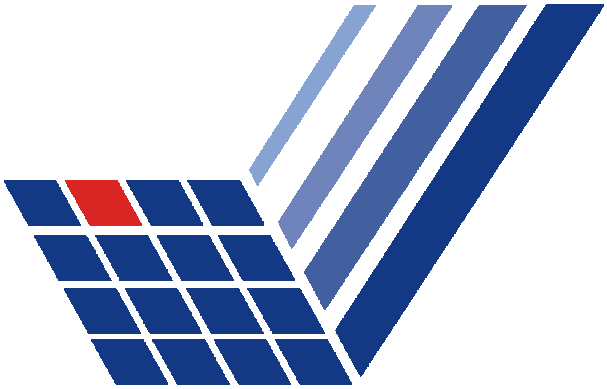
Universität Dortmund



Fachbereich Informatik
Lehrstuhl VII Graphische Systeme

Diplomarbeit

Kameraführung in virtuellen Räumen mittels reduzierter Voronoi Graphen

von
Martin Otten

Betreuer der Arbeit:
Prof. Dr. Heinrich Müller
Dipl. Inf. Jörg Ayasse

„Nicht in der Erkenntnis liegt das Glück, sondern im Erwerben der Erkenntnis“

Edgar Allan Poe (1809 - 1849)

Danksagung

Mit dieser Diplomarbeit beende ich meine knapp zwanzigjährige Schul- und Studienausbildung und möchte sie daher jenen Menschen widmen, welche mich in dieser Zeit begleitet haben.

Zuerst bedanke ich mich bei meinen Eltern und Geschwistern, welche mich auf all meinen Wegen unterstützt und gefördert haben und mir alle Türen eröffneten, durch die meine Wünsche und Ziele zu erreichen waren.

Ich bedanke mich bei meinen alten Freunden aus Oberhausen, mit denen ich unsere gemeinsame Jugend und Schulzeit durchlebt habe. Ich danke auch meinen neuen Freunden in Dortmund für die Hilfe im Studium und die wunderbare Freizeit. Ein spezieller Dank geht an jene Helfer, die mir bei der Korrektur dieser Arbeit mit Rat und Tat beiseite standen.

Natürlich bedanke ich mich bei den Lehrern, Dozenten und Professoren, welchen ich oft mehr als eine exzellente Ausbildung verdanke. Den Mitarbeitern meiner Firma danke ich für ihr Vertauen und für die Möglichkeit, an einer großartigen Arbeit teilzuhaben.

Martin Otten, Dortmund den 21.03.2003

Kurzfassung

In dieser Arbeit wird mit dem Voronoi Distanz Graph (VDG) eine Struktur vorgestellt, welche schnelle und natürliche Kameraführung in virtuellen, 3-dimensionalen Echtzeitsystemen ermöglicht. Diese Struktur verbindet die Eigenschaften der globalen Navigation über Wegegraphen und der lokalen Navigation mittels Potentialfeldern. Sie repräsentiert den statischen, freien Raum in der virtuellen Welt durch einen erweiterten Voronoi Graphen (Medial Achse). Um während der Laufzeit im Echtzeitsystem möglichst wenig Speicher- und Rechenzeitressourcen zu benötigen, wird der Voronoi Graph in einem einmaligen Vorverarbeitungsprozess auf ein mögliches Minimum reduziert. Es wird ein Kamera Management entworfen, welches auf diesem Graphen basiert und gezeigt, dass über Kostenfunktionen alle wichtigen Kamerakriterien mit dieser Struktur berechenbar sind.

Die Implementation zeigt, wie die automatische Erstellung und Reduktion dieser Voronoi Graphen in den Entwicklungsprozess des Computerspiel *Half-Life* eingebunden werden kann. Die praktischen Umsetzung des Kamera Managements in diesem Spiel ergibt, dass der Voronoi Distanz Graph seinen geforderten Eigenschaften unter realistischen Bedingungen gerecht wird.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Alle höheren Lebewesen spielen, sowohl Tiere als auch Menschen. In der Kindheit ist Spielen ein wichtiges Element des allgemeinen Lernprozesses zum Verständnis der Umwelt und Gesellschaft. Aus dem Spiel wird in der Jugend oft ein Sport, bei dem der Wettkampf und Leistungsvergleich mit anderen eine wesentliche Rolle einnimmt. Später werden Spiele meistens zur geistigen Zerstreuung und zu gesellschaftlichen Zusammenkünften genutzt. Spielen bedeutet, dass sich die teilnehmenden Spieler auf eine abstakte Welt mit formalisierten Regeln verständigen. Diese Welt ist oft ein vereinfachtes Abbild von Teilen der realen Welt, dessen Ursprung besonders bei alten Spielen nicht immer erkennbar ist. Die meisten Spiele definieren Ziele in dieser Welt, welche durch den Spieler zu erreichen sind und mit denen das Spiel endet. Im Allgemeinen haben Handlungen und Ereignisse in dieser Spielwelt keine Auswirkungen auf die Spieler in der realen Welt. Menschen lernen durch das Spielen in der Kindheit zwischen der fiktiven Spielwelt und der realen Welt zu unterschieden [Häf99].

Hauptsächlich durch die enorme Steigerung der visuellen Qualität von der monochromen Textdarstellung bis zur photorealistischen 3-dimensionalen Welt in den letzten 20 Jahren haben sich der Personal-Computer und die Spiel-Konsole zu einem sehr beliebten Spielzeug entwickelt. Die Flexibilität dieser Systeme ergibt sich durch die austauschbare Software, welche beliebig viele und unterschiedliche Spielarten ermöglichen. Die ernorme Rechenleistung heutiger Spielsysteme ermöglicht die Erschaffung sehr komplexer und realistischer Scheinwelten, in die sich der Spieler versetzen kann. Des weiteren ist ein Spieler nicht auf die direkte Präsents eines Mitspielers angewiesen. Zum einen bietet der immense Inhalt und die künstliche Intelligenz der heutigen Spiele eine dauerhafte Abwechslung, zum anderen sind durch schnelle, elektronische Kommunikationswege (DFÜ, Internet) fast immer andere menschliche Mitspieler virtuell verfügbar.

Die graphische Ausgabe ist die wichtigste Schnittstelle zwischen dem Spieler und der Spielwelt, gefolgt von der Ton- und Musikwiedergabe. Im Laufe der Zeit haben sich die meisten Spiele von 2-dimensionalen Welten mit fester Ansicht zu 3-dimensionalen Spielen mit bewegter Ansicht gewandelt. Bei den ersten Spielen dieser Generation handelte es sich um First-Person Spielen, Flug- oder Autosimulationen, bei denen sich die Sicht direkt aus der Position und Blickrichtung des vom Spieler gesteuerten Charakters ergibt. Diese Sicht bewirkt bei dem Spieler ein sehr intensives Gefühl, selber aktiver Teil der Spielwelt zu sein. Es gibt aber auch Spiele, bei denen der Spieler seinen Charakter von außen betrachtet (*Third-Person View*). In diesem Fall erlebt der Spieler die Geschichte weniger selbst als mit seinem Helden[1]. Mit dem Aufkommen von Mehrspieler-Spielen (*Multiplayer-Games*) wurde dieser Sichtmodus bei Zuschauern sehr beliebt, welche passiv das Spiel verfolgen und dabei anderen Spielern „über die Schulter“ schauen. Die Ausrichtung und Bewegungen dieser externen Zuschauersichten werden teilweise automatisch generiert, so dass ein Zuschauer die Handlungen eines gewünschten Spielers über einen längern Zeitraum ohne eigenes Eingreifen optimal verfolgen kann. Damit ergibt sich das Problem der automatischen Kameraführung.

Automatischen Kameraführung in virtuellen Räumen gliedert sich in zwei Problembereiche. Zum einen ist es wichtig, bereits vorhandenes kinematographisches Wissen über Kamerafüh-

[1] Im Spieldesign sind die Spieler immer gute Helden, die zum Schluss gewinnen werden

rung aus dem Bereich des klassischen Films in diesem neuen Kontext zu nutzen. Dies ist sinnvoll, da der narrative Film und Computer-Spiele viele Gemeinsamkeiten in Bezug auf Inhalt und Darstellung haben. Beide versetzen den Zuschauer oder Spieler in eine fiktive Welt und erzählen über bewegte Bilder und Ton eine emotionale oder dramatische Geschichte. Die Kameraführung hat dabei die Aufgabe, den Betrachter durch die Handlungen zu führen und bestimmte Stimmungen und Eindrücke gezielt hervorzuheben. Das Formalisieren, Strukturieren und Anwenden dieses Wissen wird in der Informatik durch die Teilbereiche des Wissens-Managements und der künstlichen Intelligenz abgedeckt.

Das andere Problem ergibt sich in der praktischen Umsetzung einer bestimmten Kameraeinstellung, welche aus dem kinematographischen Wissen für eine bestimmte Szene ermittelten wurde. In virtuellen Welten existieren in den meisten Fällen feste Objekte und Strukturen, welche die Sicht des Zuschauers begrenzen. Diese Hindernisse dürfen die Sicht auf den aktuellen Charakter nicht störend oder über längern Zeitraum verdecken. Bei Weltraum- oder Fugsimulationen ist dies in der Regel kein Problem, da man sich weitestgehend im freien Raum bewegt. Werden jedoch menschliche Akteure in engen, durch Hindernisse stark beschränkten Räumen (z.B. Höhlen, Wälder oder in Gebäuden) verfolgt, sind die Möglichkeiten der Kamerabewegung sehr eingeschränkt. Für menschliche Kameraoperateure bei konventionellen Filmproduktionen ist dies kein Problem, da zum einen die Handlung über die gesamte Zeit durch das Drehbuch festgelegt ist und zum anderen haben sie einen sehr ausgeprägtes räumliches Denken . Für Menschen ist es kein Problem sich schnell und zielgerichtet durch komplexe Räume zu bewegen und dabei eine stetige und harmonische Bewegungsform beizubehalten. Bei Computerspielen ist die Handlung nicht fest vorgegeben und wird durch die Spieler selbst bestimmt. Die Navigation der automatischen Kamera in komplexen, virtuellen Räumen ist ein weiteres Problem, welches dadurch verschärft wird, dass Computerspiele in der Regel Echtzeitsysteme sind und nur begrenzt Rechenzeit zur Verfügung steht. Mit diesen Problemen beschäftigen sich die Wissenschaften der virtuellen Realität, grafischen Systemen und Roboternavigation.
Diese Arbeit adressiert des letztere Problem und versucht mit dem Voronoi Distanz Graphen eine Datenstruktur und Verfahren zu entwickeln, welches einer automatischen Kamera im virtuellen Raum hilft, schnell über weite Strecken zu Navigieren und lokal eine aussagekräftige Darstellung des umgebenen Raumes zu liefern. In der Einleitung werden die historische Entwicklung der Kameraführung und Computerspiele kurz umschrieben. Die Grundlagen bereits erfolgter Forschung auf diesem Gebiet wird im zweiten Kapitel erläutert, um dann darauf basierend das neue Verfahren zu entwickelt. Anschließend muss die neue Datenstruktur wegen der geforderten Echtzeittauglichkeit auf ein mögliches Minimum an Umfang reduziert werden. Danach wird das Kamera Management für die Navigation der Kamera vorgestellt, welches die neue Struktur nutzt. Um den praktischen Nutzen dieses Verfahrens zu Testen, wurde es in ein aktuelle Computer-Spiel implementiert und die Ergebnisse zum Schluss ausgewertet.

1.2 Kameraführung

Die Geschichte der Kinematographie beginnt am 28. Dezember 1895 mit der ersten Filmvorführung vor Publikum im „Grand Café“ in Paris. Die französischen Gebrüder Lumière nannten ihre Erfindung Kinematographen und gründeten mit der Filmindustrie den wohl wichtigsten Bestandteil der modernen Unterhaltung neben der Musikbranche. Die Ausdrucksform des Films wurde im Laufe der Zeit immer durch die technischen Möglichkeiten bestimmt. Zu Beginn waren nur feste Einstellungen möglich, da die Kameraapparaturen zu empfindlich und schwer waren, um bewegt zu werden. So beschränkten sich die kreativen Möglichkeiten auf Zeitraffer, Doppelbelichtung und anderen tricktechnischen Methoden (*Reise zum Mond* von 1902). Eigen-

bewegungen der Kamera entwickelten sich relativ langsam. Der Schwenk und die Neigung ist technisch mit einem einfachen Stativ-Schwenkkopf bestehend aus Platten und Kugellagern zu realisieren. Das Rollen (Rotation um die Blickachse) ist technisch bis heute sehr aufwendig und wird als ästhetisches Mittel nur sehr selten verwendet (z.B. tiefes Fallen). Es widerspricht der menschlichen Orientierung, denn das Gehirn versucht, Rollen mittels des Gleichgewichtssinns auszugleichen. Horizontale Bewegungen sind zu einem festen Bestandteil der Kameraführung geworden. Hierzu wird die Kamera auf einen Wagen (*doll*) montiert, welcher sich entweder auf einer festen Schiene (*track*) bewegt oder freibeweglich auf Gummireifen fährt. Ein Kamerakran ermöglicht flüssige vertikale Bewegungen, wobei die Kamera auf einer Art Wippe montiert wird, welche durch Gewichte oder die Kameramannschaft am anderen Ende ausbalanciert wird. Kombiniert man ein drehbares Stativ mit einem Kamerakran, welcher wiederum auf einem Kamerawagen fixiert ist, erhält die Kamera eine beliebige räumliche Bewegungsfreiheit. Die Größe einer solchen Apparatur verhindert jedoch den Einsatz bei Innenaufnahmen.

Erst in den 50iger Jahren des 20. Jahrhunderts wurden Kameras so kompakt (35mm Arnold & Richter Arriflex), dass sie von einem einzelnen Kameramann mit der Hand geführt worden konnten und erlaubten eine vollkommen neue Freiheit und Bewegungsformen. Zum ersten mal konnten Kameras so geführt werden, als ob sie direkt durch die Augen einer beteiligten Person blicken würden und nicht von einem unbeteiligten Beobachter (diese leichte Handführung hat sich bis heute als typischer Stil des Dokumentarfilms gehalten). Durch das geringe Kameragewicht und fehlende mechanische Führung neigten manuelle Schwenks und Bewegungen jedoch dazu, stark zu verwackelten und eigneten sich für den klassischen Stil daher wenig.

Mit der Entwicklung des Steadicam-Systems in den 70igern wurde der handgeführten Kamera die Ruhe verliehen, wie man sie von den Bewegungen der mechanisch geführten Kameras kannte. Beim Steadicam-System trägt der Kameramann eine fest anliegende Weste, an welche ein kurzer mehrgelenkigen Kameraarm montiert ist. Dieser Arm ist mit Gewichten und Federn versehen und dämpft so die Bewegungen der Kamera. Der Kameramann muss die Aufnahmen nicht mehr durch einen Sucher an der Kamera kontrollieren, sondern kann ihr Bild über einen externen Video-Monitor betrachten. Die Bedienung dieses Systems erfordert einige körperlich Übung, aber ermöglicht gleichzeitig sehr dynamische und flüssige Aufnahmen (*Rocky* von 1976). Vorläufiger Stand der technischen Entwicklung ist das Skycam-System (*Star Wars* von 1979), hier sind Kameramann und Kamera komplett entkoppelt. Eine Leichtgewicht-Kamera ist an dünnen Stahldrähten aufgehängt, welche durch Elektromotoren an Führungsmasten aufgespannt und bewegt werden. Die Kamera kann so jeden Punkt zwischen den Masten erreichen und präzise Bewegungen können durch die externe, computergestützte Steuerung beliebig häufig wiederholt werden [Mon00].

1.3 Filmsprache

Betrachtet man einen aktuellen Film im Kino oder Fernsehen, wird eine ganz bestimmt Art und Weise erwartet, wie der Inhalt dem Zuschauer präsentiert wird. Diese Normen prägen die filmische Auffassung eines modernen Menschen seit seinem ersten Kontakt mit diesem Medium. Man spricht in diesem Zusammenhang von der „Sprache des Films“, welche von den meisten Zuschauern unbewusst interpretiert werden kann, aber deren inneren Strukturen und Regeln selbst nicht wahrgenommen werden. Entwickelt hat sich diese Sprache in einem evolutionären Prozess besonders in der Anfangszeit des Films zwischen den Jahren 1910 und 1940. Regisseure, Schauspieler und Produzenten „sprechen“ diese Sprache, verstehen ihre Syntax und wissen welche Elemente bestimmte Effekte erzielen. Heute existieren allgemein verwendete Standardwerke, welche das gesamte Sprachwissen strukturieren und zusammenfassen, z.B. [Ari76]. Die Filmsprache unterliegt weiterhin Änderungen, welche jedoch die Basiskonzepte nur erweitern

und kaum grundlegend ändern. Typische Hollywood Produktionen halten sich strikt an diese Konventionen, womit sichergestellt wird, dass der Film vom breiten Publikum verstanden und akzeptiert wird.

