

Einführung in die Demoszene

von

Christian Petry

Xudong Zhang

Ein wissenschaftlicher Bericht im Rahmen des Fachs

„Projektarbeit“

an der htw saar im Studiengang Informatik

Saarbrücken, den 26.08.2025

Abstract

Diese Projektarbeit befasst sich mit der Einführung in die Demoszene und der Entwicklung einer eigenen Demo. Die Arbeit behandelt die Geschichte und Entwicklung der Demoszene, von ihren Ursprüngen bis hin zu modernen Demopartys. Im praktischen Teil wird die Entstehung einer eigenen Demo dokumentiert, einschließlich der verwendeten Technologien wie OpenGL und GLSL, der visuellen Gestaltung und der technischen Implementierung. Besonderer Fokus liegt auf der Analyse verschiedener Tools und Frameworks sowie der detaillierten Beschreibung der Kernkomponenten wie Terminal-Szene, Map-Szene und Bildübergänge. Die Arbeit schließt mit einer Reflexion über Lernfortschritte, aufgetretene Probleme und mögliche Weiterentwicklungen ab.

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Insbesondere habe ich alle KI-basierten Werkzeuge angegeben, die ich bei der Erstellung, Übersetzung oder Überarbeitung des Textes verwendet habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“ zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, den 23.08.2025

Unterschriften



Christian Petry



Xudong Zhang

Inhaltsverzeichnis

1. Geschichte der Demoszene	1
1.1. Herkunft	1
1.2. Demopartys	2
1.3. Demogruppen	3
2. Evaluierung von Tools und Frameworks	6
2.1. Grafik-API: OpenGL	6
2.2. Shader: GLSL	6
2.3. Fenster- und Eingabemanagement: GLFW	7
2.4. Mathematische Operationen: GLM	7
2.5. Font-Rendering: Eigenes System mit VT323	7
2.6. Audio: SFML	8
2.7. Build-System: CMake	8
2.8. GeoJSON-Datenverarbeitung: nlohmann::json	9
2.9. Vergleich mit Alternativen	9
2.10. Zusammenfassung	9
3. Eigene Ausarbeitung	10
3.1. Entstehungsprozess	10
3.2. Ziel der Demo	10
3.3. Ursprungsidee	10
3.4. Visuelle Stilrichtung	11
4. Teile der Demo	12
4.1. Login-Szene	12
4.2. Terminal-Szene	12
4.3. CRT-Kollaps und Karten-Szene	12
4.4. Loop	12
4.5. Grafik	13
4.5.1. CRT-Effekt	13
4.5.2. Bildschirm-Kollaps	13
4.6. Sounds	13
4.7. Strukturierung	13
4.8. Effekte	14
5. Verwendete Technologien	15
5.1. Programmiersprachen & Frameworks	15
5.2. Grafik-API: OpenGL	15
5.3. Shader: GLSL	16
5.4. Fenster und Eingabemanagement: GLFW	17
5.5. Mathematische Operationen: GLM	19
5.6. Font-Rendering: Eigenes System mit VT323	19
5.7. Audio-Management: SFML	19
5.8. Build-System: CMake	20
5.9. GeoJSON-Datenverarbeitung: nlohmann::json	21

6. Kernpunkte des Codes	23
6.1. Login-Szene	23
6.2. Terminal-Szene	24
6.3. Karten-Szene	26
6.4. Übergänge (Kollaps des Bildes)	30
7. Resümee	32
7.1. Eigene Lernfortschritte	32
7.2. Entwicklungsprobleme & Lösungen	32
7.3. Denkbare Weiterentwicklungen	32
Quellenverzeichnis	34
Anhang	35
A. Screenshots der Demo	35
B. Kompilieranleitung	36
C. Zusätzliche Materialien	37

1. Geschichte der Demoszene

Laut Demoscene.info versteht man unter dem Begriff der **Demoszene** "ein weltweites, nicht kommerzielles Netzwerk kreativer Köpfe, die an der Erstellung sogenannter "Demos" beteiligt sind.", wobei Demos "computergenerierte Musikclips sind, die zeigen, welche Grafik- und Soundeffekte mit High-End-Computerhardware unter Ausnutzung ihres vollen Potenzials tatsächlich realisiert werden können". [Dem] Der Sinn ebendieser Demos besteht darin, die technischen Fähigkeiten der jeweiligen Hardware zu demonstrieren und dabei künstlerische Aspekte wie Grafikdesign und Musikkomposition zu integrieren.

Die Demoszene ist eng mit der Geschichte der Computertechnik verbunden und hat sich über die Jahre zu einer eigenen Kultur entwickelt, die sich durch Kreativität, technische Raffinesse und eine starke Gemeinschaft auszeichnet. Sie begann ursprünglich als Präsentation von Gruppen innerhalb der Cracker-Szene, die in den von ihnen verbreiteten Raubkopien ihre eigenen Intros einbauten. Daraus entstand eine eigene Kunstform inklusive Wettbewerben, in denen die Fähigkeiten der Programmierer und Künstler zur Schau gestellt wurden.

Es gibt verschiedene Kategorien von Demos, welche sich hauptsächlich auf die maximale Dateigröße konzentrieren. Dazu gehören:

- **4k/64k Demos:** Diese Demos sind auf eine maximale Dateigröße von 4 KB bzw. 64 KB beschränkt und erfordern besonders effiziente Programmierung.
- **8-Bit-Demos:** Demos, die auf 8-Bit-Computern laufen und deren Grafik- und Sounddesign die technischen Beschränkungen dieser Systeme berücksichtigen.
- **16-Bit-Demos:** Demos, die auf 16-Bit-Computern laufen und die erweiterten Möglichkeiten dieser Systeme nutzen, um komplexere Grafiken und Sounds zu erzeugen.

Diese Einschränkungen führen dazu, dass die Demoszene ein hohes Maß an Kreativität und technischem Können erfordert, da die Entwickler innovative Lösungen finden müssen, um innerhalb der vorgegebenen Grenzen zu arbeiten.

1.1. Herkunft

Die Ursprünge der Demoszene liegen in den 1980er Jahren, als eine bestimmte Gruppe früher Computerhacker, die als „Cracker“ bekannt waren, den Kopierschutz von Videospielen entfernten und damit begannen, ihre eigenen Intros in die von ihnen verbreiteten Raubkopien anstatt der Originalversionen einzubauen und somit die Gruppen, denen sie angehörten, zu repräsentieren. Diese Intros wurden unter dem Namen „Cracktros“ bekannt. Zu Beginn waren diese Intros meist sehr einfach und bestanden nur aus einem Text, der den Namen der Gruppe angab. Dies entwickelte sich im Laufe der Zeit weiter, als Programmierer begannen, komplexere Grafiken und Musik in ihre Intros zu integrieren und diese letztendlich auch außerhalb der von ihnen verbreiteten Raubkopien zu präsentieren.

Auch Firmen wie Atari erkannten das Potenzial dieser Kunstform und begannen,

eigene Demos zu erstellen, um ihre Produkte zu bewerben und die Fähigkeiten ihrer Hardware zu demonstrieren. Die erste von Atari veröffentlichte Demo wurde entwickelt, um die Fähigkeiten der Atari 400 und 800 innerhalb von Geschäften zu präsentieren. [Ata80]

Im Laufe der Zeit spaltete sich die Demoszene von der Cracker-Szene ab und entwickelte sich zu einer eigenständigen Subkultur, die sich durch Kreativität, technische Raffinesse und eine starke Gemeinschaft auszeichnet. Viele ehemalige Cracker wandten sich der Verbreitung von Raubkopien ab und begannen, ihre Fähigkeiten in der Demoszene auszuleben. Werke dieser ehemaligen Cracker-Gruppen sind heute ein wichtiger Bestandteil der Demoszene und stellen den größten künstlerischen Einfluss auf die Entwicklung der Demoszene dar. [de03]

Ein weiterer wichtiger Grund für diese Abspaltung war es, dass die Cracker-Szene zunehmend von rechtlichen Problemen und der Gefahr von Strafverfolgung betroffen war. Dies zeichnete sich durch verstärkte Hausdurchsuchungen und rechtliche Schritte gegen bekannte Cracker-Gruppen ab, was viele dazu veranlasste, sich aus der Szene zurückzuziehen oder ihre Aktivitäten einzuschränken. Dies führte auch zu einem verstärkten Fokus auf "reine" Demos, also Demos, die keine Raubkopien von Software oder Spielen beinhalteten.

Durch diese Abspaltung wurden auch die Schwerpunkte innerhalb der Gruppen neu gesetzt. Durch den Auftritt von stärkeren 16-Bit-Systemen verschoben sich die technischen Möglichkeiten und damit auch die kreativen Ansprüche an die Demos, weshalb innerhalb der Demo-Gruppen feste Aufgabenbereiche für die Mitglieder geschaffen wurden. Grafiker übernahmen zunehmend die Gestaltung der visuellen Elemente, während Musiker für die Soundgestaltung verantwortlich waren. Programmierer konzentrierten sich darauf, die einzelnen Elemente der Demo zu implementieren und miteinander zu verknüpfen.

1.2. Demopartys

Demopartys sind ein wichtiger Bestandteil der Demoszene und dienen als Treffpunkt für Demoszener aus aller Welt. Hier können Gruppen ihre neuesten Demos präsentieren, Feedback von anderen Entwicklern erhalten und sich über technische Herausforderungen austauschen. Die Programmiertechnik und das Design von Demos stehen hier im Vordergrund, und viele Entwickler nutzen diese Veranstaltungen, um neue Ideen zu entwickeln und ihre Fähigkeiten zu verbessern. Mittelpunkt jeder Demoparty sind jedoch Wettbewerbe und Vorführungen der Demos, in denen sich sowohl Gruppen als auch Einzelpersonen in verschiedenen Kategorien messen können. Über die Gewinner dieser Wettbewerbe wird von den anwesenden Besuchern der Demoparty abgestimmt. Auch ist es durchaus üblich, dass die Beiträge zu diesen Wettbewerben von den Teilnehmern noch auf der Demoparty selbst fertiggestellt werden, was als besondere Herausforderung angesehen wird.

