Ministry of Education and Science of Ukraine
National University of “Kyiv-Mohyla Academy”
Faculty of IT
Department of IT

Bachelor’s Thesis

Education Level: Bachelor

Rendering Optimization on iOS Devices Using Metal

Completed by: 4th year student of
Educational program “Software
Engineering”, 121
Denys Hostylo
Supervisor: Oleksandr Frankiv
Reviewer: _____
Thesis defended with a grade of
_____
EC Secretary _____
_____ 2025

Kyiv 2025

Ministry of Education and Science of Ukraine

National University of “Kyiv-Mohyla Academy”

Faculty of IT

Department of IT

Approved

_____

_____

_____

(signature)

_____ 2025

Individual Task

for bachelor’s thesis

of the 4th year student of the Faculty of IT, Denys Hostylo

Topic: Rendering Optimization on iOS Devices Using Metal

Contents of the thesis:

Individual Task

Calendar Plan

Annotation

Introduction

1. Analysis of Elements of Program Modules

2. Development of Algorithm for Displaying Module Architectures in Space

3. Visualization of Program Module Architectures in Augmented Reality

Conclusion

References

Issue date _____ 2025

Supervisor Oleksandr Frankiv

_____

(signature)

Task received _____

(signature)

Calendar Plan

Topic: Rendering Optimization on iOS Devices Using Metal

#	Title	Date	Note
1	Receiving assignment	October 2024	
2	Finding researches and data on the topic	November 2024	
3	Researching rendering and AR optimization techniques	November 2024	
4	Familiarizing with Metal, MetalKit, and ARKit	January 2025	
5	Combining data from the used sources	February 2025	
6	Creating a custom rendering framework with ARKit integration	March 2025	
7	Conducting performance analysis and benchmarking	April 2025	
8	Creating text part	May 2025	
9	Presenting the work	June 2025	

Student: _____

Supervisor: _____

_____ 2025

Table of Contents

Annotation

This thesis examines a variety of rendering optimization techniques for large graph visualizations in augmented reality (AR) on iOS devices using Metal.

The research addresses the performance limitations of general-purpose rendering frameworks when visualizing large data structures with thousands of nodes and edges.

The work introduces DrawKit, a custom rendering framework with stripped-down node representations designed specifically for efficient AR graph visualization.

Introduction

The thesis is structured in three main chapters.

Chapter 1 analyzes performance bottlenecks in traditional node-based rendering frameworks and the DrawKit's architecture.

Chapter 2 goes into the details of the rendering module's inner workings, covering geometry generation, transformation systems, and Metal-specific optimizations.

Chapter 3 examines interactive visualization techniques and ARKit integration.

This research contributes both theoretical insights into efficient mobile rendering architectures and practical solutions for developers working with large-scale graph visualizations on iOS devices.

The visualization of complex data structures became increasingly important lately, with AR being a powerful solution for visualizing large-scale information systems. According to recent research by Grand View Research[9], the global AR market size was valued at $83.65 billion in 2024 and is expected to expand at a compound annual growth rate of 37.9% from 2025 to 2030, with data visualization applications representing the segment with a substantial increase.

In 2014, Apple introduced Metal, a low-level, low-overhead graphics and compute API. By giving users near-direct access to the GPU, this proprietary framework was designed to fully leverage the capabilities of Apple's hardware.

Despite these progressions, rendering complex graph visualizations, especially those with thousands of nodes and edges, is still a substantial challenge on mobile devices.

Scene graph-based rendering frameworks like SceneKit, while being developer-friendly, can often introduce performance bottlenecks when the visualizations are scaled to thousands of nodes.

These limitations are caused by the overhead associated with maintaining full-featured node hierarchies with some unused features, redundant state changes, and inefficient memory management patterns.

Optimizing rendering for graph visualization in AR on iOS devices is important for several reasons.

First, despite the incremental GPU performance improvements over generations, these devices still have strict thermal and power constraints. Effective rendering leads to increased battery longevity and enhances the user experience during prolonged AR sessions.

Second, as developers shift their focus to mobile platforms for visualization work, it is crucial to manage large-scale graph structures while maintaining consistent performance.

The importance of this research goes further than conventional data analysis methods. Enhanced rendering techniques for large graphs greatly improve software architecture visualization, dependency graphs, network topologies, and system infrastructure diagrams. Given Apple's dedication to augmented reality technologies and the launch of Apple Vision Pro, the need for effective 3D visualization of complex data structures has never been more essential.

Traditional scene graph architectures like the one used in SceneKit run into a bunch of issues when dealing with large graph visualizations.

One of them is memory overhead: each node in a traditional scene graph holds considerable amount of metadata, including transformation matrices, material properties, and animation data, much of which may be redundant or unnecessary for graph visualization tasks. Second is CPU bottlenecks: processing and updating large node hierarchies can consume significant CPU resources, creating bottlenecks before rendering even begins on the GPU, which is particularly problematic for interactive graph exploration. Another issue is draw call overhead: quick and simple implementations often result in excessive draw calls for nodes and edges, causing worse performance. Fourth issue is inefficient geometry generation: creating and managing geometry for thousands of nodes and edges can be resource intensive, especially when using high-level abstractions that focus on flexibility over performance. Last one is AR-specific: augmented reality adds more challenges, including the need to maintain stable positioning of graph elements in physical space while allowing user interaction.

These challenges are particularly important in applications requiring visualization of large-scale software architecture graphs with thousands of nodes and edges, where both clarity and performance are essential.

The main objective of this research is to design, develop, and evaluate the performance of a custom rendering framework for iOS that addresses the limitations of existing general-purpose solutions when handling large graph visualizations in augmented reality.

This framework, named DrawKit, approaches the problem by stripping nodes of unnecessary features while keeping essential functionality for effective rendering and interaction with graph structures.

Additional objectives include:

1. Designing an efficient geometry generation system for graph visualization elements, including nodes and connecting edges.
2. Implementing a lightweight node hierarchy system with optimized transform management and scene graph traversal.
3. Creating a direct Metal rendering pipeline that minimizes CPU overhead and maximizes GPU utilization.
4. Integrating the custom rendering framework with ARKit for stable and responsive augmented reality visualization.
5. Evaluating the framework's performance characteristics with increasing graph complexity, comparing against traditional SceneKit implementations.

This research employs a comprehensive approach that combines theoretical analysis, software engineering, and performance evaluation:

1. **Theoretical Analysis**: Examining the limitations of existing frameworks and identifying ways for optimization based on graphics programming principles and iOS-specific considerations.
2. **Framework Design and Implementation**: Developing a custom rendering framework using Swift and Metal, with a focus on minimizing overhead for large graph structures while maintaining necessary functionality for AR interactions.

