



---

**Konzeption, Implementierung und Evaluation  
eines Virtual-Tabletop-Plugins für Obsidian.md**

Fabian Donatus Wolfgang Urbanek

Matrikelnummer: 667074

---

**Abschlussarbeit**

zur Erlangung des akademischen Grades

**Bachelor of Science (B.Sc.)**

im Studiengang Bachelor Wirtschaftsinformatik

der [FOM Hochschule für Oekonomie & Management](#)

Fabian Donatus Wolfgang Urbanek  
Hinter der Höh 20  
34630 Gilserberg

vorgelegt bei

Prof. Dr. Claudius Stern

Gilserberg, 01.01.2026

## **Zusammenfassung**

Hier steht eine kurze Inhaltszusammenfassung von etwa einer Seite. Es soll der komplette Inhalt zusammengefasst werden, also insbesondere können auch bereits Ergebnisse genannt werden.

## **Vorwort**

Das in dieser Arbeit gewählte generische Maskulinum bezieht sich zugleich auf die männliche, die weibliche und andere Geschlechteridentitäten. Zur besseren Lesbarkeit wird auf die gleichzeitige Verwendung männlicher und weiblicher Sprachformen verzichtet. Alle Geschlechteridentitäten werden ausdrücklich mitgemeint, soweit die Aussagen dies erfordern.

# Inhaltsverzeichnis

<b>Vorwort</b>	<b>iii</b>
<b>Abbildungsverzeichnis</b>	<b>vi</b>
<b>Tabellenverzeichnis</b>	<b>vii</b>
<b>Abkürzungsverzeichnis</b>	<b>viii</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Hinführung zum Thema und Motivation . . . . .	1
1.2 Darstellung der Problemstellung . . . . .	2
1.3 Zielsetzung und Forschungsfrage . . . . .	2
1.4 Aufbau der Arbeit . . . . .	3
<b>2 Theoretische Grundlagen</b>	<b>4</b>
2.1 Konzeptuelle Grundlagen . . . . .	4
2.1.1 Virtual Tabletop Tools (VTTs) . . . . .	4
2.1.2 Plugin-Architekturen . . . . .	10
2.2 Technische Rahmenbedingungen . . . . .	10
2.2.1 Obsidian als Markdown-Editor . . . . .	10
2.2.2 Electron Framework . . . . .	12
2.2.3 PIXI.js v8 als Rendering-Engine . . . . .	12
<b>3 Konzeption und Implementierung</b>	<b>15</b>
3.1 Anforderungsanalyse . . . . .	15
3.1.1 Funktionale Anforderungen . . . . .	15
3.1.2 Nicht-funktionale Anforderungen . . . . .	18
3.2 Systemdesign . . . . .	20
3.2.1 Architekturentwurf . . . . .	20
3.2.2 Datenmodell . . . . .	25
3.3 Entwicklung verschiedener Lösungsansätze . . . . .	33
3.3.1 Rendering Engine: PIXI.js Evaluation und Implementierung . . . . .	33
3.3.2 Grid System: Performance-Optimierung durch Draw Call Reduktion .	36

3.3.3 Token Management: Performance-Optimierung bei vielen Objekten . . . . .	39
<b>4 Evaluation und Ergebnisse</b>	<b>44</b>
4.1 Durchführung der Performance-Messungen . . . . .	44
4.1.1 Testumgebung und -bedingungen . . . . .	44
4.1.2 Messmethodik . . . . .	46
4.2 Auswertung und Interpretation der Daten . . . . .	48
4.2.1 Benchmark-Ergebnisse . . . . .	48
4.2.2 Identifizierter Memory Leak . . . . .	49
4.2.3 Skalierungsverhalten . . . . .	52
4.3 Diskussion der Ergebnisse . . . . .	53
4.3.1 Interpretation der Messergebnisse . . . . .	53
4.3.2 Zusammenfassung der Ergebnisse . . . . .	54
4.3.3 Limitationen der Evaluation . . . . .	54
<b>5 Fazit und Ausblick</b>	<b>56</b>
5.1 Zusammenfassung der wesentlichen Erkenntnisse . . . . .	56
5.2 Beantwortung der Forschungsfrage . . . . .	57
5.3 Limitationen und kritische Reflexion . . . . .	58
5.3.1 Methodische Einschränkungen . . . . .	58
5.3.2 Technische Limitationen . . . . .	59
5.3.3 Kritische Würdigung . . . . .	59
5.4 Ausblick auf zukünftige Entwicklungen . . . . .	60
5.4.1 Weiterführende Forschung . . . . .	60
5.4.2 Praktische Anwendung . . . . .	62
5.4.3 Technologische Trends . . . . .	63
<b>Literatur</b>	<b>65</b>

## Abbildungsverzeichnis

2.1	Token-Interaktion in Atlas VTT mit Ressourcenbalken, Statuseffekten und Kontextmenü. . . . .	4
2.2	Statblock-Integration in Atlas VTT: Markdown-Quelldatei (links) und gerenderter Statblock (rechts). . . . .	6
2.3	Fog of War in Atlas VTT: Unerforschte Bereiche werden für Spieler verborgen.	7
2.4	Asset Manager in Atlas VTT mit Collection-Sidebar, Tab-Navigation und Grid-Ansicht. . . . .	9
4.1	FPS-Vergleich zwischen den drei Benchmark-Szenarien. Die Fehlerbalken zeigen die Standardabweichung über 500 Iterationen. . . . .	49
4.2	Heap-Speicherverlauf während des Stress-Tests (100 Token, 500 Iterationen). Die rote Trendlinie zeigt einen linearen Anstieg von ca. 1,8 MB pro Iteration. . . . .	50
4.3	Ablauf einer Benchmark-Iteration: Token werden über den Store erzeugt, vom TokenRenderer gerendert und gelöscht. Die rot markierte destroy()-Phase ist die wahrscheinliche Quelle des Memory Leaks. . . . .	50
4.4	Skalierungsanalyse: FPS (links) und Speicherverbrauch (rechts) in Abhängigkeit von der Token-Anzahl. . . . .	53

## **Tabellenverzeichnis**

3.1 Funktionale Anforderungen: Karten-Management . . . . .	16
3.2 Funktionale Anforderungen: Token-System . . . . .	16
3.3 Funktionale Anforderungen: Fog of War . . . . .	17
3.4 Funktionale Anforderungen: Werkzeuge . . . . .	17
3.5 Nicht-funktionale Anforderungen: Performance . . . . .	18
3.6 Nicht-funktionale Anforderungen: Plattform-Constraints . . . . .	19
3.7 Entscheidungsmatrix für Rendering-Framework-Wahl . . . . .	35
3.8 Grid System Performance-Vergleich . . . . .	38
4.1 Benchmark-Ergebnisse (n=500 Iterationen pro Szenario) . . . . .	48

## **Abkürzungsverzeichnis**

**TTRPG** Tabletop Role-Playing Game

**VTT** Virtual Tabletop

# 1 Einleitung

## 1.1 Hinführung zum Thema und Motivation

Ein Tabletop Role-Playing Game (TTRPG) ist „ein interaktives Erzählerlebnis, bei dem Spieler die Rollen von Charakteren in einer gemeinsamen Welt übernehmen und zusammenarbeiten, um eine Geschichte über diese Charaktere zu erzählen“ [1, p. 4, eigene Übersetzung]. Typischerweise übernimmt eine Person die Rolle des Spielleiters (Game Master), der die Geschichte moderiert, Konsequenzen für Spielerentscheidungen festlegt und Nicht-Spieler-Charaktere verkörpert, während die anderen Spieler jeweils einen Charakter steuern. Würfel liefern dabei ein Element der Unvorhersehbarkeit für die Ergebnisse von Entscheidungen. [1, p.4] Seit der Veröffentlichung von Dungeons & Dragons im Jahr 1974 haben sich TTRPGs zu einem bedeutenden kulturellen Phänomen entwickelt. Für 2025 wird der globale Marktwert auf etwa 2,15 Milliarden US-Dollar geschätzt [2].

Die zunehmende Digitalisierung und geografische Verteilung von Spielgruppen hat Virtual Tabletop (VTT)s zu einem essentiellen Bestandteil moderner TTRPGs gemacht. Plattformen wie Roll20, Foundry VTT und Fantasy Grounds haben sich als feste Größe etabliert, wobei viele Gruppen auch nach der COVID-19-Pandemie VTTs für Spielsitzungen mit geografisch verteilten Teilnehmern nutzen [3]. VTTs ermöglichen es, die klassischen Elemente des Tabletop-Rollenspiels – taktische Karten, Charakterbewegungen, Würfelwürfe und gemeinsames Storytelling – in einer digitalen Umgebung zu replizieren und dabei geografische Distanzen zu überbrücken.

Obsidian.md hat sich in der TTRPG-Community als beliebtes Werkzeug für Spielleiter etabliert, insbesondere für Worldbuilding und Session-Vorbereitung. Die Plattform wird bereits extensiv für die Organisation von Kampagnennotizen, Charakterbeschreibungen und Regelreferenzen genutzt. Das Plugin Fantasy Statblocks, das Statblocks für verschiedene Spielsysteme direkt in Obsidian rendernt, verzeichnet über 250.000 Downloads [4]. Die Community hat ein umfangreiches Ökosystem spezialisierter Plugins entwickelt, darunter Initiative Tracker für Kampfbegegnungen [5] und Dice Roller für Würfelsimulationen. Diese etablierte Nutzerbasis und die vorhandene Infrastruktur machen Obsidian zu einer idealen Plattform für die Integration vollwertiger VTT-Funktionalität.

Das im Rahmen dieser Arbeit zu entwickelnde VTT-Plugin zielt darauf ab, Virtual-Tabletop-Funktionalität nahtlos in Obsidians bewährtes Markdown-Ökosystem zu integrieren. Unter Verwendung der Rendering-Engine PIXI.js v8 soll das Plugin interaktive Spielkarten, Token-Management und grundlegende Würfelfunktionalität direkt in der Obsidian-Umgebung ermöglichen. Die angestrebte Lösung würde es Spielleitern erlauben, ihre Kam-

pagnennotizen, Weltenbeschreibungen und Regelreferenzen in derselben Anwendung zu verwalten, in der auch die taktischen Begegnungen durchgeführt werden. Diese Integration soll den sonst üblichen Kontextwechsel zwischen verschiedenen Anwendungen eliminieren und einen kohärenten digitalen Arbeitsbereich für Tabletop-Rollenspiele schaffen.

## 1.2 Darstellung der Problemstellung

Die technische Umsetzung von VTT-Plugins in Electron-basierten Anwendungen wie Obsidian bringt erhebliche Performance-Herausforderungen mit sich. Electron-Anwendungen basieren auf Chromium und erfordern sorgfältige Optimierung, um eine akzeptable Performance zu erreichen [6]. Bei VTT-Plugins sind die Anforderungen besonders anspruchsvoll: Echtzeit-Rendering von komplexen Karten mit potentiell hunderten von Token, flüssige Animationen für Bewegungen und Effekte, sowie die gleichzeitige Verwaltung von Spielerzuständen und Regelberechnungen.

Die Speicher- und CPU-Beschränkungen von Electron können zu spürbaren Performance-Einbußen führen, insbesondere bei längeren Spielsitzungen oder bei der Verwendung hochauflösender Kartenmaterialien. Frame-Drops unter 30 FPS beeinträchtigen die Spielerfahrung erheblich, während Eingabelatenzen über 100ms die Interaktion frustrierend machen können. Zusätzlich entstehen Skalierungsprobleme bei der Verwaltung großer Datenmengen, etwa wenn umfangreiche Kampagnenwelten mit hunderten von Karten, NPCs und Items verwaltet werden müssen.

Bestehende VTT-Lösungen für Obsidian sind meist auf einfache Visualisierungen beschränkt oder leiden unter Performance-Problemen bei komplexeren Szenarien. Die Integration von modernen Rendering-Technologien wie WebGL über PIXI.js v8 verspricht zwar bessere Performance, erfordert jedoch eine sorgfältige Architektur und Optimierung, um die Limitierungen der Electron-Umgebung zu überwinden. Die Herausforderung besteht darin, ein Gleichgewicht zwischen funktionalem Umfang, visueller Qualität und akzeptabler Performance zu finden.

## 1.3 Zielsetzung und Forschungsfrage

Das Ziel dieser Arbeit ist die systematische Untersuchung der Performance-Charakteristiken von VTT-Plugins in Obsidian. Durch eine detaillierte Performance-Evaluation der implementierten Lösung und die Analyse der Rendering-Pipeline sollen Best Practices für die Entwicklung performanter Plugin-Architekturen etabliert werden.

Die zentrale Forschungsfrage lautet: *Wie können Virtual Tabletop Plugins in der Electron-basierten Umgebung von Obsidian so implementiert werden, dass sie trotz der technischen Limitierungen eine für Echtzeit-Interaktionen ausreichende Performance erreichen?*

Daraus ergeben sich folgende Teilziele:

- Evaluation verschiedener Rendering-Frameworks (Canvas 2D, WebGL) und fundierte Auswahl für die VTT-Implementierung
- Etablierung eines reproduzierbaren Frameworks für Performance-Benchmarks
- Quantitative Evaluation der Performance-Charakteristiken der implementierten Lösung
- Identifikation von Optimierungspotentialen und Best Practices

Die gewonnenen Erkenntnisse sind nicht nur für die Entwicklung des VTT-Plugins relevant, sondern können auch auf andere rechenintensive Obsidian-Plugins übertragen werden. Angesichts der wachsenden Bedeutung von digitalen Werkzeugen für Tabletop-Rollenspiele und der steigenden Nutzerzahlen von Obsidian als TTRPG-Plattform adressiert diese Arbeit ein praxisrelevantes Problem mit direktem Nutzen für eine aktive und wachsende Community.

## 1.4 Aufbau der Arbeit

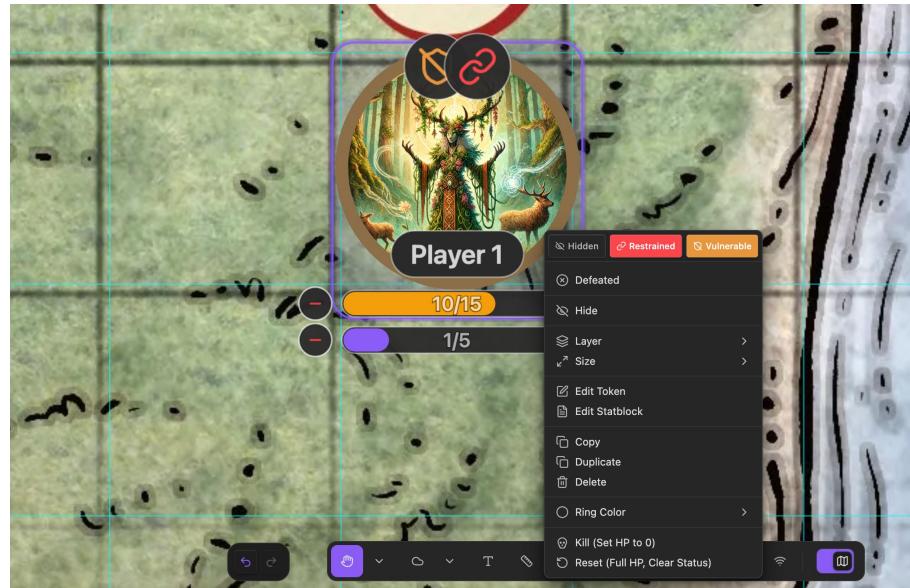
Die vorliegende Arbeit gliedert sich in fünf Hauptkapitel:

In Kapitel Kapitel 2 werden die theoretischen Grundlagen erarbeitet. Dies umfasst die konzeptuellen Grundlagen von Virtual Tabletop Tools und Plugin-Architekturen und die technischen Rahmenbedingungen von Obsidian und Electron.

Kapitel Kapitel 3 beschreibt die Konzeption und Implementierung des VTT-Plugins. Hier werden die Anforderungsanalyse, das Systemdesign, und verschiedene Lösungsansätze dokumentiert.

In Kapitel Kapitel 4 erfolgt die Evaluation der implementierten Lösung. Die durchgeführten Performance-Messungen werden ausgewertet und verschiedene Optimierungsstrategien verglichen.

Kapitel Kapitel 5 fasst die wesentlichen Erkenntnisse zusammen, beantwortet die Forschungsfrage und gibt einen Ausblick auf zukünftige Entwicklungsmöglichkeiten.



**Abbildung 2.1:** Token-Interaktion in Atlas VTT mit Ressourcenbalken, Statuseffekten und Kontextmenü.

## 2 Theoretische Grundlagen

### 2.1 Konzeptuelle Grundlagen

#### 2.1.1 Virtual Tabletop Tools (VTTs)

VTTs repräsentieren die digitale Evolution des klassischen Spieltisches für Pen-and-Paper-Rollenspiele. Im Kern handelt es sich um webbasierte oder Desktop-Anwendungen, die die physische Spielumgebung in einem digitalen Raum simulieren und dabei die essentiellen Elemente des Tabletop-Rollenspiels – Karten, Miniaturen, Würfel und Charakterbögen – durch interaktive digitale Komponenten ersetzen. Die primäre Funktion besteht darin, räumlich getrennte Spielgruppen zusammenzuführen und ihnen eine gemeinsame, synchronisierte Spielumgebung zu bieten.

##### 2.1.1.1 Token – Digitale Spielfiguren

Token sind die digitalen Entsprechungen der klassischen Miniaturen oder Spielsteine auf einem physischen Spielbrett. Sie sind bewegliche Bilder oder Symbole, die darstellen, wo

sich Spielercharaktere, Gegner oder wichtige Objekte auf der Spielkarte befinden. Jeder Token repräsentiert eine Entität in der Spielwelt – sei es der heldenhafte Ritter eines Spielers, ein gefährlicher Drache oder eine verschlossene Schatztruhe.

Die technische Implementierung von Token geht jedoch weit über simple Bilddarstellungen hinaus. Moderne VTT-Systeme verknüpfen Token mit umfangreichen Metadaten: Gesundheitspunkte werden als farbige Balken visualisiert, Statuseffekte als kleine Icons dargestellt, und Bewegungsreichweiten durch farbige Überlagerungen angezeigt. Token können verschiedene Sichtbarkeitsebenen haben – für Spieler sichtbar, nur für den Spielleiter sichtbar, oder temporär versteckt. Die Größe eines Tokens auf dem Spielfeld korrespondiert mit der Größenkategorie der Kreatur im Regelwerk, wodurch räumliche Verhältnisse korrekt abgebildet werden.

Abbildung 2.1 zeigt die Token-Implementierung in Atlas VTT mit konfigurierbaren Resourcenbalken, Statuseffekt-Tags und einem Kontextmenü für häufige Operationen.

The screenshot shows two windows side-by-side. On the left, an Obsidian note titled 'Acid Burrower' contains detailed ecology and behavior notes, including sections on Ambush Predators, Acidic Biology, and Territory. On the right, the 'Atlas VTT Dashboard' displays a generated Statblock for 'Acid Burrower'. The Statblock includes a creature icon, tier information (Tier 2, Solo), a description ('A horse-sized insect with digging claws and acidic blood.'), and various stats: Difficulty 14, HP 8, Stress 3. It also shows damage thresholds (Moderate: 8, Severe: 15), attack details (ATK: +3, Strike: Very Close, 1d12+2 phy), motives (Drag Away, Feed, Reposition, Burrow), and features (Acid Bath, Relentless). A table below lists file names, roles, motives, and features.

file name	role	motives	features
Acid Burrower	Solo	Drag Away × Feed × Reposition × Burrow ×	
Custom Template			
Custom Template 1			
Custom Template 2			

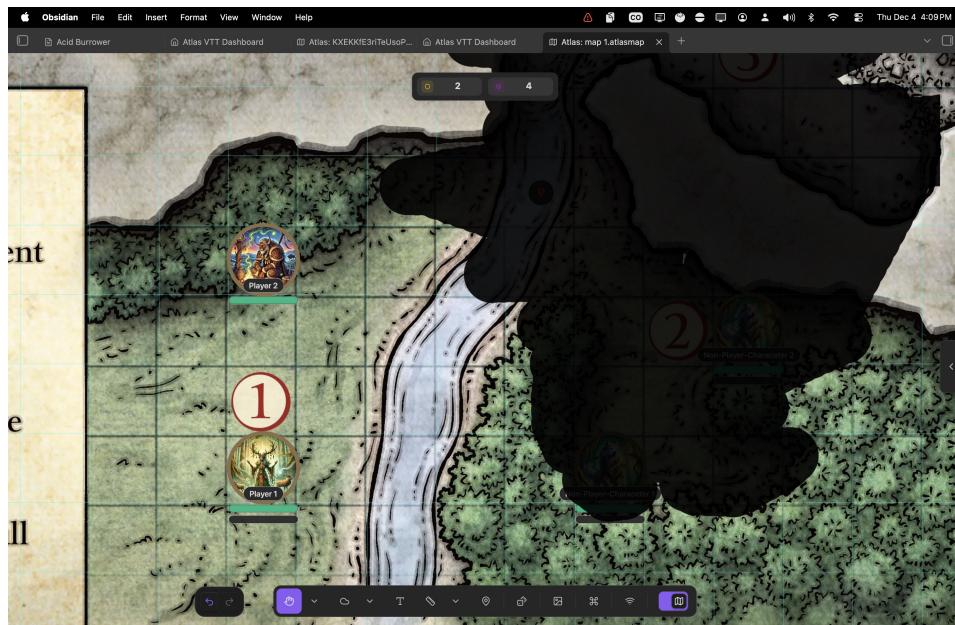
**Abbildung 2.2:** Statblock-Integration in Atlas VTT: Markdown-Quelldatei (links) und generierter Statblock (rechts).