Damit ein Vergleich der eingereichten Demos jeweils möglich ist, werden die Wettbewerbe in verschiedene Kategorien unterteilt, welche oft nach Plattformen oder Techniken differenzieren. Für PC-Systeme wird dabei traditionell anhand der Speicherbegrenzung von 64, 16 oder 4 Kilobyte unterteilt. Ferner gibt es Unterkategorien, die

sich auf spezifische Aspekte wie Grafik, Musik oder sogar ASCII-Art konzentrieren.

Zu den wichtigsten Demopartys weltweit gehören unter anderem:

- Die **Assembly** in Finnland, die jährlich in Helsinki stattfindet und als eine der ältesten Demopartys gilt. Mischung zwischen Demo- und LAN-Party.
- Die **BreakPoint** in Deutschland, die von 2003 bis 2010 in Bingen stattfand.
- Die **Revision** in Deutschland, die seit 2011 jährlich in Saarbrücken als größte Demoparty weltweit stattfindet. Nachfolger der **Breakpoint**.
- Die **Evoke** in Deutschland, die seit 1997 jährlich in Köln stattfindet und die älteste noch aktive Demoparty in Deutschland ist.
- Die **NAID** in Kanada, findet seit 1995 jährlich in Quebec statt und ist die erste Demoparty außerhalb Europas.

1.3. Demogruppen

Unter einer Demogruppe versteht man eine Gemeinschaft von Entwicklern, die gemeinsam an Demos arbeiten und ihre kreativen Ideen umsetzen. Diese Gruppen können aus Freunden, Kollegen oder sogar Online-Communities bestehen und sind oft über Jahre hinweg aktiv. Die auf Demopartys vertretenen Gruppen zeigen eine Vielzahl von Stilen und Techniken, die die Vielfalt und Kreativität der Demoszene widerspiegeln. Auch die dort präsentierten Demos sind oft das Ergebnis monatelanger oder sogar jahrelanger Arbeit und spiegeln die Leidenschaft und das Engagement der Gruppen wider. Die Gruppen arbeiten oft eng zusammen, um ihre Fähigkeiten zu verbessern und innovative Ideen zu entwickeln. Die in Demos vorhandenen Arbeitsbereiche wie Grafikdesign, Programmierung und Musikproduktion werden dabei häufig von verschiedenen Mitgliedern der Gruppe abgedeckt. Oft präsentieren diese Gruppen ihre Arbeit in den auf Demopartys veranstalteten Wettbewerben. Viele der in der Szene aktiven Entwickler gehören einer oder mehreren Demogruppen an. Zu den bekanntesten Demogruppen gehören unter anderem:

- **Future Crew** ist eine 1986 in Finnland gegründete Demogruppe, der der Durchbruch auf der Assembly 1993 mit ihrer Demo *The 2nd Reality* gelang, mit der sie den ersten Platz in der Kategorie "IBM PC Compatible" belegte und welche als Meilenstein in Sachen Grafik und Musiksynchrosynchronisation gilt. Aus dieser Gruppe ging 1995 das Entwicklerstudio **Remedy Entertainment** hervor, das für Spiele wie *Max Payne* und *Alan Wake* bekannt ist. [fam15]
- **Farbrausch** ist eine in Deutschland gegründete Gruppe, die für ihre technisch anspruchsvollen und kreativen Demos bekannt ist. Sie gewannen zahlreiche Wettbewerbe und setzten neue Maßstäbe im Intro-Design sowie in der Entwicklung des zur Demoerstellung verwendeten Tools *.werkzeug*. [the04] *.werkzeug* wurde entwickelt, um Demos mit Texturen, 3D-Modellen, Animationen und Musik-Synchronisation zu erstellen. Der Fokus des Tools liegt dabei auf minimaler Dateigröße und Echtzeit-Rendering, wobei die Dateigröße inklusive der Player-Software bis zu 64 KB klein gehalten werden kann.

- **Scoopex** ist eine aus der Cracker-Szene kommende Gruppe, die dort ursprünglich unter dem Namen **Megaforce** bekannt war und sich von dort zur Demoszene entwickelt hat, nachdem ihr ursprünglicher Leader *Ranger* 1988 aufgrund ihrer Aktivitäten in der Cracker-Szene rechtliche Probleme bekam. Die Bekanntheit der Gruppe in der Demoszene kam vor allem durch ihre Megademos, welche durch ihre Aufteilung in mehrere Teile, die Verwendung innovativer Techniken wie der Integration von Benutzereingaben und den kreativen Ladenbildschirmen zwischen den einzelnen Teilen auf sich aufmerksam machten. [Sen17]



Abbildung 1: Ausschnitt aus Demo *The 2nd Reality* der Gruppe Future Crew, zeigt Logo der Demo [Cre93]



Abbildung 2: Ausschnitt aus Demo *The Product* der Gruppe Farbrausch, Eingangsszene [Far00]



Abbildung 3: Beginn der Demo *Mental Hangover* der Gruppe Scoopex [Sco90]

2. Evaluierung von Tools und Frameworks

In diesem Kapitel werden die wichtigsten für die Entwicklung der Demo evaluierten Tools und Frameworks vorgestellt. Für jedes Tool werden die Vor- und Nachteile sowie die Gründe für die finale Entscheidung erläutert.

2.1. Grafik-API: OpenGL

OpenGL (Open Graphics Library) ist eine plattformübergreifende API zur Grafikprogrammierung, die es ermöglicht, 2D- und 3D-Graf zu rendern. In unserer Demo wird OpenGL verwendet, um die Grafiken und Effekte darzustellen.

Pro:

- Plattformunabhängig (Windows, macOS, Linux)
- Große Community und viele Ressourcen
- Direkte Kontrolle über den Grafik-Renderprozess
- Viele Tutorials und Beispielprojekte verfügbar

Contra:

- Teilweise veraltete Funktionen und komplexe Initialisierung
- Weniger komfortabel als moderne High-Level-Engines
- Keine integrierte Unterstützung für Audio oder GUI

Entscheidung: OpenGL wurde gewählt, da die Demo plattformübergreifend funktionieren soll und die direkte Kontrolle über die Grafikeffekte (z.B. CRT-Shader) benötigt wird.

2.2. Shader:GLSL

GLSL (OpenGL Shading Language) ist eine C-basierte Sprache zur Programmierung von Shadern in OpenGL. In unserer Demo wird GLSL verwendet, um verschiedene Grafikeffekte zu erzeugen.

Pro:

- Direkte Integration in OpenGL
- Hohe Flexibilität und Kontrolle über den Rendering-Prozess
- Unterstützung für moderne Grafiktechniken (z.B. Physically Based Rendering)

Contra:

- Komplexität bei der Entwicklung und Fehlersuche
- Abhängigkeit von der OpenGL-Version

Entscheidung: GLSL wurde gewählt, um die gewünschten Grafikeffekte effizient und flexibel umsetzen zu können.

2.3. Fenster- und Eingabemanagement: GLFW

GLFW ist eine plattformübergreifende Bibliothek, die die Arbeit mit Fenstern und Eingaben stark vereinfacht.

Pro:

- Sehr leichtgewichtig und einfach zu integrieren
- Gute Dokumentation und Beispiele
- Unterstützt Tastatur, Maus und Joystick
- Plattformübergreifend

Contra:

- Keine Unterstützung für Audio
- Weniger Features als SDL (z.B. Netzwerk)

Entscheidung: GLFW wurde gewählt, da die Demo keine komplexen Eingaben oder Netzwerk benötigt und die Integration mit OpenGL sehr einfach ist.

2.4. Mathematische Operationen: GLM

GLM ist eine C++ Bibliothek für Vektor- und Matrixoperationen, die sich an GLSL orientiert.

Pro:

- GLSL-kompatible Syntax
- Header-only, keine zusätzliche Abhängigkeit
- Sehr schnell und effizient
- Gut dokumentiert

Contra:

- Nur für mathematische Operationen, keine Grafikfunktionen
- Für sehr einfache Projekte eventuell Overkill

Entscheidung: GLM wurde gewählt, da die Demo viele Matrix- und Vektoroperationen für Transformationen und Effekte benötigt.

2.5. Font-Rendering: Eigenes System mit VT323

Für die Darstellung von Text wird ein eigenes Font-System verwendet, das die VT323-Schriftart nutzt.

Pro:

- Volle Kontrolle über das Rendering

- Authentischer Retro-Look
- Anpassbar für verschiedene Effekte

Contra:

- Mehr Entwicklungsaufwand als fertige Lösungen (z.B. FreeType)
- Weniger flexibel bei komplexen Schriftarten

Entscheidung: Das eigene System wurde gewählt, um die Retro-Ästhetik optimal umzusetzen und die Schriftgröße dynamisch anpassen zu können.

2.6. Audio: SFML

SFML (Simple and Fast Multimedia Library) ist eine plattformübergreifende Multimedia-Bibliothek, die die Entwicklung von Multimedia-Anwendungen erleichtert.

Pro:

- Plattformübergreifend
- Benutzerfreundlich
- Effizient bei der Verarbeitung von Audio

Contra:

- Begrenzte Funktionalität bei komplexen Anwendungen
- Keine Unterstützung für 3D-Audio

Entscheidung: SFML wurde gewählt, da die Demo nur einfache Soundeffekte benötigt.

2.7. Build-System: CMake

CMake ist ein plattformübergreifendes Build-System für C++-Projekte.