Die kleinste Einheit der Filmsprache ist die Einstellung, eine einzelne, ununterbrochene Aufnahme einer statischen oder bewegten Kamera. Mehrere Einstellungen setzten sich zu einer Szene zusammen und sind durch Schnitte oder Überblendungen getrennt. Eine Szene beschreibt eine zusammenhängende Handlungseinheit der Filmerzählung und ist kontinuierlich in Raum und Zeit. Ist diese Kontinuität unterbrochen, spricht man von einer Sequenz. Es ist unüblich, dass eine Szene aus einer einzigen Einstellung besteht, außer es ist ein gewisser Dokumentarfilm-Charakter erwünscht.

Im Mittelpunkt der meisten Szenen stehen Handlungen und Dialoge von Akteuren. Durch die Blickrichtung (Handlungsrichtung) eines einzelnen Akteurs oder beim Dialog zwischen zwei Gesprächspartner wird eine Beziehungsachse im Raum definiert. Eine generelle Regel besagt, dass alle Einstellungen einer Szene diese gedachte Achse nicht überqueren dürfen. So fällt es dem Zuschauer leichter die jeweiligen Position der Akteure und Gegenstände zu verfolgen und in Beziehung zu setzen. Kameraeinstellungen werden je nach Lage zu dieser Achse in interne, externe und Scheitelpunktaufnahmen eingeteilt. Durch Entfernung und Sichtwinkel der Kamera wird die Einstellungsgröße der Akteure in Stufen zwischen Totale, Nah- und Großaufnahmen unterteilt. Die Bildkomposition beschreibt die Lage der gezeigten Akteure und Objekte und welchen Anteil der Bildfläche sie in Anspruch nehmen. Häufig kommt dabei das Prinzip des Goldenen Schnitts zur Anwendung, wobei die agierende Person den größeren Teil beansprucht und das Zielobjekt ihrer Handlung im restlich Anteil liegt.

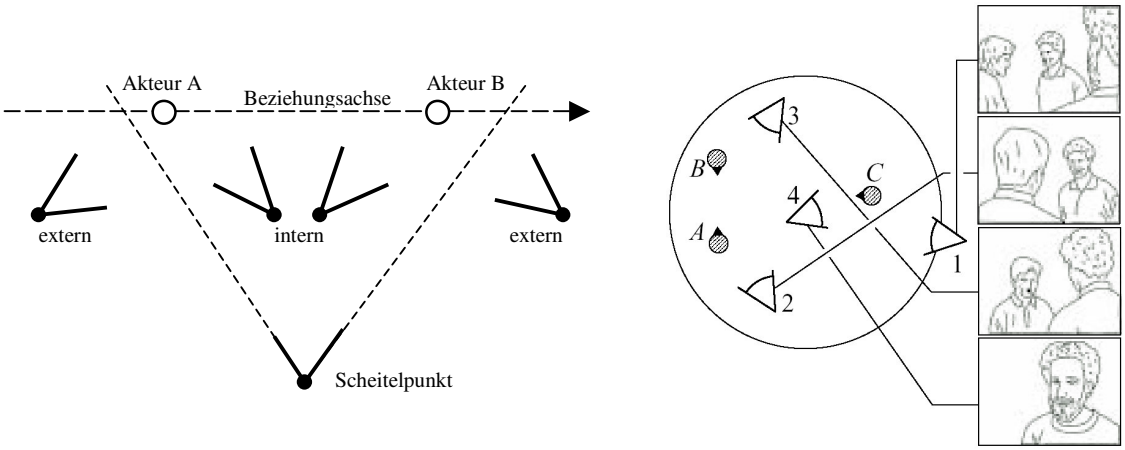


Abbildung 1.1: Elemente der Filmsprache: Kamerapositionen (l.), Ablauf eines Dialoges (r.)

Weiter kann der Regisseur durch Unschärfe und Fokussierung den Blick des Betrachters lenken und so selektives Sehen nachbilden. Dabei ist es wichtig, erst einen generellen Überblick der Szene zu vermitteln, um dann den Blick auf agierende und reagierende Akteure zu richten, wobei Aktion und folgende Reaktion klar hervorgehoben werden sollte. Elemente, welche die Aufmerksamkeit des Zuschauers auf sich ziehen, ohne dass sie zur direkten Handlung gehören, sind zu vermeiden.

Zusammenfassend kann man sagen, dass mit solchen Begriffen und Regeln bereits alle möglichen Standardsituationen wie Dialoge, Verfolgungen oder Actionszenen mit zwei, drei oder mehr beteiligten Akteuren unter allen möglichen Umständen beschrieben sind. Sie haben sich

mit der Zeit als hilfreich erwiesen, dem Zuschauer den Handlungsverlauf verständlich zu vermitteln und gewünschte Eindrücke und Stimmungen zu unterstreichen [Mon00].

1.4 Virtuelle Welten

In den letzten 15 Jahren hat sich die Idee der „virtuelle Umgebungen“ (*virtual environmens*, VE) und „virtuelle Realität“ (*virtual reality*, VR) auch in alltäglichen Anwendungen durchgesetzt. Was bis dato nur dem Militär, Forschung und der Industrie vorbehalten war, hält nun Dank rasanter Entwicklungen der Rechen- und Grafikleistung auch in kleingewerblichen und privaten Bereichen Einzug. Mittlerweile ist jeder handelsübliche PC mit einem 3D Grafikbeschleuniger ausgestattet [2].

Die ursprüngliche Idee der VR wurde 1965 von Ivan Sutherland formuliert: „Ein Fenster in eine scheinbare Welt zu schaffen, welche echt aussieht, echt klingt und sich echt anfühlt und realistisch auf Aktionen des Betrachters reagiert“ [Suth65]. VR-Systeme versuchen Eingaben des menschlichen Benutzers auf möglichst natürliche Weise aufzunehmen. Der Mensch soll mit dem VR-System wie mit der normalen Welt interagieren, also durch Körperbewegungen und Sprache.

Durch diese Eingaben verändert sich die virtuelle Welt entsprechend den physikalisch und logischen Gesetzen, wie sie man auch in der alltäglichen Welt erwarten würde. Diese Veränderungen und deren Darstellung geschehen in der virtuellen Welt mit der gleichen Geschwindigkeit wie in der realen Welt und jede Aktion des Benutzers führt zu einer sofortigen Reaktion des Systems. Diese Interaktivität ist die wichtigste Eigenschaft, die allen virtuellen Umgebungen gemeinsam ist. Ebenso soll die virtuelle Welt vom Benutzer möglichst lebensnah aufgenommen werden und viele Sinnesorgane stimulieren. Eine solche Schnittstelle zwischen Mensch und System sollte möglichst wenig ins das Bewusstsein treten und so die sensorische Aufnahmen stören.

Die Erforschung solcher lebensnaher Mensch-Maschine Schnittstellen hat seitdem eine enormes Spektrum an Lösungsansätzen hervorgebracht. Bei der Eingabe wird die Bewegung des menschlichen Körpers oder Teilen davon (Kopf, Hände) im Raum mechanisch oder optisch verfolgt. Bei der Präsentation der Welt werden vorrangig die audio- und visuellen Sinne des Menschen bedient, da sie zusammen mehr als 90% der Wahrnehmung ausmachen. Dazu zählen 3D-Bildschirme, Videobrillen und Raumklang-Systeme.

Die Anwendungsmöglichkeiten solcher VR-Systeme sind vielfältig: Simulation, Ausbildung, Entwicklung, Tele-Arbeit und Unterhaltung. Leider hat seit der Einführung der Computermaus (1965) und des Rasterbildschirms (1974) keine der oben genannten Mensch-Maschine Schnittstellen ihren Weg ins alltägliche Leben gefunden. Die Apparaturen sind entweder nicht ergonomisch genug oder in der Produktion für private Endkunden zu teuer (z.B. Head-Mouted Displays, Data Gloves etc.) [Maz96].

[2] hauptsächlich um 3D Spiele zu ermöglichen

1.5 Computer-Spiele

Das erste Computer-Spiel der Welt wurde 1962 am MIT entwickelt und hieß *Spacewar!*. Das Raumsschiffspiel wurde in 200 Stunden von dem Studenten Stephen Russell auf einer PDP-1 programmiert und die graphische Ausgabe erfolgte als Linien auf einem runden CRT Bildschirm. Zwei Spieler steuerten jeweils ein Raumsschiff und mussten versuchen, sich gegenseitig mit Raketen abzuschießen [3]. An eine kommerzielle Verbreitung war zu dieser Zeit nicht zu denken, da die benötigte Rechen- und Grafikleistung viel zu kostspielig war. Den ersten größeren Erfolg erreichte im Jahre 1970 eine für TV-Geräte entwickelte Spielkonsole *Odyssey*, welche über 200.000 mal verkauf worden ist. Diese Spielkonsole wurde über zwei Drehknöpfe (*paddles*) gesteuert und hatte einfachste Balkengraphik und mehrere Spielmodi [Kon02].

Mit der Verbreitung von vernetzten Text-Terminals entstand nach den graphischen Geschicklichkeitsspielen, welche kaum Variationen boten, ein neues Genre des Computerspiels, das Textabenteuer (*text adventure*). Das ebenfalls von MIT Studenten entwickelte *Zork* schaffte 1980 den Durchbruch und verbreitete dieses Spielkonzept auch außerhalb von Forschungs- und Firmennetzwerken. Der Spieler konnte ganze Geschichten mit seinem Held in einer fiktiven Welt erleben. Die Welt wurde durch kurze Texte beschrieben, wobei der Spieler mittels kurzer Textkommandos die Welt untersuchen konnte oder sich mit anderen Charakteren unterhalten. Die Ein- und Ausgabe war zwar etwas umständlich, brauchte jedoch keine teure Hardware und bot Stundenlage Unterhaltung.

Abbildung 1.2: Zork (1980), Pitfall (1984), Last Ninja (1988)

Seit 1980 konkurrierten auf dem Videospielemarkt hauptsächlich zwei unterschiedliche Spielsysteme: der Home-Computer und später der Personal-Computer gegen die reinen Spielkonsolen mit abwechselndem Erfolg. Beide Systeme verfügen über vergleichbare Rechen-, Speicher-, Grafik- und Tonleistung, wobei die Spielkonsolen wesentlich billiger sind, aber weniger flexible als die Computer. Damals hatten Spiele in der Regel eine mehrfarbige 2D-Darstellung der Welt, entweder fest von oben (*top-down*) oder von der Seite (*side-view*), z.B. *Pitfall* von 1984. Später bewegte sich die Ansicht parallel zu den Bewegungen der Spieler mit (*side-scoller*). Mit steigender Rechenleistung wurden pseudo-3D Spiele entwickelt, welche denn Charakter seitlich von oben zeigten und einen Tiefeneindruck erzeugten (i*sometric view*). Trotzdem waren bei diesen Spielen alle Grafiken vorher perspektivisch gemalt und Drehungen der Objekte war nur in festen Winkelabständen (z.B. 45°) möglich, wie z.B. in *Last Ninja* von 1988.

Mit Anfang der neunziger Jahre setzten sich immer mehr echte 3-dimensionalen Spiele auf den Markt durch, welche über simple Drahtgittermodelle hinausgehen. Im Bereich der Flug- oder Autosimulationen war dies ein stetiger Prozess, für Abenteuer- und Actionspiele kam der Durchbruch schlagartig. Zuerst reichte die Rechenkapazität für aufwendige, dynamische Szenen nicht aus und man erlaubte nur feste Kameraansichten (*fixed third-person*). In ein vorher erstell-

[3] Dieses grundsätzliche Spielprinzip hat sich bis heute als erfolgreich bewiesen

tes Standbild, welches auch Tiefeninformation enthält, werden freibewegliche Akteure und Gegenstände als 3-dimensionale Objekte gezeichnet (z.B. *Alone in the Dark* von 1992). Diese Technik ist sehr einfach, schnell und erlaubt eine exzellente Darstellungsqualität, weshalb sich dieses Verfahren lange gehalten hat (besonders im Abenteuer-Bereich). Mit *Wolfenstein 3D* und letztendlich *Doom* von 1993 hat id Software das Action-Spiel neu definiert. Durch schnelle Raycasting und BSP Verfahren war es nun möglich, vollkommen frei durch komplexe 3-dimensional Welten zu laufen, welche vollständig mit Texturen überzogen sind. Der Spieler sieht die Welt durch die Augen seines gesteuerten Charakters, daher der Name diese Spielgenre: First-Person-Shooter. Leider wurde fast die gesamte Rechenkapazität für die graphische Darstellung verwendet, so das der Inhalt und die Geschichte der Spiel relative simple sind. Das Spiel *Tomb Raider* von 1996 wurde als erstes 3D „Jump & Run“ Spiel bekannt, in dem der Spieler seine Heldin von außen sehen konnte und ist ebenso ein gutes Beispiel für das Versagen einer externen Kameraführung. Bewegte der Spieler die Heldin rückwärts an eine Wand, wurde die Sicht durch den eigenen Körper der Spielfigur verdeckt.

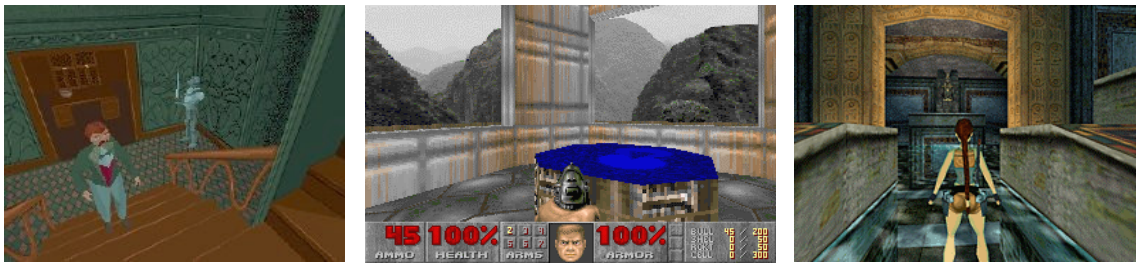


Abbildung 1.3: Alone in the Dark (1992), Doom (1993), Tomb Raider (1996)

Mit steigender Rechenkapazität und zunehmender Verbreitung von 3D-Grafikbeschleunigern für standardisierte Graphikbibliotheken (OpenGL, Direct3D) wurden die Spiele immer detaillierter, inhaltlich komplexer und hatten wieder dramatische Geschichten, Dialoge und intelligentes Verhalten von computergesteuerten Charakteren (z.B. *Half-Life* von 1998). Momentan sind in jedem handelsüblichen PC leistungsfähige 3D-Beschleuniger integriert und der Spielgraphik sind kaum noch Grenzen gesetzt. Die gewonnene Rechenzeit wird in realistische Simulationen der Spielphysik investiert, sowie intelligentes und vielfaches, künstliches Leben in diesen Spielwelten ermöglicht. Spiele die nur auf verbesserte Graphikeffekte setzten, ohne ein innovatives Spieldesign zu haben, werden kaum noch gekauft (z.B. *Unreal2*)

1.6 Online-Spiele

Mit der flächendeckenden Verfügbarkeit schneller Internetzugänge für Privatpersonen in Amerika, Europa und Asien sind Online-Spiele in den letzen 5 Jahren sehr populär geworden. Bei Online-Spielen (oder auch Multiplayer-Spiele) spielen mehrere Spieler in einer gemeinsamen virtuellen Welt mit- oder gegeneinander. Die Verbindung zwischen den Teilnehmern eines Spiels wird durch das Internet hergestellt und läuft in der Regel über einen zentralen Spiel-Server. Spieler können entweder selber neue Spiele starten oder in ständig aktualisierten Listen nach bereits laufenden Spielen suchen und dort einsteigen. Je nach Komplexität des Spiels und verfügbarer Bandbreite reicht die Anzahl der Teilnehmer pro Spiel von einem Dutzend bis in die Hunderte. In den meisten Spielen können die Spieler untereinander „chatten“, teilweise auch direkt miteinander sprechen (*Voice-Over-IP*).

In ihrem Jahresbericht 2002 zählt der amerikanische Verband der Video- und Computerspielhersteller IDSA über 145 Millionen US Bürger, die regelmäßig interaktive Videospiele zur Un-

terhaltung nutzen [IDS02]. Davon nutzen bereits ein Drittel Online-Spiele im Internet. Viele Konsolen-Hersteller statten ihre Spielkonsolen nun mit Modem- oder DSL-Adaptern aus, um nicht den Anschluss an die wachsende Gemeinschaft der Online-Spieler zu verlieren. Im Herbst 2002 startete z.B. Microsoft für seine Spielkonsole *X-Box* einen eigenen Internet-Dienst *X-Box Life*, mit dem die Benutzer über das eingebaute Modem miteinander spielen oder anderen Internet-Dienste nutzen können (WWW, E-Mail).