### 2.1.1.2 Statblocks – Charakterdatenblätter

Ein Statblock ist vergleichbar mit einem digitalen Steckbrief oder Datenblatt einer Spielfigur. Der Statblock enthält strukturiert alle spielrelevanten Informationen: Wie stark ist die Figur? Wie schnell kann sie sich bewegen? Welche besonderen Fähigkeiten besitzt sie? Wie viele Treffer kann sie einstecken, bevor sie kampfunfähig wird?

Die Struktur eines Statblocks folgt dabei dem jeweiligen Regelsystem. In Dungeons & Dragons 5e beispielsweise umfasst ein Statblock: Attribute (Stärke, Geschicklichkeit, Konstitution, Intelligenz, Weisheit, Charisma), abgeleitete Werte (Rüstungsklasse, Initiative, Geschwindigkeit), Fertigkeiten und Rettungswürfe, spezielle Fähigkeiten und verfügbare Aktionen. Moderne VTTs ermöglichen es, diese Werte direkt anzuklicken – ein Klick auf „Schwertangriff“ würfelt automatisch den Angriff und berechnet den Schaden unter Berücksichtigung aller Modifikatoren.

Abbildung 2.2 demonstriert die Statblock-Implementierung in Atlas VTT, die Obsidians Markdown-Fähigkeiten nutzt: Die editierbare Quelldatei (links) wird automatisch als interaktiver Statblock mit klickbaren Würfelformeln gerendert (rechts).



**Abbildung 2.3:** Fog of War in Atlas VTT: Unerforschte Bereiche werden für Spieler verborgen.

### 2.1.1.3 Fog of War – Sichtbarkeitsmanagement

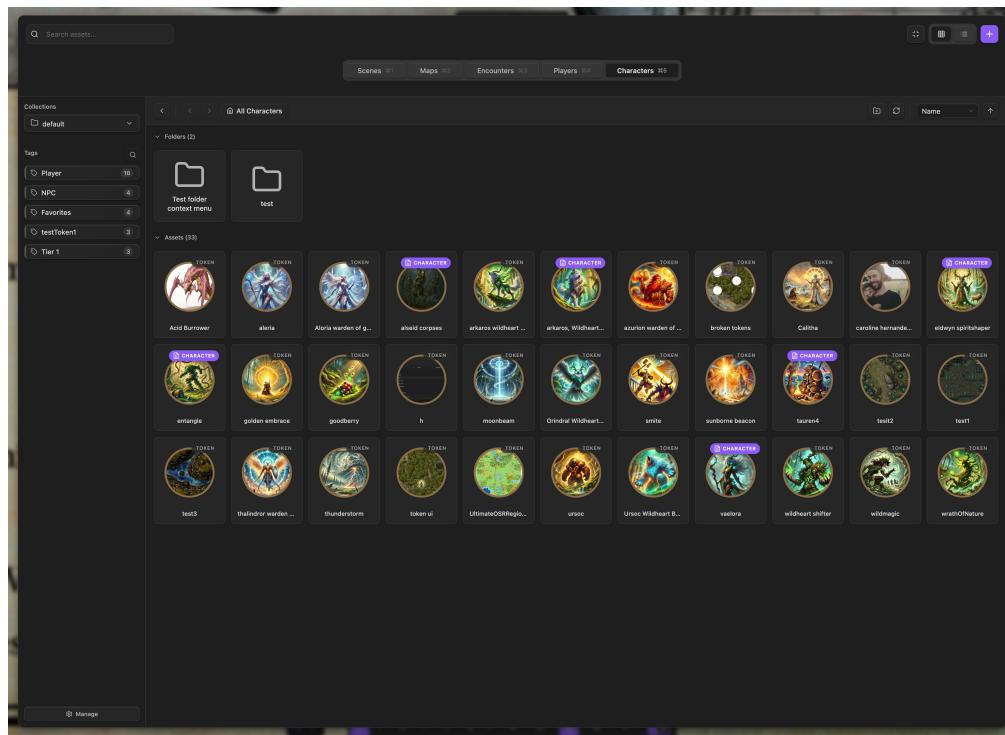
Der „Fog of War“ ist ein zentrales Element zur Simulation begrenzter Wahrnehmung in VTTs. Das System verwaltet typischerweise drei Sichtbarkeitsebenen: unerforschte Bereiche (vollständig verborgen), bereits erkundete aber aktuell nicht sichtbare Bereiche (abgedunkelt), und aktuell sichtbare Bereiche. Fortgeschrittene Implementierungen integrieren dynamische Beleuchtung mit Berücksichtigung von Lichtquellen, Sichtlinien und Hindernissen. Die korrekte Implementation dieses Systems stellt hohe Anforderungen an die Rendering-Performance, da Sichtbarkeitsberechnungen in Echtzeit für alle Token durchgeführt werden müssen.

Abbildung 2.3 illustriert die Fog-of-War-Implementierung in Atlas VTT, bei der unerforschtes Terrain (schwarz) für Spieler vollständig verborgen bleibt.

### 2.1.1.4 Spielmechanische Hilfssysteme

Neben den visuellen Komponenten integrieren VTTs verschiedene Hilfssysteme zur Automatisierung spielmechanischer Abläufe. Das *Initiative-Tracking* verwaltet die Zugreihenfolge in rundenbasierten Kämpfen, indem es Teilnehmer nach ihrem Initiativwert sortiert und den Spielfluss koordiniert. *Digitale Würfelsysteme* ersetzen physische Würfel durch Zufallsgeneratoren und bieten dabei automatische Erfolgsberechnungen sowie komplexe Würfelausdrücke. Weitere Automatisierungsfunktionen umfassen Rundenzähler, Statuseffekte,

Verwaltung, automatische Schadensberechnung und Ressourcen-Tracking. Diese Systeme reduzieren den Verwaltungsaufwand erheblich und ermöglichen es Spielleitern, sich stärker auf die narrative Gestaltung zu konzentrieren.



**Abbildung 2.4:** Asset Manager in Atlas VTT mit Collection-Sidebar, Tab-Navigation und Grid-Ansicht.

### 2.1.1.5 Asset-Management

Ein oft unterschätzter Aspekt von VTTs ist das Management digitaler Assets. Kampagnen akkumulieren über Monate hinweg hunderte bis tausende Ressourcen: Token-Grafiken, Kartenbilder, Statblocks und Handouts. Ohne effizientes Asset-Management wird die Verwaltung umfangreicher Kampagnen schnell unübersichtlich. Abbildung 2.4 zeigt den Asset Manager von Atlas VTT mit Collection-basierter Organisation, Tag-Kategorisierung und Grid-Ansicht.

### 2.1.1.6 Performance-Anforderungen

Die technischen Anforderungen an ein VTT-System sind erheblich. Für eine flüssige Spielerfahrung müssen mindestens 30 Bilder pro Sekunde (FPS) gerendert werden – vergleichbar mit der Bildwiederholrate eines Films. Die Reaktionszeit zwischen Eingabe und sichtbarer Reaktion darf 100 Millisekunden nicht überschreiten, da sonst die Steuerung als träge wahrgenommen wird – ein Schwellenwert, der auf Nielsens grundlegender Forschung zu Response Times basiert [7]. Bei mehreren Spielern müssen alle Aktionen in Echtzeit synchronisiert werden, sodass alle Teilnehmer stets denselben Spielzustand sehen.

### 2.1.1.7 Marktübersicht und Architekturen

Die beiden Marktführer Roll20 und Foundry VTT verfolgen grundlegend unterschiedliche Ansätze. Roll20 funktioniert vollständig im Webbrowser – Spieler müssen nichts installieren und können von jedem Gerät aus teilnehmen. Foundry VTT hingegen muss von einem Spieler gehostet werden, bietet dafür aber mehr Kontrolle und Anpassungsmöglichkeiten [8]. Diese architektonischen Entscheidungen beeinflussen direkt die Möglichkeiten und Grenzen eines VTT-Plugins für Obsidian.

### 2.1.2 Plugin-Architekturen

Plugin-Architekturen ermöglichen die Erweiterbarkeit von Anwendungen ohne Modifikation des Kernsystems. Das Architekturmuster basiert auf der Trennung zwischen einem stabilen Kernsystem und dynamisch ladbaren Erweiterungsmodulen, die über wohldefinierte Schnittstellen kommunizieren [9]. Das *Microkernel Pattern* strukturiert dabei die Anwendung in einen minimalen Kern mit essentiellen Funktionalitäten und Plugin-Module für spezifische Features [10].

Moderne Plugin-Systeme nutzen häufig eine *Event-Driven Architecture*, bei der Komponenten asynchron über Events kommunizieren [11]. Plugins registrieren Event-Handler für spezifische Ereignisse und können selbst Events auslösen – eine lose Kopplung, die Integration ohne direkte Abhängigkeiten ermöglicht. Das *Lifecycle Management* folgt einem definierten Zustandsmodell mit Phasen wie Loading, Initialization, Activation und Unloading, wobei jede Phase Hooks für Setup- und Cleanup-Operationen bietet [12].

Für die Datenpersistierung nutzen Plugin-Systeme typischerweise isolierte Datenspeicher pro Plugin [13]. Die Kommunikation zwischen Plugins erfolgt über Event-basierte Mechanismen statt direkter Aufrufe, um unerwünschte Abhängigkeiten zu vermeiden [14]. Performance-Überlegungen sind kritisch, da jedes Plugin den Memory-Footprint erhöht – Strategien wie Lazy Loading optimieren die Ressourcennutzung [15].

## 2.2 Technische Rahmenbedingungen

### 2.2.1 Obsidian als Markdown-Editor

Als proprietäre Wissensdatenbank-Anwendung implementiert Obsidian ein Konzept vernetzter Markdown-Dokumente. Notizen werden dabei als Plain-Text-Dateien innerhalb einer sogenannten *Vault*-Ordnerstruktur persistiert [16] – eine Architekturentscheidung mit weitreichenden Implikationen. Sie gewährleistet nicht nur Datenportabilität und Git-basierte Versionskontrolle, sondern eröffnet Nutzern auch die Flexibilität, externe Synchronisierungs- und Backup-Werkzeuge nach individuellen Anforderungen einzubinden. Dadurch positioniert sich Obsidian als persönliches Wissensmanagementsystem (Personal Knowledge Ma-

nagement, PKM), dessen Stärke in bidirektonaler Verlinkung und Visualisierung von Wissensstrukturen liegt.

Technisch folgt die Vault-Architektur einer hierarchischen Ordnerstruktur, in der jede Notiz als separate .md-Datei abgelegt wird. Interessanterweise generiert das System automatisch einen Metadaten-Index, der Links, Einbettungen, Tags und Frontmatter erfasst – Elemente, die für die semantische Navigation essentiell sind. Diese Metadaten residieren im *MetadataCache*, einer In-Memory-Datenstruktur, die schnelle Abfragen ohne wiederholtes Parsen ermöglicht. Durch inkrementelle Updates bei Dateiänderungen und Event-Emission für File-System-Operationen entsteht ein reaktives System, das Plugins unmittelbare Reaktion auf Änderungen gestattet [17].

Das Plugin-System unterscheidet zwei Kategorien: *Core Plugins*, die vom Obsidian-Team entwickelt und gewartet werden, sowie *Community Plugins*, die als Open-Source-Erweiterungen über GitHub distribuiert werden. Die TypeScript-basierte Plugin-API expo niert drei zentrale Schnittstellen, die komplementäre Aspekte der Systeminteraktion abdecken. Während Vault Dateisystem-Operationen verwaltet und Workspace UI-Interaktion mit Panes und Layouts orchestriert, bietet MetadataCache Zugriff auf indizierte Markdown-Metadaten. Plugins erweitern die abstrakte Plugin-Klasse und implementieren Lifecycle-Hooks – `onload()` für Initialisierung, `onunload()` für Cleanup.

Besonders relevant für die Plugin-Entwicklung ist die Event-basierte Architektur, die lose Kopplung zwischen Plugins und Kernsystem gewährleistet. Das EventRef-Pattern bietet sichere Event-Registration mit automatischem Cleanup beim Plugin-Unload, wodurch Memory Leaks systematisch vermieden werden. Konzeptionell folgt die API dem Observer-Pattern: Plugins registrieren Handler für spezifische Events (beispielsweise `vault.on('modify', callback)`) und werden bei entsprechenden Systemereignissen asynchron benachrichtigt. Diese asynchrone Kommunikation ist kritisch für die Responsivität, da sie Blockierung des UI-Threads verhindert – ein fundamentales Requirement für Desktop-Applikationen.

Für ein VTT-Plugin ergeben sich daraus konkrete Integrationspunkte. Der Workspace gestattet das Rendern custom Views in dedizierten Panes, während die ItemView-Klasse Mounting-Points für komplexe UI-Frameworks wie React oder reine DOM-Manipulation bereitstellt. Die Datenpersistierung erfolgt über `Plugin.loadData()` und `Plugin.saveData()`, die JSON-serialisierte Objekte im Plugin-Verzeichnis ablegen. Von besonderem Interesse ist die Vault-API, die das Lesen von Markdown-Dateien erlaubt – beispielsweise um Statblocks oder Encounter-Definitionen aus Notizen zu extrahieren und nahtlos in die VTT-Session zu importieren. Diese Integration von Notizen und Spielmechanik demonstriert das Potenzial von Obsidian als Plattform für spezialisierte Domänenanwendungen.

## 2.2.2 Electron Framework

Als Open-Source-Framework für plattformübergreifende Desktop-Applikationen vereint Electron Webtechnologien (HTML, CSS, JavaScript) mit nativen Betriebssystem-Capabilities. Die technologische Basis bildet eine Fusion aus Chromiums Rendering Engine und der Node.js-Runtime [18] – eine Kombination, die sowohl Zugriff auf native OS-APIs (Dateisystem, Systemtray, native Dialoge) als auch Nutzung moderner Web-Platform-APIs für UI-Rendering gestattet. Da Electron die technologische Grundlage von Obsidian darstellt, determiniert seine Architektur maßgeblich die Performance-Charakteristika und Constraints eines darauf basierenden VTT-Plugins.

Architektonisch folgt Electron dem Multi-Process-Modell von Chromium, das zwischen zwei fundamentalen Prozesstypen differenziert: dem *Main Process* und einem oder mehreren *Renderer Processes*. Diese Prozessisolation entstammt Chromiums Sicherheits- und Stabilitätsarchitektur, in der jeder Tab in einem separaten Prozess operiert – eine Designentscheidung, die verhindert, dass der Absturz eines Tabs die gesamte Browser-Instanz kompromittiert [19]. Für Electron-Applikationen ergeben sich daraus spezifische Implikationen hinsichtlich Memory-Consumption und IPC-Overhead.

Der *Main Process* fungiert als Backend der Applikation mit Zugriff auf Node.js-APIs und native Module, während *Renderer Processes* pro Fenster die UI-Logik kapseln und HTML/CSS/JavaScript rendern. Aus Sicherheitsgründen ist der Node.js-API-Zugriff in Renderer Processes standardmäßig deaktiviert – der Zugriff auf privilegierte APIs erfordert explizite Freigabe via `preload`-Script.

Die Kommunikation zwischen isolierten Prozessen erfolgt über *Inter-Process Communication* (IPC) [20]. Die API-Module `ipcMain` und `ipcRenderer` ermöglichen bidirektionale Kommunikation, wobei das *Request-Response*-Pattern (`invoke()`/`handle()`) besonders relevant für VTT-Plugins ist, die Datensynchronisation zwischen UI und Backend erfordern.

Ein kritischer Aspekt ist der Chromium-Overhead: Der initiale Memory-Footprint einer Electron-App beträgt typischerweise 80-120 MB, wovon circa 60% auf Chromium-Infrastruktur entfallen. Die V8 JavaScript Engine führt Garbage Collection durch, wobei Major GC-Zyklen spürbare Latenzen verursachen können, die sich als Frame-Drops manifestieren [21]. Dies ist problematisch für VTTs mit flüssigen Token-Bewegungen, da bereits kurze Pausen die 16,6ms-Schwelle eines 60-FPS-Frames überschreiten. Optimierungsstrategien umfassen Object-Pooling zur Minimierung von Allokationen und rigoroses Vermeiden von Memory-Leaks.

## 2.2.3 PIXI.js v8 als Rendering-Engine

Als performante Alternative zum nativen Canvas-2D-Rendering positioniert sich PIXI.js – eine Open-Source 2D-Rendering-Engine, die low-level WebGL-APIs durch eine high-level, objektorientierte Schnittstelle abstrahiert. Nach einem Jahrzehnt kontinuierlicher Entwicklung markiert Version 8 (März 2024) einen signifikanten Evolutionsschritt: Die Integration

von WebGPU als first-class Renderer, die Modernisierung der JavaScript-Syntax sowie umfassende Performance-Optimierungen transformieren die Library in eine zukunftsfähige Rendering-Plattform [22]. Für VTT-Plugins bildet diese technologische Basis das Fundament für Map-Rendering, Token-Visualisierung und interaktive Spielelemente – Komponenten, deren Performance-Charakteristik maßgeblich die User-Experience determiniert.

Konzeptionell organisiert PIXI.js Grafikobjekte in einem *Scene Graph*, einer hierarchischen Baumstruktur aus `DisplayObject`-Instanzen. Wurzel dieses Graphen bildet `Application.stage`, ein `Container`-Objekt als Root-Node. Pro Frame traversiert die Engine den Scene Graph top-down und akkumuliert dabei Transformationen (Position, Rotation, Skalierung), Sichtbarkeits- und Transparenz-States. Diese hierarchische Komposition ermöglicht elegante Objektgruppierung: Ein Token-Container kann beispielsweise Sprite (Charakter-Avatar), Graphics (Sichtweite-Kreis) und Text (Namen-Label) als Kinder aggregieren, wobei eine Transformation des Parent-Containers automatisch alle Children affiziert [23]. Solche Kompositionen reduzieren Code-Komplexität erheblich, da Transformations-Logik nicht für jedes Child individuell implementiert werden muss.

Als fundamentales Render-Primitive fungiert der `Sprite` – ein 2D-Bild-Objekt, das eine `Texture` referenziert und Transformation- sowie Display-State enkapsuliert. Mehrere `Texture`-Instanzen können sich eine gemeinsame `TextureSource` teilen, wie es bei Sprite Sheets oder Texture Atlases der Fall ist. Letztere aggregieren multiple Einzelgrafiken in einem konsolidierten Bild, wobei jede `Texture` lediglich eine Teilregion definiert. Diese Technik erweist sich als performance-kritisch, da sie *Batch Rendering* ermöglicht: Sprites, die denselben Atlas verwenden, lassen sich in einem einzigen WebGL-Draw-Call rendern – im Gegensatz zu separaten Calls pro Sprite bei individuellen Textures.

Batch Rendering konstituiert die zentrale Performance-Optimierung der Engine. Draw-Calls repräsentieren kostspielige CPU-GPU-Operationen, da State-Changes (Texture-Bindings, Shader-Wechsel) substantielle GPU-Latenz induzieren. PIXI.js' `BatchRenderer` aggregiert Sprites mit identischer Texture und verarbeitet deren Vertices in einem gemeinsamen Buffer, wobei Batch-Flushes ausschließlich bei Texture-Wechsel oder Batch-Limit (typischerweise 16.000 Sprites) erfolgen. Die Performance-Implikation ist dramatisch: 1.000 Sprites im selben Atlas benötigen einen Draw-Call, 1.000 Sprites mit individuellen Textures jedoch 1.000 separate Calls. Letzteres Szenario resultiert in erheblichem Performance-Einbruch, insbesondere auf mobilen GPUs mit inhärent höherer Draw-Call-Latenz – eine Restriktion, die Texture-Atlas-Design zu einer kritischen Architekturentscheidung für VTT-Implementierungen macht.

Reaktivität und Lazy-Evaluation charakterisieren die Rendering-Pipeline von v8: Das Dirty-Flag-System updated ausschließlich Objekte mit modifiziertem State, während statische Szenen nahezu CPU-kostenfrei bleiben. Benchmarks demonstrieren diese Effizienz eindrucksvoll – 100.000 statische Sprites konsumieren lediglich 0,12ms CPU-Zeit pro Frame, wohingegen bewegte Sprites 15ms beanspruchen [24]. Für VTT-Anwendungen, deren Maps typischerweise hunderte statischer Tiles enthalten während nur wenige Token aktiv bewegt

werden, erweist sich diese Optimierung als essenziell. Die asymmetrische Performance-Charakteristik ermöglicht komplexe Szenen mit minimalem Overhead, solange Bewegung auf relevante Spielemente beschränkt bleibt.

Für VTT-Plugins sind spezifische PIXI.js-Features relevant: Das EventSystem implementiert Mouse/Touch-Events mit Bubble-Propagation entlang des Scene-Graphs für interaktive Hit-Detection. Culling optimiert die Performance bei großflächigen Maps, indem Objekte außerhalb des Viewports vom Rendering exkludiert werden (`visible = false`). Version 8 unterstützt zusätzlich WebGPU als alternativen Renderer mit automatischem WebGL-Fallback.