Pro:

- Unterstützt viele Plattformen und IDEs
- Automatisiert die Abhängigkeitsverwaltung
- Gut für größere Projekte geeignet

Contra:

- Einarbeitung für Anfänger nötig
- Komplexe Projekte können unübersichtlich werden

Entscheidung: CMake wurde gewählt, um die Demo auf verschiedenen Systemen einfach kompilieren zu können.

2.8. GeoJSON-Datenverarbeitung: nlohmann::json

nlohmann::json ist eine beliebte C++ Bibliothek zur Verarbeitung von JSON-Daten. Sie bietet eine einfache und intuitive API zur Arbeit mit JSON-Objekten und -Arrays.[Loh25]

Pro:

- Einfache und intuitive API
- Gute Performance bei der Verarbeitung von JSON-Daten
- Breite Unterstützung in der C++ Community

Contra:

- Zusätzliche Abhängigkeit im Projekt
- Mögliche Komplikationen bei der Integration

Entscheidung: nlohmann::json würde gewählt, weil sie leicht zu benutzen ist und von vielen C++ Entwicklern verwendet wird. Mit einer GeoJSON-Datei kann man außerdem ganz einfach Orte auf der Karte genau anzeigen und finden.

2.9. Vergleich mit Alternativen

SDL:

- Mehr Features als GLFW (Audio, Netzwerk, Grafik)
- Größerer Overhead, für die Demo nicht nötig

FreeType:

- Sehr flexibel für Font-Rendering
- Integration aufwendig, für Retro-Look nicht nötig

FMOD:

- Professionelle Audio-Engine
- Kommerzielle Lizenz, für einfache Effekte zu groß

2.10. Zusammenfassung

Die gewählten Tools und Frameworks bieten eine gute Balance zwischen Einfachheit, Performance und Flexibilität. Die Entscheidung für OpenGL, GLFW, GLM, ein eigenes Font-System und SFML ermöglicht eine plattformübergreifende, effiziente und authentische Umsetzung der Retro-Demo.

3. Eigene Ausarbeitung

In diesem Kapitel wird die Entwicklung unserer eigenen Demo detailliert beschrieben. Dabei wird der gesamte Entstehungsprozess von der Ideenfindung bis zur Implementierung nachgezeichnet.

3.1. Entstehungsprozess

Die Entwicklung unserer Demo begann mit der Ideenfindung und der Festlegung des Themas. Die erste Phase umfasste die Konzeptualisierung, in der wir verschiedene Ansätze diskutierten und uns schließlich auf das Thema **Retro-Computer-Terminal** einigten. Die zweite Phase beinhaltete die technische Planung, in der wir die benötigten Technologien und Tools evaluiert und ausgewählt haben. Die eigentliche Implementierung erfolgte in mehreren Iterationen, in denen wir die verschiedenen Szenen und Effekte schrittweise entwickelten und innerhalb der Gesamtdemo testeten.

3.2. Ziel der Demo

Unsere Demo orientiert sich thematisch an unserem Studiengang Informatik und simuliert ein Retro-Computer-Terminal aus den 1980er und 1990er Jahren. Der visuelle Stil lehnt sich an klassische Röhrenmonitore an und wird durch einen **CRT-Effekt** sowie neongrüne Schriftzüge geprägt. Passende Soundeffekte wie das Hochfahren eines Computers sowie Tasteneingaben runden das Erlebnis ab.

3.3. Ursprungsidee

Ursprünglich haben wir drei Konzepte für unsere Demo diskutiert, die sich alle um das Thema **Informatik** drehten. Diese lauteten wie folgt:

- **Inside the Machine:** Aufzeigen, wie Code (beispielsweise in C++) durch einen Computer "fließt". Dies sollte mithilfe von animierten Datenpartikeln, durch Linien dargestellten Schaltkreisen und einer Ausgabe am Ende (z.B. "LED ON") geschehen. Außerdem sollte es mit Musik und Soundeffekten untermauert werden.
- **Digital Pulse - Rythmus der Daten:** Animation einer Laufschrift mit bunt pulsierenden Datenm angepasst an den Musiktakt, die die Funktionsweise eines Computers simuliert.
- **Retro Terminal:** Simulation eines Retro-Terminals, das zuerst bootet und im Anschluss grünen Text Zeile für Zeile ausgibt, während im Hintergrund Pieptöne und Chiptune-Musik abgespielt werden.

Daraus entstand die ursprüngliche Idee, eine Demo zu entwickeln, die die Funktionsweise eines simplen Computers anhand der Von-Neumann-Architektur veranschaulicht. Dazu sollte sich im oberen Teil der Demo ein Terminal befinden, in dem der

Benutzer Befehle eingeben kann, während im unteren Teil eine Animation der Von-Neumann-Architektur abläuft. Die Animation sollte dabei die Funktionsweise des Computers veranschaulichen, indem sie zeigt, wie Daten zwischen den verschiedenen Komponenten fließen, wobei diese Animationen abhängig von den angegebenen Befehlen im Terminal aktualisiert werden sollten. Die Idee war, dass dem Betrachter durch die die Funktionsweise des Computers verdeutlicht wird, wie die Hardware und Software zusammenarbeiten, um die eingegebenen Befehle auszuführen. Diese ursprüngliche Idee wurde jedoch im Laufe der Entwicklung verworfen, da sie sich für unseren damaligen Wissensstand als zu komplex erwies.

3.4. Visuelle Stilrichtung

Die visuelle Stilrichtung unserer Demo orientiert sich an klassischen Computern der 80er und 90er Jahre, insbesondere an den CRT-Monitoren. Die Farbpalette umfasst hauptsächlich dunkle Töne mit leuchtenden Akzenten in Neonfarben, um den Retro-Charakter zu betonen.

4. Teile der Demo

Die Demo besteht aus mehreren Komponenten, die zusammen ein kohärentes Erlebnis schaffen. Diese Szenen werden in den folgenden Unterabschnitten näher beschrieben.

4.1. Login-Szene

Die Demo startet mit einem klassischen Anmeldebildschirm, auf dem sich ein Benutzer über einen blinkenden Unterstrich-Cursor einloggt. Während der Eingabe ertönen Keystroke-Sounds. Nach erfolgreicher Anmeldung erscheint die Meldung „**ACCESS GRANTED**“, die das Ende der ersten Szene markiert.

4.2. Terminal-Szene

Nach dem Login wird ein typisches Terminal simuliert. Per Unix-artigem Befehl wird ein Verzeichniswechsel dargestellt, gefolgt vom Ausführen einer Datei namens „**RetroTerminal**“. Diese führt scheinbar C++-Code aus, der das Thema der Demo sowie die Namen des Entwicklerteams „ausgibt“

4.3. CRT-Kollaps und Karten-Szene

Nach der Codeausgabe simuliert die Demo den **Kollaps eines CRT-Bildschirms**. Der Bildschirm flackert und baut sich in der nächsten Szene neu auf: Eine Weltkarte und zum Ausschnitt passende Koordinaten zusammen mit einem Radar werden sichtbar. Das Radar findet Koordinaten, woraufhin die Kamera in mehreren Schritten bis zur **HTW Saar** zoomt. Rote Pulsanimationen und Sonar-Sounds verstärken den Effekt.

4.4. Loop

Nach dem finalen Zoom kollabiert der Bildschirm erneut. Die Demo leitet so einen erneuten Durchlauf ein.

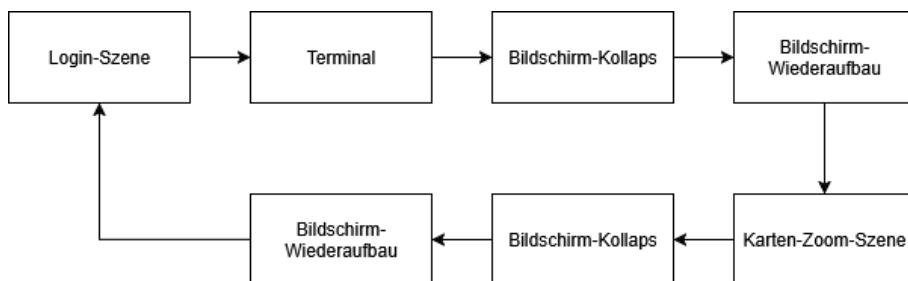


Abbildung 4: Darstellung des Szenen-Loops in der Demo

4.5. Grafik

Unsere Demo besteht aus mehreren verschiedenen Grafikelementen, die zusammen ein konsistentes visuelles Erlebnis schaffen. Dazu gehören die Login-Szene, die Terminal-Szene, die Karten-Szene sowie der Kollaps und Wiederaufbau des Bildschirms. Außerdem gibt es verschiedene visuelle Effekte, die die Retro-Ästhetik unterstreichen, wie z.B. der CRT-Effekt und die Simulation des Kollaps einer Braunschen Röhre.

4.5.1. CRT-Effekt

Der CRT-Effekt simuliert die Darstellung auf einem alten Röhrenmonitor. Er erzeugt einen leichten Scanline-Effekt und eine subtile Unschärfe, um das Gefühl eines klassischen Bildschirms zu vermitteln. Dieser Effekt wird durch einen Shader realisiert, der über allen in der Demo dargestellten Szenen liegt. Er sorgt dafür, dass die Darstellung der Grafiken authentisch wirkt und den Retro-Charakter der Demo unterstreicht.

4.5.2. Bildschirm-Kollaps

Der Bildschirm-Kollaps simuliert den Effekt eines alten CRT-Monitors, der zusammenbricht und neu aufgebaut wird. Dieser Effekt wird durch eine Kombination aus Shadern und Animationen erreicht, die den Eindruck eines physischen Bildschirms vermitteln, der sich verformt und dann wiederhergestellt wird.