Das mit Abstand meist gespielte Online-Spiel ist der 3D First-Person-Shooter *Half-Life: Counter-Strike* von Valve Software. Zu jedem beliebigen Zeitpunkt wird es weltweit von zirka 100.000 Nutzern gleichzeitig gespielt. Insgesamt gibt es über 12 Millionen aktiver Spieler, deren monatlich Online-Gesamtspielzeit 3,4 Milliarden *view minutes* beträgt [4]. Während der CPL *Counter-Strike* World Championship 2002 verfolgten laut Veranstalter über 29.000 Zuschauer die Spiele live im Internet. Die Zuschauer nutzen dabei einen speziellen in *Half-Life* integrierten TV Dienst (*HLTV*).

Wenn ein Spiel von vielen Menschen über längeren Zeitraum hinweg gespielt werden, bilden sich soziale Strukturen. Gerade bei Jugendlichen bekommt Online-Spielen einen sportlichen Charakter. Einzelspieler oder Teams suchen den Leistungsvergleich in Wettkämpfen, wie es von herkömmlichen Sportarten bekannt ist. In den letzten Jahren haben sich nationale und internationale Online-Ligen entwickelt, deren Aktivitäten und Mitgliedeszahlen stetig steigen. Ähnlich wie beim Sport werden die finanziellen Mittel hierfür aus Sponsoring und Werbung bezogen. Besonders Grafikkarten- und Prozessorhersteller (z.B. Intel, AMD, nVidia, ATI) haben in der zumeist jüngeren Spielergemeinschaft eine umsatzstarke Zielgruppe für ihre Hochleistungs-Produkte entdeckt.

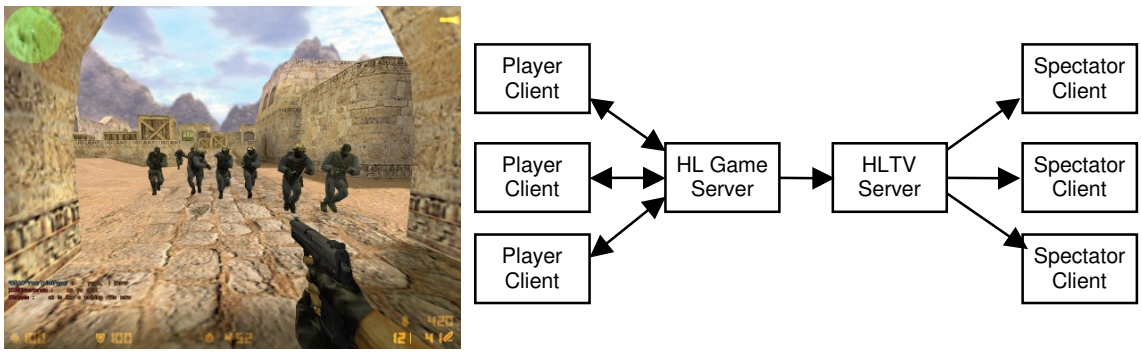


Abbildung 1.4: Counter-Strike (l.), HLTV Netzwerk Architektur (r.)

Durch diesen sportliche Charakter haben Multiplayer-Spiele einen neuen zusätzlichen Unterhaltswert neben dem primären Spielen: das gemeinschaftliche Verfolgen von Wettkämpfen. Mittlerweile können ganze Spiele mit Bild und Ton übertragen werden. Echte Video-Streams benötigen jedoch trotz Kompressionsverfahren wie MPEG4 heute noch zu viel Bandbreite um ein entsprechend großes Publikum (>10.000) zu bedienen. Deshalb findet die Übertragung der Spiele im selben proprietären Netzwerk-Protokoll statt, wie es auch zum Datenaustausch beim normalen Spielen verwendet wird. So kann die benötigte Bandbreite pro Zuschauer auf wenige Kilobyte pro Sekunde reduziert und gleichzeitig eine optimale Grafik- und Tonwiedergabe ermöglicht werden. Nachteilig ist, dass jeder Zuschauer die entsprechende Spiele-Software zum Betrachten benötigt.

[4] Gabe Newell, Game Developers Conference 2002

Die Zuschauer sind für die Spieler selbst nicht sichtbar, können sich jedoch untereinander über eine Chat-Funktion unterhalten. Bei früheren Versionen wurde die Ansicht für alle Zuschauer durch einen menschlichen Kameramann bestimmt, welcher sich im Spiel frei bewegen und beliebige Spieler verfolgen. Bei einem Spiel in dem über 10 Spieler in unterschiedlichen Räumen gleichzeitig agieren, waren die Kameramänner aber überfordert, da sie aus mangelnder Übersicht oft entscheidende Situationen verpassten und es auch keine Möglichkeit zur späteren Wiederholung gab. Problematisch ist, dass bei schnellen interaktiven Echtzeitspielen wichtige Situationen nur eine Dauer von wenigen Sekunden haben und sehr schwer vorauszusehen sind. Zwar half es, dass mehrere Kameramänner gleichzeitig das Spiel verfolgen konnten, trotzdem war das Zuschauererlebnis immer noch nicht befriedigend, da die Kameramänner dazu neigten, zu schnell die Ansicht zu wechseln [GTV].

Neuere Übertragungstechniken (z.B. HLTV) senden das gesamte Spiel, so kann jeder Zuschauer zu jeder Zeit individuell selbst bestimmen, wen oder was er sehen will. Natürlich befindet er sich damit in dem selben Entscheidungsdilemma, wie vorher die Kameramänner. Deshalb wird das Spiel vor der Übertragung um eine kurze Zeit (10s-30s) zwischengespeichert und von einer Regielogik auf interessant Ereignisse untersucht. Wurden wichtige Situationen entdeckt, wird diese Information an die Zuschauer übermittelt, noch bevor die eigentlich Szene gesendet wird. Befindet sich der Zuschauer im sogenannten „Auto-Director-Mode“, spring so die Ansicht rechtzeitig zu den entsprechenden Spielern. Allerdings enthalten diese Hinweise der Spiellogik nur den Zeitpunkt des Ereignis und welche Spieler daran beteiligt sind. Standpunkt, Blickrichtung und Bewegung der Ansicht sind nicht vorgegeben. Die Kamera verfolgt einfach den wichtigsten Spieler aus konstanter Entfernung von hinten, kinematographische Effekte werden kaum eingesetzt.

2 Grundlagen der virtuellen Kameraführung

2.1 Latombe: Motion Planning

Automatische Kameraführung und Bewegungsplanung autonomer Roboter sind vom Wesen sehr ähnlich Probleme. In beiden Fällen muss ein Objekt zielgerichtet durch eine Welt gesteuert werden, ohne dabei mit Hindernissen zu kollidieren. Robotersteuerung ist das komplexere Problem, da Roboter im Gegensatz zur virtuellen Kamera eine räumliche Ausdehnung haben und bei Bewegungen der Massenträgheit unterliegen (wobei eine simulierte Trägheit der Kamera sehr zu einem realistischen Eindruck beim Zuschauer beiträgt). In dem Buch „Robot Motion Planning“ [Lat93] definiert Latombe das theoretische Problem der Bewegungsplanung und beschreibt drei unterschiedlich Lösungsansätze.

Latombe definiert seinen Roboter als einen starren Körper $\boldsymbol{A}$, welcher sich in einem euklidischem Raum $\boldsymbol{W}$ (*workspace*) aus $\mathbb{R}^n$ bewegt, wobei gewöhnlich n = 2 oder 3 ist. Jeder Körper $\boldsymbol{A}$ hat einen Ansatzpunkt T (*reference point*) und kann bezüglich dieses Punktes eine beliebige Ausrichtung Θ (*orientation*) haben. Zusammen beschreiben (T, Θ) eine bestimmte Konfiguration q von $\boldsymbol{A}$ (*configuration*). Der Raum, welcher durch den Körper $\boldsymbol{A}$ in einer bestimmten Konfiguration *q* beschreiben wird, heißt $\boldsymbol{A}$(q). Der Konfigurationsraum $\boldsymbol{C}$ (*configuration space*) bezeichnet alle möglichen Konfigurationen von $\boldsymbol{A}$ im Raum $\boldsymbol{W}$.

Innerhalb der Raumes $\boldsymbol{W}$ gibt es eine beliebige Anzahl i von festen Hindernissen $\boldsymbol{B}_1,...,\boldsymbol{B}_i$ (*obstacles*). Diese Hindernisse reduzieren die Menge der möglichen Konfigurationen von $\boldsymbol{A}$, da sich eine beliebige Konfiguration q und ein Hindernis $\boldsymbol{B}_i$ nicht überschneiden dürfen. Die Menge $\boldsymbol{CB}$ beschreibt den für einen Körper $\boldsymbol{A}$ unerreichbaren Konfigurationsraum, der durch die Hindernisse $\boldsymbol{B}_i$ und die Ausdehnung von $\boldsymbol{A}$ beschränkt wird:

$$\boldsymbol{CB} = \{\ q \in \boldsymbol{C} \mid \boldsymbol{A}(q) \cap \bigcup \boldsymbol{B}_i \neq \varnothing\ \}$$

Der verbleibende Teilraum von $\boldsymbol{C}$, in dem sich $\boldsymbol{A}$ frei bewegen kann wird mit $\boldsymbol{C}_{free}$ bezeichnet:

$$\boldsymbol{C}_{free} = \boldsymbol{C} \setminus \boldsymbol{CB}$$

Eine virtuelle Kamera hat zwar einen Ansatzpunkt und Ausrichtung, aber keine Ausdehnung, der Teilraum $\boldsymbol{A}$(q) ist also nur ein Punkt. In diesem Fall vereinfacht sich die Definition von $\boldsymbol{C}_{free}$, da $\boldsymbol{CB} = \bigcup \boldsymbol{B}_i$.

Ein freier Weg (*trajectory*) zwischen zwei Punkten $\mathbf{p}_{start}$ und $\mathbf{p}_{goal}$ innerhalb der Arbeitsumgebung ist eine stetige Abbildung $\tau:[0,1] \rightarrow \boldsymbol{C}_{free}$, wobei $\tau(0) = \mathbf{p}_{start}$ und $\tau(1) = \mathbf{p}_{goal}$ ist. Aus der realen physikalischen Welt abgeleitet, bezieht sich die Stetigkeit auf die euklidische Entfernung $d : \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}$ zweier Punkte, also $\lim p \rightarrow p_0\, d(p,p_0) = 0$.

Der Raum $\boldsymbol{C}_{free}$ lässt sich in maximal zusammenhängende Komponenten aufteilen, so dass alle Punkte einer Komponente über einen freien Weg verbunden sind. Das generelle Problem der Bewegungsplanung ist das Finden solcher freien Wege zwischen dem Roboter und seinem Ziel, beziehungsweise der Kamera und ihrem gewünschten Standort. Latombe nähert sich diesem Problem über drei unterschiedliche Strategien: Wegegraphen, Zellzerlegung und Potential-Felder, welche im Folgenden beschrieben werden.

2.1.1 Wegegraphen

Vergleichbar mit einer Strassenkarte (*roadmap*), kann man den Freiraum $\boldsymbol{C}_{\text{free}}$ mit einem vorberechnetem Graphen aus möglichen freien Wegen durchziehen und so das Navigationsproblem auf ein Suchen nach einem Weg innerhalb dieses Graphen reduzieren. Da die Start- und Zielpunkte unseres gesuchten Weges wahrscheinlich nicht direkt auf einem dieser vorberechneten Wege liegen, müssen jeweils zwei Punkte in diesem Wegegraphen bestimmt werden, die möglichst nah und hindernisfrei zum Start und Zielpunkt liegen. Die Wegegraphen müssen den Raum so ausfüllen, das dieses Teilproblem immer und relativ einfach zu lösen ist. Für Räume mit polygonförmigen Hindernissen stellt Latombe drei unterschiedlich Wegegraphen vor: Sichtbarkeitsgraphen, Voronoi Diagramme und die Freeway Methode, wobei letztere hier nicht von Interesse ist.

Der Sichtbarkeitsgraph (*visibility graph*) ist ein ungerichteter Graph, wobei jeder Knoten jeweils eine Ecke eines Hindernisses repräsentiert. Knoten sind durch eine Kante verbunden, falls zwischen den entsprechenden Ecken im Raum ein gradlinig, freier Weg (Sichtlinie) existiert oder beide Ecken zu einer gemeinsamen Seite des selben Hindernisses gehören. Ein beliebiger Pfad innerhalb des Graphen ist immer auch ein freier Weg im Raum $\boldsymbol{W}$.

Bei der Suche nach kürzesten Pfaden innerhalb eines gewichteten Graphen, welcher Punkte und Entfernungen in einem euklidischen Raum nachbildet, wird fast immer der Algorithmus A* verwendet [Lat93]. A* ist eine Erweiterung des allgemeinen Algorithmus von Dijkstra zur Suche von kürzesten Pfaden in einem Graphen. Vorraussetzung für dieses Verfahren ist, dass die Knoten jeweils Punkte in einem euklidischen Raum sind und die Kosten einer Kante zwischen zwei Knoten entsprechend deren euklidische Entfernung ist. Der kürzeste Pfad zwischen zwei Knoten ist eine Folge von benachbarten Kanten mit der geringsten Gesamtkostensumme. Entscheidend ist, dass für zwei beliebige Knoten die direkte euklidische Entfernung eine sichere untere Grenze für die Länge des kürzesten Pfades ist. Bei der Suche berechnet sich die Kosten für einen Knoten aus der Länge des bisherigen kürzesten Weges zum Startpunkt (wie Dijkstra) plus dieser minimalen Kosten bis zum Ziel. Der Algorithmus sucht nach neuen Knoten immer vom momentan günstigsten Knoten aus und nähert sich dem Zielknoten damit wesentlich schneller als das herkömmliche Dijkstra Verfahren.

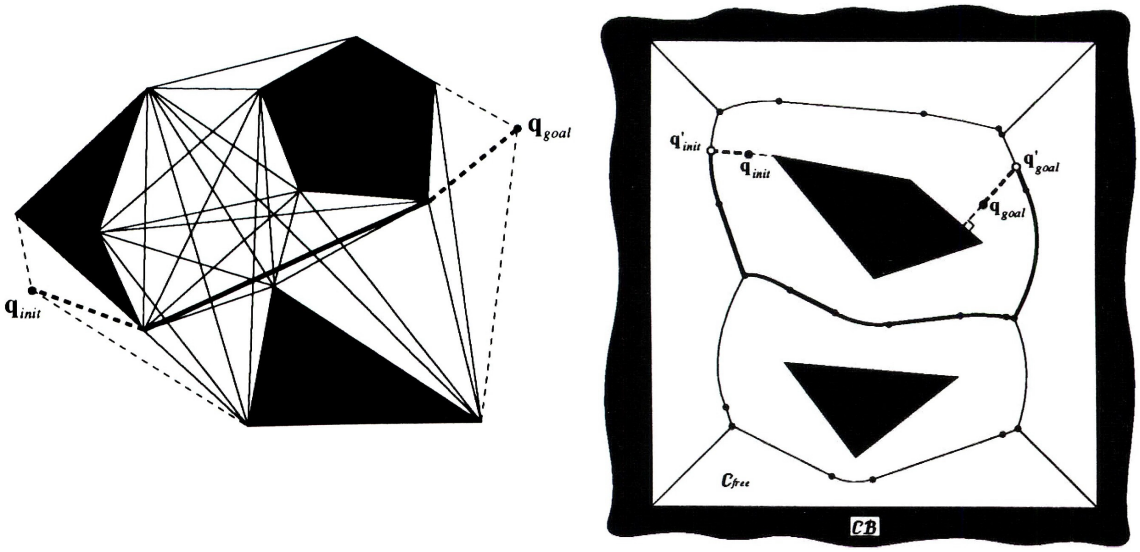


Abbildung 2.1: Sichtbarkeitsgraph (l.), Voronoi Diagramm (r.)