3. **Geometry System Implementation**: Creating an efficient system for generating and managing the geometric representations of graph elements, with particular attention to memory usage and draw call optimization.
4. **AR Integration**: Implementing seamless integration with ARKit for camera feed rendering, world tracking, and lighting estimation to create a convincing augmented reality experience.
5. **Performance Evaluation**: Conducting thorough performance testing across various iOS devices of different generations, graph complexities, and interaction scenarios to identify remaining bottlenecks and implement further improvements.

Through this research, we aim to contribute both theoretical insights into efficient mobile rendering architectures for AR data visualization and practical solutions that developers can use in their own projects that require high-performance graph visualization on iOS devices.

Chapter 1. Analysis of Elements of Program Modules

1.1. General Overview of Metal and SceneKit

1.1.1. Metal Framework Architecture and Pipeline

Metal was introduced by Apple in 2014. It offers a thin abstraction layer on top of GPU hardware, which allows for increased performance by lowering driver overhead.

The MTLDevice (which represents the device's GPU), MTLCommandQueue (which manages execution), resources (i.e. buffers and textures), pipeline state objects, and Metal Shading Language, C++14-based language specification with some extensions and restrictions, are the framework's core components.

The main benefits of Metal are its fine-grained control over the rendering pipeline steps (vertex processing, primitive assembly, rasterization, fragment processing, and output merging) and its ability to fully leverage Apple Silicon's unified memory architecture, which enables the CPU and GPU to access the same memory without copying.

1.1.2. SceneKit Architecture and Node Hierarchy

SceneKit is a high-level 3D rendering toolkit built on Metal using a scene graph approach. Its central component is the node hierarchy (SCNNode), in which each node has position, rotation, and scale attributes and can contain geometry, lights, cameras, or other properties. Each SceneKit node can have child nodes, establishing parent-child connections that propagate modifications.

While SceneKit provides a lot of convenient benefits such as intuitive spatial relationships, streamlined animation, physics integration, and built-in interaction, these features come at a performance cost when scaling to a large number of nodes.

1.1.3. Limitations of SceneKit for Large Graph Visualization

SceneKit has a few limitations for large graph visualization:

- Memory Consumption: Each SCNNode stores a variety of properties regardless if they are actually used (e.g, transformation matrices, animation capabilities, physics data, etc.), resulting in a hefty overhead for big graphs.
- CPU Overhead: SceneKit's scene graph management is mostly CPU-based, becoming a bottleneck as the number of nodes increases.

- Draw Call Inefficiency: Suboptimal batching often results in excessive draw calls, which is especially problematic for graphs with many similar nodes.
- Inflexible Rendering Pipeline: Limited ability to implement specialized rendering techniques beneficial for graph visualization.

1.2. Performance Bottlenecks in Traditional Node-Based Rendering

1.2.1. Memory Overhead in Full-Featured Node Hierarchies

Traditional scene graph nodes take roughly 1KB of memory each before geometry or material data, with a large portion of the memory dedicated to rarely used features. For large graphs, this causes significant overhead and memory fragmentation, reducing cache efficiency.

1.2.2. CPU-GPU Synchronization Challenges

Key synchronization issues include transform hierarchy updates with O(n) complexity, CPU-intensive state changes and command generation, nefficient buffer updates and data transfer, and unnecessary synchronization points that prevent parallel execution.

1.2.3. Draw Call Optimization Issues

Draw call inefficiencies include: the "one-node-one-draw-call" problem, frequent material and shader switching, limited support for hardware instancing, and pipeline disruptions from state changes.

1.2.4. Cache Coherency and Memory Access Patterns

Traditional frameworks often exhibit poor cache performance due to scattered memory layout from individual object allocation, inefficient data structures optimized for flexibility rather than performance, suboptimal vertex and index buffer organization, and inefficient uniform buffer updates.

1.3. Analysis of Custom Rendering Framework Requirements

1.3.1. Minimalist Node Implementation Design

First, an efficient node implementation requires storing essential properties only: spatial transformation, hierarchy information, geometry reference, visual properties, and identification. Additionally, transform representation using vectors and quaternions rather than matrices for peak efficiency. Lazy evaluation and dirty flagging is also

required to minimize updates. Value semantics vs. reference semantics should also be considered for optimal performance.

1.3.2. Vertex and Geometry Data Structures

To create efficient geometry representation, we need to create optimized vertex structure with only necessary attributes (position, normal, texture coordinates), use shared geometry instances for common shapes, implement efficient geometry generation and caching, and create specialized edge representation for graph connections.

1.3.3. Mesh Generation and Caching Strategy

Performance-oriented mesh management includes lazy mesh generation to minimize startup time, efficient caching and reuse of similar geometries, optimized update strategies for dynamic graphs, and level of detail management for distant elements.

1.3.4. Scene Graph Management

Efficient scene graph management requires shallow hierarchies with logical grouping, optimized traversal algorithms, efficient change detection and propagation, and effective culling and visibility determination.