4.6. Sounds

Die in unserer Demo vorhandenen Sounds werden über eine Klasse `SoundManager` gesteuert, die für das Abspielen und Verwalten der Audio-Komponenten verantwortlich ist. Zu Beginn der Demo wird durch diese Klasse der Hintergrundsound gestartet, welcher den Start eines alten Computers simuliert und während der gesamten Demo im Hintergrund läuft. Während der Login- und Terminal-Szene werden Tastaturschläge durch verschiedene Keystroke-Sounds untermauert, die bei der Animation der Texteingabe bei jedem neuen Zeichen abgespielt werden. Während der Karten-Szene werden Sonar-Sounds abgespielt, die den Suchvorgang des Radars nach bestimmten Koordinaten akustisch unterstützen. Zusätzlich werden jeweils bei Kollaps als auch bei Wiederaufbau des Bildschirms passende Soundeffekte abgespielt, die den visuellen Eindruck verstärken.

4.7. Strukturierung

Die Demo ist in mehrere Klassen und Module unterteilt, die jeweils für bestimmte Funktionen und Szenen verantwortlich sind. Diese Struktur ermöglicht eine klare Trennung der Logik und erleichtert die Wartung und Erweiterung der Demo. Die Aufteilung in verschiedene Module ermöglicht es, spezifische Funktionen unabhängig zu entwickeln und zu testen. Die generelle Ordnerstruktur sieht folgendermaßen aus:

- **assets:** Enthält alle benötigten Assets wie Schriftarten, Sounddateien und GeoJSON-Kartendaten.
- **include:** Beinhaltet die Header-Dateien der externen Bibliotheken.
- **shaders:** Enthält die Shader-Dateien für die grafischen Effekte.
- **src:** Beinhaltet den Quellcode der Anwendung.

Desweiteren ist die Ordnerstruktur innerhalb von `src` wie folgt aufgebaut:

- **audio:** Enthält den SoundManager, welcher für das Abspielen und Verwalten der Audio-Komponenten verantwortlich ist.
- **core:** Beinhaltet die Klassen WindowManager und Config, welche für die Verwaltung des Anwendungsfensters und der Konfigurationsdatei zuständig sind.
- **external:** Beinhaltet externe Bibliotheken glad.c und ini.c, die für die OpenGL-Initialisierung und das Einlesen der Konfigurationsdatei verwendet werden.
- **graphics:** Enthält ShaderManager, Font und CRTEffect, die für die Verwaltung der Grafiken und Effekte verantwortlich sind.
- **scenes:** Beinhaltet die verschiedenen Szenen der Demo, wie LoginScene, TVEffectScene, TerminalScene und LocateScene, die jeweils für die Darstellung und Logik ihrer spezifischen Bereiche zuständig sind.

4.8. Effekte

Der wichtigste visuelle Effekt in unserer Demo ist der CRT-Effekt, der die Darstellung auf einem alten Röhrenmonitor simuliert. Er erzeugt einen leichten Scanline-Effekt und eine subtile Unschärfe, um das Gefühl eines klassischen Bildschirms zu vermitteln. Dieser Effekt wird durch einen Shader realisiert, der über allen in der Demo dargestellten Szenen liegt. Er sorgt dafür, dass die Darstellung der Grafiken authentisch wirkt und den Retro-Charakter der Demo unterstreicht. Zusätzlich gibt es den Bildschirm-Kollaps-Effekt, der den Eindruck eines physischen Bildschirms vermittelt, der sich verformt und dann wiederhergestellt wird. Dieser Effekt wird ebenfalls durch einen Shader realisiert, der die Verformung und Wiederherstellung des Bildschirms simuliert.

5. Verwendete Technologien

5.1. Programmiersprachen & Frameworks

Das Projekt wurde vollständig in C++ entwickelt. Die Wahl fiel auf C++, da diese Sprache eine hohe Performance bietet und direkten Zugriff auf System- und Hardwarefunktionen ermöglicht. Zudem ist C++ plattformübergreifend und eignet sich besonders für grafikintensive Anwendungen wie Demos.

Für die Umsetzung wurden verschiedene Frameworks und Bibliotheken eingesetzt:

- **OpenGL**: Für die plattformübergreifende Grafikdarstellung und Effekte.
- **GLFW**: Für Fensterverwaltung und Eingabemanagement.
- **GLM**: Für mathematische Operationen mit Vektoren und Matrizen.
- **SFML**: Für die Audioausgabe und Soundeffekte.
- **nlohmann::json**: Für das Einlesen und Verarbeiten von GeoJSON-Daten.
- **CMake**: Als Build-System zur Verwaltung und Kompilierung des Projekts.

Durch die Kombination dieser Frameworks und Bibliotheken konnte eine performante, flexible und plattformunabhängige Demo realisiert werden.

5.2. Grafik-API: OpenGL

OpenGL (Open Graphics Library) ist eine plattformübergreifende API zur Grafikprogrammierung, die es ermöglicht, 2D- und 3D-Graf zu rendern. In unserer Demo wird OpenGL verwendet, um die Grafiken und Effekte darzustellen.

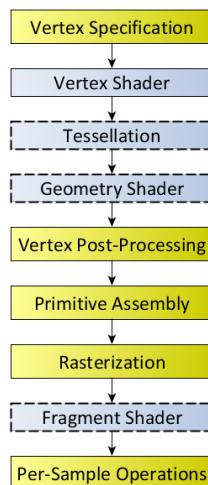


Abbildung 5: Rendering-Pipeline in OpenGL

Die Rendering-Pipeline in OpenGL beschreibt den Prozess, den jeder Vertex und jedes Fragment durchläuft, um auf dem Bildschirm dargestellt zu werden.

Sie beginnt mit der Vertex Specification, bei der Geometriedaten aus Vertex Array Objects (VAOs) bereitgestellt werden.

Anschließend durchlaufen die Daten die nachfolgenden teils programmierbaren Stufen:

- **Vertex-Shader:** Verarbeitet jeden Vertex (z.B. Transformationen)
- **(Optional) Tesselation und Geometry Shader:** Unterteilt die grundlegenden Geometrieprinzipien (z.B. Dreiecke) in kleinere Einheiten oder modifiziert sie
- **Primitive Assembly & Clipping:** Setzt die Primitiven zusammen und schneidet sie ggf. an den Bildschirm-Rändern
- **Rasterization:** Wandelt die Primitiven in Fragmente (Pixel) um
- **Fragment-Shader:** Berechnet die Farben der Fragmente (hier werden zum Beispiel unsere CRT-Shader angewendet)
- **Per-Sample Operations:** Blendet die Fragmente im Framebuffer ein

Für unsere Demo ist insbesondere die Kontrolle über den Fragment Shader entscheidend, um spezielle Effekte wie den CRT-Look oder den Bildschirmkollaps direkt pro Pixel zu berechnen.

Ein Beispiel für die Initialisierung und Nutzung von OpenGL in unserem Projekt:

```
1 #include <glad/glad.h>
2 #include <GLFW/glfw3.h>
3 ...
4 void framebuffer_size_callback(GLFWwindow* window, int width,
5     int height) {
6     if (height > 0) glViewport(0, 0, width, height);
}
```

Listing 1: OpenGL-Initialisierung und Callback in `main.cpp`

Durch diese Architektur ist es möglich, die Grafikeffekte präzise zu steuern und die volle Flexibilität von OpenGL auszunutzen. Die Entscheidung für OpenGL basiert somit auf der Notwendigkeit, plattformübergreifend zu arbeiten und individuelle Grafikeffekte wie CRT-Shader effizient umzusetzen.

5.3. Shader: GLSL

GLSL (OpenGL Shading Language) ist eine Programmiersprache, die mithilfe von OpenGL Shader-Programme erstellt. Diese Shader sind kleine Programme, die auf der Grafikkarte ausgeführt werden und für die Darstellung von Grafiken verantwortlich sind. In unserer Demo wird GLSL verwendet, um den CRT-Effekt zu realisieren. Der Shader berechnet die Scanlines und die Unschärfe, die den Eindruck eines alten Röhrenmonitors vermitteln.

Der Demo werden die Shader-Dateien mit Hilfe einer eigenen Klasse `ShaderManager` geladen und kompiliert. Diese von uns verwendeten Shader basieren dabei auf von

der Community entwickelten und auf <https://www.shadertoy.com/> veröffentlichten Shadern, welche wir für unsere Zwecke angepasst haben. Shadertoy ist dabei eine Webseite, die es Benutzern ermöglicht, die von ihnen entwickelten Shader in einer webbasierten Umgebung mit anderen zu teilen und zu experimentieren. Außerdem ermöglicht es anderen Benutzern, die Shader direkt auf der Webseite zu modifizieren und somit ihre eigenen Ideen und Konzepte zu testen. Dadurch können die verschiedenen Shader flexibel im Programm verwendet und für unterschiedliche Effekte eingesetzt werden. Die folgende Abbildung zeigt beispielhaft, wie ein Vertex-Shader und ein Fragment-Shader in OpenGL verwendet werden:

```

1 #version 330 core
2 layout(location = 0) in vec2 aPos;
3 uniform mat4 projection;
4 void main() {
5     gl_Position = projection * vec4(aPos, 0.0, 1.0);
6 }
```

Listing 2: Vertex Shader: `line.vert`

```

1 #version 330 core
2 out vec4 FragColor;
3 uniform vec4 uColor;
4 void main() {
5     FragColor = uColor;
6 }
```

Listing 3: Fragment Shader: `line.frag`

Die Shader werden im C++ Code wie folgt eingebunden und verwendet:

```

1 GLuint shaderProgram =
2     ShaderManager::loadShader("shaders/line.vert", "shaders/
3         line.frag");
4     glUseProgram(shaderProgram);
// ...
```

Listing 4: Shader-Initialisierung in C++

5.4. Fenster und Eingabemanagement: GLFW

Im Demo wird GLFW benutzt, um das Fenster zu erstellen und zu verwalten.

Mit der Klasse `WindowManager` wird die Nutzung von GLFW noch übersichtlicher und modularer gestaltet. Das Fenster kann sowohl im Vollbildmodus als auch als normales Fenster angezeigt werden, je nach Konfiguration.