2.1.2 Voronoi Diagramme

Ein Voronoi Diagramm [Lat93] beschreibt eine Untermenge von Punkten in $\boldsymbol{C}_{\text{free}}$, welche alle ein gemeinsame Eigenschaft in Bezug auf die Hindernisse $\boldsymbol{B}_{\text{i}}$ haben. Dafür definiert man die Freiheit (*clearance*) eines Punktes **p** aus $\boldsymbol{C}_{\text{free}}$ über die minimale euklidischen Entfernung:

$$Clearance(\mathbf{p}) = \min_{q \in CB} |\mathbf{p} - \mathbf{q}|$$

Durch diese Freiheit eines Punktes wird auch ein bestimmtes Volumen beschrieben, welches den freien Raum um diesen Punkt erfasst:

$$Volume(\ \mathbf{p}\ ) = \{\ \mathbf{q} \in \boldsymbol{C}_{\text{free}} \mid |\ \mathbf{p}\text{-}\mathbf{q}\ | < Clearance(\ \mathbf{p}\ )\ \}$$

Für einen Punkt **p** kann seine Freiheit durch einen oder mehrere Punkte aus ***CB*** bestimmt werden, welche durch Abbildung *Near* geschrieben werden:

$$Near(\ \mathbf{p}\ ) = \{\ \mathbf{p} \in \boldsymbol{CB} \mid |\ \mathbf{p}\text{-}\mathbf{q}\ | = Clearance(\ \mathbf{p}\ )\ \}$$

Die gemeinsame Eigenschaft aller Punkte des generellen Voronoi Diagramms von $\boldsymbol{C}_{\text{free}}$ ist, dass sie mindesten zwei Hindernispunkte in ihrer Umgebung haben:

$$Vor(\ \boldsymbol{C}_{\text{free}}\ ) = \{\ \mathbf{q} \in \boldsymbol{C}_{\text{free}} \mid |Clearance(\ \mathbf{q}\ )| > 1\ \}$$

Die Punkte des Voronoi Diagramms liegen immer in der Mitte zwischen mindestens zwei Hindernispunkten, ohne dass ein weiterer Hindernispunkt näher wäre.

Voronoi Diagramme bilden ein Netzwerk stetiger Kurven und besitzen eine surjektive Abbildung $\rho : \boldsymbol{C}_{\text{free}} \rightarrow Vor\ (\boldsymbol{C}_{\text{free}})$, welche jedem Punkt **p** im freien Raum eindeutig einem Punkt des Voronoi Diagramms zuweist. Für einen Punkt $\mathbf{p} \in \boldsymbol{C}_{\text{free}}$ existiert eine Untermenge von $Vor\ (\boldsymbol{C}_{\text{free}})$, welche alle Punkte des Diagramms enthält, die **p** in ihrem Volumen enthalten :

$$Vor\ (\ \boldsymbol{C}_{\text{free}}\ ,\ \mathbf{p}\ ) = \{\ \mathbf{q} \in Vor\ (\ \boldsymbol{C}_{\text{free}}\ ) \mid \mathbf{p} \cap Volume(\ \mathbf{q}\ ) \neq \varnothing\ \}$$

Jeweils für genau einen Voronoi Punkt dieser Menge $Vor\ (\ \boldsymbol{C}_{\text{free}}\ ,\ \mathbf{p}\ )$ gilt, dass er mit seinem Volumen einen maximalen Freiraum um Punkt **p** beschreibt. Dies ist dann der Punkt $\rho(\ \mathbf{p}\ )$:

$$\rho(\mathbf{p}) = \max_{q \in Vor(C_{free}, p)} Clearance(\mathbf{q}) - |\mathbf{p} - \mathbf{q}|$$

Diese Abbildung ρ erhält den topologische Zusammenhang des Raumes und die Abbildung eines stetigen freien Weges in $\boldsymbol{C}_{\text{free}}$ ergibt einen stetigen Pfad in $Vor\ (\ \boldsymbol{C}_{\text{free}}\ )$ (Beweis siehe [Lat93]).

Sind die Hindernisse $\boldsymbol{B}_{\text{i}}$ Polygone oder Polyeder lässt sich das Voronoi Diagram einfacher als Voronoi Graph beschreiben, dessen Kanten entweder Graden und parabolische Kurven in $\mathbb{R}^2$ oder Ebenen und parabolische Flächen in $\mathbb{R}^3$ sind. Da ein Graph keine unendlich langen Kanten haben kann, muss $\boldsymbol{C}_{\text{free}}$ komplett von ***C*B** eingeschlossen sein**.** Voronoi Diagramme, welche einen Freiraum beschreiben, der gänzlich von Hindernissen umgeben ist, werden in der Literatur auch als Skelette oder Medial Achsen (Medial Axis Transformation) bezeichnet [Blu67].

Um einen Weg von $\mathbf{p}_{start}$ zu $\mathbf{p}_{goal}$ in $\boldsymbol{C}_{free}$ zu finden, werden die beiden Punkte auf den Voronoi Graphen mittels ρ auf $\mathbf{p}'_{start}$ bzw. $\mathbf{p}'_{goal}$ abgebildet und dann ein Weg im Graphen zwischen diesen beiden Punkten gesucht.

2.1.3 Zellzerlegung

Bei der Zellzerlegung wird $\boldsymbol{C}_{free}$ in eine Menge konvexen Teilräumen aufgeteilt, wodurch eine Navigation in diesen Teilräumen trivial ist. Weiterhin soll es einfach sein, die Nachbarschaft dieser Zellen zu verwalten und freie Wege zwischen benachbarten Zellen zu finden. Dabei kann es kritische Regionen in benachbarten Zellen geben, wo ein direkter Weg zwischen zu Punkten in der anderen Zelle nicht möglich ist.

Bei der Zellzerlegung gibt es zwei Varianten: die exakte und näherungsweise Zerlegung. Bei der exakten Zellzerlegung ist der Vereinigung aller Zellen genau wieder $\boldsymbol{C}_{free}$. Die Zellen selbst sind konvexe Polygone/Polyeder und ihre jeweiligen Nachbarn und Grenzflächen in einem Verbindungsgraphen gespeichert. Falls die Punkte $\mathbf{p}_{start}$ und $\mathbf{p}_{goal}$ nicht in derselben Zelle liegen, muss nach einer (möglichst kurzen) Folge benachbarter Zellen gesucht werden, mit $\mathbf{p}_{start}$ in der ersten Zelle und $\mathbf{p}_{goal}$ in der letzten Zelle. Eine solche Folge von Zellen heißt dann Kanal und durch sie verläuft ein einfach zu konstruierender freier Weg von $\mathbf{p}_{start}$ nach $\mathbf{p}_{goal}$.

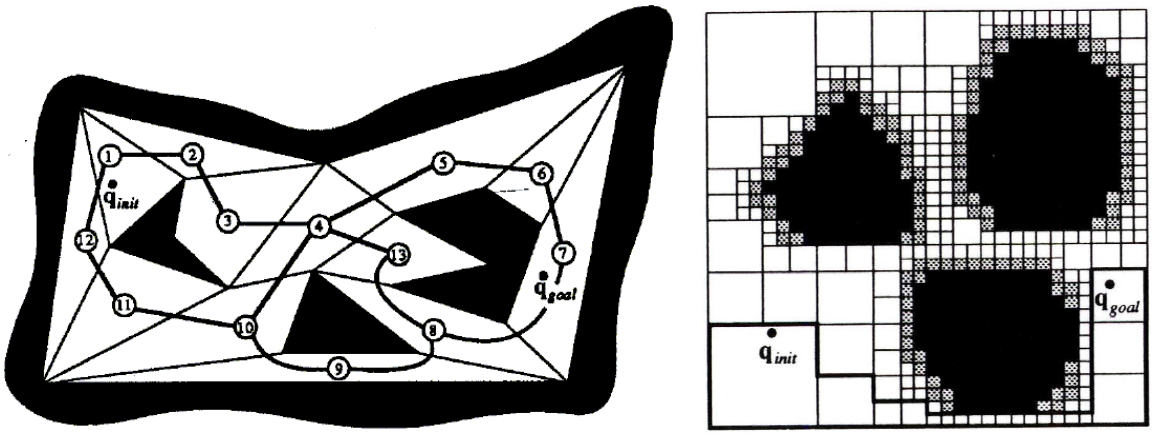


Abbildung 2.2: Zerlegung in konvexe Zellen, diskret (l.) und näherungsweise (r.)

Zellen der näherungsweisen Zerlegung können entweder vollständig frei von Hindernissen aus ***CB*** sein, oder komplett im Hindernisraum ***CB*** oder auch gemischte Zellen sein, welche teilweise Punkte aus ***CB*** und $\boldsymbol{C}_{free}$ enthalten. Die Pfadsuche funktioniert wie bei der exakten Zerlegung, jedoch dürfen nur vollständig freie Zellen nicht benutzt werden, da bei gemischten Zellen nicht sicher ist, ob der Weg wirklich kollisionsfrei ist. Die näherungsweise Zerlegung eignet sich besonders in Räumen mit vielen oder sehr unregelmäßigen Hindernissen, wo eine exakte Zerlegung sehr aufwendig werden würde. Jedoch dürfen die gemischten Zellen nicht zu groß sein, da sonst mögliche Wege verschlossen bleiben. Es bietet sich an, keine feste Zellgröße zu benutzen, sondern die gemischten Zellen solange in regelmäßige frei und gemischten Unterzellen zu unterteilen, bis alle gemischten Zellen eine bestimmte Größe unterschritten haben. Diese maximale Zellengröße muss entsprechend der Anwendung gewählt werden und Die Zellen besitzen dann neben dem Verbindungsgraphen eine Baumstruktur (Quadtree in $\mathbb{R}^2$, Octree in $\mathbb{R}^3$), die bei der Suche nach Zellenfolgen für freie Wege helfen kann. Diese Methode ist sehr universell, einfach zu implementieren und daher sehr beliebt.

2.1.4 Potentialfelder

Potentialfelder haben ihren Ursprung in der physikalischen Feldtheorie. Elektrisch geladene Teilchen haben ein elektrostatisches Feld, welches exponentiell zur Entfernung an Feldstärke abnimmt. Geladene Körper mit gleicher Polung stoßen sich voneinander ab, Körper unterschiedlicher Ladung ziehen sich an. Diese Idee wurde auf die Roboternavigation übertragen, so dass die Hindernisse im Raum von abstoßenden Feldern umgeben sind und der Zielpunkt ein anziehendes Feld besitzt. Jedes Feld definiert für einen Punkt $\mathbf{q}$ im Raum $\boldsymbol{C}_{free}$ eine Feldstärke über eine differenzierbare Funktion $U : \boldsymbol{C}_{free} \rightarrow \mathbb{R}$. Weiter wirkte jedes Feld auf ein Punkt $\mathbf{q}$ ein Kraftvektor $\boldsymbol{F}(\mathbf{q})$ aus. Dieser Kraftvektor ist gleich dem umgekehrten Steigungsvektor der Funktion U in Punkt $\mathbf{q}$, also

$$\boldsymbol{F}(\mathbf{q}) = - \nabla U(\mathbf{q})$$

Die Gesamtkraft, welche auf einen Punkt $\mathbf{q}$ einwirkt, setzt sich aus der Summe aller Kraftvektoren der anziehenden Felder U_{att} (*attractive*) und abstoßenden U_{rep} (*repulsive*) zusammen. Das anziehende Feld U_{att} des Zielpunktes $\mathbf{p}_{goal}$ steigert seine Feldkraft quadratisch zur Entfernung, wobei ε ein Skalierungsfaktor ist:

$$U_{att}(\mathbf{q}) = \varepsilon (|\mathbf{q} - \mathbf{p}_{goal}|)^2$$

Das abstoßende Feld U_{rep} lässt sich für alle Hindernisse $\mathbf{B}_i$ (vereinigt gleich $\boldsymbol{CB}$) in einer (hier vereinfachten) Formel beschreiben, wobei die Feldstärke mit zunehmenden Abstand zum nächsten Hindernis abnimmt (η ist ein linearer Skalierungsfaktor):

$$U_{rep}(\mathbf{q}) = \eta\left(\frac{1}{\rho(\mathbf{q})}\right)^2 \quad \text{mit} \quad \rho(\mathbf{q}) = \min_{q' \in CB} |\mathbf{q} - \mathbf{q}'|$$

Für die effiziente Berechnung der abstoßenden Kräfte an einem Punkt ist es hilfreich nur eine maximale Ausdehnung der einzelnen Felder zu erlauben, da ihr Einfluss sehr schnell vernachlässigbar klein wird.

Die Navigation vom Startpunkt $\mathbf{p}_{start}$ zum Zielpunkt $\mathbf{p}_{goal}$ erfolgt in kleinen Schritten, immer den summierten Kraftvektoren des aktuellen Standorts folgend. Im Gegensatz zu den bisher vorgestellten Algorithmen handelt es sich hierbei um eine lokale Methode. Der Lösungsweg wird nicht vollständig in einem einmaligen Prozess berechnet (globale Lösung), sondern es erfolgt eine iterative Annäherung an das Ziel. Ein einzelner Schritt kann dabei sehr schnell berechnet werden, da nur Hindernisse in der unmittelbaren Umgebung Einfluss auf die Kraftvektoren haben und die Gesamtkomplexität des Raumes nicht betrachtet wird.

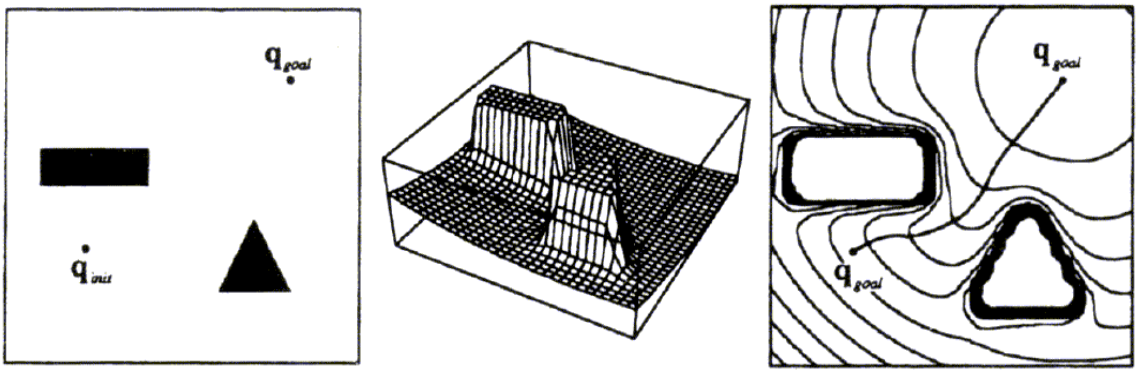


Abbildung 2.3:Potentialfeld für zwei Hindernisse

Der entscheidender Nachteil dieses Verfahrens ist, dass ein möglicher freier Weg oft nicht gefunden werden kann. Sehr wahrscheinlich haben diese Potentialfelder neben dem Punkt $\mathbf{p}_{goal}$ als globales Funktionsminimum noch viel andere lokale Minima. Dort heben sich der anziehende Kraftvektor und der abstoßende Kraftvektor gegenseitig auf und der resultierende Vektor ist Null. Gelangt das Verfahren an einen solches lokales Minima, bewegt sich die Lösung nicht weiter in Richtung Zielpunkt, da alle Schritte eine Potentialerhöhung bedeuten würden. In der Praxis kommt es häufig vor, dass simple Suchverfahren von solchen lokalen Minima „eingefangen" werden und dann endlos um sie herum pendeln.

Um diesem Problem der lokalen Minima zu begegnen, wurden eine Reihe von Lösungen entwickelt. Der Algorithmus kann erkennen, ob er in ein lokales Minimum geraten ist und dann entsprechende „Fluchtmaßnahmen" ergreifen. Eine Möglichkeit besteht darin, das gefundene Minimum mit einem temporären, abstoßenden Feld zu füllen. Oft helfen auch zufällige Bewegungsschritte, deren Länge mit der Verweildauer in einem lokalen Minimum zunehmen (*random motion*). Auch Bewegungen entlang der Tangentialebene des abstoßenden Vektors sind sinnvoll, sich so an der „Wand" des Hindernisses entlang tasten. Oder der bisherige Weg wird zurück gelaufen und andere Wege ausprobiert, wobei bereits bekannten lokalen Minima ausgewichen wird (*back tracking*).

Optimal bei der Erstellung der Potentialfelder wäre, dass überhaupt keine lokalen Minima auftreten können. Man kann z.B. für die Berechnung des anziehenden Potentialfelds von $\mathbf{p}_{goal}$ anstatt der euklidisch Metrik eine andere Metrik wählen, welche bei der Distanz die Hindernisse berücksichtigt, z.B. die Manhatten Metrik L^1. Diese Lösung erfordert meist eine Zerlegung des Raumes in ein diskretes Raster und ist eigentlich eher mit Zellzerlegung verwand als mit Potentialfeldern.

2.2 Drucker : Intelligent Camera Control

In seiner Arbeit „Intelligent Camera Control for Graphical Environments" [Dru94] gibt Steven Drucker einen umfassenden Überblick und Lösungen für unterschiedliche Probleme der Kameraführung in interaktiven Applikationen. Der Zustand einer Kamera wird durch sieben Komponenten beschrieben: Koordinaten im Raum, horizontalen, vertikalen und Neigungswinkel der Blickrichtung (*azimuth, pitch*, *roll*) sowie die Größe des Blickwinkels (*zoom*). In dem ältern CINEMA System [Dru92] wurde das Kameraverhalten in prozeduralen Funktionen fest programmiert und je nach Bedarf abgerufen. Diese reine Reproduktion von bekannten Kameraeinstellungen lieferte sehr unflexible Lösungen, welche nur unter optimalen Bedingungen (freier Raum, keine Bewegung der Akteure) erfolgreich waren.