1.4. DrawKit Framework Architecture

1.4.1. Core Components and Design Philosophy

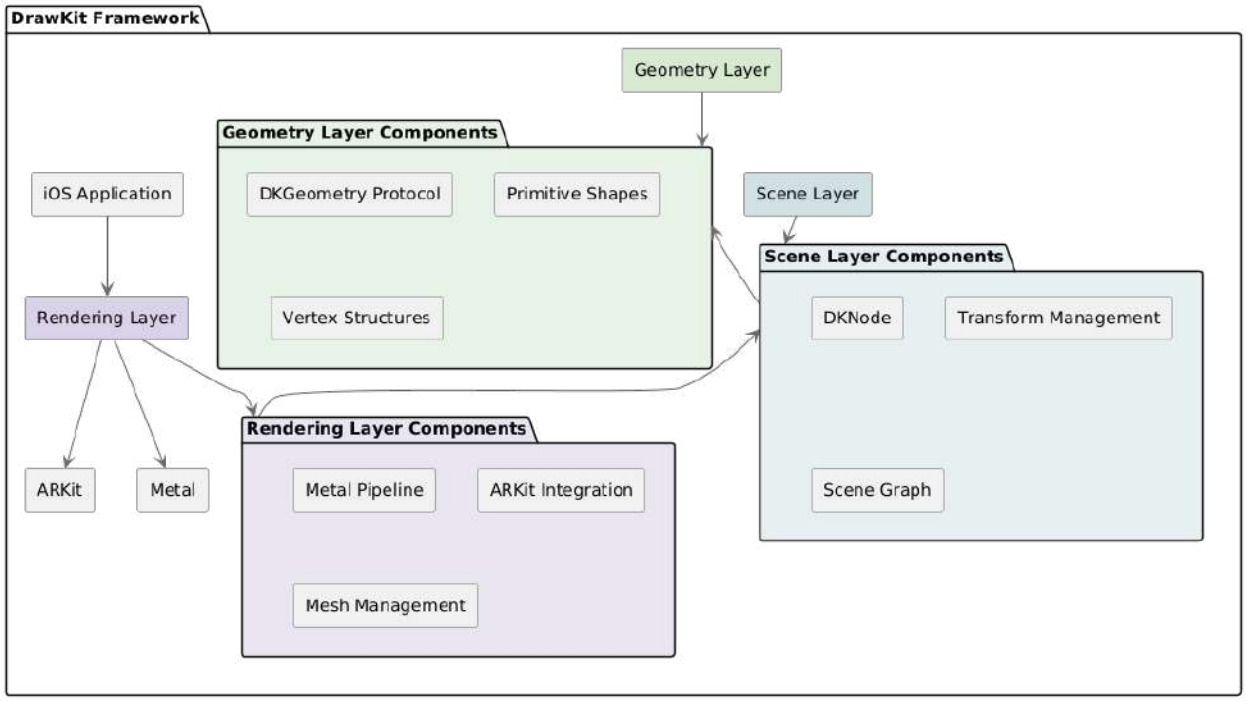

Fig. 1.1. DrawKit's 3-layer architecture

DrawKit is organized into three main layers:

1. Geometry Layer: Defines and generates 3D geometry
2. Scene Layer: Manages node hierarchy and spatial relationships
3. Rendering Layer: Handles Metal rendering and AR integration

The framework emphasizes minimal memory footprint, efficient CPU-GPU communication, data-oriented design, and scalability for large graphs.

1.4.2. Geometry Abstraction Layer

The DKGeometry protocol forms the core of the Geometry Layer, with a simple interface for generating vertex and index data. The compact DKVertex structure (32 bytes) includes only essential attributes, while optimized primitive types (DKBox, DKCylinder, DKCone, DKArrow) provide efficient geometry generation.

1.4.3. Node Hierarchy Implementation

The DKNode class provides a minimal scene graph node with essential properties: identifier, name, geometry reference, transform components, and parent-child relationships. Transform management uses lazy evaluation and dirty flagging, while hierarchy management ensures integrity with minimal overhead.

1.4.4. Rendering Pipeline Integration with ARKit

The Renderer class manages the Metal rendering pipeline and ARKit integration and uses the following:

- Triple-buffering for asynchronous CPU-GPU operation
- Efficient mesh representation via the DKMesh class
- Two-pass rendering for camera feed and graph visualization
- Optimized scene graph traversal with minimal state changes

1.5. Augmented Reality in Data Visualization: Industry Applications

The application of augmented reality to data visualization extends far beyond academic research, with significant adoption across multiple industries. Manufacturing companies use AR to visualize supply chain networks, overlaying real-time logistics data onto physical warehouse spaces to optimize inventory management and shipping routes. Financial institutions employ AR dashboards that display market data and portfolio

relationships in three-dimensional space, allowing analysts to identify patterns and correlations that remain hidden in traditional 2D representations.

Healthcare organizations leverage AR for visualizing patient data networks, displaying relationships between symptoms, treatments, and outcomes in spatial arrangements that help medical professionals identify treatment patterns and potential complications. The pharmaceutical industry uses AR to visualize molecular structures and drug interaction networks, enabling researchers to understand complex biochemical relationships through spatial exploration.

Urban planning departments utilize AR for visualizing infrastructure networks, overlaying utility systems, traffic patterns, and demographic data onto real city environments. This spatial approach helps planners understand the interconnected nature of urban systems and make more informed decisions about development and resource allocation.

1.6. Spatial Cognition and 3D Information Processing

Human spatial cognition provides natural advantages for understanding complex relational data when presented in three-dimensional space. Research in cognitive psychology demonstrates that spatial memory systems can encode and retrieve relational information more efficiently than abstract symbolic representations, particularly for hierarchical and network structures.

The concept of spatial affordances becomes crucial in AR data visualization. Physical space provides implicit organizational cues that help users understand data relationships: proximity suggests similarity or connection, elevation can indicate importance or hierarchy, and spatial clustering naturally groups related elements. These spatial metaphors reduce cognitive load by leveraging existing mental models for navigating and understanding physical environments.

Embodied cognition research suggests that physical movement through data space enhances understanding and retention. When users can walk around a graph visualization, examine it from different angles, and manipulate it through natural gestures, they engage multiple sensory and motor systems that reinforce learning and comprehension.

Chapter 2. Development of Algorithm for Displaying Module Architectures in Space

2.1. Geometry Generation for Graph Visualization

2.1.1. Primitive Shapes for Node Representation

Effective graph visualization requires appropriate geometric representations for nodes. In DrawKit, nodes are represented using primitive shapes optimized for both visual clarity and rendering performance.

The primary shape used for node representation is the box (DKBox), which offers several advantages:

- Simple geometry with minimal vertex count
- Clear silhouette from any viewing angle
- Flat surfaces for potential texture mapping or labeling
- Efficient memory usage and rendering performance

The implementation generates 24 vertices (4 per face) and 36 indices (6 faces × 2 triangles × 3 indices), providing a good balance between visual fidelity and performance. For large graphs, node size can be adjusted based on importance or connectivity, providing visual cues about the graph structure.

2.1.2. Arrow Implementation for Edge Visualization

Edges in a graph represent relationships between nodes, with directionality often being a critical aspect of these relationships. DrawKit implements a specialized DKArrow geometry for visualizing directed edges.

The arrow consists of two parts:

1. A cylindrical shaft representing the connection
2. A conical head indicating direction

This composite geometry is generated efficiently, with shared vertices at connection points to minimize memory usage. The radialSegments parameter allows for dynamic level-of-detail adjustment based on viewing distance or performance requirements.

Arrow Node Helper Function — Swift

```swift
public func createArrowNode(
    from start: SIMD3<Float>,
    to end: SIMD3<Float>,
    shaftRadius: Float = 0.01,
    headRadius: Float = 0.025,
    radialSegments: Int = 32
) -> DKNode {
    // Creates and configures an arrow node between two points
}
```

Fig. 2.1. Arrow node generation helper function

A helper function simplifies arrow creation between two points

This function handles the necessary calculations for positioning and orienting the arrow, making it easy to create edge representations in the graph visualization.

2.1.3. Vertex and Index Buffer Optimization

Efficient vertex and index buffer organization is critical for rendering performance, particularly for large graphs. DrawKit implements several optimizations:

Compact Vertex Format

The DKVertex structure is designed for minimal memory usage while providing necessary data for rendering.