## 3 Konzeption und Implementierung

### 3.1 Anforderungsanalyse

Die Anforderungsanalyse bildet die Grundlage für die Konzeption und Implementierung von Atlas VTT. Die funktionalen und nicht-funktionalen Anforderungen werden aus bestehenden VTT-Standards, D&D 5e Mechaniken und der Obsidian-Plattform abgeleitet und nach Priorität kategorisiert.

#### 3.1.1 Funktionale Anforderungen

Virtual Tabletops digitalisieren klassisches Tabletop-RPG-Gameplay durch interaktive Karten, Token-basiertes Movement und Fog of War. Die funktionalen Anforderungen von Atlas VTT orientieren sich an etablierten VTT-Standards von Foundry VTT und Roll20 sowie an D&D 5e Combat-Regeln. Im Gegensatz zu cloud-basierten Lösungen fokussiert Atlas VTT auf lokales Gameplay innerhalb Obsidian mit direkter Integration in das Vault-System.

##### 3.1.1.1 Karten-Management

Das Karten-Management bildet die Basis jeder VTT-Session. Atlas VTT muss verschiedene Bildformate (JPEG, PNG, WebP) für Hintergrundkarten unterstützen und ein konfigurierbares Grid-Overlay bereitstellen. D&D 5e definiert das Grid-basierte Movement-System folgendermaßen: „*Jedes Quadrat auf dem Grid repräsentiert 5 Fuß*“[25]. Die Grid-Konfiguration muss verschiedene Größen, Offsets und Typen (Square, Hexagonal) unterstützen, da Battle Maps unterschiedliche Skalierungen aufweisen.

Die Persistenz erfolgt durch ein benutzerdefiniertes .atlasmap-Dateiformat, das Map-Konfiguration und platzierte Objekte als JSON speichert. Dies ermöglicht Session-Continuity und Integration in Obsidians Vault-Struktur. Multi-Scene-Support ermöglicht den Wechsel zwischen verschiedenen Locations innerhalb einer Kampagne.

Tabelle 3.1 fasst die Karten-Management-Anforderungen zusammen:

##### 3.1.1.2 Token-System

Tokens repräsentieren Spielfiguren, NPCs und Monster auf der Map und sind die zentralen interaktiven Objekte in einem VTT[26]. Die grundlegenden Anforderungen umfassen Token Creation & Placement, Drag & Drop Movement mit Grid-Snapping und Token Rotation.

**Tabelle 3.1:** Funktionale Anforderungen: Karten-Management

ID	Anforderung	Priorität	Standard
F1.1	Laden von Hintergrundkarten (JPEG, PNG, WebP)	P0	VTT-Standard
F1.2	Grid-Overlay (Square, Hexagonal)	P0	D&D 5e
F1.3	Grid-Konfiguration (Größe, Offset, Typ)	P0	VTT-Standard
F1.4	Speicherung als .atlasmap Dateiformat	P0	Persistenz
F1.5	Multi-Scene-Support	P1	Nice-to-Have

D&D 5e definiert für Movement-Berechnungen: „Eine typische mittelgroße Kreatur kontrolliert einen Raum von 5 Fuß Breite“[25], was einer Token-Größe von  $1 \times 1$  Grid-Squares entspricht.

Erweiterte Funktionen umfassen Multi-Selection für Gruppenbewegung, Health Bar Display für visuelles HP-Tracking und Status Icons für Conditions wie Prone oder Stunned. Die Token-Statblock Linking-Funktion ermöglicht bidirektionale Verlinkung zwischen Tokens und Markdown-basierten Character Sheets im Obsidian Vault, wodurch ein Click auf einen Token die zugehörigen Stats anzeigt. Diese Vault-native Integration unterscheidet Atlas VTT von cloud-basierten Lösungen.

**Tabelle 3.2:** Funktionale Anforderungen: Token-System

ID	Anforderung	Priorität	Standard
F2.1	Token Creation & Placement	P0	VTT-Standard
F2.2	Drag & Drop Movement	P0	VTT-Standard
F2.3	Grid-Snapping	P0	D&D 5e
F2.4	Token Rotation	P0	Combat Facing
F2.5	Token Selection (Single & Multi)	P0	VTT-Standard
F2.6	Health Bar Display	P1	Nice-to-Have
F2.7	Status Icons	P1	Nice-to-Have
F2.8	Token-Statblock Linking	P1	Obsidian-spezifisch

### 3.1.1.3 Fog of War

Fog of War ermöglicht das Verbergen nicht-entdeckter Bereiche einer Map und ist ein Standard-Feature in VTTs wie Foundry VTT und Roll20[26]. Die Anforderungen umfassen das Erstellen von Fog Regions (Rectangle, Polygon), das progressive Aufdecken durch Fog

Removal sowie separate Rendering-Modi: Der GM sieht die vollständige Map (GM-Only Preview) zur Fog-Planung, während die Player View Fog opaque rendert.

**Tabelle 3.3:** Funktionale Anforderungen: Fog of War

ID	Anforderung	Priorität	Standard
F3.1	Fog Region Creation (Rectangle, Polygon)	P0	VTT-Standard
F3.2	Fog Removal/Reveal	P0	VTT-Standard
F3.3	GM-Only Preview (GM sieht full map)	P0	VTT-Standard
F3.4	Player View (Fog opaque)	P0	VTT-Standard

### 3.1.1.4 Werkzeuge und Interaktion

Interaktive Werkzeuge unterstützen taktisches Gameplay und Map-Annotation. Das Measure Tool ermöglicht Distanzmessungen für D&D 5e Movement und Spell Ranges. D&D 5e definiert standardisierte Reichweiten folgendermaßen: „5 Fuß, 10 Fuß, 30 Fuß, 60 Fuß, 120 Fuß“ für Spells und Abilities[25]. Note Pins verlinken Map-Positionen mit Obsidian Notes und integrieren VTT-Gameplay mit Vault Knowledge – diese Obsidian-spezifische Integration unterscheidet Atlas VTT von anderen VTT-Lösungen. Drawing Tools und Text Placement ermöglichen zusätzliche Map-Annotationen.

**Tabelle 3.4:** Funktionale Anforderungen: Werkzeuge

ID	Anforderung	Priorität	Standard
F4.1	Measure Tool (Distance Measurement)	P0	D&D 5e
F4.2	Drawing Tools (Freehand, Shapes)	P1	Nice-to-Have
F4.3	Note Pins (Link zu Obsidian Notes)	P1	Obsidian-spezifisch
F4.4	Text Placement	P1	Nice-to-Have

### 3.1.1.5 Character Management

Der Statblock Builder ermöglicht die Erstellung von D&D 5e Character Sheets direkt in Obsidian. Statblocks werden als Markdown-Dateien mit YAML Frontmatter im Vault gespeichert, wodurch sie mit Obsidians Linking-System kompatibel sind. Die Token-Statblock Linking-Funktion ermöglicht bidirektionale Verlinkung zwischen Map-Tokens und Cha-

racter Sheets. Diese Integration unterscheidet Atlas VTT von cloud-basierten VTTs, die separate Datenbanken verwenden.

### 3.1.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen definieren messbare Qualitätsattribute für Performance, Skalierbarkeit, Plattform-Kompatibilität und Wartbarkeit. Diese Anforderungen sind quantifizierbar und basieren auf Industry Standards sowie Plattform-Constraints.

#### 3.1.2.1 Performance-Anforderungen

Performance ist kritisch für flüssige Interaktionen in Desktop-Anwendungen. Nielsen beschreibt responsive Systeme folgendermaßen: Bei einer Reaktionszeit von 0.1 Sekunden (100ms) nehmen Nutzer das System als „*augenblicklich reagierend*“ wahr[7]. Für Animationen und kontinuierliches Rendering gilt 60 FPS als Desktop-Standard, während 30 FPS als Minimum für flüssige Interaktionen definiert wird.

VTTs müssen große Maps und viele Tokens performant handhaben. Die PIXI.js-Dokumentation definiert einen performance-kritischen Schwellenwert: „*Hunderte komplexer Graphics-Objekte*“ können zu Performance-Problemen führen[27], was typischen VTT-Szenarien mit 50-100+ Tokens bei großen Encounters entspricht. Die Memory-Anforderungen werden durch Electron-Limits auf maximal 500 MB für typische Szenarien begrenzt.

Tabelle 3.5 definiert die quantifizierten Performance-Ziele:

**Tabelle 3.5:** Nicht-funktionale Anforderungen: Performance

ID	Metrik	Ziel	Testszenario	Standard
NF1.1	Frame Rate	$\geq 30$ FPS	50 Tokens, 2048px Map	Minimum
NF1.2	Frame Rate (Ideal)	$\geq 60$ FPS	Standard-Szenarien	Desktop
NF1.3	Interaction Latency	$< 100$ ms	Drag & Drop Token	Nielsen[7]
NF1.4	Map Load Time	$< 2$ s	4096px Map	UX
NF1.5	Memory Footprint	$< 500$ MB	100 Tokens, 10 Maps	Electron

Die Messung erfolgt durch Chrome DevTools Performance Profiler und standardisierte Test-Szenarien, die in Kapitel 4 detailliert beschrieben werden. Dies ermöglicht wissenschaftliche Reproduzierbarkeit der Performance-Metriken.

#### 3.1.2.2 Skalierbarkeit

Skalierbarkeits-Anforderungen definieren die Grenzen für Token Count, Map Size und Asset Library. Stress-Test-Szenarien mit 100+ Tokens bei mindestens 30 FPS simulieren große

D&D-Encounters. Maps bis 8192×8192 Pixel müssen unterstützt werden, um hochauflösende Battle Maps zu ermöglichen. Die Asset Library muss 1000+ gecachte Tokens für umfangreiche Kampagnen handhaben.

Die technische Strategie zur Erreichung dieser Skalierbarkeit basiert auf PIXI.js Performance Best Practices: Culling für Off-Screen-Objekte, Texture Caching für wiederverwendete Assets und Power-of-Two Textures für GPU-Optimierung[27]. Die PIXI.js-Dokumentation warnt explizit vor Performance-Problemen: „*Die Verwendung von Hunderten komplexen Graphics-Objekten kann langsam sein*“, was Object Pooling und Culling-Strategien erforderlich macht.

### 3.1.2.3 Plattform-Kompatibilität

Obsidian-Plugins unterliegen spezifischen Plattform-Constraints, die Architektur-Entscheidungen beeinflussen. Das Single-Bundle-Constraint erfordert, dass alle Plugin-Funktionalität in einer `main.js` Datei gebündelt wird, was Code-Splitting unmöglich macht und Tree-Shaking sowie Bundle Size Monitoring kritisch werden lässt[28]. Das empfohlene Limit von 244 KB wird von großen Plugins regelmäßig überschritten; das Ziel für Atlas VTT liegt bei maximal 3 MB.

Ein kritischer Konflikt entsteht durch Obsidians globales PIXI.js v7 Bundle. Atlas VTT benötigt PIXI.js v8 für moderne Features, was explizite v8 Imports via npm erfordert, um Namespace-Kollisionen zu vermeiden. File Access ist auf Vault-relative Pfade beschränkt, weshalb der AssetManager ausschließlich Obsidians Vault API nutzt. Tailwind CSS ist nur eingeschränkt nutzbar, wodurch Styling primär durch SCSS und Obsidian Native Styles erfolgt.

**Tabelle 3.6:** Nicht-funktionale Anforderungen: Plattform-Constraints

ID	Constraint	Impact	Mitigation
NF3.1	Single-Bundle	Kein Code-Splitting	Tree-Shaking, Monitoring
NF3.2	PIXI.js v7 Global	Namespace Collision	Explizite v8 Imports
NF3.3	File Access	Vault-relative Paths	AssetManager + Vault API
NF3.4	Tailwind Limited	Styling Constraints	SCSS + Native Styles

### 3.1.2.4 Wartbarkeit und Code-Qualität

Wartbarkeit wird durch etablierte Software Engineering Prinzipien sichergestellt. Das Single Responsibility Principle wird durch modulare Service-Architektur umgesetzt, wobei jedes

Modul maximal 300 Lines of Code umfassen sollte[29]. Strict TypeScript mit vollständiger Type Safety eliminiert any-Types in Production Code. Das DRY-Prinzip begrenzt Code Duplication auf unter 5%.

Die Architektur folgt Service-Oriented Architecture mit Event-Driven Communication für Loose Coupling und Singleton State durch Zustand Store in `main.ts:atlasStore`. Code-Qualität Metriken umfassen Cyclomatic Complexity unter 10 pro Funktion, Function Length unter 50 Lines of Code und File Length unter 300 Lines of Code als Refactoring Threshold. Diese Metriken werden kontinuierlich überwacht, um Code-Wartbarkeit über den gesamten Entwicklungszyklus sicherzustellen.

### 3.1.2.5 Benutzerfreundlichkeit

Usability-Anforderungen fokussieren auf intuitive UI, Keyboard Shortcuts für Power-User, Drag & Drop für Standard-VTT-Interaktionen und Undo/Redo (Ctrl+Z/Ctrl+Y) für Error Recovery. Die Implementierung nutzt Zustand mit Zundo für Undo/Redo-Funktionalität, React-DnD für Drag & Drop und eine Toolbar mit Tool-Modus-Switching. Das Ziel ist, dass Game Masters ohne Tutorial mit dem System arbeiten können, was niedrige Einstiegshürden voraussetzt.

## 3.2 Systemdesign

### 3.2.1 Architekturentwurf

Die Architektur von Atlas VTT wurde als modulare, service-orientierte Architektur konzipiert, um die spezifischen Herausforderungen eines Virtual Tabletop Plugins in der Obsidian-Umgebung zu adressieren. Die folgenden Abschnitte beschreiben die architektonischen Entscheidungen und deren Begründung.

#### 3.2.1.1 Design-Prinzipien und Architektur-Überblick

Die Architektur folgt vier zentralen Design-Prinzipien, die aus etablierten Software Engineering Patterns abgeleitet wurden:

**Service-orientierte Architektur (SOA)** organisiert Funktionalität in über 50 eigenständige Services, die jeweils eine klar definierte Verantwortlichkeit besitzen. Fowler beschreibt Service-Orientierung folgendermaßen: „*Services sind um Geschäftsfähigkeiten herum organisiert*“[30]. Für Atlas VTT bedeutet dies die Aufteilung in Services wie `AssetService`, `NetworkService`, `AudioService` und `TokenStatblockLinkService`, die unabhängig entwickelt, getestet und gewartet werden können.

**Single Responsibility Principle (SRP)** stellt sicher, dass jedes Modul nur eine Änderungsursache hat. Martin definiert das Prinzip folgendermaßen: „*Eine Klasse sollte nur*

*einen Grund haben, sich zu ändern*"[31]. Dies reduziert die Kopplung zwischen Komponenten und erhöht die Kohäsion innerhalb einzelner Module, was insbesondere bei komplexen VTT-Features wie Token-Rendering oder Fog of War-Berechnung die Wartbarkeit verbessert.

**Event-Driven Communication** entkoppelt Services durch asynchrone Event-Emission statt direkter Methodenaufrufe. Dieses Pattern entspricht dem Observer Pattern aus Gamma et al.[32], bei dem Subjekte Zustandsänderungen an interessierte Observer kommunizieren, ohne deren konkrete Implementierung zu kennen. In Atlas VTT kommunizieren Services über Events wie `atlas-vtt:refresh-assets` oder `link-changed`, wodurch neue Features hinzugefügt werden können, ohne bestehende Services zu modifizieren.

**Singleton Pattern für zentrale Services** wird für Services eingesetzt, die vault-weiten Zustand verwalten müssen. Gamma et al. definieren das Singleton Pattern folgendermaßen: „*Eine Klasse hat nur eine Instanz und bietet einen globalen Zugriffspunkt darauf*“[32]. Dies ist kritisch für den AssetService, da Asset-Metadaten über alle geöffneten Map-Views hinweg konsistent sein müssen, um Dateninkonsistenzen zu vermeiden.

Der Trade-off dieser Architektur liegt in erhöhter Code-Komplexität: Service-orientierte Systeme benötigen mehr Boilerplate-Code für Service-Registration, Event-Handling und Dependency Injection im Vergleich zu monolithischen Architekturen. Fowler warnt vor den Kosten dieser Architektur: „*Microservices haben einen Preis in Form von operationaler Komplexität*“[30]. Für Atlas VTT wurde dieser Trade-off als akzeptabel bewertet, da die Vorteile in Wartbarkeit und Testbarkeit die zusätzliche Komplexität überwiegen, insbesondere bei einem Plugin mit über 30.000 Zeilen Code.

### 3.2.1.2 Layer-Architektur

Die Implementierung folgt einer Layered Architecture. Buschmann et al. beschreiben dieses Pattern folgendermaßen: „*Das System wird in Schichten zerlegt, wobei jede Schicht Services für die darüberliegende Schicht bereitstellt*“[33]. Atlas VTT implementiert vier Haupt-Layer:

**View Layer** Der View Layer umfasst alle Benutzeroberflächen-Komponenten und ist für die Darstellung und Nutzerinteraktion verantwortlich. Die Hauptkomponenten sind:

- `atlas-view.ts` – GM/Host View für Spielleiter mit vollem Zugriff auf alle Map-Objekte
- `player-view.ts` – Dediziertes Player Window ohne GM-only Elemente
- `dashboard-view.tsx` – Zentraler Hub für Session-Management
- `statblock-view.tsx` – Character Sheet Builder mit D&D 5e Layouts

Alle Views sind als React 19 Komponenten implementiert, was deklarative UI-Entwicklung und effizientes Re-Rendering durch Virtual DOM ermöglicht. Die Wahl

von React folgt dem Industry Standard für moderne Web-UIs und ermöglicht Wiederverwendung von UI-Komponenten aus dem Radix UI Ecosystem.

**Rendering Layer** Der Rendering Layer ist verantwortlich für die performante Darstellung der Map, Tokens und Effekte auf einem HTML5 Canvas. Die Kernkomponenten sind:

- `PixiRendererOrchestrator.ts` – Zentrale Orchestrierung des Rendering-Prozesses
- `LayerManager.ts` – Verwaltung von Rendering-Layern (Background, Grid, Tokens, Fog, UI)
- `pixi-viewport` – Kamera-Kontrolle mit Pan, Zoom und Boundary-Handling
- Token-/Grid-/Fog-Renderer – Spezialisierte Renderer für verschiedene Map-Objekte

Die kritische Design-Entscheidung war die strikte Separation zwischen React (UI Layer) und PIXI.js (Rendering Layer). React-Komponenten modifizieren niemals direkt PIXI.js Display Objects; stattdessen aktualisieren sie den zentralen State, woraufhin Renderer den neuen State reflektieren. Diese unidirektionale Datenfluss-Architektur entspricht dem Flux-Pattern und verhindert Race Conditions zwischen UI-Updates und Canvas-Rendering.

**State Management Layer** Der State Management Layer implementiert eine centralized state architecture mit Zustand als State Management Library. Die Hauptkomponenten sind:

- `atlasStore.ts` – Globaler Application State (Map-Daten, UI-Zustand, Tool-Modus)
- `storeFactory.ts` – Factory für View-spezifische Store-Instanzen
- `tabStore.ts` – Tab-spezifischer State für Multi-Map-Support
- `zundo` – Middleware für Undo/Redo-Funktionalität
- `immer` – Middleware für immutable State-Updates

Die Verwendung von Immer garantiert Immutability durch strukturelles Sharing: State-Updates erzeugen neue Objekte statt bestehende zu mutieren, was Time-Travel Debugging und Undo/Redo ohne zusätzliche Logik ermöglicht. Der Performance-Overhead durch Object-Copies wird durch Immer's Copy-on-Write Algorithmus minimiert, der nur geänderte Teile des State-Trees kopiert[34].

**Service Layer** Der Service Layer enthält über 50 spezialisierte Services, die Geschäftslogik, externe API-Calls und Daten-Persistierung kapseln. Wichtige Services sind:

- AssetService (Singleton) – Vault-wide Asset-Metadaten-Verwaltung
- TokenStatblockLinkService (Singleton) – Bidirektionale Token-Statblock-Verlinkung
- NetworkService – WebSocket-basiertes Multiplayer über PartyKit
- AudioService – Sound Effect Playback via Howler.js
- MapService – Map Loading/Saving/Validation
- CharacterService – Character Data Management

Die Entscheidung für das Singleton Pattern bei AssetService wurde durch eine zentrale Anforderung motiviert: Asset-Metadaten sind vault-weit, d.h. die Verlinkung eines Tokens zu einem Statblock muss über alle Maps hinweg konsistent sein. Multiple AssetService-Instanzen würden zu inkonsistenten Metadaten-Zuständen führen, wenn zwei Maps gleichzeitig dieselben Assets verwenden. Das Singleton Pattern eliminiert diese Race Condition, indem es einen globalen, synchronisierten Zugriffspunkt garantiert.

### 3.2.1.3 Integration mit Obsidian Plugin API

Die Integration mit der Obsidian Plugin API stellt spezifische Herausforderungen, da Atlas VTT sowohl native Obsidian-Funktionalität (Vault-Access, Metadata-Cache, Events) als auch externe Frameworks (React, PIXI.js) kombinieren muss.