Über Callback-Funktionen werden Tastendrücke und die Änderung der Fenstergröße direkt behandelt. Dadurch kann das Programm flexibel auf Benutzereingaben und Fensterereignisse reagieren.

Die Hauptschleife sorgt dafür, dass das Fenster immer aktuell bleibt, die Grafiken gerendert werden und alle Eingaben verarbeitet werden. So entsteht eine stabile und interaktive Anwendung, die auf verschiedenen Betriebssystemen funktioniert.

```

1 #include <GLFW/glfw3.h>
2 WindowManager windowManager;
3 GLFWmonitor* primary = glfwGetPrimaryMonitor();
4 const GLFWvidmode* mode = glfwGetVideoMode(primary);
5 glfwWindowHint(GLFW_DECORATED, GLFW_FALSE);
6
7 if (!windowManager.initialize(
8     Config::fullscreen ? mode->width : Config::width,
9     Config::fullscreen ? mode->height : Config::height, "Retro
10    ↴Terminal",
11     Config::fullscreen)) {
12     return -1;
13 }
14
15 windowManager.setKeyCallback(close_window_on_escape);
16 windowManager.setFramebufferSizeCallback(
17     framebuffer_size_callback);

```

Listing 5: GLFW window setup in `main.cpp`

```

1 while (!windowManager.shouldClose()) {
2     // ... rendering and logic ...
3     windowManager.swapBuffers();
4     windowManager.pollEvents();
5 }

```

Listing 6: Main loop in `main.cpp`

Der `WindowManager` steuert das Fenster im Programm. Er erstellt das Fenster, setzt die Größe und den Titel, und sorgt dafür, dass das Fenster angezeigt wird. Außerdem kann er auf Tastendrücke und Änderungen der Fenstergröße reagieren. Mit Methoden wie `swapBuffers()` und `pollEvents()` bleibt das Fenster immer aktuell und verarbeitet Eingaben.

```

1 bool WindowManager::initialize(int width, int height, const
2     char* title, bool fullscreen) {
3     window = glfwCreateWindow(width, height, title, nullptr,
4         nullptr);
5     // ...
6     glfwMakeContextCurrent(window);
7     glfwSetWindowUserPointer(window, this);
8     return true;
9 }

```

Listing 7: WindowManager implementation in `WindowManager.cpp`

5.5. Mathematische Operationen: GLM

Im Demo benutzen wir die GLM-Bibliothek, um mathematische Operationen wie Vektoren und Matrizen zu verwalten.

Mit der GLM-Bibliothek können wir in Demo einfach mit Vektoren und Matrizen rechnen. Zum Beispiel, um Koordinaten zu transformieren, Farben zu speichern oder Projektionen für die Kartenanzeige zu berechnen. Dadurch werden mathematische Operationen im Grafikprogramm sehr übersichtlich und schnell.

```
1 glm::mat4 projection = glm::ortho(0.0f, float(width), 0.0f,
2                                     float(height));
3 glm::vec4 color(1.0f, 0.0f, 0.0f, 1.0f); // Rot mit voller
4                                         Deckkraft
5 glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"),
6                     1, GL_FALSE, &projection[0][0]);
7 glUniform4fv(glGetUniformLocation(shaderProgram, "uColor"), 1,
8             &color[0]);
```

Listing 8: GLM für Projektion und Farbe

5.6. Font-Rendering: Eigenes System mit VT323

Die Schriftart VT323 wird im Demo in verschiedenen Dateien benutzt.

In `main.cpp` wird die Schrift geladen und für die Anzeige vorbereitet.

In `LocateScene.cpp` wird die Schrift genutzt, um Text direkt auf dem Bildschirm darzustellen. So entsteht ein individueller Retro-Look für die Demo.

```
1 Font font;
2 if (!font.load("assets/fonts/VT323-Regular.ttf", 50)) {
3     std::cerr << "Failed to load font.\n";
4     return -1;
5 }
6 // ...
7 font.renderText(msg, x, y, scale, color);
```

Listing 9: Font loading and usage in `main.cpp`

```
1 font.renderText(hud.str(), cx - 180, cy + viewH / 2 - 40, 0.7f
2 , glm::vec3(1.0f, 0.2f, 0.2f));
font.renderText(msg, cx - 180, cy + viewH / 2 - 80, 1.0f, glm
    ::vec3(0, 1, 0));
```

Listing 10: Text rendering in `LocateScene.cpp`

5.7. Audio-Management: SFML

In Demo wird SFML verwendet, um Soundeffekte abzuspielen.

Mit dem SoundManager, der auf SFML basiert, können wir verschiedene Sounddateien im Programm laden und abspielen. Die Methode `loadSounds` lädt alle benötigten Sounds, und mit `playSound(index)` wird ein Sound abgespielt, zum Beispiel ein Tastenton oder ein Hintergrundsound. So bekommt die Demo passende Audioeffekte für verschiedene Aktionen.

```

1  bool SoundManager::loadSounds(const std::vector<std::string>&
2      soundFiles) {
3          // ... SFML Buffer und Source erzeugen, WAV-Dateien laden
4          ...
5      }
6
7  void SoundManager::playSound(int index) {
8      // ... SFML Sound abspielen ...
9 }
```

Listing 11: Audio laden und abspielen in SoundManager.cpp

```

1 SoundManager soundManager;
2 std::vector<std::string> soundFiles = {
3     "assets/sounds/keystroke-01.wav",
4     "assets/sounds/keystroke-02.wav",
5     // ...
6 };
7 soundManager.loadSounds(soundFiles);
8 soundManager.playSound(0); // Beispiel: Tastenton abspielen
```

Listing 12: Verwendung in main.cpp

5.8. Build-System: CMake

Das Projekt verwendet CMake als Build-System, um die Kompilierung und das Verlinken aller Quell- und Bibliotheksdateien zu automatisieren.

In der Datei `CMakeLists.txt` werden die benötigten Header- und Bibliothekspfade angegeben, alle Quellcodes eingebunden und die externen Bibliotheken wie GLFW, Freetype, OpenGL und SFML für die Audioausgabe verlinkt. Dadurch ist eine plattformübergreifende und flexible Projektstruktur gewährleistet.

```

1 cmake_minimum_required(VERSION 3.10)
2 project(RetroTerminal)

3
4 set(CMAKE_CXX_STANDARD 17)

5
6 include_directories(
7     include
8     C:/msys64/mingw64/include
9     C:/msys64/mingw64/include/GLFW
10    C:/msys64/mingw64/include/freetype2
11    C:/msys64/mingw64/include/glm
12    C:/msys64/mingw64/include/SFML
```

```

13 )
14
15 link_directories(
16     C:/msys64/mingw64/lib
17 )
18
19 file(GLOB SOURCES
20     src/*.cpp
21     src/*.c
22 )
23
24 add_executable(RetroTerminal ${SOURCES})

```

Listing 13: Projektkonfiguration in CMakeLists.txt

```

1 target_link_libraries(RetroTerminal
2     glfw3
3     freetype
4     opengl32
5     gdi32
6     user32
7     kernel32
8     sfml-audio
9     sfml-system
10 )

```

Listing 14: Bibliotheken verlinken in CMakeLists.txt

5.9. GeoJSON-Datenverarbeitung: nlohmann::json

In der Demo, die Verarbeitung und Darstellung von GeoJSON-Kartendaten wird die `json.hpp` (nlohmann::json) verwendet. Damit können die geografischen Daten aus JSON-Dateien effizient eingelesen und in Vektoren umgewandelt werden, sodass die Karten dynamisch im Programm angezeigt und animiert werden können.

Mit der Funktion `loadMapFromGeoJSON` wird eine GeoJSON-Datei geöffnet und eingelesen. Die geografischen Koordinaten aus der Datei werden mit Hilfe von `json.hpp` ausgelesen und in Bildschirmkoordinaten umgerechnet. So entstehen Vektoren, die die Umrisse der Karte beschreiben und später mit OpenGL als Linien oder Flächen angezeigt werden. Dadurch kann die Karte aus echten Geodaten direkt im Programm dargestellt werden.

```

1 #include "json.hpp"
2 using json = nlohmann::json;
3
4 ...
5
6 std::vector<std::vector<glm::vec2>> loadMapFromGeoJSON(const
7     std::string& filename, const MapNorm& norm) {

```

```
8     std::ifstream infile(filename);
9     json j;
10    infile >> j;
11
12    for (const auto& feature : j["features"]) {
13        // ... Verarbeitung der GeoJSON-Daten ...
14    }
15    return map;
16 }
```

Listing 15: Verwendung von nlohmann::json für GeoJSON in LocateScene.cpp

6. Kernpunkte des Codes

6.1. Login-Szene

Die Login-Szene ist der Startpunkt unserer Demo und simuliert den Anmeldeprozess an einem Retro-Computer-Terminal. Sie besteht aus mehreren Teilen, die nachfolgend beschrieben werden:

1. **Username-Typing-Animation:** Diese Animation zeigt, wie ein Benutzername auf einem Terminal eingegeben wird. Sie wird durch die *typedUsername*-Variable gesteuert, die den aktuellen Text anzeigt, der auf dem Terminal angezeigt wird.
2. **Password-Typing-Animation:** Diese Animation zeigt, wie ein Passwort auf einem Terminal eingegeben wird. Die Animation wird durch die *typedPassword*-Variable gesteuert, die den aktuellen Text anzeigt, der auf dem Terminal angezeigt wird.
3. **Verifying-Animation:** Diese Animation zeigt, wie der Login-Prozess verifiziert wird. Die Animation wird durch die *verifyingDots*-Variable gesteuert, die die Anzahl der Punkte anzeigt, die auf dem Terminal angezeigt werden.
4. **Access Granted-Animation:** Diese Animation zeigt, dass der Login-Prozess erfolgreich war und der Zugriff gewährt wurde. Sie zeigt nach dem erfolgreichen Login die Meldung „ACCESS GRANTED“ an.