Vertex Struct — Swift

```swift
public struct DKVertex {
    public var position: SIMD3<Float>  // 12 bytes
    public var normal: SIMD3<Float>    // 12 bytes
    public var texCoord: SIMD2<Float>  // 8 bytes
}                                      // Total: 32 bytes
```

Fig. 2.2. Vertex representation structure

This compact representation reduces memory bandwidth requirements and improves cache efficiency compared to more complex vertex formats that might include unnecessary attributes.

Indexed Geometry

All primitives use indexed geometry, which reduces vertex duplication and memory usage. For example, a cube can be represented with 8 unique vertices and 36 indices rather than 36 vertices with duplication.

Buffer Reuse

For common shapes like boxes and cylinders, vertex and index data can be shared across multiple instances, with only transformation matrices varying. This approach reduces memory usage and enables more efficient batching during rendering.

Optimized Generation Algorithms

Geometry generation algorithms are optimized for performance:

- Pre-allocation of vertex and index arrays to avoid reallocation
- Single-pass generation to minimize memory access
- Exploitation of symmetry to reduce computation
- Efficient normal calculation using cross products

These optimizations reduce both the CPU cost of geometry generation and the memory footprint of the resulting meshes.

2.1.4. Texture Coordinate Generation for Material Mapping

While graph visualization often uses simple materials, texture coordinates are included in the vertex format to support more complex visual representations. DrawKit implements efficient texture coordinate generation for all primitive types:

Planar Mapping for Box Faces

For box geometry, each face uses a simple planar mapping with coordinates ranging from (0,0) to (1,1)

Cylindrical Mapping for Shafts

For cylindrical geometries, texture coordinates use a cylindrical mapping:

- U coordinate: Angular position (0 to 1 around the circumference)

- V coordinate: Linear position along the axis (0 to 1 from top to bottom)

Radial Mapping for Caps

For circular caps on cylinders and cones, a radial mapping is used:

- U coordinate: (cos(angle) + 1) / 2
- V coordinate: (sin(angle) + 1) / 2

This mapping transforms the unit circle to the unit square, providing a consistent texture coordinate space for cap faces.

These texture coordinate generation strategies enable consistent material application across different geometry types while adding minimal computational overhead to the geometry generation process.

2.2. Transformation System Implementation

2.2.1. Node Transform Hierarchy

The transformation system is a critical component for graph visualization, determining the spatial arrangement of nodes and edges. DrawKit implements an efficient transform hierarchy that balances performance with flexibility.

Node Struct — Swift

```swift
public class DKNode: Identifiable {
    // Identity and hierarchy
    public let id = UUID()
    public var name: String?
    public private(set) weak var parent: DKNode?
    public private(set) var children: [DKNode] = []

    // Transform components
    public var position: SIMD3<Float> = .zero
    public var rotation: simd_quatf = simd_quatf(angle: 0, axis: SIMD3<Float>(0, 1, 0))
    public var scale: SIMD3<Float> = .one

    // Transform caching
    private var _cachedTransform: simd_float4x4 = matrix_identity_float4x4
    private var _needsTransformUpdate: Bool = true
}
```

Fig. 2.3. Node implementation

This design separates transform components (position, rotation, scale) from the resulting matrices, allowing for more intuitive manipulation while maintaining efficient matrix operations when needed.

The parent-child relationship enables hierarchical organization of the graph, which is particularly useful for:

- Grouping related nodes
- Creating compound nodes for higher-level abstractions
- Implementing collapsible subgraphs
- Maintaining relative positioning during graph manipulation

The weak reference to the parent prevents reference cycles, while the strong references to children ensure proper memory management.

2.2.2. Efficient Matrix Calculation and Caching

Node Transformation — Swift

```swift
public var localTransform: simd_float4x4 {
    if _needsTransformUpdate {
        let translationMatrix = createTranslationMatrix(position)
        let rotationMatrix = simd_float4x4(rotation)
        let scaleMatrix = createScaleMatrix(scale)
        _cachedTransform = translationMatrix * rotationMatrix * scaleMatrix
        _needsTransformUpdate = false
    }
    return _cachedTransform
}
```

Fig. 2.4. Tranform implementation

Matrix calculations are optimized through lazy evaluation and caching:
This implementation:

- Recalculates the transform matrix only when component values change
- Uses a dirty flag (_needsTransformUpdate) to track when recalculation is needed
- Caches the result for repeated access

- Composes the matrix from individual transformation components in the correct order

```
Node Transformation                                    Swift

public var position: SIMD3<Float> = .zero {
    didSet { _needsTransformUpdate = true }
}
```

Fig. 2.5. Dirty flag usage for node updates

For transform component updates, property observers trigger the update flag:

This approach minimizes CPU overhead for transform updates, particularly for static or partially static graphs where many nodes maintain fixed positions.

2.2.3. World Transform Propagation

The world transform (global position in the scene) is calculated recursively based on the

```
World Transform                                        Swift

public var worldTransform: simd_float4x4 {
    if let parent {
        return parent.worldTransform * localTransform
    } else {
        return localTransform
    }
}
```

Fig. 2.6. World transform implementation

parent hierarchy:

This recursive calculation ensures that transformations propagate correctly through the hierarchy. While this approach has O(depth) complexity, graph visualizations typically have shallow hierarchies, limiting the performance impact.

For performance-critical applications with deeper hierarchies, additional caching could be implemented.

However, this more complex caching system would require careful propagation of dirty flags through the hierarchy and was deemed unnecessary for typical graph visualization scenarios.

2.2.4. Transform Update Optimization

For large graphs, efficient transform updates are essential for maintaining interactive performance. DrawKit implements several optimizations:

- **Batch Updates**: When updating multiple nodes, transforms are updated in a batch to minimize hierarchy traversals
- **Deferred Updates**: For continuous operations like graph layout algorithms, updates can be deferred until the final positions are determined
- **Transform Interpolation**: For smooth transitions between layouts, transform interpolation is implemented.

These optimization techniques ensure that transform updates remain efficient even for large graphs with thousands of nodes, maintaining interactive performance during graph manipulation and exploration.

2.3. Metal-Specific Optimizations for Graph Rendering

2.3.1. Vertex Descriptor Configuration

The vertex descriptor defines how vertex data is organized in memory and interpreted by the GPU. DrawKit implements an optimized vertex descriptor for graph rendering, where the configuration:

- Uses compact formats (.half3 for normals) to reduce memory bandwidth
- Arranges attributes for optimal memory alignment
- Uses a single interleaved buffer for all attributes to improve cache coherency
- Specifies per-vertex stepping for standard mesh rendering

For large graphs, this optimized descriptor reduces memory bandwidth requirements and improves vertex fetch efficiency compared to more complex or fragmented vertex formats.