Ein besser Ansatz ist es, die charakteristische Eigenschaften einer speziellen Kameraeinstellungen genauer zu analysieren und daraus Bedingungen zu formulieren, wie die oben genannten Komponenten des Kamerazustands möglichst gut einzuhalten sind (*soft constraints*). Eine Kameraeinstellung ist dann eine Zielfunktion, welche unter den aktuellen Umständen zu minimieren ist. Die Zielfunktion setzen sich aus unterschiedlichsten Funktionen zusammen, z.B.:

- Entfernung zu Akteuren
- Winkel und Geschwindigkeit relativ zu Akteuren
- Sichtbarkeit von Akteuren
- Differenz zur vorherigen Position

Andere zwingende Eigenschaften (*hard constraints*) können in expliziten Nebenbedingungen formuliert werden, z.B. dass sich die Kamera nur innerhalb eines gewissen Freiraumes bewegen darf. Da diese Zielfunktionen und Nebenbedingungen nichtlinear (meistens auch nicht differenzierbar) sind, ist diese Suche ein nichtlineares Optimierungsproblem und damit generell nicht effizient lösbar. Drucker verwendet zur Minimierung seiner Zielfunktionen eine sequenzielle, quadratische Programmierung, welche von einem möglichst guten Startwert ausgeht. Trotz dieser aufwendigen Suche sind lokale Minima ein Problem und Drucker empfiehlt mehrere Strategien um dieses zu umgehen:

- Zielfunktionen und Eingabemenge so definieren, dass keine lokalen Minima auftreten
- Startwerte aufgrund von Erfahrungswerten und geometrischen Kenntnissen des Raums möglichst nah am Optimum wählen
- Aktive Hilfe durch den Anwender
- Optimumsuche mit möglichst vielen unterschiedlichen Startwerten
- Geeigneten diskreten Lösungsraum wählen und Tiefensuche nutzen

Neben dem Entwurf von grafischen Benutzerschnittstellen geht Drucker noch auf das Problem der kollisionsfreien Wegplanung ein. In seinem Beispiel kann der Benutzer sich eine Liste von Bildern und Objekten in einem virtuellen Museum anschauen. Die Kameralogik plant dann automatisch einen kürzesten Weg an allen gewählten Ausstellungsstücken vorbei, ohne mit einer Wand oder anderen Kunstwerken zu kollidieren. Die Architektur des Museum ist dabei relativ einfach, da alle Räume rechteckig und auf einer Ebene nebeneinander durch Türen verbunden sind.

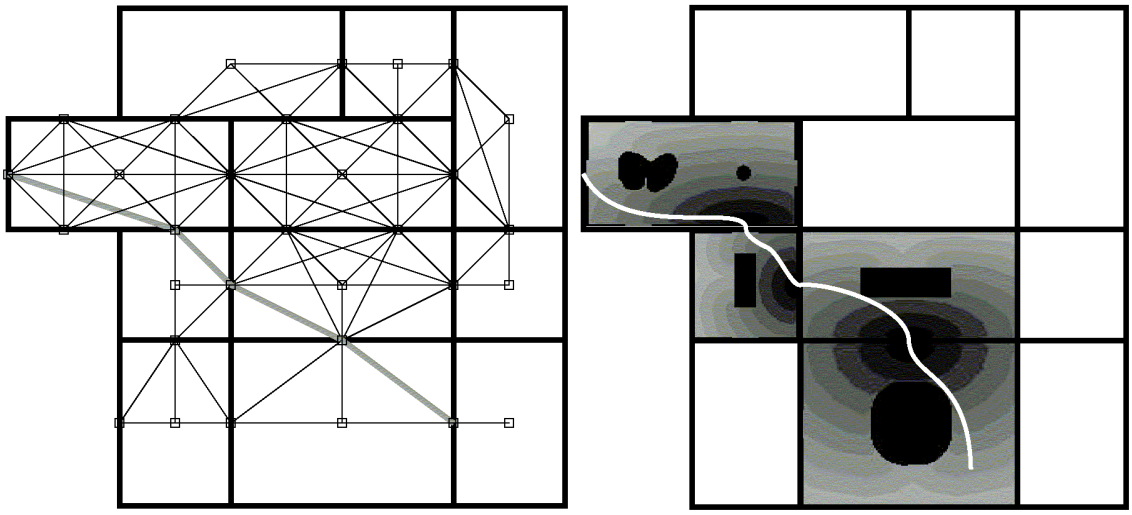


Abbildung 2.4: Globale Navigation über Sichtbarkeitsgraphen (l.), lokale Navigation über Potentialfelder (r.)

Die Navigation erfolgt dabei auf zwei unterschiedlichen Ebenen und nutzt unterschiedliche Techniken. Die globale Wegplanung durch die Räume erfolgt über einen Sichtbarkeitsgraphen. Alle Kunstobjekt und Türen sind Knoten in dem Sichtbarkeitsgraphen, Knoten in einem Raume sind alle untereinander verbunden (starke Zusammenhangskomponenten). Durch den Suchalgorithmus A* wird jeweils der kürzeste Weg über alle gewünschte Knoten ermittelt und so ein Weg von Tür zu Tür durch die Räume beschrieben.

Die Navigation innerhalb eines Raumes muss gesondert behandelt werden, da der Sichtbarkeitsgraph nicht die Ausdehnung der Objekte oder die breite Türen berücksichtigt. Eine geradlinige Bewegung zwischen den Knoten des gefundenen A* Pfades würde mit diesen kollidieren. Als Lösung wird in einer einmaligen Vorverarbeitung für jede Tür in einem Raum, dieser Raum in ein diskretes Raster zerlegt. Jede Zelle dieses Rasters wird speziell markiert, falls sie durch ein Hindernis nicht passierbar ist. Andernfalls enthält sie die Entfernung zur Tür in einer erweiterten Manhatten Metrik, welche Hindernisse bei der Messung der Entfernung zum Zielpunkt berücksichtigt. Diese Metrik wird ausgehend vom Mittelpunkt der jeweiligen Tür für jede Zelle im Raum berechnet (*global numerical navigation function*) [Bar89].

Wenn sich die Kamera während der Laufzeit dann zu einer bestimmten Tür in einem Raum bewegen muss, bewegt sie sich in über dieses Raster, jeweils immer der Zelle mit dem kleinsten Wert folgend. Diese Lösung ist frei von lokalen Minima und sehr schnell, da fast alle Rechenlast in der Vorverarbeitung stattfindet.

Drucker ist es erfolgreich gelungen, erprobte Techniken aus dem Bereich der Roboterforschung auf dem Gebiet der Kameraführung anzuwenden. Trotzdem beschränken sich seine Lösungen auf relativ spezielle Fälle. Die Geometrie des Gesamtraumes beruht auf achsenparallelen Teilräumen, welche durch ausgezeichnete Portale verbunden sind. Der Weg durch den 3-dimensionalen Raum wird nur auf einer 2-dimensionale Grundfläche berechnet. Die Bewegung innerhalb eines Teilraumes funktioniert nur von „Tür zu Tür“, für eine Navigation zwischen zwei beliebigen Punkten in einem Teilraum muss während der Laufzeit ein neues diskretes Navigationsraster mittels Potentialfelder berechnet werden. Diese Lösung ist dann wiederum anfällig für lokale Minima.

2.3 Cohen, Li-wei, Salesin : The Virtual Cinematographer

Eine der am häufigsten zitierten Quelle im Bereich der virtuellen Kameraführung ist der „Virtual Cinematographer“ von M.Cohen, Li-wie He und D. Salesin [Coh96]. Ihr System modelliert kinematographisches Wissen in einer hierarchischen, endlichen Zustandsmaschine. Dieses System befindet sich zwischen der VR Anwendung, in welcher sich die Akteure bewegen und der grafischen Ausgabe für unbeteiligte Zuschauer.

Das kinematographische Wissen wird in Ausdrücken (*idioms*) über die beteiligten Akteure und ihre Tätigkeiten beschreiben, wobei jeder Ausdruck eine bestimmte Charakteristik eines Szenetyps widerspiegelt. Diese Ausdrücke sind in einer Baumstruktur geordnet, wobei die Wurzel ein leerer, immer gültiger Ausdruck ist. Der Baum kann solange nach unten durchlaufen werden, wie die virtuelle Welt die Ausdrücke in den Knoten erfüllt. Mit jedem Schritt wird der Charakter der Szene etwas mehr beschrieben. Um eine möglichst genaue Klassifizierung der Szene zu erhalten, wird nach möglichst langen gültigen Pfaden gesucht, da ein Blatt des Baumes die aktuelle Szene besonders detailliert beschreibt. Sollte im Laufe der Zeit eine der Ausdrücke des aktuellen Pfades durch Veränderungen in der Welt verletzt werden, muss bis zum nächsten vollständig gültigen Knoten aufgestiegen werden, um von dort aus in seinen Nachkommen nach besseren Szenetypen zu suchen.

Jedem Szenetyp, also Knoten in diesem Baum, wird eine Menge von Kameraeinstellungen (*camera modules*) zugewiesen, welche in diesem Fall den kinematographischen Charakter der Szene besonders hervorheben. Die Kameraeinstellungen sind mit Übergangsregeln versehen, um eine Sequenz oder Kombination von Einstellungen je nach Verhalten der beteiligten Akteure zu ermöglichen. Je tiefer der Szenetype im Baum angesiedelt ist, desto differenzierter sind diese

Einstellungen und ihre Übergänge innerhalb dieses Szenetyps. Das System befindet sich also immer in einem Knoten des Wissensbaums, vorzugsweise in einem Blatt. Des weiteren befindet es sich immer in einer aktuellern Kameraeinstellung (Zustand) dieses Szenetypen.

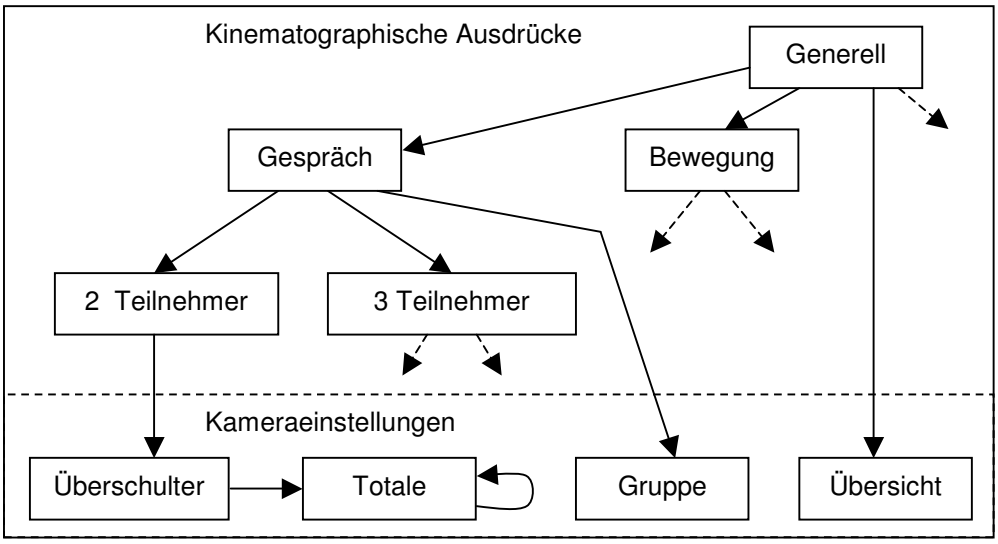


Abbildung 2.5: Strukturierung kinematographischen Wissens

In ihrem praktischen Beispiel lassen sie Gäste in einer virtuellen Party beliebig umherlaufen und miteinander Sprechen. Der „Virtual Cinematographer“ versucht aus dieser Welt möglich zusammenhängende Szene von Gesprächen mit wechselnder Anzahl von Teilnehmern zu erfassen. Die Ausdrücke beschreiben und verfeinern hauptsächlich Gesprächs- und Bewegungsszenen, welche Kombinationen von 16 verschiedenen Kameraeinstellungen nutzen. Dabei werden Bewegungen und Tätigkeit der Akteure in den Ausdrücken („ A spricht mit B“, „A geht zu B“) ausgewertet. Wird kein spezieller Szenetyp erkannt, bleibt das System im Zustand „Generell“ und zeigt den Raum mit allen Akteuren. Wird ein Gespräch erkannt, wird ja nach Anzahl der Teilnehmer weiter unterschieden. Sprechen zwei Gäste miteinander, wird erst eine Überschulter-Einstellung mit beiden Sprechern gezeigt, dann abwechselnd eine Gesichts-Totale des jeweils sprechenden Akteurs. Kommen zu diesem Gespräch weitere Teilnehmer hinzu, ist der Ausdruck des Knotens „2 Teilnehmer“ nicht mehr erfüllt und das System springt zum Szenentyp „Gespräch“ zurück. Sind es dann mehr als 3 Teilnehmer, kann die Szene nicht weiter differenziert werden und eine feste Gruppeneinstellung wird gewählt.

Insgesamt ist der „Virtual Cinematographer“ ein robustes System um kinematographisches Wissen zu strukturieren und für automatische Kameraführung nutzbar zu machen. Jedoch bleiben kritische Probleme ungelöst. Zum einen wird keine Möglichkeit angegeben, welcher Szenentyp bei mehreren Möglichkeiten zu wählen ist, es findet keine Bewertung statt. Das System ist ziemlich unflexibel. Werden die Bedingungen für eine Szene nicht genau erfüllt, bleiben die Kameraeinstellungen zu allgemein. Zudem kennt das System keine Raumobjekte, welche die Sicht auf die Akteure verdecken könnte und somit gewählte Kameraeinstellungen unmöglich machen. Der virtuelle Raum in ihrem Beispiel ist komplett leer und die Kamera kann sich beliebig bewegen. Bei Verdeckung der Sicht durch andere Akteure, wurden diese entgegen ihrer Bewegung künstlich im Raum verschoben. Minimale Distanzen konnten nicht eingehalten werden, so dass Akteure sich selbst verdeckten und schnelle Schnitte scheiterten an den fest programmierten Mindestzeiten in den Übergangsregeln der Kameraeinstellungen. Die Autoren kommen zu dem Schluss, dass die Bedingungen für einen Szenentyp und die Regeln für die

Kameraeinstellungen wesentlich flexibler gestaltet werden müssen. Die Kameraeinstellungen sollten dann während der Laufzeit nach diesen Regeln dynamischen optimiert werden (*incorporating constraint solver*).

2.4 Tsai-Yen Li, Tzong-Hann Yu : Planning an Intelligent Observer

Die Autoren beschreiben in "On-Line Planning For An Intelligent Observer In A Virtual Factory" [LiYu00] eine Methode zur Verfolgung eines bewegten Objektes durch einen Raum mit Hindernissen. Dabei kann zur Laufzeit der Betrachter seine gewünschte Entfernung und Blickwinkel auf das Objekt verändern. Ihre Notation von Raum, Hindernissen und Wegen folgt grundsätzlich Latombe (siehe Kapitel 2.1), jedoch wird eine Konfiguration der Kamera nicht absolut, sondern relativ zum verfolgten Objekt angegeben. Eine Konfiguration ist nur gültig, falls sie innerhalb $\boldsymbol{C}_{\text{free}}$ liegt und das Objekt von ihr aus sichtbar ist.

In ihrem Modell ist der Weg des Objektes über die gesamte Zeit bekannt und es kann einmalig ein sicherer Standardweg für die Kamera berechnet werden. Von diesem Standardweg weicht die Kamera während der Laufzeit entsprechend den Eingaben des Betrachters ab, falls dieser seine gewünschten Winkel oder Entfernung ändert. Konfigurationen können entsprechend verschiedener Benutzerpräferenzen über eine Kostenfunktion bewertet werden (hier vereinfacht) :

$$f(\, d, \alpha, q \,) = w_1 f_1(\, l \,) + w_2 f_2(\, \alpha \,) + w_3 f_3(\, q \,)$$

mit w_1 bis w_3 als lineare Gewichtungsfaktoren:

$f_1(\, d \,) = \lvert\, l - l_0 \,\rvert$	Abweichung von optimaler Entfernung l_0
$f_2(\, \alpha \,) = \lvert\, \alpha - \alpha_0 \,\rvert$	Abweichung von optimalem Blickwinkel α_0
$f_3(\, q \,) = \lvert\, (d, \alpha) - q \,\rvert$	Entfernung zur vorherigen Konfiguration q

Der Benutzer kann über die Faktoren w_1 bis w_3 bestimmen ob eher die Entfernung l_0, der Blickwinkel α_0 oder die Trägheit der Kamera bei der Suche nach optimalen Konfigurationen dominieren soll. Die Benutzerpräferenzen für die Kamera werden durch einen 5-Tupel $P=(l_0, \alpha_0, w_1, w_2, w_3)$ definiert.