Die Szene verwendet auch Callback-Funktionen, um bestimmte Ereignisse zu handhaben. Dies wird verwendet, um bei der Typing-Animation bei jedem neuen Zeichen einen Soundeffekt abzuspielen. Der Code für die Login-Szene ist in mehreren Funktionen aufgeteilt:

- **update():** steuert die Animation im Login-Bereich. Zuerst wird der Benutzername Buchstabe für Buchstabe angezeigt, danach das Passwort. Nach dem Passwort folgt eine kurze Verifizierung mit Punkten, und am Ende erscheint „ACCESS GRANTED“. Bei jedem neuen Zeichen wird ein Soundeffekt abgespielt.
- **render():** Zeichnet die aktuellen Inhalte des Terminals auf den Bildschirm.
- **reset():** Setzt die Szene auf ihren Anfangszustand zurück.

```
1 void LoginScene::update(float deltaTime) {
2     timer += deltaTime;
3     switch(stage) {
4         case SHOW_USERNAME:
5             if (charIndex < (int)username.length() && timer >
6                 0.18f) {
7                 typedUsername += username[charIndex++];
8                 timer = 0.0f;
9                 if (onTypeCallback) {
10                     onTypeCallback(); // Play typing sound
11                 }
12             }
13     }
14 }
```

```

12     if (charIndex == (int)username.length()) {
13         stage = WAIT_PASSWORD;
14         timer = 0.0f;
15     }
16     break;
17 // ... other cases for password and verification ...
18 }
19 }
```

Listing 16: *update*-Methode in LoginScene.cpp

Außerdem werden einige Variablen zur Verwaltung des Animations-Status verwendet.

- **stage:** Diese Variable speichert den aktuellen Zustand der Login-Szene (z.B. SHOW_USERNAME, WAIT_PASSWORD, etc.).
- **username:** Speichert den Benutzernamen, der in der Animation angezeigt wird.
- **password:** Speichert das Passwort, das in der Animation angezeigt wird.
- **typedUsername:** Speichert den aktuellen Text, der in der Username-Typing-Animation angezeigt wird.
- **typedPassword:** Speichert den aktuellen Text, der in der Password-Typing-Animation angezeigt wird.
- **timer:** Speichert die Zeit, die seit dem letzten Update vergangen ist.

6.2. Terminal-Szene

Die Terminal-Szene ist ein wichtiger Teil der Demo, da sie die Funktionsweise eines Retro-Terminals simuliert. Die Szene besteht aus mehreren Teilen:

1. **Directory-Typing-Animation:** Diese Animation zeigt den Benutzer, wie ein Verzeichnis auf einem Retro-Terminal angezeigt wird. Die Animation wird durch die *displayedTextDirectory*-Variable gesteuert, die den aktuellen Text anzeigt, der auf dem Terminal angezeigt wird.
2. **File-Typing-Animation:** Diese Animation zeigt den Benutzer, wie eine Datei auf einem Retro-Terminal angezeigt wird. Die Animation wird durch die *displayedTextFile*-Variable gesteuert, die den aktuellen Text anzeigt, der auf dem Terminal angezeigt wird.
3. **Demo-Code-Animation:** Diese Animation zeigt den Benutzer, wie ein Code-Demo auf einem Retro-Terminal angezeigt wird. Sie wird durch die *demos*-Variable gesteuert, die den Code und die Ausgabe der Demo enthält.

Die Szene verwendet auch Callback-Funktionen, um bestimmte Ereignisse zu handhaben, was wie in `LoginScene.cpp` verwendet wird, um bei der Typing-Animation bei jedem neuen Zeichen einen Soundeffekt abzuspielen. Der Code für die Terminal-Szene ist in mehreren Funktionen aufgeteilt:

- **update()**: steuert die Animation für das Tippen im Terminal. Zuerst wird das Verzeichnis Zeichen für Zeichen angezeigt, danach die Datei. Wenn beide Animationen fertig sind, startet die Demo-Code-Animation, bei der der Code und die Ausgabe wie auf einem echten Terminal angezeigt werden. Bei jedem neuen Zeichen wird ein Soundeffekt abgespielt.
- **render()**: Zeichnet die aktuellen Inhalte des Terminals auf den Bildschirm.
- **reset()**: Setzt die Szene auf ihren Anfangszustand zurück.
- **initializeDemos()**: Diese Funktion initialisiert die Code-Demos, die in der Terminal-Szene angezeigt werden.

```

1  oid TerminalAnimation::update(float currentTime) {
2      // Update directory typing animation
3      if (animationIndex < animationTextDirectory.size() &&
4          currentTime - animationLastTime >= 0.1) {
5          displayedTextDirectory += animationTextDirectory[
6              animationIndex];
7          animationIndex++;
8          animationLastTime = currentTime;
9          if (onTypeCallback) {
10             onTypeCallback(); // Soundeffekt beim Tippen
11         }
12     }
13     // ... other animations for file and demo code ...
14 }
```

Listing 17: *update*-Methode in TerminalScene.cpp

Außerdem werden auch einige Variablen definiert, die den Zustand der Szene speichern. Die wichtigsten dieser Variablen sind:

- **animationIndex**: Speichert den aktuellen Index der Directory-Typing-Animation.
- **animationIndexFile**: Speichert den aktuellen Index der File-Typing-Animation.
- **codeLineIndex**: Speichert aktuellen Index der Code-Zeile in der Demo-Code-Animation.
- **codeCharIndex**: Speichert aktuellen Index des Zeichens in der Demo-Code-Animation.
- **demoStarted**: Gibt an, ob die Demo gestartet wurde.
- **finished**: Gibt an, ob die Demo beendet wurde.

Insgesamt ist die Terminal-Szene ein wichtiger Teil der Demo, da sie die Funktionsweise eines Retro-Terminals simuliert und den Benutzer durch eine interaktive Erfahrung führt.

6.3. Karten-Szene

Ein weiterer wichtiger Teil unserer Demo ist die aus vier verschiedenen Kartenansichten bestehende Karten-Szene. Diese Szene zeigt verschiedene geografische Regionen und beinhaltet ein Radar, welches in der Demo festgelegte Koordinaten anzeigt. Beim Treffen ebendieser Koordinaten wird ein Soundeffekt abgespielt, um den Treffer zu signalisieren. Ebenso existiert ein visuelles Feedback, das dem Benutzer hilft, die aktuellen Ziele auf der Karte zu identifizieren. Dies besteht aus rot pulsierenden Umrissen um die Ziele herum, beispielsweise die rot pulsierenden Umrisse des Saarlandes in Abbildung 10.

Die Karten-Szene besteht aus den folgenden Teilen:

1. **Kartenanzeige:** Die Karten werden aus GeoJSON-Daten geladen und in Vektoren umgewandelt, die dann auf dem Bildschirm dargestellt werden. Aktuell sind dies eine Weltkarte, eine Deutschlandkarte, eine Karte des Saarlandes und eine Karte der HTW Saar. Die Funktion `loadMapFromGeoJSON(filename, norm)` übernimmt das Einlesen und Umwandeln der GeoJSON-Daten.
2. **Radar:** Der Radar ist ein visuelles Feedback, das dem Benutzer hilft, die aktuellen Ziele auf der Karte zu identifizieren. Die Animation wird durch Variablen wie `locateAnimTimer` und entsprechende Zeichenfunktionen gesteuert.
3. **Zielmarker:** Die Zielmarker sind visuelle Indikatoren, die die Position der aktuellen Ziele auf der Karte anzeigen. Methoden wie `drawCross`, `drawCircle` und `drawPoint` übernehmen die grafische Darstellung.
4. **Treffer-Feedback:** Beim Treffen der Koordinaten wird ein Soundeffekt abgespielt, um den Treffer zu signalisieren. Dies wird durch die Integration der `SoundManager`-Klasse und die Methode `playSound` realisiert.
5. **Visuelles Feedback:** Dies besteht aus rot pulsierenden Umrissen um die Ziele herum. Die Methode `drawMapPulseCenter` erzeugt diesen Effekt.

Für die HTW Saar war keine fertige GeoJSON-Datei zum Download verfügbar. Um die Karte der HTW Saar für die Demo zu erstellen, haben wir das Online-Tool geojson.io <https://geojson.io> verwendet. Mit diesem Tool kann man direkt auf einer echten Karte Linien und Flächen zeichnen. Das Programm wandelt die gezeichneten Elemente automatisch in eine GeoJSON-Datei um, die dann heruntergeladen und im Projekt verwendet werden kann. So konnten wir die Umgebung der Hochschule präzise nachzeichnen und als GeoJSON-Daten in unsere Anwendung integrieren. Die Datei `htwsaar.geo.json` enthält alle Koordinaten von HTW Saar und wird von der Karten-Szene eingelesen und angezeigt.

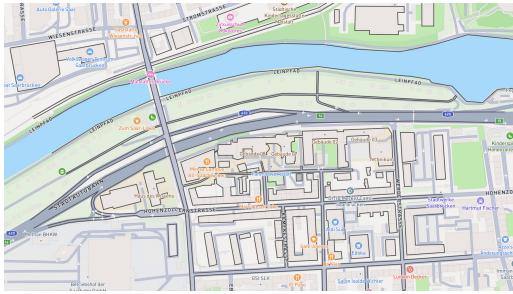


Abbildung 6: htwsaar auf geojson.io



Abbildung 7: htwsaar in Demo

Die Szene verwendet verschiedene Funktionen, um die Karten und Animationen zu aktualisieren und zu rendern.