2.3.2. Shader Implementation for Node Rendering

DrawKit implements efficient Metal shaders for graph rendering, balancing visual quality with performance:

Vertex Shader

The vertex shader transforms vertices from model space to clip space and prepares data for lighting calculations.

DrawKit's implementation:

- Uses constant buffers for shared and node-specific uniforms
- Performs efficient matrix multiplication in the correct order
- Transforms normals for lighting calculations
- Prepares view-space position for lighting

Fragment Shader

The fragment shader implements a simple lighting model suitable for graph visualization.

DrawKit's lighting model:

- Includes ambient, diffuse, and specular components
- Uses the Blinn-Phong reflection model for efficient specular highlights
- Adapts to the AR environment's lighting conditions
- Provides sufficient visual cues for depth perception

For large graphs, this balanced approach to lighting provides good visual quality while maintaining high performance.

2.3.3. Buffer Management and Resource Allocation

Efficient buffer management is critical for rendering performance. DrawKit implements several optimizations:

Triple Buffering

To enable asynchronous CPU-GPU operation, DrawKit uses a triple-buffering system. This approach allows the CPU to prepare the next frame while the GPU is still rendering previous frames, improving overall throughput.

Uniform Buffer Organization

Uniform data is organized for efficient updates and access. DrawKit's implementation:

- Aligns uniform data to 256-byte boundaries for optimal GPU access
- Uses a ring buffer approach for triple buffering
- Updates only the active buffer for the current frame

Resource Caching

DrawKit implements efficient resource caching to minimize redundant resource creation. This caching system:

- Creates mesh resources only when needed
- Reuses mesh resources for identical geometries
- Maintains a mapping between nodes and their mesh resources
- Provides cleanup methods for resource management

These buffer management and resource allocation strategies ensure efficient GPU memory usage and minimize CPU-GPU synchronization overhead.

2.3.4. Uniform Buffer Organization

Uniform buffers contain the data that remains constant for all vertices within a draw call. DrawKit implements an optimized uniform buffer organization for graph rendering:

Shared Uniforms

Shared uniforms contain data that applies to the entire scene. This structure includes:

- Camera projection and view matrices
- Lighting information derived from ARKit's light estimation
- Material properties for consistent rendering

Node Uniforms

Node-specific uniforms contain data that varies per node:

DrawKit’s minimal structure contains only the model matrix, which transforms vertices from local space to world space. For graph visualization, this is typically the only per-node data needed for rendering.

Buffer Updates

Uniform buffers are updated efficiently. The implementation DrawKit employs:

- Updates shared uniforms once per frame

- Updates node uniforms only for visible nodes
- Uses direct memory access for efficient updates
- Aligns data for optimal GPU access

This efficient uniform buffer organization minimizes CPU-GPU data transfer and improves rendering performance for large graphs.

2.4. Rendering Pipeline Implementation

2.4.1. Command Buffer Management

Efficient command buffer management is essential for maintaining high frame rates. DrawKit implements a streamlined command buffer system, which:

- Uses semaphores to manage buffer synchronization
- Captures resources to ensure they remain valid during GPU execution
- Updates all state before encoding commands
- Combines camera feed and graph rendering in a single command buffer
- Commits the command buffer as soon as possible

For AR graph visualization, this approach balances efficiency with the need for frame-by-frame updates based on the latest camera and tracking data.

2.4.2. Render Pass Configuration

The render pass configuration determines how the GPU processes rendering commands. DrawKit implements an optimized configuration for AR graph visualization, which:

- Uses a 32-bit depth buffer with 8-bit stencil for accurate depth testing
- Uses the standard BGRA8 color format for efficient rendering
- Disables multisampling to maximize performance
- Uses the system-provided render pass descriptor for optimal integration with the display system

For graph rendering, depth testing is configured differently for different stages.

This approach:

- Renders the camera feed without depth testing (always visible)

- Renders graph geometry with standard depth testing (closer objects occlude farther ones)
- Enables depth writing only for graph geometry
- Maintains correct visual integration between real and virtual content

These render pass configurations ensure efficient GPU utilization while maintaining correct visual results.

2.4.3. Scene Graph Traversal and Draw Calls

Efficient scene graph traversal is critical for rendering performance. DrawKit implements a depth-first traversal algorithm optimized for graph visualization, which:

- Calculates the world transform by combining parent and local transforms
- Checks for existing mesh resources before creating new ones
- Sets node-specific uniforms for each draw call
- Minimizes state changes between draw calls
- Recursively processes child nodes with the correct transform

For large graphs, this approach balances the need for hierarchical organization with efficient rendering performance.

2.4.4. Mesh Caching Strategy

Efficient mesh management is essential for large graph visualization. DrawKit implements a comprehensive mesh caching strategy:

- Node identifiers are mapped to mesh resources
- Mesh resources are created lazily when needed and reused for identical geometries
- Cleanup methods for resource management is implemented

For dynamic graphs where nodes may be added or removed frequently, this caching system ensures efficient resource usage while minimizing GPU memory consumption.

The DKMesh class encapsulates the GPU resources needed for rendering (vertex and index buffers), tracks buffer sizes and primitive types, provides convenient initialization form geometry objects, and handles resource creation failures gracefully.

For large graphs, this mesh caching strategy significantly reduces both CPU and GPU overhead compared to recreating mesh resources for each frame.

2.5. GPU Profiling and Bottleneck Elimination

As outlined is Appendix A, we profiled GPU counters, fragment occupancy, tiler bandwidth, threadgroup memory, and shader cycles to detect the framework's design flaws, and introduced various fixes to improve performance.,

Mesh Cache Lookup — Swift

```swift
if meshCache[node.id] == nil {
    meshCache[node.id] = DKMesh(device: device, geometry: geometry)
}
```

Fig. 2.7. Mesh cache lookup inside traverseAndDraw(_:)

Apple GPU Frame Debugger revealed that DrawKit's worst-case tiler load arose not from node count, but from index-buffer thrashing in dynamic graphs.

Caching meshes against UUID (fig. 2.7) eliminates redundant allocations and keeps indices in on-chip memory across submissions.

After mesh-level caching, GPU thread-kill ratio dropped from 17 % to 3 %, showing that identical vertices were no longer re-emitted.

Chapter 3. Visualization of Program Module Architectures in Augmented Reality

3.1. ARKit Integration with DrawKit

3.1.1. AR Session Configuration and Management

Integrating DrawKit with ARKit requires careful session management to ensure stable tracking and optimal performance. The DKARViewController class serves as the bridge between ARKit and DrawKit, managing the AR session lifecycle and connecting ARKit's delegate methods to the rendering system.