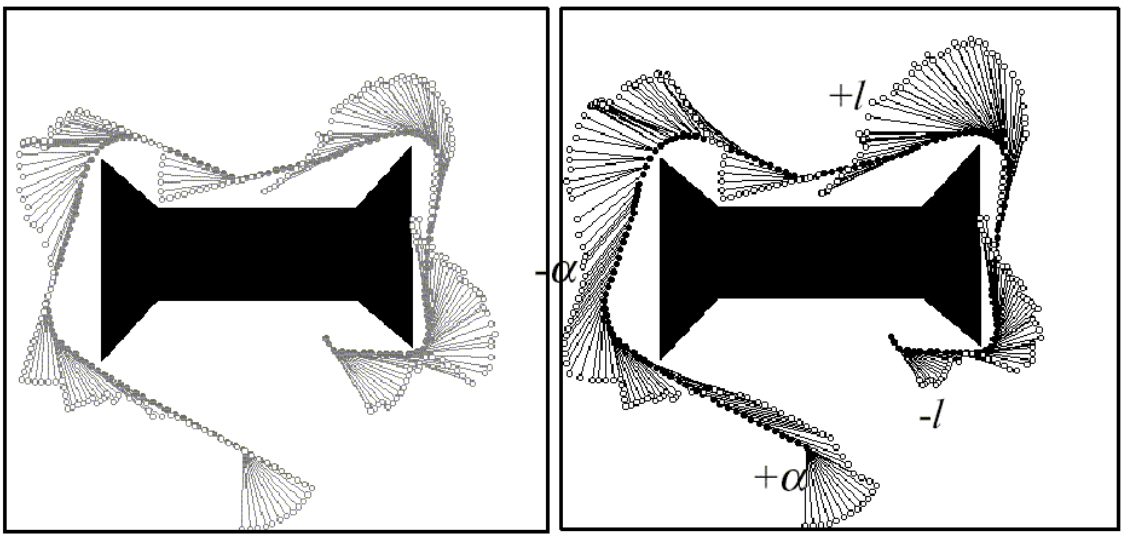


Abbildung 2.6: Verfolgung eines Objektes (l.) und Veränderung der Parameter zur Laufzeit (r.)

Zur Verfolgung des Zielobjektes wird die aktuelle Konfiguration schrittweise geändert, wobei lokal erreichbare Nachbarkonfigurationen bewertet werden und jeweils jene mit den geringsten Kosten als Nachfolger gewählt wird (*Best-First search*). Wie bei Potentialfeldern kann dieses iterative Vorgehen an lokalen Minima scheitern, wobei dann zu einem Punkt auf dem sicheren Standardpfad zurückgegegangen wird (*back tracking*).

Bei der Implementierung ihrer Methode mußten die Autoren testen, ob eine Konfiguration gültig in Bezug auf die Sichtbarkeit des Zielobjektes ist. Dabei haben sie auf die schnellste Möglichkeit der Vorberechnung zurückgegriffen. Der Konfigurationsraum bestehend aus $\mathbb{R}^2$ und den Blickrichtungen [0,360[wurde in eine diskretes Raster zerlegt und für jede Kombination die Sichtweite bis zum nächsten Hindernis berechnet. In ihrem Fall war das ein Raster aus 128∗128 Zellen und Winkelschritten von jeweils 3 Grad, das ergibt 128∗128∗120 ≅ 2 Millionen Konfigurationen im Speicher.

2.5 Zusammenfassung und Bewertung

Betrachten man die bisherigen Ergebnisse in einem gemeinsamen Rahmen, ergibt sich dass, kinematographisches Wissen umfangreich vorhanden ist und gut hierarchisch strukturiert werden kann. Ist die Charakter einer Szene, ihre handelnden Akteure und die Zeitpunkte ihrer Aktion und Reaktion bekannt, kann relativ leicht auf vorgegebene Kameraeinstellungen zurückgegriffen werden. Da die räumliche Umgebung, die Bewegungen und Ausrichtung der Akteure sehr stark variieren können, ist es unvorteilhaft, diese Kameraeinstellungen in festen Sequenzen abzuspielen. Ein bessere und flexiblere Lösung ist es die Kameraeinstellung in Form von Kostenfunktionen über Sichtbarkeit, Entfernung, Winkel und Geschwindigkeit in Bezug auf den dominierenden (handelnden) Akteur zu definieren und dynamisch nach Lösungen zu suchen, welche die Gesamtkosten minimieren.

Grundsätzlich wird zwischen globalen und lokalen Lösungsansätzen unterschieden. Die globale Ansätze sind meist Graphen orientiert und ermöglichen eine Kamerabewegung über lange Wege um so die Distanz zwischen Kamera und Ziel zu verringern. Dies ist sehr nützlich bei sich schnell bewegenden Objekten, welche durch die Kamera verfolgt werden müssen. Allerdings sind die Möglichkeiten der Kamerabewegung sehr eingeschränkt, da sich die Kamera fast nur auf den vordefinierten Wegen bewegen kann. Sichtbarkeitsgraphen liefern zwar die kürzesten Wege, sind jedoch sehr unergonomisch, da sie immer direkt an den Hindernissen entlang laufen und sprunghafte Richtungswechsel enthalten. Voronoi Graphen ergeben bei Kamerabewegungen eine für den Menschen natürlichere Bewegungsform, da sie möglichst viel Abstand zu den Hindernissen halten und in geschwungenen Kurven verlaufen.

Lokale Methoden wie die Potentialfelder ergeben sehr natürlich Bewegungsformen und beschränken sich nicht auf eine (kleine) endlich Menge von Lösungswegen. Ihre Natur von anziehenden und abstoßenden Kräftesummen entspricht sehr der Idee von Kostenfunktionen bei Kameraeinstellungen. Jedoch leiden diese Methoden extrem unter dem Problem der lokalen Minima und eigenen sich nicht zur ausgedehnten Navigation in komplexen Räumen.

Keine der beiden Ansätze kann unser Planungsproblem universell lösen, was auch Drucker erkannt hat. Er benutzt beide Methoden auf unterschiedlichen Ebenen, jedoch erfolgt die Unterteilung von globalen und lokalen Räumen per Hand, indem er abgeschlossene Räume mittels menschlichem Wissens unterteilt und sie durch vorgegebene Portale verbindet. Diese Lösung ist sehr speziell und kann nicht automatisiert werden. Wünschenswert wäre eine Methode, welche die Vorteile der schnellen globalen Navigation mittels Wegegraphen mit den ergonomischen

Wegführung von Potentialfeldern vereint. Dies wird mit den nun beschriebenen Voronoi Distanz Graphen erreicht.

3 Voronoi Distanz Graphen

Bevor die eigentliche Idee der reduzierten Voronoi Graphen entwickelt wird, werden Rahmenbedingungen festgelegt, welche sich durch aktuelle Leistung der Zielsysteme und die Eigenschaften der VR-Anwendungen ergeben. Das neue Verfahren trägt den Namen reduzierte Voronoi Distanz Graphen, kurz VDG.

3.1 Anforderungen zur Laufzeit

Beim Entwurf eines neuen Verfahrens der automatischen Kameraführung müssen verschiedenste Rahmenbedingungen eingehalten werden. Unsere Zielanwendungen sind interaktive VR 3D-Echtzeitsysteme, deren Akteure sich relativ schnell und unvorhersehbar bewegen und handeln. In diesen Systemen konkurrieren die beteiligten Primärmodule wie Eingabeverarbeitung, Simulationslogik, Graphik- und Tonausgabe um vorhandene Ressourcen wie Rechenleistung und Speicherkapazität. Ein zusätzliches Kameramodul darf nur einen kleinen Teil dieser Ressourcen beanspruchen, da eine signifikante Leistungsbeschneidung der oben genannten Primärmodule nicht verhältnisgemäß wäre. Die CPU ist in den letzten Jahren durch günstige und leistungsfähige Graphikprozessoren (GPUs) stark entlastet worden, jedoch beanspruchen aufwendige physikalische Simulationen und die künstliche Intelligenz der vom Computer gesteuerten Chartere (NPCs) beliebig viel Rechenleistung. Die Qualität dieser Funktionen gehören neben der Graphikleistung zu den wichtigsten Vorzügen eines guten Computerspiels. Eine realistische Physik hilft dem Spieler sich in der virtuellen Welt schneller zurecht zu finden, erhöht deren Glaubhaftigkeit und lädt zum ausgedehnten Spielen mit der Welt an sich ein. Intelligentes Verhalten von computergesteuerten Charakteren erhöht die Interaktivität, ermöglicht weniger lineare Handlungsverläufe und steigert das Langzeitvergnügen durch Variationen des Spielablaufes.

Folglich müsste eine automatische Kameraführung mit maximal 5% der verfügbaren Rechenleistung zur Laufzeit auskommen. Während der Laufzeit muss eine relative homogene Rechenverteilung des Algorithmus gewährleistet sein. Auch wenn nur in sehr seltenen Fällen überdurchschnittlich aufwendige Operationen nötig sind, darf es nicht zu spürbaren Verzögerungen kommen. Die menschliche Empfindung, ob eine Darstellung flüssig erscheint, hängt nicht nur von der mittleren Bildfrequenz ab, sondern auch von deren Varianz.

Mit dem Speicherbedarf verhält es sich ähnlich, die benutzte Datenstruktur der Kameraführung sollte nicht größer als die geometrischen Daten der Spielwelt sein, eher kleiner. Die Ladezeiten eventuell vorberechneter Daten dürfen nur wenige Sekunden dauern, da längere Wartezeiten den kontinuierlichen Spielfluss erheblich stören.

3.2 Anforderungen an das Verfahren

Nach Betrachten der Ergebnisse des letzten Kapitel kann man einen relativ genauen Anforderungskatalog für neue Verfahren definieren. Grundsätzlich sollten die Bewegungen der Kamera stetig sein, also frei von Positions- und Richtungssprüngen. Möglichst harmonische Bewegungen sind vorteilhaft. Es sollte ein universelles Verfahren mit einer einzigen Datenstruktur verwendet werden, dessen Berechungen zur Laufzeit oder in einer einmaligen Vorverarbeitungsphase automatisch und ohne menschliche Hilfe ablaufen können. Das Verfahren muss auf beliebige 3-dimensionale Räume anwendbar sein, ohne dass besondere Einschränkungen deren Geometrie erforderlich sind.

Kameraeinstellungen sollten durch Optimierung von Kostenfunktionen über Entfernung, Winkel und Sichtbarkeit des Zielobjektes berechnet werden. Optimale Entfernung und Winkel sind relativ leicht zu finden, jedoch ist die Sichtbarkeit ein zwingendes Kriterium, welches den Lösungsraum einschränkt und die Optimierung wesentlich erschwert. Das Verfahren sollte die Berechnung von Sichtbarkeit zwischen zwei Punkten im Raum unterstützen, besser noch die Größe des maximalen Sichtkegels. Weiterhin sollte für die Optimierung der Kostenfunktion eine Verkleinerung des möglichen Lösungsraum ermöglicht und so die Suche nach „günstigen" Ausgangsposition erleichtert werden.

Wünschenswert ist die Eigenschaft der Potentialfelder bei der Navigation, um einem möglichst gleichmäßigen Abstand zu umliegenden Hindernissen einzuhalten, da dies sehr natürliche Bewegungsformen ergibt. Zum anderen werden relativ schnelle Spieler verfolgt, deren Bewegungen unvorhersehbar sind und abrupte Richtungswechsel auftreten können. Dann ist es für die Kameranavigation sinnvoll in alle Richtung möglichst viel Spielraum zu haben.

3.3 Entwicklung des neuen Verfahrens

Die Grundidee des neuen Verfahrens ist es, einen Voronoi Graphen, wie bereits von Latombe vorgestellt, so zu erweitern, dass auch die Eigenschaften der Potentialfelder genutzt werden können. Das abstoßende Potential eines Punktes im Raum wurde über den quadratischen Kehrwert der Entfernung zum nächstgelegen Hindernis im Raum definiert. Dieses Potential kann auch als eine Kostenfunktion angesehen werden. Als Vorlage dienen die Kostenfunktion von Tsai-Yen Li und Tzong-Hann Yu aus Kapitel 2.5 und die Kostenfunktion für eine Kamerakonfiguration c besteht aus einer Summe von unterschiedlichen Teilfunktionen. Nach Latombe wird die Vereinigung aller Hindernisse als ***CB*** bezeichnet und eine Konfiguration c ist ein Tupel (*Position*×*Angle*×*Velocity*), wobei die Koordinaten von *Position*(c) in $\boldsymbol{C}_{\text{frei}}$ liegen müssen (zwingende Nebenbedingung), *Angle*(c) ist eine Polarkoordinate (θ, ϕ), welche die Bewegungsrichtung angibt. Dabei ist θ der horizontale und ϕ der vertikale Winkel, beide in normierter Radialform $[-\pi,\pi]$. Die *Velocity*(c) $\in \mathbb{R}$ ist die aktuelle Geschwindigkeit. Es wird angenommen, dass die Blickrichtung des Zielobjektes gleich seiner Bewegungsrichtung ist und die Kamera immer direkt auf das Zielobjekt schaut.

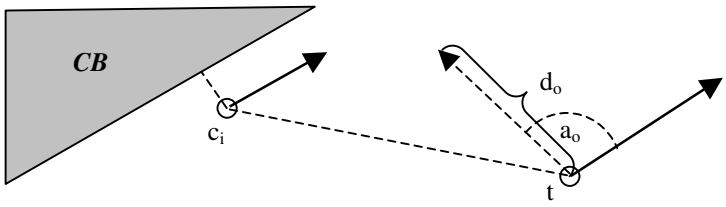


Abbildung 3.1: Grundelemente des Verfahrens

Eine Szene ist eine Folge von stetigen Kamerakonfigurationen, welche einen kontinuierlichen Bewegungsablauf der Kamera ergeben. Die erste Kamerakonfiguration ist c_0, die aktuelle Kameraposition wird mit c_i und deren Nachfolgekonfiguration mit c_{i+1} bezeichnet. Zwischen den Kamerakonfigurationen liegt eine Zeitspanne $\Delta time$, welche der Einfachheit halber als konstant angesehen werden kann (=konstante Bildwiederholfrequenz). Die aktuelle Konfiguration des verfolgten Zielobjektes wird mit t bezeichnet.

Das Kamera Management, welches die Kamerakonfigurationen berechnet, erhält die kinematographischen Vorgaben von einem übergeordneten Regie-System, welches die aktuelle Spielsituation kennt und entsprechende Einstellungen vorgibt. Dieses Regie-System wird hier nicht

näher definiert und entspricht z.B. dem „Virtual Cinematographer“ aus Kapitel 2.3. Der Regie stehen verschiedene Kameraparameter zur Verfügung wie z.B. Abstand und Blickwinkel zum Zielobjekt.

Die optimale Entfernung zwischen Kamera und Objekt ist d_μ und der gewünschte Winkel zwischen Kamera und Bewegungsrichtung des Objekts wird durch a_μ beschriebe. Die Kamera sollte sich träge verhalten um die harmonischen Bewegungen einer menschlichen Kameraführung nachzuahmen. Dafür wird ein Trägheitsfaktor i_σ gegeben, der bestimmt, wie stark eine neue Kamerakonfiguration von einer nur durch Trägheit (*inertia*) bestimmten Nachfolgekonfiguration von c_i abweichen darf. Der gewünschte Abstand der Kamera zu ***CB*** ist abhängig von der Geschwindigkeit und wird durch p_σ bestimmt. Je schneller sich die Kamera bewegt, desto größer sollte die Entfernung zu möglichen Hindernissen sein um mehr Bewegungsfreiheit zu garantieren. Die Sichtbarkeit ist eigentlich eine zwingende Nebenbedingung, jedoch sollten Verdeckungen über eine kurze Zeitspanne v_σ (1 bis 2 Sekunden) erlaubt sein. So wird der Kamera eine Möglichkeit gegeben, das Zielobjekt wiederzufinden, ohne dass ein Sprung nötig ist.
Die genaue Definition und Gewichtung dieser Faktoren und Kostenfunktionen ist an dieser Stelle nicht wichtig und wird in Kapitel 5 beschrieben. Grundsätzlich soll nur festgelegt werden, welche Kostenfaktoren von der gesuchten Datenstrukturen berechnet werden müssen. Es gibt also fünf Kriterien, welche bei der Bestimmung einer Nachfolgekonfiguration c von Bedeutung sind:

$f_{vis}(c)$:	Sichtbarkeit von t von c aus unter Berücksichtigung von ***CB***.
$f_{dist}(c)$:	Differenz zwischen der optimalen Entfernung d_μ und der tatsächlichen Entfernung zwischen c und t.
$f_{angle}(c)$:	Differenz zwischen optimalem Winkel a_μ und des tatsächlichem Blickwinkel von c auf t.
$f_{inertia}(c)$:	Differenz zwischen c und der trägen Nachfolgekonfiguration von c_i.
$f_{clear}(c)$:	Potentialfunktion über kürzeste Entfernung von c nach *CB*.