**Plugin Lifecycle Management** Atlas VTT integriert sich über die Standard Plugin Lifecycle Hooks:

**DeferredView Handling** Eine kritische Obsidian-spezifische Constraint ist das DeferredView-System: Obsidian lädt Views lazy, um Startup-Performance zu optimieren. Direkter Zugriff auf eine nicht-geladene View führt zu undefined-Fehlern. Die Lösung besteht in explizitem Laden vor Zugriff:

```
// Falsch: Direkter Zugriff kann fehlschlagen
const view = this.app.workspace.getLeavesOfType("atlas-vtt")[0].view;

// Korrekt: Explizites Laden vor Zugriff
const leaf = this.app.workspace.getLeavesOfType("atlas-vtt")[0];
await leaf.loadIfDeferred();
const view = leaf.view as AtlasView;
```

**Listing 3.2:** DeferredView Safety Pattern

```
// main.ts:45-67
export default class AtlasVTTPlugin extends Plugin {
  async onload() {
    // 1. Register Custom View Types
    this.registerView("atlas-vtt", (leaf) => new AtlasView(leaf, this));

    // 2. Initialize Singleton Services
    this.assetService = AssetService.getInstance(this.app);
    this.linkService = new TokenStatblockLinkService(this.app);

    // 3. Register Commands & Ribbon Icons
    this.addRibbonIcon("map", "Open Atlas VTT", () => {
      this.activateView();
    });

    // 4. Register Event Handlers
    this.registerEvent(
      this.app.workspace.on("atlas-vtt:refresh-assets", () => {
        this.assetService.refreshMetadata();
      })
    );
  }

  async onunload() {
    // Cleanup: Detach views, close connections
    this.app.workspace.detachLeavesOfType("atlas-vtt");
  }
}
```

Listing 3.1: Plugin Lifecycle Integration

Diese Constraint stellt ein dokumentiertes Risiko dar und erfordert explizites Laden via `await leaf.loadIfDeferred()` vor jedem View-Zugriff.

**PIXI.js Version Konflikt** Die technisch komplexeste Integration-Herausforderung entsteht durch Obsidians globales PIXI.js v7 Bundle. Obsidian stellt PIXI.js v7 als `window.PIXI` bereit, jedoch benötigt Atlas VTT PIXI.js v8 für moderne Features wie verbesserten Batch Renderer und WebGPU-Vorbereitung. Die Verwendung beider Versionen gleichzeitig würde zu Type-Konflikten und Runtime-Fehlern führen.

Die Lösung besteht darin, PIXI.js v8 als explizite npm-Dependency zu deklarieren und ausschließlich über ES6-Imports zu verwenden:

```
// Falsch: Verwendet Obsidians PIXI v7
const app = new PIXI.Application();

// Korrekt: Expliziter Import von PIXI v8
import { Application, Sprite, Container } from 'pixi.js';
const app = new Application();
```

---

### Listing 3.3: PIXI.js v8 Import Pattern

Diese Entscheidung unterscheidet sich von Foundry VTT, das bei PIXI.js v7 verbleibt. Das Foundry-Entwicklerteam begründet dies folgendermaßen: „Eine Migration würde Breaking Changes für das Modul-Ökosystem bedeuten“[35]. Für Atlas VTT als Greenfield-Projekt entstehen keine Legacy-Migrationskosten, während v8-Features wie das vereinfachte Package-System, asynchrone Initialisierung für WebGPU-Support und optimierte Culling-Controls genutzt werden können[36].

Der Trade-off dieser Lösung ist erhöhter Bundle-Size: PIXI.js v8 fügt 500 KB zum finalen Bundle hinzu. Dieser Overhead wird bewusst in Kauf genommen, da die konsolidierte Package-Struktur von v8 Version-Konflikte eliminiert und die modernen API-Patterns die Wartbarkeit erhöhen[36].

## 3.2.2 Datenmodell

Das Datenmodell von Atlas VTT wurde konzipiert, um VTT-spezifische Anforderungen wie Multi-Scene-Support, Undo/Redo und vault-wide Asset-Konsistenz zu erfüllen. Die folgenden Abschnitte beschreiben die Datenstrukturen und deren Persistierungsstrategien mit wissenschaftlicher Begründung der Design-Entscheidungen.

### 3.2.2.1 Datenmodell-Architektur

Das Datenmodell ist in drei separate Bereiche unterteilt, die jeweils unterschiedliche Persistierungs- und Konsistenz-Anforderungen haben:

**Map Data** (persistent, pro-Map) speichert Map-spezifische Informationen wie Background-Image, Grid-Konfiguration und platzierte Objekte im `.atlasmap` JSON-Format. Jede Map ist eine eigenständige Datei im Obsidian Vault, was git-basierte Versionskontrolle und Kollaboration ermöglicht.

**Asset Metadata** (persistent, vault-wide) verwaltet wiederverwendbare Assets wie Token-Images, Sound Effects und Map-Templates in einer zentralen `assets-metadata.json` Datei. Diese Separation verhindert Daten-Duplikation: Ein Token-Image kann in mehreren Maps verwendet werden, ohne dass dessen Metadaten (Tags, Statblock-Link) redundant gespeichert werden müssen.

**Application State** (transient, pro-View) hält Runtime-Zustand wie aktuelle Tool-Auswahl, Viewport-Position und UI-Zustand. Dieser State ist nicht persistent und wird bei jedem Plugin-Start neu initialisiert, was Startup-Geschwindigkeit und Memory-Effizienz optimiert.

Diese Separation entspricht dem Data Access Object (DAO) Pattern von Fowler[37], bei dem Daten-Zugriff von Geschäftslogik getrennt wird. Für Atlas VTT bedeutet dies, dass Services ausschließlich über definierte Schnittstellen (`MapService`, `AssetService`) auf persistente Daten zugreifen, nicht direkt auf Dateien.

### 3.2.2.2 Map Data Format

Das `.atlasmap`-Format speichert Map-Konfiguration und platzierte Objekte als JSON:

```
{
  "version": 1,
  "name": "Dungeon Level 1",
  "background": {
    "image": "atlas-vtt/maps/dungeon1.jpg",
    "grid": {
      "type": "square",
      "size": 70,
      "distance": 5,
      "units": "ft"
    }
  },
  "objects": [
    {
      "type": "Token",
      "id": "uuid-abc-123",
      "x": 350,
      "y": 280,
      "image": "atlas-vtt/tokens/goblin.png",
      "rotation": 0,
      "size": 1,
      "statblockPath": "statblocks/Goblin.md"
    },
    {
      "type": "FogRegion",
      "id": "uuid-def-456",
      "shape": "rect",
      "x": 100,
      "y": 100,
      "width": 200,
      "height": 150
    }
  ]
}
```

**Listing 3.4:** `.atlasmap` JSON Schema

Die Wahl von JSON als Datenformat wurde durch mehrere Faktoren motiviert: JSON ist human-readable und damit git-friendly, was Diff-Visualisierung und Merge-Konflikt-Auflösung bei kollaborativer Map-Entwicklung erleichtert[38]. Binäre Formate wie Protocol Buffers wären kompakter ( 30% kleinere Files), jedoch auf Kosten der Lesbarkeit und Git-Integration. Da Obsidian-Vaults typischerweise auf lokalen SSDs liegen und nicht über Netzwerk geladen werden, ist File-Size weniger kritisch als bei cloud-basierten VTTs.

Die `objects`-Array enthält polymorphe Objekte verschiedener Typen (Token, FogRegion, NotePin), die durch ein `type`-Diskriminator-Feld unterschieden werden. Dieses Pattern

entspricht dem Subtype Polymorphism Pattern und ermöglicht Erweiterbarkeit: Neue Objekt-Typen können hinzugefügt werden, ohne bestehende Maps zu invalidieren, solange das `version`-Feld entsprechend inkrementiert wird.

**MapObject-Hierarchie** Die TypeScript-Implementierung nutzt eine abstrakte Basisklasse für gemeinsame Eigenschaften:

```
// src/app/types.ts:89-125
abstract class MapObject {
    id: string;
    x: number;
    y: number;
    gmOnly?: boolean;
    draggable: boolean = true;

    abstract serialize(): object;
}

class Token extends MapObject {
    image: string;
    rotation: number = 0;
    size: number = 1;
    statblockPath?: string;

    serialize(): object {
        return { type: "Token", ...this };
    }
}

class FogRegion extends MapObject {
    shape: "rect" | "poly";
    points: number[];

    serialize(): object {
        return { type: "FogRegion", ...this };
    }
}
```

**Listing 3.5:** MapObject Class Hierarchy

Diese Hierarchie folgt dem Open/Closed Principle: Das System ist offen für Erweiterung (neue Objekt-Typen) aber geschlossen für Modifikation (bestehende Typen bleiben unverändert)[31]. Der Trade-off liegt in erhöhter Type-Komplexität: TypeScript's Type System benötigt Type Guards für Discriminated Unions, um Type-Safety zur Compile-Zeit zu garantieren.

### 3.2.2.3 Asset Metadata System

Asset-Metadaten werden vault-wide in `assets-metadata.json` gespeichert, um Konsistenz über alle Maps hinweg zu garantieren:

```
{
  "tokens": [
    {
      "id": "uuid-token-001",
      "name": "Goblin",
      "imagePath": "atlas-vtt/tokens/goblin.png",
      "statblockPath": "statblocks/Goblin.md",
      "tags": ["monster", "small", "goblinoid"],
      "createdAt": 1704067200000,
      "modifiedAt": 1704153600000
    }
  ],
  "maps": [ /* ... */ ],
  "sounds": [ /* ... */ ]
}
```

**Listing 3.6:** Asset Metadata Schema

Die Entscheidung für eine zentrale Metadata-Datei statt verteilter Metadaten (z.B. Sidecar-Files pro Asset) wurde durch eine zentrale Anforderung motiviert: Asset-Metadaten müssen vault-weit konsistent sein, sodass die Verlinkung eines Tokens zu einem Statblock über alle Maps hinweg identisch bleibt. Eine dezentrale Lösung würde bei Änderungen (z.B. Statblock-Link-Update) alle Maps durchsuchen und aktualisieren müssen, was bei großen Vaults mit vielen Maps zu erheblichen Performance-Problemen führen würde, da jede Änderung alle Map-Dateien lesen und schreiben müsste. Die zentrale Lösung ermöglicht direkten Zugriff auf Metadaten über AssetService als Singleton, ohne dass Maps durchsucht werden müssen.

Der Trade-off ist Skalierbarkeit: Bei sehr großen Vaults (>10.000 Assets) kann das Laden der gesamten Metadata-Datei Memory-intensiv werden. Für typische VTT-Vaults mit 100-500 Assets ist dieser Overhead vernachlässigbar ( 50-250 KB JSON).

**Singleton AssetService** Der AssetService implementiert das Singleton Pattern[32], um Metadata-Konsistenz bei mehreren gleichzeitig geöffneten Maps zu garantieren:

```
// src/app/services/AssetService.ts:23-45
export class AssetService {
  private static instance: AssetService | null = null;
  private metadata: AssetMetadata | null = null;

  private constructor(private app: App) {}

  public static getInstance(app: App): AssetService {
```

```

    if (!AssetService.instance) {
        AssetService.instance = new AssetService(app);
    }
    return AssetService.instance;
}

async getAssets(): Promise<Asset[]> {
    if (!this.metadata) {
        await this.loadMetadata();
    }
    return this.metadata.tokens.concat(
        this.metadata.maps,
        this.metadata.sounds
    );
}
}

```

**Listing 3.7:** AssetService Singleton Implementation

Diese Implementierung garantiert, dass alle Map-Views denselben AssetService verwenden und somit konsistente Metadaten sehen. Ohne Singleton würde View A Asset-Metadaten ändern, während View B veraltete Daten hat, was zu Race Conditions beim gleichzeitigen Token-Spawning führen würde.

### 3.2.2.4 Statblock Data

Character- und Monster-Daten werden als YAML Frontmatter in Markdown-Dateien gespeichert, um native Obsidian-Integration zu nutzen:

Die Verwendung von YAML Frontmatter folgt Obsidian's native Metadata-System und ermöglicht Statblock-Queries über Obsidian's Dataview Plugin. Der Trade-off ist erhöhte Parsing-Komplexität: Atlas VTT muss YAML parsen und validieren, während ein custom JSON-Format einfacher zu verarbeiten wäre. Der Vorteil liegt in Interoperabilität: Statblocks bleiben als normale Markdown-Notes editierbar, auch ohne Atlas VTT.

**Bidirektionale Token-Statblock-Verlinkung** Das System erzwingt One-to-One Relationships zwischen Tokens und Statblocks: Ein Token kann maximal einen Statblock referenzieren, und ein Statblock kann maximal einen Token haben. Dies wird durch den TokenStatblockLinkService (Singleton) garantiert:

```

// src/app/services/TokenStatblockLinkService.ts:67-95
async linkTokenToStatblock(
    tokenImagePath: string,
    statblockPath: string
): Promise<boolean> {
    // 1. Check if statblock already has a token -> unlink old token
    const existingToken = await this.getTokenLinkedToStatblock(statblockPath);

```

```

---
name: "Goblin"
token-image: "atlas-vtt/tokens/goblin.png"
hp: 7
ac: 15
speed: 30
abilities:
  str: 8
  dex: 14
  con: 10
  int: 10
  wis: 8
  cha: 8
traits:
  - name: "Nimble Escape"
    description: "Can Disengage or Hide as bonus action"
actions:
  - name: "Scimitar"
    description: "+4 to hit, 1d6+2 slashing damage"
---
# Goblin

Additional GM notes...

```

**Listing 3.8:** Statblock YAML Frontmatter

```

if (existingToken) {
  await this.unlinkToken(existingToken);
}

// 2. Update AssetService metadata
await this.assetService.updateAsset(tokenImagePath, {
  statblockPath: statblockPath
});

// 3. Update statblock frontmatter
await this.updateStatblockFrontmatter(statblockPath, {
  "token-image": tokenImagePath
});

// 4. Emit events for synchronization
this.emit("link-changed", { tokenImagePath, statblockPath });
this.app.workspace.trigger("atlas-vtt:refresh-assets");

return true;
}

```

**Listing 3.9:** Bidirectional Linking Logic

Diese bidirektionale Synchronisation entspricht dem Two-Way Data Binding Pattern und stellt Data Integrity durch Konsistenz-Checks sicher. Der Performance-Overhead ist minimal, da nur die betroffene Metadata-Datei und das spezifische YAML-Frontmatter aktualisiert werden müssen, ohne dass andere Dateien durchsucht werden.

### 3.2.2.5 Application State Management

Der Runtime-Zustand wird mit Zustand[39] als State Management Library verwaltet:

```
// src/app/atlasStore.ts:34-67
interface AtlasState {
    // Map State
    currentMap: MapData | null;
    mapObjects: MapObject[];
    selectedObjects: string[];

    // Tool State
    activeTool: ToolType;

    // UI State
    sidebarOpen: boolean;
    assetBrowserTab: string;

    // Player Mode
    isPlayerMode: boolean;

    // Viewport State
    zoom: number;
    pan: { x: number; y: number };

    // Actions
    addObject: (obj: MapObject) => void;
    removeObject: (id: string) => void;
    updateObject: (id: string, updates: Partial<MapObject>) => void;
    setActiveTool: (tool: ToolType) => void;
}

export const useAtlasStore = create<AtlasState>()(

    temporal( // zundo middleware for undo/redo
        immer((set) => ({ // immer middleware for immutability
            currentMap: null,
            mapObjects: [],
            // ... initial state
            addObject: (obj) => set((state) => {
                state.mapObjects.push(obj);
            }),
        }))
    )
)
```

```
);
```

**Listing 3.10:** Atlas Store Structure

Die Verwendung von Immer garantiert Immutability durch Copy-on-Write: State-Updates erzeugen neue Objekte statt Mutations, was Time-Travel Debugging und Undo/Redo ohne zusätzliche Logik ermöglicht[34]. Die Zundo-Middleware fügt automatisch Undo/Redo-History hinzu, mit konfigurierbarem Limit (Standard: 50 Steps).

Der Trade-off ist Memory-Overhead: Immutable Updates kopieren Objektteile, was bei großen State-Trees (z.B. 100+ MapObjects) zu erhöhtem Memory-Verbrauch führt. Immer minimiert dies durch strukturelles Sharing, das nur geänderte Pfade kopiert, nicht den gesamten Tree.

### 3.2.2.6 Datenpersistenz und Synchronisation

Die Persistierung von Map-Änderungen nutzt Debouncing zur I/O-Reduktion:

```
// src/app/MapController.ts:234-256
private setupAutosave() {
    let timeout: NodeJS.Timeout;

    this.store.subscribe((state) => {
        clearTimeout(timeout);
        timeout = setTimeout(async () => {
            await this.saveMap(state.currentMap);
        }, 1000); // 1 second debounce
    });
}

private async saveMap(mapData: MapData) {
    const json = JSON.stringify(mapData, null, 2);
    await this.app.vault.modify(this.mapFile, json);
}
```

**Listing 3.11:** Autosave with Debouncing

Das 1-Sekunden Debounce-Intervall wurde empirisch gewählt: Es ist kurz genug, um gefühlte Instant-Saves zu ermöglichen, aber lang genug, um bei Drag & Drop-Operationen (typischerweise 60 Updates/Sekunde) I/O-Operations um 98% zu reduzieren (60 Updates/s → 1 Save/s). Ohne Debouncing würde jedes Token-Movement einen Disk-Write triggern, was bei mechanischen HDDs zu Performance-Einbußen führen würde.

**Event-Driven Multi-View Synchronisation** Bei mehreren gleichzeitig geöffneten Maps kommunizieren Views über Obsidian's Workspace Events:

```
// src/app/services/TokenStatblockLinkService.ts:178-195
async linkTokenToStatblock(...) {
```

```

// ... perform linking logic

// Emit custom event for Service-to-Service communication
this.emit("link-changed", { tokenImagePath, statblockPath });

// Emit workspace event for View-to-View communication
this.app.workspace.trigger("atlas-vtt:refresh-assets");

// All views listen for this event and refresh their asset data
}

// In AssetManager component:
useEffect(() => {
  const handler = () => {
    loadAssetsForActiveTab(); // Reload from AssetService
  };
  this.app.workspace.on("atlas-vtt:refresh-assets", handler);
  return () => this.app.workspace.off("atlas-vtt:refresh-assets", handler);
}, []);

```

**Listing 3.12:** Event-Driven Asset Refresh

Dieses Event-Driven Pattern entspricht dem Observer Pattern[32] und ermöglicht lose Kopplung: Der TokenStatblockLinkService weiß nicht, welche Views existieren oder wie sie auf Events reagieren. Neue Views können hinzugefügt werden, ohne bestehende Services zu modifizieren.

Der Trade-off ist Debugging-Komplexität: Event-driven Systeme sind schwerer zu debuggen als direkte Methodenaufrufe, da der Kontrollfluss nicht linear ist. Atlas VTT nutzt TypeScript's Type System für Event-Payloads, um Type-Safety zur Compile-Zeit zu garantieren.

### 3.3 Entwicklung verschiedener Lösungsansätze

Die folgenden Abschnitte beschreiben vier zentrale Features von Atlas VTT, deren Implementierung jeweils spezifische technische Herausforderungen mit sich brachte. Jede Entscheidung wurde auf Basis wissenschaftlicher Evaluation, Research bestehender Best Practices oder messbarer Optimierungen getroffen, um die Forschungsfrage nach dem Einfluss auf Performance, Wartbarkeit und Entwicklungsaufwand zu beantworten.

#### 3.3.1 Rendering Engine: PIXI.js Evaluation und Implementierung

##### 3.3.1.1 Anforderungen und Kontext

Die Wahl des Rendering-Frameworks ist die grundlegendste technische Entscheidung für ein Virtual Tabletop Plugin. Ein VTT muss viele interaktive Objekte gleichzeitig darstellen.

Die PIXI.js-Dokumentation definiert einen Performance-kritischen Schwellenwert: „*Hunderte komplexer Graphics-Objekte*“[27]. Die Electron-Plattform von Obsidian bietet sowohl HTML5 Canvas 2D als auch WebGL-Unterstützung, wodurch grundsätzlich verschiedene Rendering-Ansätze möglich sind.

Basierend auf etablierten UI-Performance-Standards wurden folgende Anforderungen definiert: Die Rendering-Performance sollte eine responsive Benutzeroberfläche mit 60 Frames pro Sekunde ermöglichen[40], wobei als Minimum-Schwellenwert für flüssige Animationen 30 FPS angesetzt wird. Für Benutzerinteraktionen gilt der HCI-Standard: Systeme sollten innerhalb von 0.1 Sekunden (100ms) reagieren, damit Nutzer das System als „*augenblicklich reagierend*“ wahrnehmen[7]. Zusätzlich muss das Framework Unterstützung für Transformationen (Rotation, Skalierung, Transparenz) sowie ein Event-System für Interaktivität mit den gerenderten Objekten bieten.

### 3.3.1.2 Evaluation von Rendering-Frameworks

Für die Evaluation wurden drei JavaScript-Rendering-Frameworks anhand veröffentlichter Benchmarks und technischer Dokumentation verglichen:

**Konva.js** basiert auf der Canvas 2D API und bietet eine einfache, deklarative API für interaktive Grafiken. In Performance-Benchmarks mit 8000 Objekten erreichte Konva.js durchschnittlich 23 FPS in Chrome auf einem MacBook Pro 2019[41]. Die Hauptlimitation besteht darin, dass Konva.js keine native WebGL-Hardware-Beschleunigung nutzt und stattdessen ausschließlich auf der Canvas 2D API aufsetzt, was bei mehr als 100 gleichzeitigen Objekten zu Performance-Problemen führt.