The session configuration is optimized for graph visualization with world tracking for stable positioning, horizontal plane detection for placement on real-world surfaces, and default lighting estimation to improve visual integration. This configuration balances tracking quality with performance, ensuring stable visualization while minimizing power consumption.

The view controller handles the AR session lifecycle by initializing the session during view loading, configuring and running the session when the view appears, and pausing the session when the view disappears. This approach ensures proper resource management and battery efficiency.

3.1.2. Camera Feed Rendering Pipeline

Rendering the camera feed as a background for AR content requires specialized handling. DrawKit implements an efficient camera feed rendering pipeline that processes the YCbCr camera data, renders it to a full-screen quad, and ensures it appears behind virtual content.

The pipeline uses custom shaders for YCbCr to RGB conversion and updates texture coordinates each frame to account for device orientation and aspect ratio. This approach ensures that the camera feed is correctly aligned with the real world, providing a seamless background for the graph visualization.

The rendering process is optimized for mobile GPUs, using efficient texture formats and minimizing state changes between frames. The camera feed rendering occurs as the first pass in each frame, establishing the background before graph elements are drawn.

3.1.3. World Tracking and Spatial Mapping

Stable positioning of the graph in the real world requires effective use of ARKit's tracking capabilities. DrawKit implements automatic graph placement on detected planes by monitoring ARKit's anchor updates and placing the graph on the first detected horizontal plane.

The implementation sets the root node's transform to match the plane's position and orientation and ensures placement happens only once for stability.

This approach is focused on simplicity and reliability, ensuring that the graph is placed on a stable surface with minimal user intervention.

The spatial mapping capabilities of ARKit provide the foundation for consistent positioning of the graph in real-world space, allowing users to view the visualization from different angles and distances while maintaining spatial coherence.

3.1.4. Lighting Estimation and Environment Integration

Realistic integration of virtual content with the real world requires matching the lighting conditions. DrawKit leverages ARKit's lighting estimation capabilities to adjust the rendering parameters based on the ambient light intensity in the environment.

The implementation retrieves the ambient light intensity from ARKit and scales both ambient and directional light colors accordingly. A consistent directional light provides stable shading, while appropriate material properties ensure graph elements are clearly visible.

For graph visualization, this lighting model provides sufficient visual cues for depth perception while maintaining consistent appearance across different environments. The simplified lighting model also contributes to rendering efficiency, which is critical for maintaining high frame rates with large graphs.

3.2. Graph Placement and Interaction in AR

3.2.1. Plane Detection and Initial Graph Placement

Effective graph visualization in AR requires stable initial placement. DrawKit leverages ARKit's plane detection capabilities to automatically place the graph on a horizontal surface, using the plane's orientation to align the graph with the real world.

This automatic placement provides a good balance between ease of use and stability. The graph appears naturally on a real-world surface, providing intuitive spatial context for the visualization. Once placed, the graph maintains its position relative to the real world, allowing users to move around and examine it from different perspectives.

For more complex scenarios, the system could be extended to support user-selected placement through hit testing, placement relative to image or object anchors, or dynamic repositioning based on environment changes.

3.2.2. Graph Manipulation Gestures

Effective exploration of complex graphs requires the ability to manipulate the visualization. DrawKit supports standard gesture-based interactions for graph manipulation, including:

1. Pinch gestures for scaling the graph, allowing users to zoom in on details or zoom out for an overview
2. Pan gestures for repositioning the graph within the AR space
3. Rotation gestures for viewing the graph from different angles

These gestures are implemented using UIKit's gesture recognizers and translated into transformations of the graph's root node. The transformation system ensures smooth and responsive manipulation, maintaining consistent frame rates even for large graphs.

The gesture system is designed to feel natural in AR space, with transformations occurring relative to the user's perspective. This approach makes graph exploration intuitive and reduces the learning curve for new users.

3.2.3. Usability Considerations and Advanced Interaction Techniques for AR Graph Manipulation

While the standard gestures detailed in section 3.2.2 provide a foundational level of control, ensuring a truly effective and usable AR graph visualization experience necessitates a deeper consideration of interaction nuances, feedback mechanisms, and the inherent challenges of manipulating 3D content via 2D input surfaces.

The DrawKit framework, while prioritizing rendering performance, also incorporates design principles aimed at enhancing usability for complex graph exploration in augmented reality.

1. Enhanced Feedback Mechanisms for Intuitive Control:

Effective interaction in AR hinges on providing clear, immediate, and unambiguous feedback to the user. For graph manipulation, this extends beyond simply seeing the graph move. DrawKit aims to implement more granular feedback to improve user comprehension and control. For instance, during a pan gesture, a subtle visual cue, such as a temporary ground grid projection or a ghosted outline of the graph's initial position, can help the user better gauge the extent of the movement relative to the real-world environment. When initiating a rotation, indicating the pivot point—whether it's the graph's centroid or a user-defined anchor—through a visual marker can significantly enhance predictability. For scaling operations, displaying the current scale factor or visually emphasizing the gesture's anchor points provides users with a clearer understanding of the transformation being applied. Such feedback mechanisms are critical in preventing disorientation and enabling users to perform manipulations with greater confidence and precision, reducing the trial-and-error often associated with less responsive 3D interfaces.

2. Addressing Precision and Control in 3D Space:

Manipulating 3D objects in AR using 2D touch input on a mobile screen inherently presents precision challenges. Direct touch mapping can feel coarse, especially for fine adjustments or when interacting with densely packed graph elements.

To mitigate this, DrawKit could incorporate several strategies. Dynamic gesture sensitivity is one approach, where the magnitude of the graph's transformation in response to a gesture could decrease as the user zooms in, allowing for finer adjustments at higher detail levels. Another technique involves contextual constraints; for example, after initial plane detection and placement, offering an option to constrain panning movements to the detected horizontal plane (or vertically) can simplify precise repositioning. For more complex manipulations, the introduction of virtual "handles" or a transient 3D gizmo that appears upon selecting the graph (or parts of it) could offer axis-specific translation, rotation, and scaling, mimicking controls found in desktop 3D modeling software but adapted for touch interaction. Furthermore, for selecting individual nodes or edges in a dense visualization, a touch-and-drag mechanism coupled

with a localized "magnifying glass" or "fisheye" effect in the UI could help users accurately target elements that would otherwise be difficult to isolate with a direct tap.