Die gesamte Kostenfunktion für eine Kamerakonfiguration c berechnet sich aus der Summe der Einzelkosten:

$$f_{total} = f_{vis} + f_{nvis} * (f_{dist} + f_{angle} + f_{inertia} + f_{clear})$$

Der Faktor f_{nvis} liegt zwischen 0 und 1 und ist abhängig von v_σ und der Zeit, wie lange das Zielobjekt t nicht mehr sichtbar gewesen ist. Ist das Objekt länger als v_σ nicht sichtbar, strebt der Wert sehr schnell gegen 0 damit die Sichtbarkeit zum absolut dominierenden Anteil wird.

Die Funktionen f_{dist}, f_{angle} und $f_{inertia}$ sind direkt berechenbar, ohne dass der Raum ***CB*** diese Ergebnisse beeinflusst. Das neue Verfahren muss die Berechnung von f_{vis} und f_{clear} unterstützen und dieses möglichst schnell. Die neue Datenstruktur wird zuerst so entwickelt, dass mit ihr für jeden Punkt f_{clear} berechnet werden kann und dann wird gezeigt, dass mit ihr auch f_{vis} berechenbar ist. Die exakte Definition von f_{clear} lautet:

$$f_{clear}(c) = \left(\frac{Velocity(c) * p_\sigma}{\min_{q \in CB} |Position(c) - q|} \right)^2$$

Der Gewichtungsfaktor p_σ bestimmt den Einfluss der Kamerageschwindigkeit auf die Stärke des abstoßenden Potentials. Problematisch ist die Berechnung des minimalen Abstand zum nächsten Hindernis in ***CB***. Die Berechnung der minimalen Entfernungen (Freiheit) wird jedoch auch bei den Voronoi Graphen durchgeführt.

Für einen Punkt **v** eines Voronoi Diagramms *Vor*($\boldsymbol{C}_{\text{free}}$) gilt, dass die minimale Entfernung d = *Clearance*(**v**) zu ***CB*** durch mindestens 2 Punkte in ***CB*** erfüllt wird. Das bedeutet auch, dass keine Punkte aus ***CB*** innerhalb dieses Radius d um **v** liegen. Für einen beliebigen Punkt **q** aus $\boldsymbol{C}_{\text{free}}$ ergibt sich aus der Abbildung ρ(**q**) ein Punkt **v**, welcher für **q** die maximale Freiheit bestimmt.

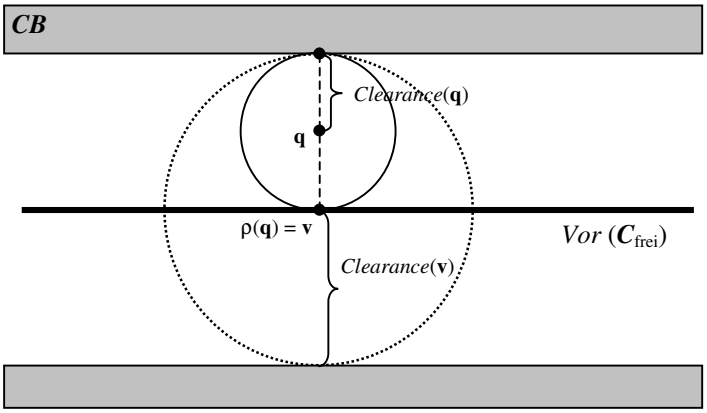


Abbildung 3.2: Berechnung der Freiheit um Punkt q

Ist ein Voronoi Diagram *Vor*($\boldsymbol{C}_{\text{free}}$) vorhanden, kann die Freiheit für einen Punkt $\mathbf{q} \in \boldsymbol{C}_{\text{free}}$ berechnet werden:

$$\textit{Clearance}(\mathbf{q}) = \textit{Clearance}(\mathbf{v}) - |\mathbf{v}-\mathbf{q}| \quad \text{mit } \mathbf{v} = \rho(\mathbf{q})$$

Durch die Reduktion des Problems auf die Bestimmung von ρ(**q**) ist die Komplexität jedoch nicht wesentlich geringer geworden, da ein generalisiertes Voronoi Diagramm immer noch eine unendliche Menge von Punkten haben kann und die Abbildung ρ(**q**) nicht direkt berechenbar ist. Es kann jedoch davon ausgegangen werden, dass der Raum der Hindernisse ***CB*** komplett durch Polyeder beschrieben wird. Dann reduziert sich das Voronoi Diagramm auf einen Voronoi Graphen, welcher aus einer endliche Menge von ebenen oder parabolischen Fläche besteht.

Dieser Schritt reduziert zwar die Komplexität des Voronoi Diagramms, jedoch hat ein Voronoi Graph immer noch sehr viele Flächen, mindestens die Anzahl der Flächen der Polyeder, welche ***CB*** beschreiben. Bei einer einzelnen Berechnung der Freiheit eines Punktes **q** müssten also im ungünstigsten Fall alle Flächen des Graphen untersucht werden. Geht man jedoch davon aus, dass ***CB*** konstant ist und sich unsere Kamera nur stetig bewegen soll, kann die Tatsache genutzt werden, dass die Abbildung eines Weges in $\boldsymbol{C}_{\text{free}}$ mittels ρ wieder einen stetigen Weg in *Vor*($\boldsymbol{C}_{\text{free}}$) ergibt. Sei für den Punkt **q** bereits der Punkt **v** = ρ(**q**) berechnet, so liegt **v** in einer Fläche f des Voronoi Graphen *Vor*($\boldsymbol{C}_{\text{free}}$). Verschiebt sich der Punkt **q** um einen hinreichend kleinen Betrag zu Punkt **q'** = **q**+ε, so liegt Punkt **v'** = ρ(**q'**) aufgrund der Stetigkeit wieder in Fläche f oder einer in *Vor*($\boldsymbol{C}_{\text{free}}$) benachbarten Fläche f'. Durch diese Eigenschaft kann die praktische Berechnung der Freiheit um einen stetig bewegten Punkt wesentlich vereinfacht werden, da die Menge an Nachbarflächen im allgemeinen sehr klein und unabhängig von der Gesamtgröße des Graphen ist.

3.4 Vereinfachung des Voronoi Diagramms

Der Voronoi Graph $Vor(\boldsymbol{C}_{free})$ für einen entsprechenden Raum ***CB*** aus Polyedern in $\mathbb{R}^3$ besteht aus Knoten und unterschiedlichen Arten von Kanten und Flächen. Ein Knoten entsteht an einem Punkt im Raum, dessen Freiheit durch vier oder mehr Punkte im ***CB*** definiert wird. Zwischen diesen Knoten sind Kanten aufgespannt, wobei die Freiheit eines Punktes auf den Kanten durch genau drei Punkte in $Vor(\boldsymbol{C}_{free})$ beschrieben wird. Diese Kanten könne Geraden oder parabolische Kurven sein. Die restlichen Punkte in $Vor(\boldsymbol{C}_{free})$ haben genau 2 Punkte aus ***CB*** in ihrer Umgebung und bilden die Voronoi Flächen, welche wiederum durch die Kanten aufgespannt werden. Entsprechend den Kanten können diese Flächen ebene oder parabolische Form haben. In $\mathbb{R}^2$ gelten die gleichen Regeln, nur dass die Knoten und Kanten jeweils einen Punkt aus ***CB*** weniger in ihrer Umgebung habe und es keine Flächen gibt.

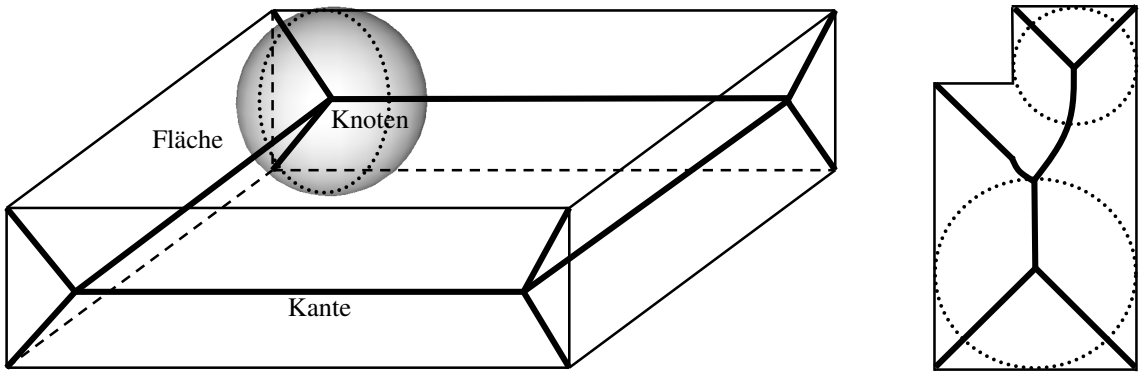


Abbildung 3.3: Knoten, Kanten und Flächen von Voronoi Graphen in $\mathbb{R}^3$ (l.) und $\mathbb{R}^2$ (r.)

Zwei beliebige Punkte **v** und **v'** einer Voronoi Fläche f in $\mathbb{R}^3$ haben gemeinsam, dass die zwei Punkte aus ***CB***, welche ihre Umgebung definieren, jeweils zu den selben Flächen in ***CB*** gehören. Die Punkte aus ***CB*** können entweder in einer Fläche eines Polyeder liegen oder auf einer Kante zwischen 2 Flächen aus ***CB***. Sei $\mathbf{v} \in Vor(\boldsymbol{C}_{free})$ und $\mathbf{p}_1, \mathbf{p}_2 \in \boldsymbol{CB}$ für die gilt:

$$Near(\mathbf{v}) = \{ \mathbf{p}_1, \mathbf{p}_2 \}$$

Dann gilt für die Voronoi Fläche f , in der **v** liegt**:**

- falls $\mathbf{p}_1$ und $\mathbf{p}_2$ beide Ecke in ***CB*** sind, ist f eben
- falls $\mathbf{p}_1$ und $\mathbf{p}_2$ beide auf Kanten in ***CB*** liegen, welche koplanar sind, ist f eben
- falls $\mathbf{p}_1$ und $\mathbf{p}_2$ beide innerhalb von Fläche in ***CB*** liegen, ist f eben
- ansonsten ist f parabolisch

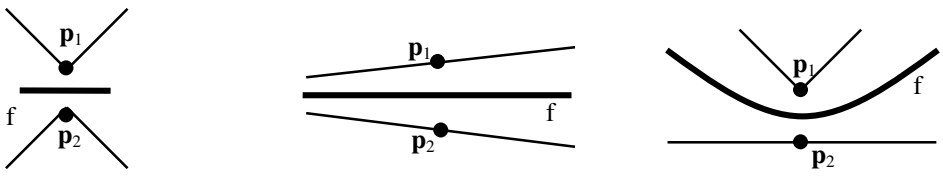


Abbildung 3.4: p_1 Ecke – p_2 Ecke (l.), p_1 Fläche – p_2 Fläche (m.), p_1 Ecke – p_2 Fläche (r.)

Besteht ***CB*** nicht aus Polyedern, sondern nur aus einer Punktemenge, gilt immer der erste Fall und der Voronoi Graph enthält nur Ebenen in $\mathbb{R}^3$. Dieser Spezialfall der generalisierten Voronoi Diagramme ist die ursprüngliche Form der Voronoi Diagramme und wurde so historisch zuerst

von Dirichlet (1850) und Voronoi (1907) beschrieben [Berg00]. Die Definition der allgemeineren „generalisierten Voronoi Diagramme“ erfolgt erst 1991 durch F. Aurenhammer [Aur91]. Folgenden wird mit Voronoi Graphen dieser Spezialfall des generalisiert Voronoi Diagramms gemeint.

Unterschiedliche Verfahren zur Berechnung von generalisierten Voronoi Diagrammen sind bekannt, z.B. durch stetiges hinzufügen neuer Hindernisse oder „divide-and-conquer“ Methoden [Oka92] jedoch sind sie sehr zeitintensiv und teilweise numerisch nicht sehr robust [Hoff99]. Zudem sind die generalisierten Voronoi Diagramme in der Darstellung und der Weiterverarbeitung für unsere Zwecke nicht sehr effizient in Bezug auf die Rechenzeit. Die Schnittberechnung der parabolischen Kanten und Flächen in $\mathbb{R}^3$ sind sehr aufwendig und nicht mit den schnellen, linearen Gleichungssysteme lösbar, wie sie häufig in graphischen Systemen mit polygonaler Geometrie genutzt werden.

Wünschenswert wäre eine Voronoi Graph, welcher nur aus geradlinigen Kanten und dreieckigen Flächen besteht. Ein solcher Graph aus Dreiecken kann den ursprünglichen generalisierten Voronoi Graphen nur näherungsweise wiedergeben, was für unsere Zwecke ausreichend ist.

Einen solchen Voronoi Graphen aus dreieckigen Voronoi Flächen kann auf zwei unterschiedliche Arten für einen Raum $\boldsymbol{C}_{\text{free}}$ erzeugt werden:

a) Berechnung des generalisierten Voronoi Diagramms und anschließend Transformation der parabolischen Flächen in Netze von Dreiecken.
b) Reduktion der Hindernisse in ***CB*** auf eine Punktemenge, Berechnung des Voronoi Graphens, welcher dann nur aus konvexen Ebenen besteht.

Das VDG Verfahren nutzt die zweite Möglichkeit und rastert die Begrenzungsflächen von ***CB*** zu Punktmengen bis zu einer vorgegebenen Auflösung. Diese Entscheidung geht auf die Tatsache zurück, dass für die Berechnung von generalisierten Voronoi Diagrammen in $\mathbb{R}^3$ weder frei verfügbare Werkzeuge noch Quellcode verfügbar ist. Für Voronoi Graphen von Punktmengen aus $\mathbb{R}^n$ steht mit Qhull [Qhull02] jedoch eine ausgezeichnete Quellcode-Bibliothek zu Verfügung, welche sehr schnell und robust ist, sowie exzellent dokumentiert und langjährig, wissenschaftlich erprobt ist.

Für die Berechnung von $Vor(\boldsymbol{C}_{\text{free}})$ kann (und muss) nicht ganz ***CB*** in eine Punktmenge gerastert werden, da ***CB*** ein unendlicher Raum ist und er per Definition den endlichen Raum $\boldsymbol{C}_{\text{free}}$ komplett umschließt. Es brauchen nur die polygonalen Begrenzungsflächen zwischen ***CB*** und $\boldsymbol{C}_{\text{free}}$ gerastert werden.

3.5 Rasterung von *CB*

Durch die Rasterung der Polyedern in ***CB*** durch eine endliche Menge von Punkten entsteht Informationsverlust, dessen Auswirkungen zu berücksichtigen und abzuschätzen ist. Es kann davon ausgegangen werden, dass diese Begrenzungsflächen als Dreiecke vorliegen. Bei der Rasterung einer Fläche f müssen die Rasterpunkte $\mathbf{Q} \in \mathbb{R}^3$ dermaßen verteilt sein, dass maximale Abstände zwischen ihnen einen maximalen Grenzradius m einhalten. Eine Rasterung $\mathbf{Q} = \textit{Samples}(\text{ f, m })$ der Fläche f wird als gültig definiert, falls gilt:

$$\forall\, p \in f : \exists\, q \in \mathbf{Q} : |q-p| \leq m$$

Die drei Eckpunkte eines Dreiecks f sind eine gültige Rasterung für f, falls die Kantenlängen alle kleiner 2*m sind. Für eine algorithmische Rasterung kann man Dreiecke mit größeren Kantenlängen sukzessive entlang ihrer längsten Kante in zwei Teildreiecke solange aufteilen, bis alle Teildreiecke klein genug sind. Die entgültige Rasterung ist dann die Vereinigung aller Eckpunkte. Die Menge der Rasterpunkte alle Flächen wird mit $\boldsymbol{CBR} \subset \boldsymbol{CB}$ bezeichnet, entsprechend vergrößert sich der freie Raum zu $\boldsymbol{CR}_{\text{free}} = \boldsymbol{C} \setminus \boldsymbol{CBR}$.