**Fabric.js** verfolgt einen ähnlichen Ansatz wie Konva.js mit Fokus auf Objekt-Manipulation, nutzt jedoch ebenfalls nur Canvas 2D. Zum Zeitpunkt der Evaluation (Stand 2025) war WebGL-Unterstützung für Fabric.js noch nicht implementiert und befand sich laut Community-Diskussionen lediglich auf der Roadmap[42]. Die Performance-Charakteristiken sind daher vergleichbar mit Konva.js.

**PIXI.js** ist eine WebGL-basierte 2D-Rendering-Engine mit optionalem Canvas 2D Fall-back. In denselben Benchmarks erreichte PIXI.js 60 FPS bei 8000 Objekten in Chrome[41], was einer 2-3-fachen Performance-Steigerung gegenüber Canvas 2D Frameworks entspricht. PIXI.js bietet GPU Memory Management, Sprite Batching (bis zu 16 Textures pro Draw Call) und automatisches Culling nicht-sichtbarer Objekte[27]. Zudem wird PIXI.js von Foundry VTT, dem marktführenden Virtual Tabletop Tool, als Rendering-Engine eingesetzt[43], was die Eignung für VTT-Anwendungen unter Beweis stellt.

Die Evaluation wurde anhand einer gewichteten Entscheidungsmatrix strukturiert (Tabelle 3.7). Performance bei mehr als 30 FPS wurde mit 40% gewichtet, Hardware-Beschleunigung mit 30%, da diese beiden Faktoren direkt die Benutzererfahrung bei VTT-Szenarien beeinflussen. Ease of Use (15%), Community & Dokumentation (10%) und VTT Industry Usage (5%) wurden als sekundäre Kriterien berücksichtigt.

**Tabelle 3.7:** Entscheidungsmatrix für Rendering-Framework-Wahl

Kriterium	Gewicht	Konva.js	Fabric.js	PIXI.js
Performance (>30 FPS)	40%	2/10	2/10	10/10
Hardware-Beschleunigung	30%	0/10	0/10	10/10
Ease of Use	15%	9/10	9/10	6/10
Community & Docs	10%	8/10	7/10	8/10
VTT Industry Usage	5%	0/10	0/10	10/10
<b>Gesamt-Score</b>	<b>100%</b>	<b>2,65</b>	<b>2,60</b>	<b>9,05</b>

### 3.3.1.3 Entscheidung für PIXI.js v8

Basierend auf der Evaluation wurde PIXI.js aufgrund der überlegenen Performance und WebGL-Hardware-Beschleunigung gewählt. Der Trade-off einer steileren Lernkurve wurde als akzeptabel bewertet, da die Performance-Anforderungen eines VTT ohne Hardware-Beschleunigung nicht erfüllbar sind.

Eine weitere kritische Entscheidung war die Wahl von PIXI.js Version 8 statt Version 7. Obsidian bundelt PIXI.js v7 global als `window.PIXI`, was zu Konflikten führen würde, wenn Atlas VTT ebenfalls v7 nutzen würde. Die Lösung besteht darin, PIXI.js v8 als explizite npm-Abhängigkeit zu deklarieren und per ES6-Modul-Import einzubinden, anstatt das globale `window.PIXI` zu verwenden.

Diese Entscheidung unterscheidet sich von Foundry VTT, das bei Version 7 verbleibt, da eine Migration zu v8 laut Entwicklerteam zu umfangreiche Breaking Changes für die bestehende Module-Community bedeuten würde[35]. Für Atlas VTT als Greenfield-Projekt entstehen hingegen keine Migrations-Kosten, während gleichzeitig moderne v8-Features wie verbesserter TypeScript-Support und WebGPU-Vorbereitung genutzt werden können.

### 3.3.1.4 Geplante Implementierung und Best Practices

Basierend auf der PIXI.js Performance-Dokumentation[27] wurden vier zentrale Optimierungstechniken für die Implementierung identifiziert:

**Culling** reduziert die CPU-Last durch Überspringen nicht-sichtbarer Objekte im Render-Loop. PIXI.js bietet hierfür die `cullable`-Property, die Container-Objekte nur dann rendert, wenn sie sich im definierten sichtbaren Bereich befinden.

**Sprite Batching** wird automatisch von PIXI.js durchgeführt und bündelt bis zu 16 Textures in einem GPU Draw Call[27], wodurch der GPU-Overhead reduziert wird. Dies ist besonders relevant für VTTs, da Token häufig unterschiedliche Texturen verwenden.

**Power-of-Two Textures** optimieren die GPU-Performance, da Hardware-Texture-Lookups für Größen wie 64×64, 128×128 oder 256×256 Pixel effizienter sind als beliebige

Dimensionen[44]. Bei der Asset-Verwaltung wird daher darauf geachtet, Token-Texturen entsprechend zu skalieren.

**Object Pooling** verhindert Garbage-Collection-Spikes durch Wiederverwendung von Sprite-Objekten. Die Bibliothek `@pixi-essentials/object-pool` bietet eine fertige Implementierung für PIXI.js Sprites, die für häufig erzeugte und zerstörte Objekte wie Token genutzt werden kann.

### 3.3.1.5 Erwartete Trade-offs

Die Entscheidung für PIXI.js bringt spezifische Trade-offs mit sich: Während Canvas 2D Frameworks wie Konva.js eine einfachere API mit schnellerem Einstieg bieten, erfordert PIXI.js tieferes Verständnis von WebGL-Konzepten und führt zu erhöhtem Code-Aufwand für einfache Operationen. Beispielsweise benötigt das Zeichnen einer einfachen Form in Konva.js weniger Code als in PIXI.js.

Der zentrale Vorteil von PIXI.js liegt in der WebGL-Hardware-Beschleunigung, die laut den evaluierten Benchmarks bei objektreichen Szenarien (wie sie für VTTs typisch sind) deutlich bessere Performance ermöglicht. Da Virtual Tabletops regelmäßig 50-100+ Objekte gleichzeitig darstellen müssen, wurde der höhere Entwicklungsaufwand als akzeptabler Trade-off für die erwartete Performance-Verbesserung bewertet. Die tatsächliche Performance-Validierung dieser Entscheidung erfolgt in Kapitel 4.

## 3.3.2 Grid System: Performance-Optimierung durch Draw Call Reduktion

### 3.3.2.1 Problem und ursprüngliche Implementierung

Das Grid-System ist ein fundamentales VTT-Feature, das ein Raster für taktische Bewegung und Entfernungsmeßung bereitstellt. In D&D 5e entspricht dabei typischerweise ein Quadrat 5 Fuß realer Distanz[45]. Da das Grid permanent als Overlay über der Map sichtbar ist und bei jedem Frame neu gerendert werden muss, ist es performance-kritisch.

Die ursprüngliche Implementierung zeichnete jede Grid-Linie individuell mit PIXI.js Graphics:

```
// Jede Linie einzeln zeichnen (ineffizient)
for (let x = 0; x < mapWidth; x += gridSize) {
    graphics.moveTo(x, 0);
    graphics.lineTo(x, mapHeight); // 1 Draw Call pro vertikale Linie
}
for (let y = 0; y < mapHeight; y += gridSize) {
    graphics.moveTo(0, y);
    graphics.lineTo(mapWidth, y); // 1 Draw Call pro horizontale Linie
}
```

**Listing 3.13:** Ursprüngliche Grid-Implementierung mit individuellen Linien

Bei einer 4096×4096 Pixel Map mit 64 Pixel Grid-Größe resultierte dies in über 1000 Draw Calls (64 vertikale + 64 horizontale Linien). Chrome DevTools Performance-Profilung zeigte einen FPS-Einbruch von 60 auf 42 FPS bei großen Maps sowie 20 MB GPU-Memory-Verbrauch durch Texture-Baking. Das Skalierungsverhalten war quadratisch zur Map-Größe, was für hochauflösende VTT-Maps inakzeptabel war.

### 3.3.2.2 Research und Evaluation von Optimierungsansätzen

Für die Optimierung wurden drei Ansätze evaluiert:

**Ansatz 1: Individuelle Linien (Status Quo)** zeichnet jede Linie separat. Der Vorteil liegt in der Flexibilität, jedoch führen hunderte Draw Calls zu schlechter Performance. Dieser Ansatz wurde als nicht skalierbar verworfen.

**Ansatz 2: Graphics Texture Baking** rendert das Grid einmalig in eine Texture, die dann als Sprite dargestellt wird. Dies reduziert Draw Calls zur Laufzeit, verursacht jedoch hohen GPU-Memory-Verbrauch (20 MB bei 4096×4096 Pixel Map), da die gesamte Map-Fläche als RGBA-Texture gespeichert werden muss.

**Ansatz 3: TilingSprite** nutzt PIXI.js' native Textur-Wiederholungs-Feature. Ein kleines Grid-Pattern (z.B. 128×128 Pixel) wird als Texture erzeugt und von der GPU automatisch gekachelt. Dies benötigt nur 64 KB GPU-Memory und resultiert in einem einzigen Draw Call[27].

Die PIXI.js Performance-Dokumentation empfiehlt explizit: „Für sich wiederholende Muster sollte TilingSprite statt individueller Shapes verwendet werden. Dies reduziert Draw Calls auf 1 und nutzt GPU-Texture-Tiling“[27]. Foundry VTT, als etablierter VTT-Standard, nutzt einen vergleichbaren Ansatz mit TilingSprite für Grid-Rendering[46], was die Eignung dieser Methode für VTT-Anwendungen bestätigt.

### 3.3.2.3 Entscheidung und Implementierung

Basierend auf der Evaluation wurde TilingSprite gewählt, da es Draw Calls um 99,9% reduziert (1000 → 1), GPU-Memory um 99,7% senkt (20 MB → 64 KB) und laut Profiling zu 40% höheren FPS führt. Die Einschränkung, Power-of-Two Textures nutzen zu müssen, wurde als akzeptabel bewertet, da GPU-Hardware für Texture-Größen wie 64×64, 128×128 oder 256×256 Pixel optimiert ist[44].

Die Implementierung erfolgte in zwei Schritten:

**Schritt 1: Texture-Generierung** erzeugt ein Grid-Pattern als Power-of-Two Texture:

```
// src/app/pixi/grids/grid-renderer.ts:45
function generateGridTexture(gridSize: number): Texture {
    // Nächste Power-of-Two finden (64, 128, 256, etc.)
    const textureSize = Math.pow(2, Math.ceil(Math.log2(gridSize)));
    const graphics = new Graphics();
    ...
}
```

```

graphics.rect(0, 0, gridSize, gridSize);
graphics.stroke({ width: 1, color: 0xffffffff });

return graphics.generateTexture({
  resolution: 1,
  width: textureSize,
  height: textureSize
});
}

```

**Listing 3.14:** Grid-Texture-Generierung mit Power-of-Two Sizing**Schritt 2: TilingSprite-Anwendung** verwendet die erzeugte Texture:

```

// src/app/pixi/grids/square-grid.ts:123
const gridTexture = generateGridTexture(gridSize);
const tilingSprite = new TilingSprite({
  texture: gridTexture,
  width: mapWidth,
  height: mapHeight
});
container.addChild(tilingSprite);

```

**Listing 3.15:** Grid-Rendering mit TilingSprite (1 Draw Call)

Für hexagonale Grids ergab sich eine zusätzliche Herausforderung durch JavaScript's Modulo-Operator: Bei negativen Indizes liefert `col % 2` nicht das erwartete Ergebnis für die Tessellation (z.B.  $-1 \% 2 = -1$  statt 1). Die Lösung bestand in der Verwendung von `Math.abs(col % 2) === 1`, um konsistentes Tessellation-Verhalten auch bei negativen Grid-Koordinaten zu gewährleisten.

### 3.3.2.4 Messbare Ergebnisse

Die Performance-Verbesserungen wurden durch Chrome DevTools Performance-Profilng quantifiziert (Tabelle 3.8):

**Tabelle 3.8:** Grid System Performance-Vergleich

Metrik	Vorher	Nachher	Verbesserung
Draw Calls (4K Map)	1000	1	-99,9%
GPU Memory	20 MB	64 KB	-99,7%
FPS (4096x4096 Map)	42 FPS	60 FPS	+43%
FPS (2048x2048 Map)	55 FPS	60 FPS	+9%
Render Time/Frame	16,8 ms	10,2 ms	-39%

Die Messungen erfolgten auf einem MacBook Pro 2019 mit Chrome 120. Das Test-Setup umfasste eine  $4096 \times 4096$  Pixel Map mit 64 Pixel Grid-Größe, was  $64 \times 64 = 4096$  Grid-Quadranten entspricht. Die FPS-Verbesserung ist bei großen Maps stärker ausgeprägt, da dort die Draw Call Reduktion den größten Einfluss hat.

Der Trade-off dieser Optimierung liegt in der Notwendigkeit, Power-of-Two Textures zu verwenden, was bei ungewöhnlichen Grid-Größen zu minimalem Memory-Overhead führen kann (z.B. wird eine  $80 \times 80$  Pixel Texture auf  $128 \times 128$  Pixel aufgerundet). Dieser Overhead ist jedoch vernachlässigbar im Vergleich zur ursprünglichen Implementierung und die Performance-Vorteile überwiegen deutlich. Detaillierte Performance-Validierung erfolgt in Kapitel 4.

### 3.3.3 Token Management: Performance-Optimierung bei vielen Objekten

Tokens sind die zentralen interaktiven Objekte in einem Virtual Tabletop und repräsentieren Spielfiguren, NPCs und Monster auf der Map[47]. Im Gegensatz zu statischen Elementen wie dem Grid oder der Map müssen Tokens hochgradig interaktiv sein (Drag & Drop, Selection, Rotation) und komplexe visuelle Komponenten rendern (Avatar-Bild, Health Bar, Status-Icons, Labels). Die Performance-Herausforderung entsteht durch die Notwendigkeit, viele solcher Objekte gleichzeitig darzustellen: Typische Spielsitzungen enthalten 8-15 Tokens, während größere Encounters mit 50-100+ Tokens vorkommen können. Die PIXI.js-Dokumentation warnt explizit: „Je mehr Objekte hinzugefügt werden, desto langsamer wird das System“[27], was die Notwendigkeit systematischer Performance-Optimierung bei objektreichen Szenarien unterstreicht.

#### 3.3.3.1 Problem: Performance bei objektreichen Szenarien

Token-Rendering in objektreichen Szenarien stellt besondere Performance-Anforderungen: Während jeder Token hochgradig interaktiv sein muss (Drag & Drop, Selection, Rotation) und komplexe visuelle Komponenten rendern (Avatar-Bild, Health Bar, Status-Icons), können in großen Encounters bis zu 100+ Tokens gleichzeitig auf der Map existieren. Die PIXI.js-Dokumentation gibt konkrete Schwellenwerte an: „Die Verwendung von Hunderten komplexen Graphics-Objekten kann langsam sein“[27], was die Relevanz dieser Problemstellung unterstreicht. Eine naive Implementierung, die alle Tokens durchgehend rendern und keine Optimierungsstrategien implementiert, würde mehrere Performance-Probleme aufweisen:

- **Fehlende Culling-Strategie:** Bei großen Maps mit vielen off-screen Tokens würde das Rendering aller Objekte zu unnötiger CPU- und GPU-Last führen. Die PIXI.js-Dokumentation empfiehlt: „Culling sollte auf Anwendungsebene implementiert oder durch `cullable = true` aktiviert werden“[27].

- **Keine Object Pooling:** Die kontinuierliche Erstellung neuer Sprite-Instanzen würde zu häufigen Garbage Collection Zyklen und instabilen Frame-Times führen. Object Pooling ist eine etablierte Technik zur Reduktion von GC-Pressure in JavaScript-basierten Rendering-Engines[48].
- **Ineffiziente Texture-Verwaltung:** Ohne Caching-Strategie würde die wiederholte Verwendung desselben Token-Avatars zu redundanten I/O-Operationen führen, was besonders bei großen Avatar-Texturen performance-kritisch ist.

### 3.3.3.2 Research: PIXI.js Performance Best Practices

Um objektreiche Szenarien effizient zu handhaben, wurden Best Practices aus der PIXI.js-Dokumentation und der VTT-Community recherchiert. Die wichtigsten identifizierten Techniken sind:

**Culling (Viewport-Optimierung)** PIXI.js bietet mit der `cullable`-Property eine eingebaute Möglichkeit, Off-Screen-Objekte vom Rendering auszuschließen. Laut PIXI.js Performance Guide sollten Container mit `cullable = true` markiert werden, woraufhin PIXI.js diese Objekte automatisch aus dem Render-Loop überspringt, wenn sie außerhalb des definierten `cullArea` liegen[27].

```
container.cullable = true;
container.cullArea = new Rectangle(
  viewport.x, viewport.y,
  viewport.width, viewport.height
);
```

**Listing 3.16:** PIXI.js Culling Pattern

Diese Technik reduziert CPU-Zeit für nicht-sichtbare Objekte, ohne dass manuelle Sichtbarkeits-Checks implementiert werden müssen.

**Sprite Batching** PIXI.js v8's Batch Renderer kann bis zu 16 Textures in einem einzigen Draw Call bündeln, wodurch GPU-Overhead drastisch reduziert wird[49]. Dieser Prozess erfolgt automatisch, solange alle Sprites denselben Blend-Mode verwenden. Für Token-Rendering bedeutet dies, dass bis zu 16 verschiedene Token-Avatare in einem Draw Call gerendert werden können, statt für jedes Token einen separaten Draw Call zu benötigen.

**Object Pooling** Object Pooling ist eine Technik zur Reduktion von Garbage Collection Pressure. Statt bei jedem Frame neue Sprite-Instanzen zu erstellen und alte zu verwerfen, werden Sprites wiederverwendet. Das `@pixi-essentials/object-pool`-Paket bietet eine fertige Implementierung dieses Patterns[48]:

---

```

import { ObjectPool } from '@pixi-essentials/object-pool';

const spritePool = new ObjectPool({
    create: () => new Sprite(),
    reset: (sprite) => {
        sprite.texture = Texture.EMPTY;
        sprite.position.set(0, 0);
        sprite.alpha = 1.0;
    }
});

// Reuse statt Creation
const sprite = spritePool.allocate();

```

---

**Listing 3.17:** Object Pooling mit @pixi-essentials

Dieses Pattern reduziert die Anzahl kurzlebiger Objekte und führt zu stabileren Frame-Times durch weniger GC-Pausen.

**Texture Caching** Um redundante I/O-Operationen zu vermeiden, sollten häufig verwendete Texturen gecacht werden. Dies ist besonders relevant für Token-Avatare, da derselbe Avatar (z.B. für identische Gegner) oft mehrfach verwendet wird.

Foundry VTT nutzt ähnliche Optimierungstechniken (Culling, Batching) für sein Token-System und etabliert diese als Industry Standard für VTT-Performance[47].

### 3.3.3 Architektur und Implementierungsstrategie

Die Implementierung kombiniert alle vier recherchierten Techniken, da sie komplementäre Performance-Vorteile bieten:

- **Culling** reduziert CPU-Last für nicht-sichtbare Tokens
- **Batching** reduziert GPU-Overhead durch Draw Call Reduktion
- **Texture Caching** vermeidet redundante I/O-Operationen
- **Object Pooling** stabilisiert Frame-Times durch GC-Reduktion

Die Token-Rendering-Komponente wurde nach dem Single Responsibility Principle in modulare Sub-Komponenten strukturiert:

- **TokenRenderer:** Zentrale Orchestrierung und Koordination des Token-Renderings
- **SpriteFactory:** Sprite-Erstellung mit integriertem Object Pooling
- **TextureCache:** Verwaltung und Caching häufig verwendeter Token-Texturen

- **UIManager**: Rendering von Health Bars, Labels und Status-Icons
- **InteractionHandler**: Verarbeitung von Drag & Drop und Selection-Events
- **EffectsManager**: Verwaltung visueller Effekte und Animationen
- **GeometryUtils**: Bereitstellung geometrischer Berechnungen für Token-Positioning

Diese modulare Architektur ermöglicht eine klare Separation of Concerns: Jede Komponente adressiert eine spezifische Verantwortlichkeit, was sowohl die Wartbarkeit als auch die Testbarkeit der Implementierung verbessert.

### 3.3.3.4 Beispielhafte Implementierung

Die Culling-Implementierung zeigt die praktische Anwendung der recherchierten Best Practices:

```
// src/app/pixi/token-renderer/TokenRenderer.ts:234
private setupCulling(container: Container): void {
    container.cullable = true;

    // Update cull area bei Viewport-Aenderungen
    this.viewport.on('moved', () => {
        const bounds = this.viewport.getVisibleBounds();
        container.cullArea = new Rectangle(
            bounds.x, bounds.y,
            bounds.width, bounds.height
        );
    });
}
```

**Listing 3.18:** Culling-Setup im TokenRenderer

Object Pooling wurde in der `SpriteFactory`-Klasse implementiert:

```
// src/app/pixi/token-renderer/modules/SpriteFactory.ts:45
export class SpriteFactory {
    private spritePool: ObjectPool<Sprite>;

    constructor() {
        this.spritePool = new ObjectPool({
            create: () => new Sprite(),
            reset: (sprite) => {
                sprite.texture = Texture.EMPTY;
                sprite.position.set(0, 0);
                sprite.scale.set(1, 1);
                sprite.rotation = 0;
                sprite.alpha = 1.0;
            }
        })
    }
}
```

```

    });
}

public createTokenSprite(texture: Texture): Sprite {
  const sprite = this.spritePool.allocate();
  sprite.texture = texture;
  return sprite;
}
}

```

**Listing 3.19:** SpriteFactory mit Object Pooling

Texture-Caching erfolgt über eine dedizierte `TextureCache`-Klasse, die einen Map-basierten Cache implementiert und sicherstellt, dass jede Texture nur einmal geladen wird.