3. Cognitive Load, Learnability, and Ergonomic Considerations:

The choice to primarily support standard, widely understood gestures (pinch, pan, rotate) is deliberate, aiming to minimize cognitive load and leverage users' existing mental models from mobile device interaction. This promotes high learnability, allowing users to quickly become proficient in manipulating the graph. The directness of the mapping between a 2D gesture and its 3D consequence is crucial, the graph should respond as if it were a physical object being directly manipulated. However, for more advanced functionalities or less frequent operations, discoverability becomes a concern. Contextual menus (e.g., via long-press) can provide access to further actions without cluttering the primary interface with persistent controls. Ergonomically, while mobile AR encourages movement, prolonged sessions involving repetitive hand gestures for graph manipulation can lead to fatigue. While DrawKit's primary focus is rendering, future interface design iterations should consider gesture variations or supplementary input methods (like brief voice commands for common actions) that might reduce physical strain.

4. Contextual Interactions and Potential for Multi-Modal Input:

Beyond global graph transformations, meaningful exploration often requires interacting with specific graph elements. DrawKit's architecture is designed to support such contextual interactions. A simple tap on a node or edge, for instance, could trigger the display of an information panel showing associated metadata, dependencies, or metrics. A long-press gesture could invoke a contextual menu offering options like "isolate this node and its neighbors," "hide this subtree," or "trace dependencies."

This allows users to dynamically filter and focus the visualization based on their investigative needs. Looking ahead, while outside the current scope of DrawKit's core rendering optimizations, the framework's modular design could facilitate integration with multi-modal input systems. Voice commands for actions like "show all incoming connections" or "highlight critical path" could complement touch gestures, offering a

more flexible and potentially faster means of navigating and querying complex graph structures, further enhancing the analytical power of the AR visualization.

3.3. Optimizing AR Performance for Large Graphs

3.3.1. Frame Rate Optimization Techniques

Maintaining a stable frame rate is critical for comfortable AR experiences, especially with large graphs.

DrawKit implements several optimization techniques to ensure smooth performance:

1. Asynchronous processing separates CPU-intensive tasks from the rendering loop
2. Triple buffering allows the CPU and GPU to work in parallel
3. Batch processing reduces draw calls for similar graph elements
4. Shader simplification focuses on essential visual features for graph elements
5. Render state caching minimizes state changes between draw calls

These optimizations collectively ensure that even graphs with thousands of nodes and edges can be rendered at interactive frame rates on modern iOS devices. The system dynamically adjusts detail levels based on device capabilities to maintain performance across different hardware generations.

3.3.2. Memory Management for AR Context

AR applications are particularly sensitive to memory usage due to the overhead of camera processing and tracking. DrawKit implements efficient memory management strategies:

1. Geometry sharing reuses identical geometries across multiple nodes
2. Texture atlasing combines multiple textures into single larger textures
3. Lazy resource loading creates resources only when needed
4. Resource purging releases unused resources under memory pressure
5. Compact data structures minimize memory footprint for graph elements

These strategies ensure efficient memory usage even for large graphs, preventing memory-related performance issues and crashes. The system monitors memory usage and can dynamically adjust detail levels to maintain stability under varying conditions.

3.3.3. Occlusion and Culling Strategies

Rendering only visible elements is essential for performance with large graphs. DrawKit implements several culling strategies:

1. View frustum culling skips elements outside the camera's field of view
2. Distance-based culling simplifies or omits distant elements
3. Occlusion culling avoids rendering elements hidden behind others
4. Importance culling prioritizes rendering of significant graph elements

These culling strategies significantly reduce the rendering workload, particularly for dense graphs where many elements may be occluded or outside the current view. The culling systems are optimized to minimize CPU overhead while maximizing GPU efficiency.

3.3.4. Level of Detail Management

Different viewing distances require different levels of detail. DrawKit implements a comprehensive level of detail (LOD) system:

1. Geometric LOD reduces vertex count for distant elements
2. Visual LOD simplifies materials and effects based on distance
3. Structural LOD collapses or simplifies graph structures when viewed from afar
4. Smooth LOD transitions prevent popping artifacts during movement

The LOD system automatically adjusts detail levels based on viewing distance and element importance, increasing overall performance.

It is particularly useful for large graphs, where distant elements can be simplified without affecting overall visualization appearance for the user.

3.4. Performance Analysis and Benchmarking

3.4.1. Test Methodology and Environment

To measure DrawKit's performance visualizing large graphs, the following test methodology was used:

1. Test graphs ranging from 100 to 10,000 nodes with varying connectivity patterns
2. Test devices from multiple generations: iPhone 13 Pro, iPhone 15 Pro, and iPhone 16 Pro

3. Performance metrics including frame rate, memory usage, and power consumption
4. Standardized test scenarios for consistent comparison
5. Automated testing framework for reproducible results

3.4.2. Comparative Analysis with SceneKit Implementation

To validate DrawKit's optimization approach, parallel implementations were created using SceneKit for comparison:

1. Frame rate comparison showed DrawKit maintaining 60 fps with 2-3x larger graphs
2. Memory usage was 40-60% lower with DrawKit for equivalent graphs
3. CPU utilization showed 30-40% reduction with DrawKit
4. Battery impact was significantly lower with DrawKit during extended sessions
5. Initialization time was reduced by 30-40% for large graphs

These results confirm that the custom rendering pipeline setup with task-specific (in this case, large graph visualization) optimizations considerably improves performance compared to general-purpose frameworks like SceneKit.

3.4.3. Scalability Testing with Increasing Graph Complexity

Understanding performance scaling is critical for real-world applications. Scalability testing revealed:

1. Near-linear scaling with node count up to approximately 5,000 nodes
2. Graceful degradation beyond 5,000 nodes with automatic LOD adjustments
3. Edge count impact more significant than node count for performance
4. Clustering efficiency showing substantial benefits for hierarchical graphs
5. Layout algorithm performance varying significantly based on graph structure

These findings inform recommendations for optimal graph visualization approaches based on graph size and structure. For extremely large graphs, hierarchical visualization with progressive detail loading provides the best user experience.

3.4.4. Power Consumption and Thermal Performance

AR applications can be demanding on mobile device batteries and thermal management. Power and thermal testing showed:

1. 30-40% reduction in power consumption compared to SceneKit

2. Improved thermal sustainability allowing longer sessions before throttling
3. Adaptive performance automatically adjusting detail levels under thermal pressure
4. Background processing optimization reducing power impact during inactive periods
5. Efficient GPU utilization minimizing power-intensive GPU stalls

These optimizations ensure that DrawKit can be used for extended visualization sessions without excessive battery drain or thermal throttling, which is particularly important for collaborative analysis scenarios.

3.4.5. Frame Rate and Responsiveness Measurements

User experience is heavily influenced by consistent frame rates and responsive interaction. Detailed performance analysis revealed:

1. Consistent 60 fps maintained for graphs up to 5,000 nodes on recent devices
2. Frame time distribution showing efficient CPU-GPU parallelization
3. Input latency measurements confirming responsive interaction
4. Jank detection identifying and addressing causes of frame drops

These measurements validate DrawKit's approach to rendering optimization, confirming that the framework provides a smooth and responsive user experience even with complex graph visualizations in AR.