Die wichtigsten Methoden der Klasse `LocateScene` sind:

- `LocateScene()`: Konstruktor, lädt die Karten und initialisiert die Szene.

```

1 LocateScene::LocateScene() {
2     // ...
3     worldMap = loadMapFromGeoJSON("assets/maps/world.geo.
4         json", norm_de);
5     germanyMap = loadMapFromGeoJSON("assets/maps/germany.
6         geo.json", norm_de);
7     // ...
}
```

Listing 18: Karten laden in LocateScene.cpp

- `loadMapFromGeoJSON()`: wird eine GeoJSON-Datei geöffnet und eingelesen. Die geografischen Koordinaten aus der Datei werden ausgelesen, normalisiert und in Vektoren umgewandelt. So entstehen die Umrisse der Karte, die später im Programm angezeigt werden.

```

1 #include "json.hpp"
2 using json = nlohmann::json;
3
4 std::vector<std::vector<glm::vec2>> loadMapFromGeoJSON(
5     const std::string& filename, const MapNorm& norm) {
6     std::vector<std::vector<glm::vec2>> map;
7     std::ifstream infile(filename);
8     json j;
9     infile >> j;
10
11    for (const auto& feature : j["features"]) {
12        const auto& geom = feature["geometry"];
13        std::string type = geom["type"];
14        // ... Koordinaten auslesen und normalisieren ...
15    }
16    return map;
```

16 }

Listing 19: GeoJSON-Datei einlesen und umwandeln in LocateScene.cpp

- **update()**: Aktualisiert die Animationen, Zoom und Position der Karte sowie den Fortschritt der Lokalisierung.
- **drawMap()**: zeichnet die Umrisse der Karte auf den Bildschirm. Sie bekommt die Koordinaten als Vektoren und wandelt sie mit einer Projektion und einem Maßstab um. Mit OpenGL werden die Linien dann als Polygone angezeigt, sodass die Karte sichtbar wird.

```
1 void LocateScene::drawMap(const std::vector<std::vector<
2     glm::vec2>>& mData, const glm::mat4& projection,
3                                     float cx, float cy,
4                                     float scale, glm::
5                                     vec4 color) {
6
7     glUseProgram(shaderProgram);
8     glUniformMatrix4fv(glGetUniformLocation(shaderProgram,
9         "projection"), 1, GL_FALSE, &projection[0][0]);
10    glUniform4fv(glGetUniformLocation(shaderProgram, "
11        uColor"), 1, &color[0]);
12    glBindVertexArray(vao);
13    glBindBuffer(GL_ARRAY_BUFFER, vbo);
14
15    for (const auto& polyline : mData) {
16        std::vector<float> verts;
17        for (const auto& pt : polyline) {
18            verts.push_back(cx + pt.x * scale);
19            verts.push_back(cy + pt.y * scale);
20        }
21        glBufferData(GL_ARRAY_BUFFER, verts.size() *
22                      sizeof(float), verts.data(), GL_DYNAMIC_DRAW);
23        glEnableVertexAttribArray(0);
24        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
25            (void*)0);
26        glDrawArrays(GL_LINE_STRIP, 0, verts.size() / 2);
27    }
28}
```

Listing 20: Karte zeichnen in LocateScene.cpp

- **drawCircle()**, **drawPoint()**, **drawLine()**, **drawCross()**: Übernehmen die grafische Darstellung der Karten und Markierungen.
- **drawMapPulseCenter()**: erzeugt einen pulsierenden Effekt auf der Karte. Sie berechnet den Mittelpunkt jeder Linie und skaliert die Punkte mit einem Pulsfaktor. Dadurch erscheinen die Umrisse der Zielregion auf der Karte kurzzeitig größer und heben sich visuell hervor. So wird das aktuelle Ziel für den Benutzer deutlich sichtbar gemacht.

```
1 void LocateScene::drawMapPulseCenter(
```

```

2     const std::vector<std::vector<glm::vec2>>& mapData,
3     const glm::mat4& projection, float cx, float cy, float
4     scale,
5     float pulseScale, glm::vec4 color) {
6     for (const auto& polyline : mapData) {
7         if (polyline.empty()) continue;
8         float sumX = 0, sumY = 0;
9         for (const auto& pt : polyline) {
10             sumX += pt.x;
11             sumY += pt.y;
12         }
13         float centerX = sumX / polyline.size();
14         float centerY = sumY / polyline.size();
15
16         std::vector<float> verts;
17         for (const auto& pt : polyline) {
18             float px = centerX + (pt.x - centerX) *
19                 pulseScale;
20             float py = centerY + (pt.y - centerY) *
21                 pulseScale;
22             verts.push_back(cx + px * scale);
23             verts.push_back(cy + py * scale);
24         }
25         // ... OpenGL commands to draw the pulsating
26         // effect ...
27     }
28 }

```

Listing 21: Pulsierender Effekt mit `drawMapPulseCenter` in `LocateScene.cpp`

- **render()**: Zeichnet die gesamte Szene inklusive Karte, Radar und Text.
- **reset()**: Setzt die Szene auf ihren Anfangszustand zurück und startet die Animation neu.

Außerdem werden einige Variablen definiert, die den Zustand der Szene speichern. Wichtige Variablen sind:

- **currentStep**: Gibt die aktuelle Kartenebene an (Erde, Deutschland, Saarland, HTW Saar).
- **zoom, centerX, centerY**: Steuern die Ansicht und den Fokus der Karte.
- **locateAnimTimer**: Steuert die Animation des Radar-Effekts.
- **finished**: Gibt an, ob die Lokalisierung abgeschlossen ist.

Insgesamt bietet die Klasse `LocateScene` eine dynamische und interaktive Visualisierung geografischer Daten und führt den Benutzer Schritt für Schritt durch die verschiedenen Ebenen der Standortbestimmung.

6.4. Übergänge (Kollaps des Bildes)

Ein wichtiger visueller Effekt in unserer Demo ist der Übergangseffekt, der den Bildschirm kollabieren lässt, wenn von einer Szene zur nächsten gewechselt wird.

Die Klasse `TVEffectScene` ist für die Darstellung dieses Effekts zuständig. Sie verwendet einen speziellen Shader, der mit `loadShader()` geladen wird. Das zentrale Element ist die Methode `render()`, die den Kollaps-Effekt abhängig vom Animationsfortschritt (`closeAnim`) und der aktuellen Zeit (`time`) zeichnet.

Die Szene nutzt ein Quad, das mit Texturkoordinaten versehen ist und als Bildschirmfläche dient. Die Methode `createQuad()` erzeugt die notwendigen OpenGL-Objekte (vao, vbo) und richtet die Attribute für Position und Textur ein.

Während des Übergangs werden verschiedene Uniforms an den Shader übergeben, darunter:

- **iResolution**: Die aktuelle Bildschirmauflösung.
- **iTime**: Die aktuelle Zeit für Animationseffekte.
- **closeAnim**: Der Fortschritt des Kollaps-Effekts.
- **screenTexture**: Die Textur des aktuellen Bildschirms.

Zusätzlich werden Soundeffekte ausgelöst, wenn der Übergang beginnt oder endet. Dies wird durch die Variablen `snowSoundPlayed` und `biboiSoundPlayed` gesteuert und mit der `SoundManager`-Klasse realisiert.

Die wichtigsten Methoden der Klasse `TVEffectScene` sind:

- **initialize()**: Initialisiert die Szene und lädt den Shader.
- **createQuad()**: erstellt ein Rechteck (Quad) für den Bildschirm. Sie definiert die Eckpunkte und die Texturkoordinaten, richtet die OpenGL-Objekte (VAO, VBO) ein und aktiviert die Attribute für Position und Textur. Dieses Quad wird später verwendet, um den Übergangseffekt über den ganzen Bildschirm darzustellen.

```
1 void TVEffectScene::createQuad() {
2     float quadVertices[] = {
3         // positions      // texCoords
4         0.0f, 0.0f,      0.0f, 0.0f,
5         1.0f, 0.0f,      1.0f, 0.0f,
6         1.0f, 1.0f,      1.0f, 1.0f,
7         0.0f, 1.0f,      0.0f, 1.0f
8     };
9     // ... OpenGL VAO/VBO Setup ...
10 }
```

Listing 22: Quad erzeugen in `TVEffectScene.cpp`

- **loadShader()**: Lädt den benötigten Shader für den Übergangseffekt.

```
1 bool TVEffectScene::loadShader() {
```

```

2     shaderProgram = ShaderManager::loadShader("shaders/
3         tv_effect.vert", "shaders/tv_effect.frag");
4     if (!shaderProgram) {
5         std::cerr << "Failed to load TVEffectScene shader.
6             " << std::endl;
7         return false;
8     }

```

Listing 23: Shader laden in TVEffectScene.cpp

- **render()**: Zeichnet den Kollaps-Effekt und steuert die Animation.
- **cleanup()**: Gibt die verwendeten OpenGL-Ressourcen frei.

Durch diese Struktur entsteht ein flüssiger und eindrucksvoller Übergangseffekt, der den Szenenwechsel in der Demo visuell unterstützt.

7. Resümee

7.1. Eigene Lernfortschritte

Über den Verlauf dieses Projekts haben wir viele neue Kenntnisse und Fähigkeiten erworben, darunter:

- Vertiefte Kenntnisse in OpenGL und GLSL
- Bessere Kenntnisse in der Programmiersprache C++
- Verständnis für die Funktionsweise von Demos und deren technische Umsetzung
- Hintergrundwissen über die Geschichte und Kultur der Demoszene
- Kenntnisse in der Verwendung von CMake zur Projektorganisation

7.2. Entwicklungsprobleme & Lösungen

Während der Entwicklung der Demo sind wir auf verschiedene Herausforderungen gestoßen, darunter:

- Schwierigkeiten bei der Implementierung von OpenGL-Funktionen, insbesondere bei der Handhabung von Shadern und Texturen. Implementierung der Shader an den richtigen Stellen war nicht immer trivial.
- Herausforderungen bei der Synchronisation von Audio und Video, Audio wurde nicht immer zu den gewollten Zeitpunkten abgespielt. Dieses Problem besteht zum Teil weiterhin mit dem Sonar-Sound innerhalb der Karten-Szene.
- Implementierung des Radars innerhalb unserer Karten-Szene war komplexer als erwartet, insbesondere die korrekte Darstellung und Aktualisierung der Radar-Elemente.