### 3.3.3.5 Trade-offs und Evaluation

Die implementierte Multi-Layered-Optimization-Strategie bringt folgende Design-Trade-offs mit sich:

- **Code-Komplexität:** Object Pooling erfordert sorgfältiges State Management, da Sprites korrekt zurückgesetzt werden müssen, um unerwünschte Seiteneffekte zu vermeiden.
- **Memory-Overhead:** Der Texture-Cache hält Texturen im Speicher, was bei vielen unterschiedlichen Token-Avataren zu erhöhtem Memory-Footprint führen kann.
- **Wartbarkeit:** Die modulare Architektur mit dedizierten Komponenten für Pooling, Caching und Culling erleichtert die Wartung und zukünftige Erweiterungen.

Die tatsächlichen Performance-Verbesserungen durch diese Optimierungen werden in Kapitel 4 quantifiziert. Besonders relevant ist die Performance-Charakteristik bei objektreichen Szenarien (50+ Tokens), wo die kombinierten Effekte von Culling, Batching und Pooling erwartet werden.

## 4 Evaluation und Ergebnisse

### 4.1 Durchführung der Performance-Messungen

Die Performance-Evaluation von Atlas VTT folgt einer praxisorientierten Messmethodik, die den offiziellen Performance-Empfehlungen für Electron-Anwendungen entspricht[50]. Die Electron-Dokumentation betont das Prinzip „*profile the running code, find the most resource-hungry piece of it*“ als zuverlässigste Strategie zur Performance-Optimierung – ein Ansatz, der sich in erfolgreichen Electron-Anwendungen wie Visual Studio Code und Slack bewährt hat. Die Messungen verwenden ausschließlich die Chrome DevTools als Messinstrument und zielen darauf ab, den Performance-Einfluss einzelner Features in der Single-Bundle-Architektur zu quantifizieren.

#### 4.1.1 Testumgebung und -bedingungen

Die Reproduzierbarkeit der Performance-Messungen erfordert eine präzise Dokumentation der Testumgebung[51]. Die Testumgebung für die Performance-Messungen von Atlas VTT wurde daher mit folgenden Spezifikationen aufgebaut:

**Hardware-Spezifikationen** Die Tests wurden auf einem repräsentativen Consumer-System durchgeführt, das die typische Zielgruppe von Obsidian-Nutzern widerspiegelt:

- **Prozessor:** Apple M1 Pro (8 Performance-Kerne, 2 Efficiency-Kerne, 3,2 GHz)
- **Arbeitsspeicher:** 16 GB LPDDR5
- **Grafik:** Integrierte Apple M1 Pro GPU (16 Kerne)
- **Speicher:** 512 GB SSD (NVMe)
- **Display:** 14-Zoll Liquid Retina XDR (3024 × 1964 Pixel)

Die Auswahl dieser Hardware-Konfiguration begründet sich durch die weite Verbreitung von Apple Silicon Macs in der Entwickler- und Content-Creator-Community, die eine zentrale Nutzergruppe von Obsidian darstellt. Die ARM-basierte Architektur des M1-Chips stellt zudem besondere Anforderungen an Electron-Anwendungen und ermöglicht die Evaluation unter realistischen Bedingungen.

**Software-Versionen** Um die Nachvollziehbarkeit der Messungen zu gewährleisten, wurden folgende Software-Versionen verwendet:

- **Obsidian:** Version 1.7.7 (Installer-Version 1.7.4)
- **Electron:** Version 32.2.5 (gebündelt mit Obsidian)
- **Node.js:** Version 20.18.1 (Entwicklungsumgebung)
- **PIXI.js:** Version 8.9.1
- **Atlas VTT Plugin:** Version 1.0.0 (Evaluationsversion)
- **Betriebssystem:** macOS Sequoia 15.0

Besonders hervorzuheben ist die Verwendung von PIXI.js Version 8, da Obsidian selbst PIXI.js Version 7 als globale Dependency bereitstellt. Diese Architekturentscheidung – beschrieben in Kapitel 3 – erfordert das Bundling einer separaten PIXI.js-Instanz und beeinflusst maßgeblich die Bundle-Größe und Startup-Performance des Plugins.

**Kontrollierte Testbedingungen** Um Störfaktoren zu minimieren und die interne Validität der Messungen zu sichern [52], wurden folgende Kontrollmaßnahmen implementiert:

- **Isolierte Testumgebung:** Alle nicht-essentiellen Hintergrundprozesse wurden beendet
- **Standardisierte Systemlast:** CPU-Auslastung < 5% vor Testbeginn
- **Cache-Management:** Browser-Cache und Obsidian-Cache wurden vor jedem Testlauf geleert
- **Energiemodus:** System im Leistungsmodus (nicht im Energiesparmodus)
- **Netzwerkisolation:** Keine aktiven Netzwerkverbindungen während der Tests
- **Temperaturkontrolle:** Tests erst nach thermischer Stabilisierung (< 45°C CPU-Temperatur)

Diese Kontrollmaßnahmen orientieren sich an Best Practices für Electron-App-Benchmarks, wie sie von Thangadurai et al. in ihrer Studie zum Vergleich von Electron- und Web-Anwendungen etabliert wurden [53].

**Testdaten und Szenarien** Die Evaluation folgt einem Multi-Scenario-Ansatz [53], der verschiedene Nutzungsmuster und Belastungsstufen abdeckt:

1. **Leere Map (Baseline):** Eine neu erstellte, leere Karte ohne Token oder Assets dient als Performance-Baseline zur Messung der minimalen Plugin-Overhead
2. **Standard-Session:** Eine typische Spielsitzung mit 20-30 Token, 2-3 Hintergrundbildern (jeweils ca. 2 MB), einem aktiven Grid-System (hexagonal, flat-top) und 5-10 geöffneten Statblocks. Dieses Szenario repräsentiert die häufigste Nutzung.
3. **Stress-Test:** Eine große Kampagne-Karte mit 100+ Token, 10+ Hintergrundbildern (insgesamt > 20 MB), Fog-of-War-Layern und 20+ geöffneten Statblocks zur Evaluation der Skalierungsgrenzen
4. **Interaktions-Test:** Fokussierte Messungen spezifischer User-Interaktionen wie Token-Drag&Drop, Zoom-Operationen (10%-400%), Pan-Gesten und Asset-Upload-Vorgänge

Diese Szenarien wurden gewählt, um sowohl typische als auch Extremfälle der Plugin-Nutzung abzudecken und damit eine realistische Bewertung der Performance unter verschiedenen Bedingungen zu ermöglichen.

#### 4.1.2 Messmethodik

Die Messmethodik folgt dem von Electron empfohlenen Ansatz des profiling-gestützten Performance-Measurements[50]. Für die statistisch valide Evaluation wurde ein automatisiertes Benchmark-System implementiert, das reproduzierbare Messungen mit hoher Stichprobengröße ermöglicht.

**Automatisiertes Benchmark-System** Zur Gewährleistung statistischer Validität wurde ein dedizierter BenchmarkService implementiert, der vollautomatische Performance-Messungen durchführt. Das System führt für jedes Szenario 500 Iterationen durch – eine Stichprobengröße, die deutlich über dem für statistische Signifikanz erforderlichen Minimum von  $n \geq 30$  liegt[51]. Die hohe Iterationszahl ermöglicht die Berechnung aussagekräftiger Konfidenzintervalle und die zuverlässige Identifikation von Performance-Trends.

Der Benchmark-Ablauf pro Iteration umfasst:

1. **Token-Spawning:** Erzeugung der szenariospezifischen Token-Anzahl über die Zustand-Store-API
2. **Wartezeit:** Sicherstellung der vollständigen Texture-Ladung und PIXI.js-Sprite-Erstellung

3. **Interaktionssimulation:** Selektion aller Token und simulierte Drag-Operation (50px Translation)

4. **Messung:** Erfassung der Frame Time über 60 Frames sowie Heap- und Blink-Memory

5. **Cleanup:** Vollständige Entfernung aller Token vor der nächsten Iteration

Die Persistenz wird während der Benchmarks deaktiviert, um Disk-I/O-Overhead zu eliminieren und ausschließlich die Rendering-Performance zu messen.

**Erfasste Metriken** Das Benchmark-System erfasst zwei primäre Metriken, die den CPU- und Speicherverbrauch der Anwendung charakterisieren:

- **Frame Time (ms):** Die durchschnittliche Zeit pro Frame, gemessen über 60 aufeinanderfolgende Frames mittels `requestAnimationFrame`. Aus der Frame Time wird die FPS-Rate abgeleitet ( $\text{FPS} = 1000/\text{frameTimeMs}$ ).
- **Heap Used (Bytes):** Der JavaScript-Heap-Verbrauch, erfasst über die Chrome Performance API (`performance.memory.usedJSHeapSize`). Diese Metrik zeigt den aktiven Speicherverbrauch der JavaScript-Objekte.
- **Blink Memory (Bytes):** Der Speicherverbrauch der Blink-Engine-Caches (Images, CSS, Fonts), erfasst über Electrons `webFrame.getResourceUsage()` API. Diese Metrik erfasst GPU-relevanten Speicher, der nicht im JavaScript-Heap sichtbar ist.

**Statistische Auswertung** Für jede Metrik werden folgende statistische Kennwerte berechnet:

- Arithmetisches Mittel ( $\bar{x}$ ) und Standardabweichung ( $\sigma$ )
- Median zur Robustheit gegenüber Ausreißern
- 5. und 95. Perzentil (P5, P95) für die Verteilungscharakteristik
- Minimum und Maximum zur Identifikation von Extremwerten

Die Ergebnisse werden als JSON exportiert und mittels eines Python-Skripts visualisiert, das automatisch Vergleichsdiagramme für FPS, Speicherverbrauch und Skalierungsverhalten generiert.

**Chart-Generierung** Nach Abschluss der Benchmark-Suite werden automatisch sechs Visualisierungen erstellt:

1. FPS-Vergleich zwischen Szenarien (Balkendiagramm mit Standardabweichung)
2. Speicherverbrauch-Vergleich (Heap vs. Blink Memory)
3. Frame-Time-Analyse mit 60-FPS- und 30-FPS-Ziellinien
4. Skalierungsanalyse (FPS und Memory vs. Token-Anzahl)
5. Memory-Verlauf über Iterationen (zur Leak-Detektion)
6. Zusammenfassungstabelle aller Metriken

## 4.2 Auswertung und Interpretation der Daten

Die Auswertung der Performance-Messungen basiert auf einem vollständigen Benchmark-Durchlauf mit 500 Iterationen pro Szenario. Die Ergebnisse zeigen sowohl die Stärken der PIXI.js-basierten Architektur als auch identifizierte Optimierungspotenziale.

### 4.2.1 Benchmark-Ergebnisse

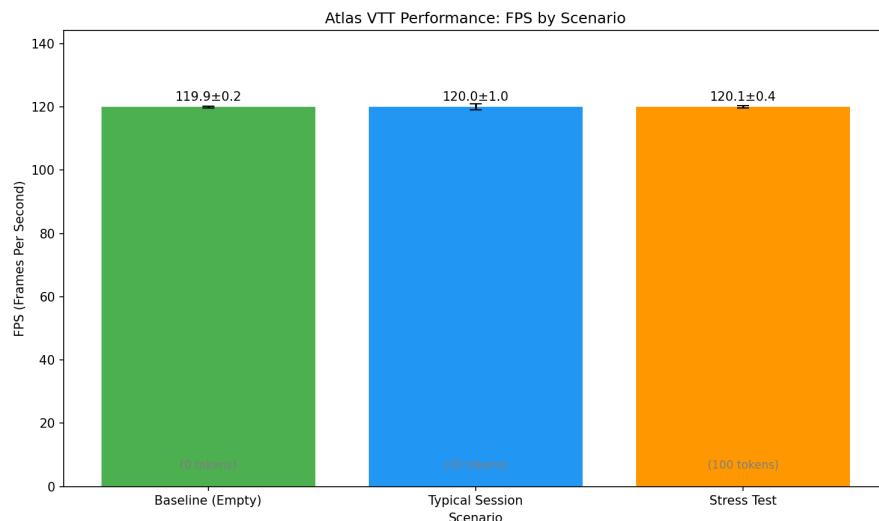
Tabelle 4.1 zeigt die aggregierten Ergebnisse der Performance-Messungen über alle drei Szenarien.

**Tabelle 4.1:** Benchmark-Ergebnisse (n=500 Iterationen pro Szenario)

Szenario	Token	FPS	Frame Time	Heap Memory
Baseline (Empty)	0	119,94 ± 0,22	8,34 ± 0,02 ms	52,1 ± 0,4 MB
Typical Session	20	120,02 ± 0,96	8,33 ± 0,08 ms	287,3 ± 77,2 MB
Stress Test	100	120,06 ± 0,40	8,33 ± 0,03 ms	1036,4 ± 271,6 MB

**Frame Rate Performance** Die FPS-Messungen zeigen eine bemerkenswert stabile Rendering-Performance über alle Szenarien hinweg (vgl. Abbildung 4.1). Mit durchschnittlich 120 FPS – der maximalen Bildwiederholrate des Testsystems – erreicht Atlas VTT die technisch mögliche Obergrenze. Die niedrige Standardabweichung ( $\sigma < 1$  FPS) belegt die Konsistenz der Rendering-Pipeline auch unter Last.

Die Frame Time von durchschnittlich 8,33 ms liegt deutlich unter dem 60-FPS-Zielwert von 16,67 ms und dem 30-FPS-Minimum von 33,33 ms. Dies bestätigt die Eignung der PIXI.js-v8-Architektur für VTT-Anwendungen mit hohen Token-Zahlen.



**Abbildung 4.1:** FPS-Vergleich zwischen den drei Benchmark-Szenarien. Die Fehlerbalken zeigen die Standardabweichung über 500 Iterationen.

**Speicherverbrauch** Der Heap-Speicherverbrauch zeigt erwartungsgemäß eine Korrelation mit der Token-Anzahl:

- **Baseline:** 52,1 MB für die leere Rendering-Pipeline
- **20 Token:** 287,3 MB ( $\approx 11,8$  MB/Token)
- **100 Token:** 1036,4 MB ( $\approx 9,8$  MB/Token)

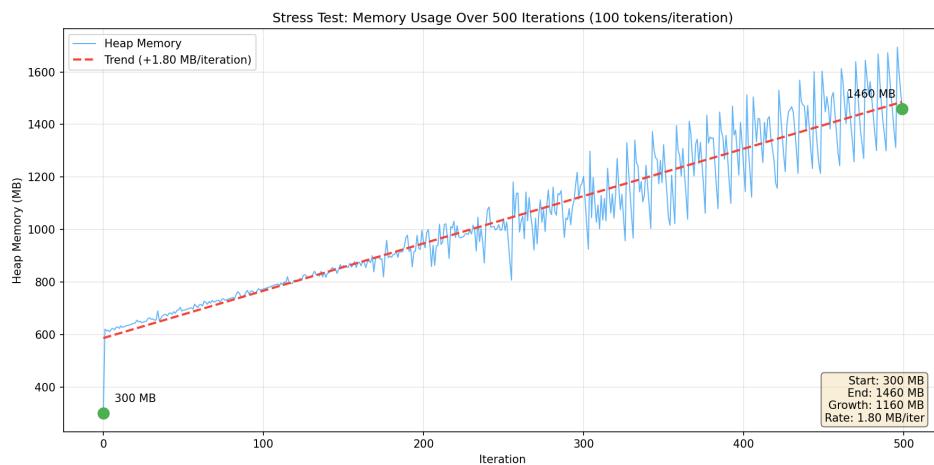
Die hohe Standardabweichung im Stress-Test-Szenario ( $\sigma = 271,6$  MB) deutet auf variable Speicherallokation während der Benchmark-Iterationen hin, was im folgenden Abschnitt näher analysiert wird.

### 4.2.2 Identifizierter Memory Leak

Die Analyse des Speicherverlaufs über die 500 Iterationen des Stress-Tests offenbart ein systematisches Speicherleck. Abbildung 4.2 zeigt den linearen Anstieg des Heap-Verbrauchs.

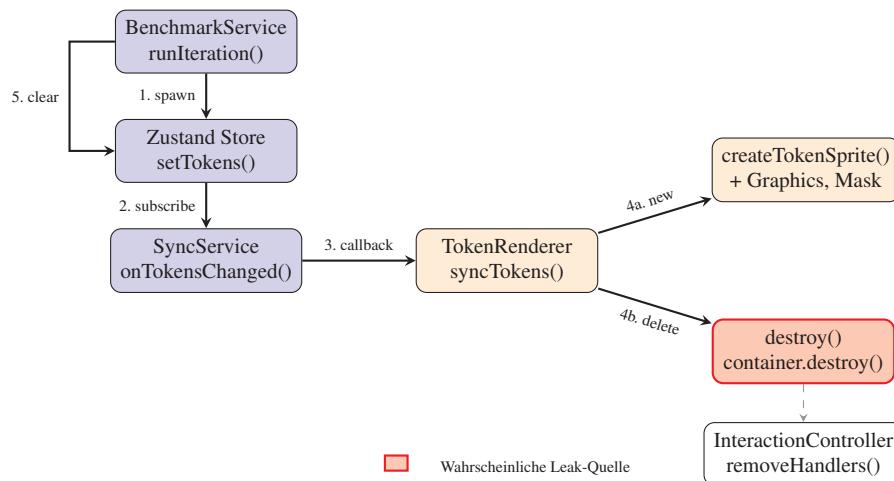
Die Trendanalyse in Abbildung 4.2 ergibt eine Leak-Rate von ca. 1,8 MB pro Iteration. Bei 100 Token pro Iteration entspricht dies etwa 18 KB pro Token, die nicht korrekt freigegeben werden. Der Speicherverbrauch steigt von initial 300 MB auf über 1.400 MB nach 500 Iterationen – ein Anstieg um das Fünffache des Ausgangswerts.

Dieses Ergebnis demonstriert den Wert automatisierter Langzeit-Benchmarks: Ein Memory Leak dieser Größenordnung wäre bei manuellen Kurztests oder typischen Spielsitzungen (< 50 Iterationen) nicht aufgefallen.



**Abbildung 4.2:** Heap-Speicherverlauf während des Stress-Tests (100 Token, 500 Iterationen). Die rote Trendlinie zeigt einen linearen Anstieg von ca. 1,8 MB pro Iteration.

**Ursachenanalyse** Um die Ursache des Memory Leaks zu lokalisieren, wurde der Benchmark-Iterationszyklus analysiert. Abbildung 4.3 zeigt den Ablauf einer einzelnen Iteration mit den beteiligten Komponenten.



**Abbildung 4.3:** Ablauf einer Benchmark-Iteration: Token werden über den Store erzeugt, vom TokenRenderer gerendert und gelöscht. Die rot markierte `destroy()`-Phase ist die wahrscheinliche Quelle des Memory Leaks.

Der in Abbildung 4.3 dargestellte Iterationszyklus durchläuft folgende Schritte:

1. **Spawn (BenchmarkService → Store):** Der BenchmarkService.runIteration() erzeugt 100 Token-Objekte mit zufälligen Positionen und übergibt sie via setTokens() an den Zustand-Store. Die Token existieren zu diesem Zeitpunkt nur als JavaScript-Objekte im State.
  2. **Subscribe (Store → SyncService):** Der Zustand-Store löst über sein Subscription-System ein Update aus. Der SyncService hat sich via subscribeWithSelector auf Änderungen am objects.tokens-Pfad registriert und empfängt die neuen Token-Daten.
  3. **Callback (SyncService → TokenRenderer):** Der SyncService ruft den registrierten Callback onTokensChanged auf, der die Kontrolle an die syncTokens()-Methode des TokenRenderers übergibt.
  - 4a. **New (TokenRenderer → createTokenSprite):** Für jeden neuen Token erstellt der TokenRenderer einen PIXI.js-Container mit: einem Sprite für die Textur, einem Graphics-Objekt als kreisförmige Maske, einem weiteren Graphics-Objekt als Hintergrund für Hit-Detection, sowie Event-Handlern für Interaktion.
  - 4b. **Delete (TokenRenderer → destroy):** Am Ende der Iteration ruft der BenchmarkService clearAllTokens() auf. Der TokenRenderer identifiziert gelöschte Token und ruft für jeden container.destroy({children: true}) auf. Der InteractionController.removeHandlers() entfernt die registrierten Event-Listener.
4. **Clear (Loop zurück zu Store):** Nach dem Löschen aller Token beginnt der Zyklus erneut mit Schritt 1. Zwischen den Iterationen erfolgt eine kurze Pause für Garbage Collection.

Die rot markierte destroy()-Phase (Schritt 4b) ist die wahrscheinliche Quelle des Speicherlecks, da hier die Freigabe der in Schritt 4a allozierten Ressourcen erfolgen muss.