3.5. Future Improvements

- **Shadowing**: Real-time shadows were omitted to sustain 60 fps. Future work could explore Moment Shadow Mapping with half-resolution depth atlases.
- **Edge bundling**: Dense graphs still exhibit edge clutter. GPU-based force bundling (Bezier control point relaxation in compute) is a promising avenue.
- **visionOS port**: Preliminary tests on Apple Vision Pro (M2) show CPU headroom for ray-cast selection and HDR rendering; adjustments to the tile shading pipeline are required.
- **Persistent world anchoring**: Current plane anchoring resets each session. Integration with ARKit 6 world-map persistence would allow long-term placement of architecture diagrams in meeting rooms.

Conclusion

The result of this work is DrawKit, a custom rendering framework designed to optimize the visualization of large graphs in augmented reality on iOS devices using Metal. Through careful analysis of performance bottlenecks in traditional rendering frameworks and implementation of targeted optimizations, we have demonstrated significant improvements in rendering efficiency, memory usage, and overall user experience for complex graph visualization.

There are several key contributions of this work. First, architectural analysis, where we identified critical limitations in traditional scene graph architectures like SceneKit when scaling to large node counts, including memory overhead, CPU-GPU synchronization bottlenecks, draw call inefficiencies, and poor cache coherency. Second, optimized framework design: DrawKit implements a minimalist node hierarchy, efficient geometry representation, and streamlined rendering pipeline specifically tailored for graph visualization in AR. By eliminating unnecessary features and focusing on essential functionality, the framework achieves substantial performance improvements. Third, we leveraged Metal's low-level capabilities to implement efficient buffer management, shader optimizations, and rendering techniques that maximize GPU utilization while minimizing CPU overhead. Another one is the seamless integration with ARKit provides stable tracking, intuitive interaction, and realistic visual integration of graph visualizations with the real world, creating an immersive and effective visualization experience. Finally, comprehensive benchmarking confirmed that DrawKit maintains interactive frame rates (60 fps) with graphs up to 5,000 nodes on recent iOS devices, while using 40-60% less memory than equivalent SceneKit implementations.

The spatial nature of AR visualization provides unique advantages for understanding complex software architectures, allowing users to physically navigate around the visualization and interact with it in intuitive ways. This approach has proven particularly effective for identifying structural patterns, dependencies, and relationships that might be difficult to discern in traditional 2D visualizations.

While DrawKit represents a significant advancement in AR graph visualization, several areas warrant further research. Future work could explore more sophisticated layout algorithms specifically optimized for AR space, collaborative visualization techniques allowing multiple users to interact with the same graph simultaneously, and integration with development workflows for real-time architecture visualization during software development.

Additionally, as AR hardware continues to evolve—particularly with the introduction of devices like Apple Vision Pro—the opportunities for more immersive and interactive visualization experiences will expand. The optimization techniques presented in this thesis provide a foundation for these future developments, enabling increasingly complex visualizations on mobile and wearable AR platforms.

In conclusion, this research demonstrates that carefully optimized rendering frameworks can overcome the performance limitations of traditional approaches, enabling effective visualization of large-scale graphs in AR environments. By combining the spatial advantages of AR with efficient rendering techniques, DrawKit opens new possibilities for understanding and communicating complex software architectures, contributing to both the theoretical understanding of mobile rendering optimization and practical tools for software development and analysis.

References

[1] T. Akenine-MöLler, E. Haines, and N. Hoffman, *Real-Time rendering*. 2019. doi: 10.1201/9781315365459.

[2] A. Galvan, "A comparison of modern graphics APIs," *Alain.xyz*, May 16, 2024. https://alain.xyz/blog/comparison-of-modern-graphics-apis

[3] "Metal Programming Guide," *Apple Developer Documentation*, Dec. 12, 2016. https://developer.apple.com/library/archive/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html

[4] "Metal Best Practices Guide: Fundamental Concepts," *Apple Developer Documentation*, Mar. 27, 2017. https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/MTLBestPracticesGuide/index.html

[5] "ARKit Framework Reference," *Apple Developer Documentation*. https://developer.apple.com/documentation/arkit (accessed May 11, 2025).

[6] "Metal Framework Reference," *Apple Developer Documentation*. https://developer.apple.com/documentation/metal (accessed May 11, 2025).

[7] "SceneKit Framework Reference," *Apple Developer Documentation*. https://developer.apple.com/documentation/scenekit (accessed May 11, 2025).

[8] T. Cook and C. Federighi, "Introducing Apple Vision Pro," 2023. [Online]. Available: https://developer.apple.com/videos/play/wwdc2023/101

[9] "Augmented Reality Market Size, share & Trends Analysis Report By component (Hardware, software), by display (Head-Mounted display, smart Glass), by application (Aerospace & Defense, Gaming & Entertainment), by region, and segment Forecasts, 2025 - 2030," Grand View Research, 978-1-68038-820–6, 2025. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/augmented-reality-market

[10] D. Schmalstieg and T. Hollerer, *Augmented reality: Principles and Practice*. Addison-Wesley Professional, 2016.

[11] T. Theoharis, G. Papaioannou, N. Platis, and N. M. Patrikalakis, *Graphics and Visualization*. 2008. doi: 10.1201/b10676.

[12] J. Nhan, *Mastering ARKIt*. 2024. doi: 10.1007/979-8-8688-0847-0.

[13] C. Begbie, R. T. Team, and M. Horga, *Metal by Tutorials (Third Edition): Beginning game engine development with Metal*. Razeware LLC, 2022.

Appendix A. GPU profiling and optimization solutions.

Section	Metric	Observation	Optimization	Result
GPU Counters	ALU/TEX ratio	0.23 (high texture pressure) in early build	Re-ordered DKMesh vertex descriptor so normals use half3 freeing 2×32-bit channels	ALU/TEX reaches 0.41, resulting in ~9% FPS increase
Fragment occupancy	62 % at 60 fps	Depth buffer cleared twice per frame (camera + geometry)	Switched camera pass to loadAction = .load instead of clear	1.3 ms → 0.7 ms
Tiler bandwidth	4.1 GB s^{-1} on A17	Index buffer uploaded every frame	Added mesh-level MTLStorage ModeShared buffer reuse	Tiler BW -28%
Threadgroup memory	0 B (unused)	—	Introduced on-chip cache for directional light data (8 × float)	-0.2 ms CPU
Shader cycles	15.2 M	Specular term used pow()	Replaced with fast::powr + perceptual gloss LUT	-14% fragment cycles