7.3. Denkbare Weiterentwicklungen

Die Demo könnte um weitere Funktionen und Verbesserungen erweitert werden, wie z.B.:

- Aktuell gibt es keine Hintergrund-Musik, die gesamte Audio-Untermalung besteht aus Soundeffekten. Es gibt die Möglichkeit, dies durch die Implementierung von passender Chiptune-Musik zu verbessern.
- Die Bildschirmauflösung der Demo kann nach Kompilierung mithilfe einer Config-Datei angepasst werden. Allerdings skalieren noch nicht alle Elemente der Demo korrekt mit. Dies könnte durch eine verbesserte Berechnung der Positionen und Größen der Elemente behoben werden.
- Die Demo läuft aktuell in einer Endlosschleife mit der Möglichkeit, die Demo durch drücken von Escape zu beenden. Es wäre denkbar, einen Mechanismus

einzbauen, der es dem Benutzer ermöglicht, die Demo zu pausieren oder zu stoppen.

- Zusätzlich zur Möglichkeit der Pausierung wäre es denkbar, dem Benutzer die Möglichkeit zu geben, zwischen den verschiedenen Szenen zu wechseln.

Quellenverzeichnis

- [Ata80] Atari. Atari in-store demonstration program by atari, 1980 | atari 8 bit. <https://www.youtube.com/watch?v=Cj5qt1Q74J0>, 1980. Zugriff am 23.08.2025.
- [Cre93] Future Crew. Second reality. <https://www.youtube.com/watch?v=iw17c70uJes>, 1993. Zugriff am 25.08.2025.
- [de03] digitalekultur e.V. Was ist die demoscene? https://www.digitalekultur.org/files/dk_wasistdiedemoszene.pdf, 2003. Zugriff am 23.08.2025.
- [Dem] Demoscene.info. The demoscene. <https://www.demoscene.info/index.html>. Zugriff am 24.08.2025.
- [fam15] Famous people who came from the demoscene. <https://chipflip.wordpress.com/2015/06/12/famous-people-who-came-from-the-demoscene/>, 2015. Zugriff am 25.08.2025.
- [Far00] Farbrausch. The product. <https://www.youtube.com/watch?v=QqHTwtXvYW4>, 2000. Zugriff am 25.08.2025.
- [Loh25] Niels Lohmann. Json for modern c++. <https://github.com/nlohmann/json>, 2013–2025. Zugriff am 24.08.2025.
- [Sco90] Scoopex. Mental hangover. <https://www.youtube.com/watch?v=zR9a4vTuuoo>, 1990. Zugriff am 25.08.2025.
- [Sen17] Dr. William Sen. Demoscene: Scoopex - generations ahead. <https://www.digitalwelt.org/themen/subkulturen/demoszene-scoopex>, 2017. Zugriff am 25.08.2025.
- [the04] theprodukt. .werkzeug1 v1.201. https://files.scene.org/view/resources/demomaker/theprodukt/pno0002_werkzeug1_v1201.zip, 2004. Downloadlink, Zugriff am 25.08.2025.

Anhang

A. Screenshots der Demo

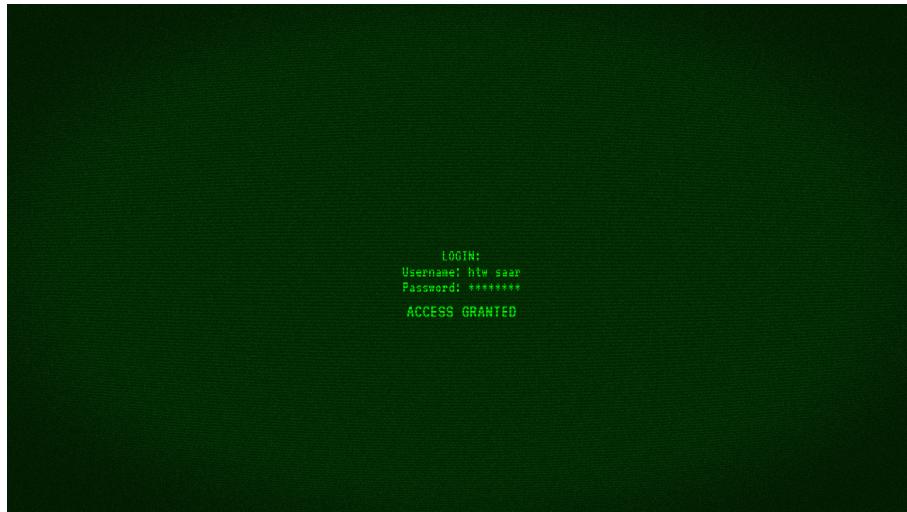


Abbildung 8: Login-Screen unserer Demo

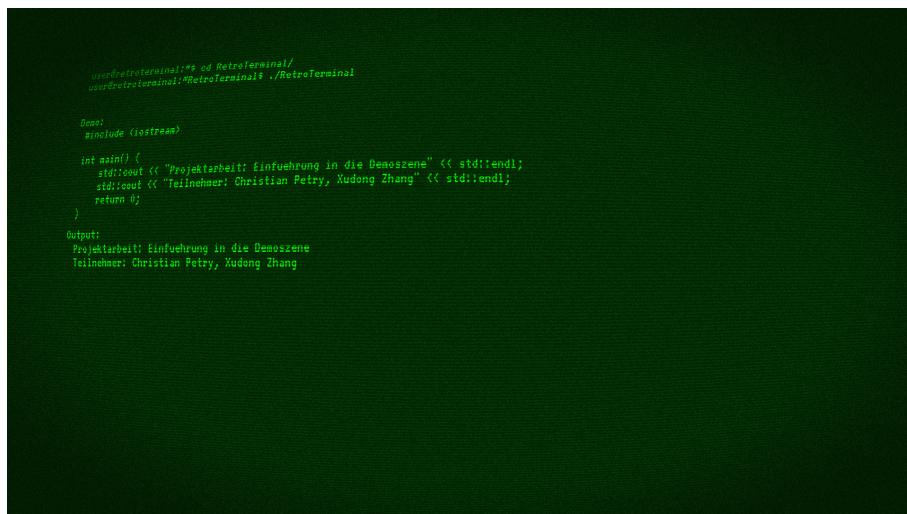


Abbildung 9: Terminal-Screen unserer Demo

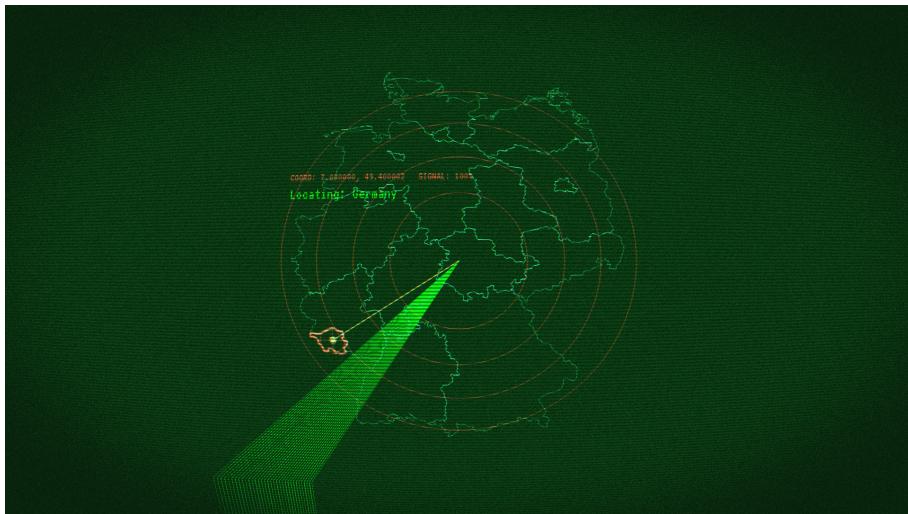


Abbildung 10: Teil der Karten-Szene unserer Demo, zweite Stage, zeigt Deutschland

B. Komplilieranleitung

Die Demo kommt standardmäßig mit einer kompilierten Version der Anwendung. Zur Komplilierung der Demo wird die MSYS2 MinGW64-Umgebung benötigt. Die folgenden Schritte sind erforderlich:

1. Installieren Sie MSYS2 von <https://www.msys2.org/>.
2. Öffnen Sie die MSYS2 MinGW64-Shell.
3. Aktualisieren Sie die Paketdatenbank und installieren Sie die benötigten Pakete:

```
pacman -Syy
pacman -S --needed mingw-w64-x86_64-toolchain
pacman -S mingw-w64-x86_64-cmake
pacman -S mingw-w64-x86_64-glfw
pacman -S mingw-w64-x86_64-freetype
pacman -S mingw-w64-x86_64-glm
pacman -S mingw-w64-x86_64-sfml
```

4. Klonen Sie das Repository und wechseln Sie in das Verzeichnis `pa_demoszene`
 5. Komplilierung des Projekts:
- ```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
mingw32-make
```
6. Wechseln Sie zurück in das Oberverzeichnis durch `cd ..`
  7. Nach der Komplilierung finden Sie die ausführbare Datei im Verzeichnis `pa_demoszene`.

8. Starten der Anwendung entweder durch Ausführen der beiliegenden `start.bat` oder durch Eingabe des Befehls `./RetroTerminal` in der MinGW64-Shell.

## C. Zusätzliche Materialien

- Quellcode der Demo (im Verzeichnis `Demo-Code/src`)
- Assets wie Sounds und Karten (im Verzeichnis `Demo-Code/assets`)
- Quellcode der LaTeX-Ausarbeitung (im Verzeichnis `Ausarbeitung`)