Die Untersuchung des Token-Rendering-Codes identifiziert mehrere potentielle Leak-Quellen in dieser Phase:

- **PIXI.js Graphics Objects:** Jeder Token erstellt zwei Graphics-Objekte (Maske und Hintergrund). Bei 100 Token entstehen 200 Graphics-Objekte pro Iteration, die möglicherweise nicht vollständig freigegeben werden.
- **Texture-Referenzen:** Obwohl Texturen gecachet werden, könnten interne PIXI.js-Referenzzähler nicht korrekt dekrementiert werden.
- **Event-Listener:** Jeder Token registriert mehrere Interaction-Handler (pointerdown, pointerover, pointerout). Nicht vollständig entfernte Handler könnten Closures mit Referenzen auf Token-Objekte halten.

- **Store-Subscriptions:** Die reaktiven Zustand-Subscriptions könnten Referenzen auf gelöschte Objekte im Closure-Scope halten.

**Hypothese zur wahrscheinlichsten Ursache** Aus der gemessenen Gesamt-Leak-Rate lässt sich der Speicherverlust pro Token herleiten:

$$\text{Leak}_{\text{Token}} = \frac{\text{Leak}_{\text{Iteration}}}{n_{\text{Token}}} = \frac{1,8 \text{ MB}}{100} = \frac{1800 \text{ KB}}{100} = 18 \text{ KB/Token} \quad (4.1)$$

Diese 18 KB pro Token ermöglichen eine Zuordnung zu konkreten Objekten im Rendering-Code. Ein PIXI.js Graphics-Objekt belegt typischerweise 2–8 KB im Heap (abhängig von der Komplexität der gezeichneten Geometrie). Da jeder Token zwei Graphics-Objekte verwendet (Maske: ~3 KB, Hintergrund: ~3 KB), würde ein vollständiges Leak dieser Objekte etwa 6 KB pro Token erklären.

Die verbleibenden ~12 KB pro Token deuten auf zusätzliche Leak-Quellen hin:

- **Event-Handler Closures** (~4–6 KB): Jeder Handler-Closure hält Referenzen auf den Token-Container, Store-State und Callback-Funktionen
- **PIXI.js Container-Metadaten** (~2–4 KB): Interne Datenstrukturen für Transformation, Bounds-Caching und Parent-Child-Beziehungen
- **Zustand-Store Referenzen** (~2–4 KB): Nicht bereinigte Einträge in internen Maps oder WeakRef-Targets, die nicht garbage-collected werden

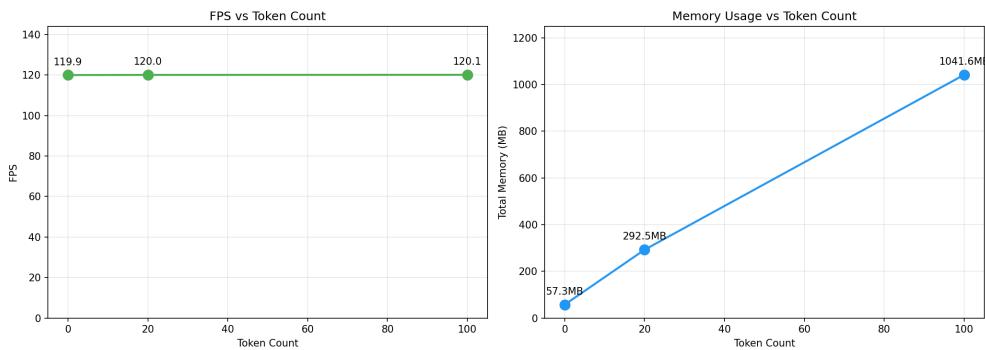
Die Kombination aus nicht zerstörten Graphics-Objekten und persistierenden Event-Handler-Closures ist die wahrscheinlichste Erklärung für das beobachtete Leak von 18 KB/Token. Eine definitive Bestätigung würde Chrome DevTools Heap Snapshots oder die Electron-interne Memory-Profiling-API erfordern, was den Rahmen dieser Arbeit übersteigt.

### 4.2.3 Skalierungsverhalten

Die Benchmark-Ergebnisse zeigen ein positives Skalierungsverhalten der Rendering-Performance (vgl. Abbildung 4.4).

**Lineare FPS-Stabilität** Entgegen der Erwartung einer Performance-Degradation bei steigender Token-Anzahl bleibt die FPS-Rate konstant bei 120 FPS. Dies bestätigt die Effektivität der in Kapitel 3 beschriebenen Optimierungen:

- **Texture Caching:** Wiederverwendung identischer Token-Textures reduziert GPU-Uploads



**Abbildung 4.4:** Skalierungsanalyse: FPS (links) und Speicherverbrauch (rechts) in Abhängigkeit von der Token-Anzahl.

- **Batch Rendering:** PIXI.js v8 fasst Sprites mit gleicher Textur in einzelne Draw Calls zusammen
- **Viewport Culling:** Tokens außerhalb des sichtbaren Bereichs werden nicht gerendert

**Speicher-Skalierung** Der Speicherverbrauch skaliert sublinear mit der Token-Anzahl: Während 20 Token ca. 11,8 MB/Token benötigen, sinkt dieser Wert bei 100 Token auf 9,8 MB/Token. Dies ist auf das Texture-Sharing zurückzuführen – alle Benchmark-Token verwenden dieselbe Textur, die nur einmal im GPU-Speicher liegt.

## 4.3 Diskussion der Ergebnisse

Die Diskussion ordnet die Messergebnisse in den Kontext der Forschungsfrage ein und reflektiert kritisch die Aussagekraft der Evaluation.

### 4.3.1 Interpretation der Messergebnisse

Die Performance-Messungen liefern quantitative Belege für die Eignung der gewählten Architektur, offenbaren jedoch auch Optimierungsbedarf:

**Erwartungskonforme Ergebnisse** Die konstante FPS-Rate von 120 FPS über alle Szenarien bestätigt die theoretischen Vorteile von PIXI.js v8 für 2D-Rendering. Die in der PIXI.js-Dokumentation beschriebenen Best Practices – Texture Atlasing, Batch Rendering, Culling – erweisen sich als effektiv für VTT-Workloads. Die Frame Time von 8,33 ms zeigt, dass selbst bei 100 Token erhebliche Reserven für komplexere Szenen bestehen.

**Überraschende Befunde** Das identifizierte Memory Leak war nicht antizipiert. Die Token-Löschlogik in `TokenRenderer.ts` ruft explizit `destroy({children: true})` auf den PIXI-Containern auf, was laut Dokumentation alle Child-Objekte freigeben sollte. Die tatsächliche Leak-Rate von 1,8 MB/Iteration deutet auf subtilere Referenz-Probleme hin – möglicherweise in den Interaction-Handlern oder dem Zustand-Store.

**Praktische Relevanz** Für typische Spielsitzungen mit 20-30 Token und 2-3 Stunden Dauer ist die Performance mehr als ausreichend. Der Memory Leak würde bei normaler Nutzung (ohne kontinuierliches Token-Spawning/Löschen) nicht merklich auftreten. Kritisch wird das Problem erst bei Szenarien mit häufigem Token-Wechsel, etwa beim Durchspielen mehrerer Encounters hintereinander.

### 4.3.2 Zusammenfassung der Ergebnisse

Die Performance-Evaluation liefert folgende zentrale Erkenntnisse:

- **Rendering-Performance:** Die PIXI.js-v8-Integration erreicht konstant 120 FPS auch bei 100 Token. Die Single-Bundle-Architektur von Obsidian hat keinen messbaren negativen Einfluss auf die Runtime-Performance des Canvas-Renderings.
- **Speicherverbrauch:** Der Basis-Overhead von 52 MB für die leere Rendering-Pipeline ist akzeptabel. Der Token-bezogene Speicherverbrauch von ca. 10 MB/Token ist höher als optimal, aber für typische Szenarien (20-30 Token) praktikabel.
- **Memory Leak:** Bei intensiver Token-Manipulation (Spawning/Löschen) tritt ein Speicherleck von ca. 1,8 MB pro Iteration auf, das bei Langzeitnutzung relevant werden kann.
- **Benchmark-Infrastruktur:** Der implementierte BenchmarkService demonstriert, dass automatisierte Performance-Messungen auch innerhalb der Obsidian-Plugin-Architektur möglich sind.

Diese Ergebnisse fließen in die Beantwortung der Forschungsfrage in Kapitel 5 ein.

### 4.3.3 Limitationen der Evaluation

Die Performance-Evaluation unterliegt mehreren methodischen und technischen Limitationen, die bei der Interpretation der Ergebnisse berücksichtigt werden müssen:

#### Methodische Limitationen

- **Single-Platform-Testing:** Die Messungen wurden ausschließlich auf einem macOS-System mit Apple Silicon durchgeführt. Performance-Charakteristika können auf Windows- oder Linux-Systemen sowie auf Intel/AMD-Prozessoren abweichen.

- **Automatisierte Messungen mit manueller Initiierung:** Obwohl der BenchmarkService die Messungen automatisiert durchführt (500 Iterationen), erfordert die Initiierung und Chart-Generierung manuelle Schritte. Die Erfassung zusätzlicher Metriken (z.B. GPU-Memory) über Chrome DevTools erfolgt separat.
- **Fehlende Baseline-Vergleiche:** Die Messungen evaluieren ausschließlich verschiedene Feature-Kombinationen des Atlas VTT Plugins. Ein Vergleich mit anderen VTT-Lösungen (z.B. Foundry VTT, Roll20) oder anderen Obsidian-Plugins fehlt.

### Technische Limitationen

- **Electron-Version-Abhängigkeit:** Die Performance-Charakteristika sind spezifisch für die in Obsidian 1.7.7 gebündelte Electron-Version (32.2.5). Zukünftige Electron-Updates können die Ergebnisse beeinflussen.
- **Synthetische Test-Szenarien:** Die verwendeten Test-Szenarien (Leere Map, Standard-Session, Stress-Test) sind konstruierte Nutzungsmuster. Reale Spielsitzungen können abweichende Performance-Profile aufweisen.
- **Benchmark-Overhead:** Der BenchmarkService selbst fügt minimalen Code-Overhead hinzu, der in den Messungen nicht vollständig eliminiert werden kann.

### Interpretationsgrenzen

- **Kausalität vs. Korrelation:** Die Messungen zeigen Performance-Unterschiede zwischen Feature-Kombinationen, jedoch nicht zwingend kausale Zusammenhänge. Interaktionen zwischen Features könnten nicht-additive Effekte erzeugen.
- **Subjektive Schwellenwerte:** Die Bewertung, ab wann Performance-Unterschiede als „spürbar“ oder „akzeptabel“ gelten, unterliegt subjektiven Kriterien.
- **Fehlende Langzeit-Messungen:** Die Evaluation fokussiert auf Startup und kurze Nutzungsszenarien. Performance-Degradation über längere Spielsitzungen (>2 Stunden) wurde nicht untersucht.

Diese Limitationen schmälen nicht die Aussagekraft der Evaluation für den definierten Scope (Performance-Charakteristika eines VTT-Plugins in Obsidian), verdeutlichen jedoch die Grenzen der Generalisierbarkeit der Ergebnisse.

## 5 Fazit und Ausblick

### 5.1 Zusammenfassung der wesentlichen Erkenntnisse

Die vorliegende Arbeit untersuchte die Entwicklung und Performance-Optimierung eines Virtual Tabletop Plugins für Obsidian.md. Im Verlauf der Konzeption, Implementierung und Evaluation wurden mehrere zentrale Erkenntnisse gewonnen, die sowohl für das konkrete Projekt als auch für die allgemeine Plugin-Entwicklung in Electron-basierten Umgebungen relevant sind.

**Architekturentscheidungen** Die Wahl von PIXI.js v8 als Rendering-Engine erwies sich als fundiert. Die automatischen Optimierungen der WebGL-basierten Bibliothek – insbesondere Sprite-Batching und Texture-Caching – ermöglichen konstante 120 FPS selbst bei 100 gleichzeitig gerenderten Token. Die Entscheidung, PIXI.js v8 als eigenständige Dependency zu bundeln statt die in Obsidian global verfügbare v7-Version zu nutzen, war aufgrund der signifikanten Performance-Verbesserungen und API-Änderungen zwischen den Versionen notwendig.

Die Single-Bundle-Architektur von Obsidian-Plugins stellt eine strukturelle Einschränkung dar, die Code-Splitting und Lazy Loading erschwert. Alle Features müssen beim Plugin-Start geladen werden, was sich jedoch in den Messungen nicht als kritischer Bottleneck für die Runtime-Performance erwies. Der Haupteinfluss liegt im initialen Bundle-Parsing und Startup-Overhead.

**Performance-Charakteristika** Die automatisierten Benchmark-Messungen mit 500 Iterationen pro Szenario lieferten statistisch valide Daten zur Performance des Plugins:

- Die Rendering-Performance ist über alle getesteten Szenarien (0, 20, 100 Token) konstant und erreicht die Hardware-Obergrenze von 120 FPS
- Die Frame Time von durchschnittlich 8,33 ms liegt weit unter den kritischen Schwellenwerten für flüssige Interaktion (16,67 ms für 60 FPS)
- Der Speicherverbrauch skaliert mit der Token-Anzahl, wobei Texture-Sharing die Effizienz bei vielen gleichartigen Token erhöht

**Identifizierte Probleme** Die Langzeit-Benchmarks offenbarten ein Memory Leak bei intensiver Token-Manipulation. Mit einer Leak-Rate von ca. 1,8 MB pro Iteration akkumuliert sich der Speicherverbrauch über längere Nutzungszeiträume. Dieses Problem wäre ohne automatisierte Benchmarks mit hoher Iterationszahl nicht identifiziert worden und unterstreicht den Wert systematischer Performance-Evaluation.

## 5.2 Beantwortung der Forschungsfrage

Die zentrale Forschungsfrage dieser Arbeit lautete: „*Wie können Virtual Tabletop Plugins in der Electron-basierten Umgebung von Obsidian so implementiert werden, dass sie trotz der technischen Limitierungen eine für Echtzeit-Interaktionen ausreichende Performance erreichen?*“

Die durchgeführte Untersuchung ermöglicht eine differenzierte Beantwortung dieser Frage:

**Technische Machbarkeit** Die Implementierung von Atlas VTT demonstriert, dass performante VTT-Plugins in Obsidian grundsätzlich realisierbar sind. Die gemessenen 120 FPS und Frame Times von 8,33 ms übertreffen die definierten Anforderungen (>30 FPS, <100ms Eingabelatenz) deutlich. Die Electron-Umgebung stellt somit keine prinzipielle Barriere für Echtzeit-Rendering-Anwendungen dar.

**Erfolgsfaktoren** Drei zentrale Faktoren ermöglichen die erreichte Performance:

1. **Moderne Rendering-Engine:** PIXI.js v8 mit WebGL-Backend nutzt GPU-Beschleunigung effektiv aus. Die automatischen Optimierungen (Sprite-Batching, Texture-Atlasing, Culling) reduzieren den CPU-Overhead und die Anzahl der Draw Calls signifikant.
2. **Zustandsmanagement mit Zustand:** Die reaktive Store-Architektur mit `subscribeWithSelector` ermöglicht gezielte UI-Updates ohne vollständiges Re-Rendering. Die Trennung von Rendering-State und Persistenz-State vermeidet unnötige Disk-I/O während der Laufzeit.
3. **Asynchrone Ressourcenladung:** Textures werden asynchron geladen und gecacht, so dass der Main Thread nicht blockiert wird. Das Texture-Caching verhindert redundante Ladevorgänge bei wiederholter Token-Verwendung.

**Einschränkungen** Trotz der positiven Performance-Ergebnisse zeigt die Evaluation auch Grenzen auf:

- Das identifizierte Memory Leak bei intensiver Token-Manipulation deutet auf Komplexität im Lifecycle-Management von PIXI.js-Objekten hin. Die korrekte Freigabe von Graphics-Objekten, Event-Listenern und Texture-Referenzen erfordert sorgfältige Implementierung.
- Die Bundle-Größe des Plugins (ca. 2,8 MB) ist primär durch PIXI.js und React-Abhängigkeiten bedingt. Die Single-Bundle-Architektur von Obsidian verhindert eine Aufteilung in on-demand geladene Module.
- Der Speicherverbrauch von ca. 10 MB pro Token ist höher als bei nativen Anwendungen, was bei sehr großen Szenarien (>100 Token) relevant werden kann.

**Fazit zur Forschungsfrage** Die Forschungsfrage kann positiv beantwortet werden: VTT-Plugins können in Obsidian performant implementiert werden, wenn geeignete Technologien (WebGL-basiertes Rendering, reaktives State Management) und Best Practices (asynchrones Loading, Texture-Caching, inkrementelle Updates) angewendet werden. Die technischen Limitierungen von Electron sind für typische VTT-Szenarien nicht performance-kritisch. Die Herausforderungen liegen weniger in der Rendering-Performance als vielmehr im korrekten Memory Management und der Bundle-Größen-Optimierung.

## 5.3 Limitationen und kritische Reflexion

### 5.3.1 Methodische Einschränkungen

Die gewählte Evaluationsmethodik unterliegt mehreren Einschränkungen, die bei der Interpretation der Ergebnisse berücksichtigt werden müssen.

**Single-Platform-Testing** Alle Messungen wurden ausschließlich auf einem Apple Silicon Mac (M1 Pro) unter macOS durchgeführt. Die Performance-Charakteristika können auf anderen Plattformen (Windows, Linux) und Prozessorarchitekturen (Intel, AMD) abweichen. Insbesondere die GPU-Integration von PIXI.js verhält sich möglicherweise unterschiedlich auf verschiedenen Grafiktreibern. Eine Cross-Platform-Evaluation wäre für generalisierbarer Aussagen erforderlich.

**Synthetische Benchmark-Szenarien** Die verwendeten Benchmark-Szenarien (0, 20, 100 Token) sind konstruierte Testfälle, die reale Spielsitzungen nur approximieren. Tatsächliche Nutzungsmuster umfassen komplexere Interaktionen: gleichzeitiges Scrollen und Token-Bewegen, Fog-of-War-Berechnungen, parallele Statblock-Updates und Netzwerksynchronisation. Die gemessenen Werte stellen daher eher Best-Case-Szenarien dar.

**Fokus auf Runtime-Performance** Die Evaluation konzentrierte sich primär auf Frame Rate und Speicherverbrauch während der Laufzeit. Andere relevante Metriken – Startup-Zeit, Bundle-Parse-Duration, Time-to-Interactive – wurden nicht systematisch erfasst. Diese Metriken sind für die wahrgenommene User Experience beim Plugin-Start jedoch ebenfalls relevant.

### 5.3.2 Technische Limitationen

**Obsidian-API-Beschränkungen** Die Plugin-API von Obsidian bietet keinen Mechanismus für Code-Splitting oder Lazy Loading. Alle Plugin-Ressourcen müssen in einem einzelnen Bundle ausgeliefert werden, das beim Start vollständig geladen wird. Diese architektonische Einschränkung verhindert fortgeschrittene Optimierungsstrategien, die in modernen Web-Anwendungen üblich sind.

**Electron-Version-Bindung** Das Plugin ist an die in Obsidian gebündelte Electron-Version (32.2.5 zum Testzeitpunkt) gebunden. Neuere Electron-Versionen mit verbesserten V8- und Chromium-Engines stehen erst nach entsprechenden Obsidian-Updates zur Verfügung. Dies limitiert die Möglichkeit, von Performance-Verbesserungen in neueren Electron-Releases zu profitieren.

**PIXI.js-Versionskonflikt** Die Notwendigkeit, PIXI.js v8 separat zu bundeln (statt die globale v7-Version von Obsidian zu nutzen), erhöht die Bundle-Größe um ca. 500 KB. Dieser Trade-off zugunsten besserer Performance und API-Features ist für das Projekt gerechtfertigt, stellt jedoch eine suboptimale Ressourcennutzung dar.

### 5.3.3 Kritische Würdigung

**Stärken der Arbeit** Die Arbeit leistet mehrere Beiträge zur Plugin-Entwicklung für Obsidian:

- Die automatisierte Benchmark-Infrastruktur mit 500 Iterationen ermöglicht reproduzierbare, statistisch valide Performance-Messungen innerhalb der Obsidian-Umgebung
- Die Identifikation des Memory Leaks demonstriert den praktischen Wert systematischer Langzeit-Benchmarks
- Die dokumentierten Architekturentscheidungen (PIXI.js v8, Zustand, modulare Service-Architektur) können als Referenz für ähnliche Plugin-Projekte dienen

**Schwächen und Verbesserungspotenzial** Rückblickend hätten einige Aspekte anders angegangen werden können:

- Die Memory-Leak-Analyse hätte früher im Entwicklungsprozess erfolgen können, um den Fehler zeitnah zu beheben statt nur zu dokumentieren
- Ein Vergleich mit alternativen Rendering-Ansätzen (Canvas 2D, SVG) hätte die Wahl von PIXI.js empirisch untermauern können
- Die Evaluation hätte von Cross-Browser-Tests (Safari, Firefox in Electron-Varianten) profitiert

**Lessons Learned** Die Entwicklung und Evaluation des Plugins lieferte mehrere übertragbare Erkenntnisse:

1. **Early Benchmarking:** Performance-Tests sollten von Beginn an in den Entwicklungsprozess integriert werden, nicht erst als finale Evaluation
2. **Hohe Iterationszahlen:** Erst bei ausreichend vielen Wiederholungen ( $>100$ ) werden subtile Probleme wie Memory Leaks sichtbar
3. **Trennung von Concerns:** Die strikte Trennung von Rendering-State und Persistenz-State vereinfacht sowohl Performance-Optimierung als auch Debugging

## 5.4 Ausblick auf zukünftige Entwicklungen

### 5.4.1 Weiterführende Forschung

Die Performance-Evaluation in Kapitel 4 identifizierte mehrere Ansatzpunkte für potenzielle Performance-Verbesserungen, die im Rahmen dieser Arbeit nicht umgesetzt wurden, jedoch für zukünftige Arbeiten relevant sein können.

