // Define the half-length of the axis crosshair
    const float L = 0.1f;

    // Define the endpoints for the 3 orthogonal lines (X, Y, Z axes)
    glm::vec3 pts[] = {
       P + glm::vec3(L, 0, 0), P - glm::vec3(L, 0, 0), // X axis segment
       P + glm::vec3(0, L, 0), P - glm::vec3(0, L, 0), // Y axis segment
       P + glm::vec3(0, 0, L), P - glm::vec3(0, 0, L)  // Z axis segment
    };

    // Calculate the base index in the global model vertex buffer
    int base = ((uint8_t*)V_vertex - (uint8_t*)&M->vertices[0]) / VD.Bindings[0].stride;

    // Write the 6 vertices into the vertex buffer
    for(int i = 0; i < 6; i++) {
       (V_vertex++)->pos = pts[i];
    }

    // Define the 3 line segments in the index buffer (6 indices total)
    M->indices[ib++] = base + 0; M->indices[ib++] = base + 1;
    M->indices[ib++] = base + 2; M->indices[ib++] = base + 3;
    M->indices[ib++] = base + 4; M->indices[ib++] = base + 5;
}
```

List of Figures

Acknowledgements

To my family, thank you for your unconditional support and for always believing in me. I dedicate a special thought to my grandmother, Anna, whose comforting meals and our shared moments playing cards provided a much-needed haven during these years.