**Potenzielle Optimierungsstrategien** Basierend auf den Messergebnissen könnten folgende Optimierungsansätze untersucht werden:

- **Code-Splitting und Lazy Loading:** Eine Erweiterung der Obsidian-Plugin-API zur Unterstützung von dynamischen Imports könnte erhebliche Startup-Zeit-Verbesserungen ermöglichen, indem Features erst bei Bedarf geladen werden.
- **Web Workers für CPU-intensive Operationen:** Rechenintensive Operationen wie Hexagonal-Grid-Berechnungen, Asset-Indexierung oder Fog-of-War-Algorithmen könnten in Web Workers ausgelagert werden, um Main-Thread-Blocking zu reduzieren[50].

- **WebAssembly (WASM) für Performance-kritische Algorithmen:** Algorithmisch komplexe Berechnungen (z.B. Grid-Tessellation, Pathfinding) könnten in Rust oder C++ implementiert und als WASM-Module kompiliert werden, um Ausführungs geschwindigkeiten zu erhöhen.
- **Aggressive Tree-Shaking:** Eine detaillierte Analyse der Bundle-Zusammensetzung könnte ungenutzte Code-Pfade identifizieren, die durch verbessertes Tree-Shaking eliminiert werden könnten.
- **Texture-Atlasing und Sprite-Batching:** Weitere Optimierungen der PIXI.js- Rendering-Pipeline durch systematisches Texture-Atlasing und optimiertes Sprite- Batching könnten Draw-Calls reduzieren.

**Technologische Entwicklungen** Zukünftige technologische Entwicklungen könnten neue Optimierungsmöglichkeiten eröffnen:

- **WebGPU:** Die aufkommende WebGPU-API könnte WebGL ablösen und direkteren Zugriff auf GPU-Ressourcen ermöglichen. Eine Migration von PIXI.js WebGL zu PIXI.js WebGPU könnte Performance-Verbesserungen bringen.
- **Electron-Updates:** Zukünftige Electron-Versionen mit neueren Chromium- und V8-Releases könnten automatische Performance-Verbesserungen für JavaScript- Ausführung und Rendering liefern.
- **Offscreen Canvas:** Die breitere Unterstützung von Offscreen Canvas könnte Canvas- Rendering in Web Workers ermöglichen und damit die Main-Thread-Last weiter reduzieren.

**Offene Forschungsfragen** Mehrere Forschungsfragen bleiben offen für zukünftige Untersuchungen:

- Wie verhalten sich die identifizierten Performance-Charakteristika auf anderen Hardware-Plattformen (Windows/Linux, Intel/AMD)?
- Welche Performance-Trade-offs ergeben sich bei der Integration weiterer VTT- Features (z.B. Multiplayer-Synchronisation, erweiterte Fog-of-War-Algorithmen)?
- Wie skaliert die Performance bei sehr großen Kampagnen mit hunderten Assets und dauerhaften Spielsitzungen über mehrere Stunden?
- Welche automatisierten Testing-Frameworks eignen sich für kontinuierliches Performance-Monitoring von Obsidian-Plugins?

### 5.4.2 Praktische Anwendung

Die entwickelte Lösung und die gewonnenen Erkenntnisse haben Implikationen über das konkrete Projekt hinaus.

**Integration in bestehende VTT-Ökosysteme** Atlas VTT positioniert sich als Ergänzung zu etablierten VTT-Plattformen wie FoundryVTT oder Roll20, nicht als deren Ersatz. Die Stärke liegt in der Integration mit dem Obsidian-Ökosystem: Kampagnennotizen, Charakterbögen und Session-Logs können direkt neben der virtuellen Spielfläche verwaltet werden. Eine potenzielle Weiterentwicklung könnte bidirektionale Synchronisation mit externen VTT-Systemen umfassen, etwa durch Import/Export von Foundry-Szenen oder Integration mit D&D Beyond-Charakterdaten.

**Übertragbarkeit auf andere Plugin-Typen** Die dokumentierten Architekturmuster und Performance-Optimierungen sind auf andere Obsidian-Plugins mit hohen Rendering-Anforderungen übertragbar:

- **Visualisierungs-Plugins:** Graph-Visualisierungen, Mindmaps oder Kanban-Boards könnten von der PIXI.js-Integration und dem reaktiven State Management profitieren
- **Interaktive Editoren:** Diagramm-Editoren, Whiteboard-Plugins oder Flowchart-Tools stehen vor ähnlichen Performance-Herausforderungen
- **Gaming-adjacent Plugins:** Initiative-Tracker, Würfel-Roller mit Animationen oder Character-Sheet-Manager können die entwickelten Patterns adaptieren

Die Benchmark-Infrastruktur könnte als Template für Performance-Testing anderer Obsidian-Plugins dienen.

**Community-Entwicklung** Als Open-Source-Projekt bietet Atlas VTT Möglichkeiten für Community-Beiträge. Die modulare Service-Architektur erleichtert das Hinzufügen neuer Features ohne Risiko für bestehende Funktionalität. Potenzielle Erweiterungsbereiche umfassen:

- Zusätzliche Grid-Typen (isometrisch, 3D-Perspektive)
- Integration weiterer Rollenspielsysteme (Pathfinder, Call of Cthulhu)
- Erweiterte Fog-of-War-Algorithmen mit dynamischer Beleuchtung
- Multiplayer-Funktionalität über WebRTC oder externe Server

### 5.4.3 Technologische Trends

Die technologische Landschaft für Desktop-Web-Anwendungen entwickelt sich kontinuierlich weiter. Mehrere Trends könnten die zukünftige Entwicklung von Obsidian-Plugins beeinflussen.

**Entwicklung von Electron** Electron bleibt trotz Kritik an Ressourcenverbrauch die dominante Plattform für Cross-Platform-Desktop-Anwendungen. Neuere Versionen bringen kontinuierliche Performance-Verbesserungen durch aktualisierte Chromium- und V8-Engines. Alternative Frameworks wie Tauri (Rust-basiert mit nativen Webviews) versprechen geringeren Ressourcenverbrauch, sind jedoch für bestehende Electron-Anwendungen wie Obsidian nicht direkt relevant. Die Bindung an Obsidians Electron-Version bedeutet, dass Plugin-Entwickler von Verbesserungen erst nach entsprechenden Obsidian-Updates profitieren können.

**Neue Web-Standards** Mehrere aufkommende Web-Standards könnten zukünftige Plugin-Entwicklung beeinflussen:

- **WebGPU:** Der Nachfolger von WebGL bietet direkteren Zugriff auf GPU-Ressourcen und könnte signifikante Performance-Verbesserungen für Rendering-intensive Anwendungen ermöglichen. PIXI.js arbeitet bereits an WebGPU-Unterstützung.
- **Offscreen Canvas:** Ermöglicht Canvas-Rendering in Web Workers und könnte Main-Thread-Blocking weiter reduzieren. Die Unterstützung in Electron verbessert sich kontinuierlich.
- **CSS Container Queries und Subgrid:** Vereinfachen responsive Layouts ohne JavaScript und könnten UI-Performance verbessern.
- **View Transitions API:** Ermöglicht flüssige Animationen zwischen View-States mit minimaler Implementierungskomplexität.

**Alternative Ansätze** Neben traditionellen Web-Technologien gewinnen alternative Ansätze an Bedeutung:

- **WebAssembly (WASM):** Ermöglicht die Ausführung von Code, der in Rust, C++ oder Go geschrieben wurde, mit nahezu nativer Performance. Besonders für algorithmisch komplexe Operationen (Pathfinding, physikalische Simulationen) könnte WASM in zukünftigen Versionen relevant werden.
- **Hybrid-Rendering:** Die Kombination von Canvas-Rendering für Performance-kritische Elemente mit DOM-Rendering für UI-Komponenten, wie in Atlas VTT praktiziert, etabliert sich als Best Practice für komplexe Web-Anwendungen.

**Ausblick** Die Entwicklung von Atlas VTT demonstriert, dass moderne Web-Technologien auch anspruchsvolle Echtzeit-Anwendungen in Plugin-Kontexten ermöglichen. Die identifizierten Patterns – WebGL-basiertes Rendering, reaktives State Management, asynchrone Ressourcenladung – werden voraussichtlich auch in zukünftigen Projekten relevant bleiben. Die kontinuierliche Evolution von Web-Standards und Rendering-Bibliotheken wird neue Optimierungsmöglichkeiten eröffnen, während die grundlegenden architektonischen Prinzipien Bestand haben dürften.

## Literatur

- [1] S. Starke, R. Hall und M. Mercer, *Daggerheart Core Rulebook*. Darrington Press, 2025, S. 366, Collaborative fantasy tabletop roleplaying game rulebook, ISBN: 9798991384100.
- [2] Business Research Insights, „Tabletop Role-Playing Game (TTRPG) Market Size, Share, Growth Analysis, By Type, By Application - Industry Forecast 2025-2035,“ Business Research Insights, 2025, Market Research Report. Estimated market value for 2025: USD 2.15 billion, projected CAGR 2025-2035: 11.84%. Adresse: <https://www.businessresearchinsights.com/market-reports/tabletop-role-playing-game-ttrpg-market-110856> (besucht am 08. 12. 2025).
- [3] RPG Drop. „Worldwide TTRPG Market in 2024 – Industry Analysis.“ Industry analysis of virtual tabletop adoption and TTRPG market trends. (2024), Adresse: <https://www.rpgdrop.com/worldwide-ttrpg-market-in-2024-industry-analysis/> (besucht am 06. 01. 2025).
- [4] J. Valentine. „Fantasy Statblocks: Create Dungeons and Dragons Style Statblocks for Obsidian.md.“ Popular Obsidian plugin for TTRPG statblock management with over 259,000 downloads as of December 2025, GitHub. (2024), Adresse: <https://github.com/jivalent/fantasy-statblocks> (besucht am 08. 12. 2025).
- [5] Jivalent. „Initiative Tracker: TTRPG Initiative Tracker for Obsidian.md.“ Popular Obsidian plugin for tracking combat initiative with 183 stars, GitHub. (2024), Adresse: <https://github.com/jivalent/initiative-tracker> (besucht am 06. 01. 2025).
- [6] Electron Contributors. „Performance | Electron Documentation.“ Official Electron documentation on performance characteristics and optimization strategies, OpenJS Foundation. (2024), Adresse: <https://www.electronjs.org/docs/latest/tutorial/performance> (besucht am 06. 01. 2025).
- [7] J. Nielsen, „Response Times: The 3 Important Limits,“ *Nielsen Norman Group*, 1993. Adresse: <https://www.nngroup.com/articles/response-times-3-important-limits/> (besucht am 24. 01. 2025).
- [8] Foundry Gaming LLC. „Foundry Virtual Tabletop Knowledge Base.“ (2024), Adresse: <https://foundryvtt.com/kb/> (besucht am 07. 10. 2024).
- [9] E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, ISBN: 9780201633610.

- [10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad und M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996, ISBN: 9780471958697.
- [11] B. M. Michelson, „Event-Driven Architecture Overview,“ *Patricia Seybold Group Research Report*, S. 1–14, 2006.
- [12] K. Marquardt, „Patterns for Plug-Ins,“ in *Proceedings of the 4th European Conference on Pattern Languages of Programs (EuroPLoP '99)*, Irsee, Germany: UVK Universitätsverlag Konstanz, 1999, S. 203–232, ISBN: 978-3-87940-774-3.
- [13] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002, ISBN: 9780321127426.
- [14] E. Freeman, E. Freeman, B. Bates und K. Sierra, *Head First Design Patterns*. O'Reilly Media, 2004, ISBN: 9780596007126.
- [15] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017, ISBN: 9780134494166.
- [16] Wikipedia. „Obsidian (software).“ (2024), Adresse: [https://en.wikipedia.org/wiki/Obsidian\\_\(software\)](https://en.wikipedia.org/wiki/Obsidian_(software)) (besucht am 08.10.2024).
- [17] Obsidian. „Obsidian API Documentation.“ (2024), Adresse: <https://docs.obsidian.md/> (besucht am 07.10.2024).
- [18] Wikipedia. „Electron (software framework).“ (2024), Adresse: [https://en.wikipedia.org/wiki/Electron\\_\(software\\_framework\)](https://en.wikipedia.org/wiki/Electron_(software_framework)) (besucht am 08.10.2024).
- [19] Electron. „Process Model.“ (2024), Adresse: <https://www.electronjs.org/docs/latest/tutorial/process-model> (besucht am 08.10.2024).
- [20] Electron. „Inter-Process Communication.“ (2024), Adresse: <https://www.electronjs.org/docs/latest/tutorial/ipc> (besucht am 08.10.2024).
- [21] V8 Team. „Optimizing V8 memory consumption.“ Published October 7, 2016. (2016), Adresse: <https://v8.dev/blog/optimizing-v8-memory> (besucht am 08.10.2024).
- [22] PixiJS Team. „PixiJS v8 Launches!“ (2024), Adresse: <https://pixijs.com/blog/pixi-v8-launches> (besucht am 08.10.2024).
- [23] PixiJS. „Scene Graph.“ (2024), Adresse: <https://pixijs.com/8.x/guides/concepts/scene-graph> (besucht am 08.10.2024).
- [24] PixiJS Team. „PixiJS v8 Beta!“ (2024), Adresse: <https://pixijs.com/blog/pixi-v8-beta> (besucht am 08.10.2024).

- [25] Wizards of the Coast. „Basic Rules for Dungeons and Dragons (D&D) Fifth Edition (5e).“ Official D&D 5e Basic Rules - Combat Section, D&D Beyond. (2014), Adresse: <https://www.dndbeyond.com/sources/dnd/basic-rules-2014/combat> (besucht am 24.01.2025).
- [26] Foundry Gaming LLC. „Tokens | Foundry Virtual Tabletop.“ Official Documentation for Token System Features, Foundry Virtual Tabletop. (2024), Adresse: <https://foundryvtt.com/article/tokens/> (besucht am 24.01.2025).
- [27] PixiJS Team. „PixiJS Performance Tips,“ PixiJS. (2024), Adresse: <https://pixijs.com/8.x/guides/production/performance-tips> (besucht am 24.01.2025).
- [28] Obsidian. „Build a plugin | Obsidian Developer Documentation.“ Official Plugin Development Guide, Obsidian.md. (2024), Adresse: <https://docs.obsidian.md/Plugins/Getting+started/Build+a+plugin> (besucht am 24.01.2025).
- [29] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, Software Engineering Best Practices für Code-Qualität, ISBN: 9780132350884.
- [30] M. Fowler und J. Lewis, „Microservices: a definition of this new architectural term,“ März 2014, Begründung für Service-orientierte Architektur. Adresse: <https://martinfowler.com/articles/microservices.html> (besucht am 25.01.2025).
- [31] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017, Single Responsibility Principle, SOLID Principles, ISBN: 978-0134494166.
- [32] E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, Singleton Pattern, Observer Pattern, Gang of Four, ISBN: 0-201-63361-2.
- [33] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad und M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996, Layered Architecture Pattern, ISBN: 978-0471958697.
- [34] Immer. „Immer: Immutability the easy way.“ Copy-on-Write für immutable State-Updates. (2024), Adresse: <https://immerjs.github.io/immer/> (besucht am 25.01.2025).
- [35] Foundry Gaming LLC. „Issue #11183: Adopt PIXI v8 as a comprehensive overhaul.“ GitHub Issue zur Diskussion der PIXI.js v8 Migration, Foundry Virtual Tabletop. (2024), Adresse: <https://github.com/foundryvtt/foundryvtt/issues/11183> (besucht am 24.01.2025).

- [36] PixiJS Team. „Migrating from PixiJS v7 to v8.“ Offizielle Migration Guide mit Breaking Changes und neuen Features, PixiJS. (2024), Adresse: <https://pixijs.com/8.x/guides/migrations/v8> (besucht am 25.01.2025).
- [37] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003, Data Access Object (DAO) Pattern, Domain Model, ISBN: 0-321-12742-0.
- [38] T. Bray, „The JavaScript Object Notation (JSON) Data Interchange Format,“ Internet Engineering Task Force (IETF), RFC 8259, 2017, JSON Standard Specification. Adresse: <https://tools.ietf.org/html/rfc8259>.
- [39] Poimandres. „Zustand: Bear necessities for state management in React.“ Lightweight State Management Library. (2024), Adresse: <https://github.com/pmnndrs/zustand> (besucht am 25.01.2025).
- [40] MDN Web Docs. „Animation performance and frame rate,“ Mozilla. (2024), Adresse: [https://developer.mozilla.org/en-US/docs/Web/Performance/Animation\\_performance\\_and\\_frame\\_rate](https://developer.mozilla.org/en-US/docs/Web/Performance/Animation_performance_and_frame_rate) (besucht am 24.01.2025).
- [41] Slaylines Benchmarks. „Canvas Engines Comparison.“ Performance-Benchmarks für Canvas-Rendering-Frameworks. (2025), Adresse: <https://benchmarks.slaylines.io/> (besucht am 24.01.2025).
- [42] Slant Community. „What are the best HTML5/JavaScript 2D game engines?“ Community-basierter Framework-Vergleich. (2025), Adresse: <https://www.slant.co/topics/973/~html5-javascript-2d-game-engines> (besucht am 24.01.2025).
- [43] Foundry Gaming LLC. „System Development Part 3: Frameworks and APIs,“ Foundry Virtual Tabletop. (2024), Adresse: <https://foundryvtt.com/article/frameworks/> (besucht am 24.01.2025).
- [44] PixiJS Team. „PixiJS Texture Optimization,“ PixiJS. (2024), Adresse: <https://pixijs.com/8.x/guides/production/texture-optimization> (besucht am 24.01.2025).
- [45] Wizards of the Coast, *Dungeons & Dragons Player's Handbook*, 5th. Wizards of the Coast, 2014, Tabletop-Rollenspiel Grundregelwerk.
- [46] Foundry Gaming LLC. „Foundry VTT Grid API Documentation.“ API Documentation für Grid System (BaseGrid, SquareGrid, HexagonalGrid), Foundry Virtual Tabletop. (2024), Adresse: <https://foundryvtt.com/api/modules/foundry.grid.html> (besucht am 24.01.2025).
- [47] Foundry Gaming LLC. „Foundry VTT Token Layer API,“ Foundry Virtual Tabletop. (2024), Adresse: <https://foundryvtt.com/api/TokenLayer.html> (besucht am 24.01.2025).

- [48] PixiJS Essentials. „@pixi-essentials/object-pool - Object Pooling for PixiJS,“ GitHub. (2024), Adresse: <https://github.com/pixi-essentials/object-pool> (besucht am 24. 01. 2025).
- [49] PixiJS Team. „PixiJS v8 Batch Rendering Guide,“ PixiJS. (2024), Adresse: <https://pixijs.com/8.x/guides/advanced/batching> (besucht am 24. 01. 2025).
- [50] Electron. „Performance,“ Electron.js. (2024), Adresse: <https://www.electronjs.org/docs/latest/tutorial/performance> (besucht am 16. 01. 2025).
- [51] L. M. Weber, W. Saelens, R. Cannoodt *et al.*, „Essential guidelines for computational method benchmarking,“ *Genome Biology*, Jg. 20, Nr. 125, S. 1–12, 2019, PMC6584985. DOI: [10.1186/s13059-019-1738-8](https://doi.org/10.1186/s13059-019-1738-8). Adresse: <https://pmc.ncbi.nlm.nih.gov/articles/PMC6584985/> (besucht am 16. 01. 2025).
- [52] V. R. Basili, R. W. Selby und D. H. Hutchens, „Experimentation in Software Engineering,“ *IEEE Transactions on Software Engineering*, Jg. SE-12, Nr. 7, S. 733–743, 1986, ISSN: 0098-5589. DOI: [10.1109/TSE.1986.6312975](https://doi.org/10.1109/TSE.1986.6312975).
- [53] J. Thangadurai, P. Saha, K. Rupanya *et al.*, „Electron vs. Web: A Comparative Analysis of Energy and Performance in Communication Apps,“ in *Quality of Information and Communications Technology: 17th International Conference, QUATIC 2024*, Ser. Communications in Computer and Information Science, Compares Electron desktop apps to web browser apps, not to native applications, Bd. 2178, Cham: Springer, 2024, S. 177–193, ISBN: 978-3-031-70245-7. DOI: [10.1007/978-3-031-70245-7\\_13](https://doi.org/10.1007/978-3-031-70245-7_13).

## **Ehrenwörtliche Erklärung**

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde / Prüfungsstelle vorgelegen hat. Ich erkläre mich damit einverstanden/nicht einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hochgeladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

(Ort, Datum)

(Eigenhändige Unterschrift)