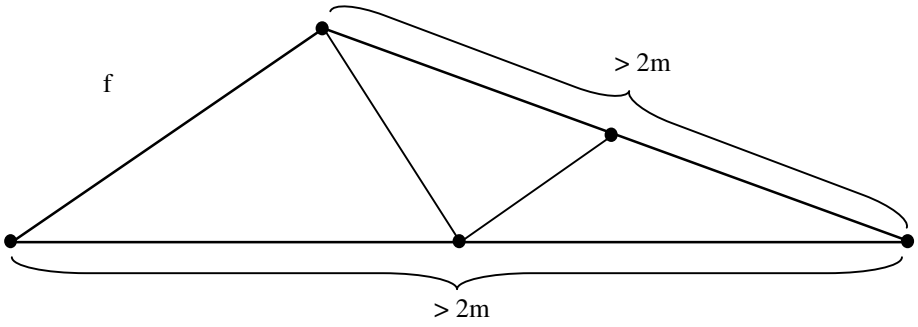


Abbildung 3.5: Unterteilung von Dreiecksflächen an ihrer längsten Kante

3.6 Berechnung des Voronoi Graphen

Nachdem ***CB*** zu einer Punktemenge ***CBR*** in $\mathbb{R}^3$ gerastert wurde, wird der Voronoi Graph aus diesen Punkten berechnet. Die Berechnung wird auf die Konstruktion einer Delaunay Triangulation über eine konvexe Hülle in $\mathbb{R}^4$ zurückgeführt. Zuerst werden die Punkte auf einen Paraboloid in $\mathbb{R}^4$ projiziert mit $w = x^2 + y^2 + z^2$. Für diese Punktmenge wird mit einem modifizierten QuickHull Verfahren die konvexe Hülle berechnet [Qhull02]. Jeder Punkt dieses Paraboloids ist auch Bestandteil der konvexen Hülle, wobei die Kanten der konvexen Hülle wiederum die Kanten der Delaunay Triangulation derselben Punkte in $\mathbb{R}^3$ sind [Berg00].

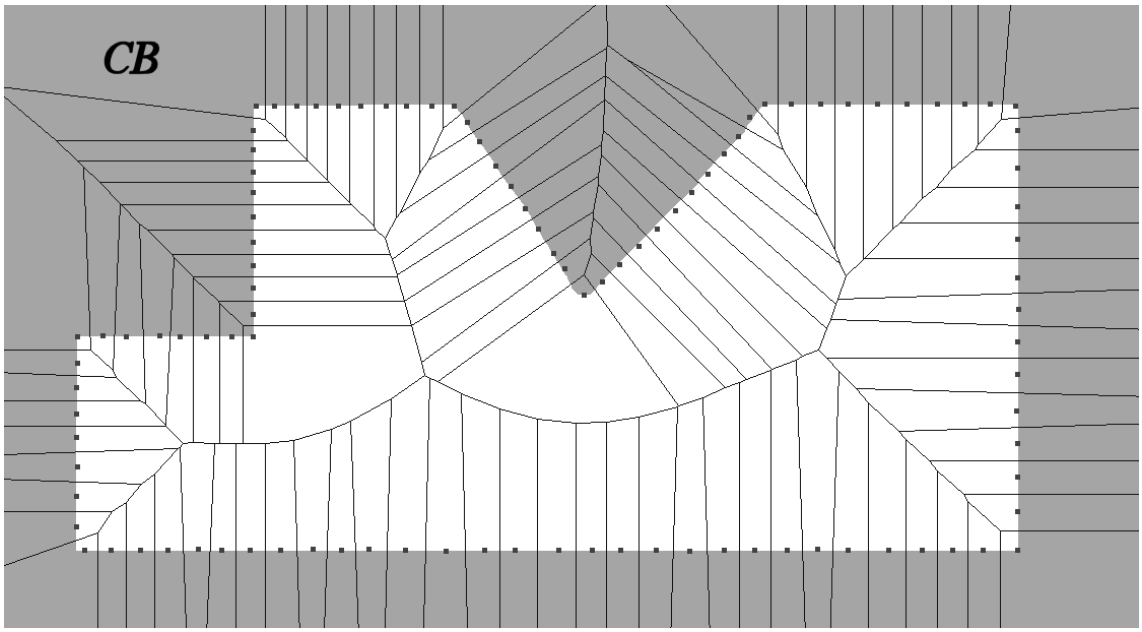


Abbildung 3.6: Kompletter Voronoi Graph aus Ratserpunkte CBR

Aufgrund der Dualität zwischen Voronoi und Delaunay-Graphen entspricht der Mittelpunkt des Umkreises eines Delaunay-Tetraeder einem Knotenpunkt im Voronoi Graphen. Voronoi Knoten von benachbarten Delaunay-Tetraeder sind durch eine Voronoi Kante verbunden. Eine Menge von drei oder mehr benachbarten Voronoi Knoten bilden eine Voronoi Fläche, falls deren entsprechenden Delaunay-Tetraeder eine gemeinsame Kante haben.

Durch die Rasterung wird der Raum $\boldsymbol{CR}_{\text{free}}$ nicht mehr von ***CBR*** eingeschlossen und die Knoten des berechnete Voronoi Graph erstreckt sich auch über den ursprünglichen Raum ***CB***. Des weiteren entstehen unbegrenzte Kanten mit nur einem Knoten, da der Raum $\boldsymbol{CR}_{\text{free}}$ unendliche Ausdrehung hat. Für die Näherung eines generalisierten Voronoi Diagramms, sind diese unbegrenzte Kanten und Voronoi Knoten außerhalb des ursprünglichen Freiraums $\boldsymbol{C}_{\text{free}}$ zu entfernen, sowie alle ihnen inzidente Kanten. Dabei werden auch Kanten entfernt, welche teilweise durch $\boldsymbol{C}_{\text{free}}$ laufen. Diese Kanten schneiden die Begrenzungsfläche zwischen $\boldsymbol{C}_{\text{free}}$ und ***CB*** und ihre Umgebungspunkte aus ***CBR*** stammen alle von der selben Grenzfläche aus ***CB*** ab. Dieser Fall kann auch bei den generalisierten Voronoi Graphen nicht vorkommen, da Punkte einer Ebene keine Kugel eindeutig beschreiben können.

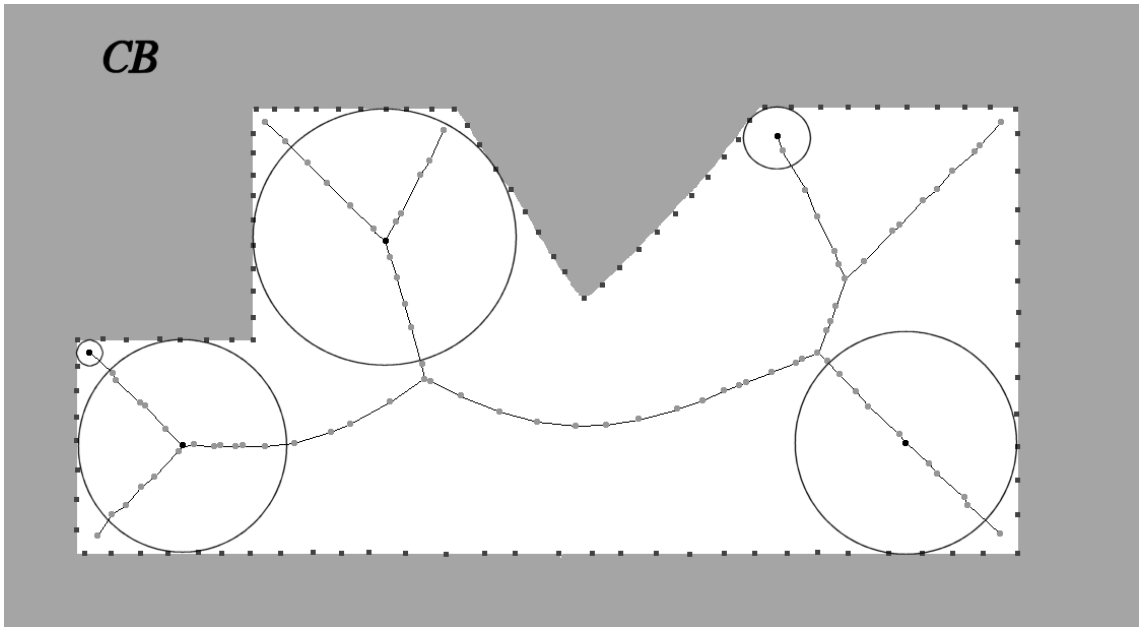


Abbildung 3.7: Reduktion des Voronoi Graphen auf Knoten in C_{free}

3.7 Berechnung der Freiheiten

Die eigentliche Aufgabe des Voronoi Graphen für unseren Zweck ist es, die Freiheit für einen beliebigen Punkt **q** in $\boldsymbol{C}_{\text{free}}$ zu berechnen. Unser Voronoi Graph besteht aus polygonalen Flächen in $\mathbb{R}^3$, wobei ihre Eckpunkte die Knoten des Voronoi Graphen sind. Der Einfachheit halber sind diese Voronoi Flächen trianguliert und die Freiheit eines Voronoi Knoten wurde bei der Berechnung von $Vor(\boldsymbol{C}_{\text{free}})$ am jeweiligen Knoten gespeichert. Zwei Flächen sind adjazent, falls sie mindestens einen gemeinsamen Knoten haben. Diese Datenstruktur wird Voronoi Distanz Graph genannt, kurz VDG.

Sei VDG = (V, F, P, C) mit

- einer endlichen, nichtleeren Menge Knoten V
- einer endlichen, nichtleeren Menge Flächen $F \subset V^3$
- einer Abbildung $P : V \rightarrow \mathbb{R}^3$, welche jedem Knoten eine Position im Raum zuweist
- einer Abbildung $C : V \rightarrow \mathbb{R}$, welche jedem Knoten eine umgebende Freiheit zuordnet

Für den VDG gilt, dass Knoten und Flächen eindeutig sind :

- falls $v_1, v_2 \in V : P(v_1) = P(v_2) \rightarrow C(v_1) = C(v_2) \rightarrow v_1 = v_2$
- falls $f = (v_1, v_2, v_3) \in F \rightarrow v_1 \neq v_2, v_2 \neq v_3, v_3 \neq v_1$

Zwei Knoten v_1,v_2 sind adjazent, falls ein dritter Knoten v_3 existiert, so dass eine Fläche(v_1, v_2, v_3) $\in$ F exitiert. Zwei Flächen sind benachbart, falls sie einen gemeinsamen Knoten haben. Die Mengen *Neighbours*(v) $\subseteq$ V bzw. *Neighbours*(f) $\subseteq$ F beschreiben die Menge der jeweiligen Nachbarn des Elements. Lose Knoten, welche nicht Eckpunkt einer einzigen Fläche sind, darf es nicht geben.

Ein Weg im VDG ist eine Folge von Knoten $w = (v_1,...,v_n)$, wobei v_i und v_{i+1} adjazent sind mit $0 < i < n$. Ein Weg im VDG ist immer auch ein freier Weg in $\boldsymbol{C}_{free}$. Die Länge eines Weges ist die Summe der euklidischen Abstände der Knotenpaare:

$$Length(\ w\ ) = \sum_{i=1}^{n-1} \left| P(v_i) - P(v_{i+1}) \right|$$

Der kürzeste Weg zwischen zwei Knoten v_{start} und v_{goal} ist der Weg mit der geringsten Länge aus der Menge aller Wege, deren erster Knoten v_{start} und der Letzte v_{goal} ist. Die Menge aller Punkte auf einer Geraden zwischen zwei Punkten $\mathbf{p}_1, \mathbf{p}_2 \in \mathbb{R}^3$ wird mit *Line*($\mathbf{p}_1$, $\mathbf{p}_2$) beschrieben:

$$Line(\mathbf{p}_1, \mathbf{p}_2) = \{\ \mathbf{p} \in \mathbb{R}^3 \mid x \in [0,1] : \mathbf{p} = \mathbf{p}_1 + x*(\ \mathbf{p}_2 - \mathbf{p}_1)\ \}$$

Für die Berechnung der Freiheit für einen Punkt $\mathbf{p} \in \mathbb{R}^3$, welcher in einer Fläche f liegt, werden die drei Freiheiten der Knoten dieser Fläche linear interpoliert. Punkt **p** liegt in der Dreiecksfläche $f = (v_1, v_2, v_3)$, falls für $x, y \in \mathbb{R}$ gilt:

$$\mathbf{p} = P(v_1) + x * \overrightarrow{P(v_1)\,P(v_2)} + y * \overrightarrow{P(v_1)\,P(v_3)} \ : x \geq 0,\ y \geq 0,\ (x+y) \leq 1$$

Dann ist seine Freiheit:

$$Clearance(p) = C(v_1) + x * (\ C(v_2) - C(v_1)\ ) + y * (C(v_3) - C(v_1)\ )$$

Für einen Punkt **q** in $\boldsymbol{C}_{free}$, welcher im Freiraumvolumen einer Voronoi Fläche f liegt, kann ein Freiraumradius (nicht unbedingt der Maximale) für diesen Punkt berechnet werden:

$$Clearance(\ \mathbf{q}, f\ ) = Clearance\ (\mathbf{q'}) - |\ \mathbf{q} - \mathbf{q'}\ |\ \text{ mit } q' = \min_{v \in \mathbf{f}} \left\| q - v \right\|$$

Der Punkt **q'**, welcher in Fläche f liegt und **q** am Nächsten ist, lässt sich geometrisch durch das Lot von Punkt **q** auf Fläche f bestimmen. Liegt **q'** auf einer Kante von f, muss das Lot zweimal gefällt werden.

Für die Berechnung der maximalen Freiheit eines Punktes **q** muss die Fläche f ∈ F gefunden werden, welche diese Freiheit maximiert:

$$Clearance(\mathbf{q}) = \max_{f \in F} Clearance(\mathbf{q}, f)$$

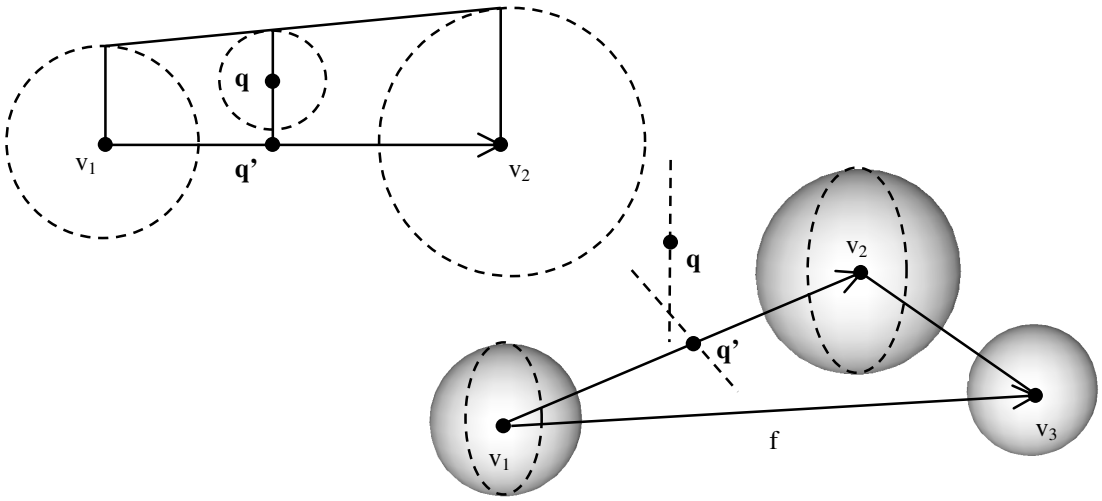


Abbildung 3.8: Berechnung des nächsten Punkt q' in Fläche (v_1 ,v_2, v_3) zu Punkt q

Für die Berechnung der Freiheit eines Punktes **q** müssen also alle Flächen des VDG betrachtet werden. Ist die maximierende Fläche f und der Lotfußpunkt **q'** gefunden, gilt wie bei den generalisierten Voronoi Diagrammen **q'** = ρ(**q**). Wandert **q** stetig in kleinen Schritten ε durch $\boldsymbol{C}_{free}$, läuft auch Lotfußpunkt **q'** stetig über adjazente Flächen des VDGs.

Die Berechnung der Freiheit durch lineare Interpolation ist zwar sehr schnell, jedoch nicht immer korrekt. Das Problem wird hier in $\mathbb{R}^2$ beschrieben, existiert aber analog auch in $\mathbb{R}^3$. Sei eine Kante f aus den zwei Rasterpunkten $\mathbf{r}_1$, $\mathbf{r}_2$ entstanden und v_1 und v_2 die beiden Voronoi Knoten dieser Kante (siehe Abbildung 3.9). Dann liegen in den meisten Fällen die beiden Punkte $\mathbf{r}_1$, $\mathbf{r}_2$ innerhalb des Freiraumvolumens, wie es durch die lineare Interpolation zwischen v_1 und v_2 beschrieben wird. Dies ist natürlich ein Fehler, welcher in der Praxis jedoch sehr gering ist, da die einzelnen Voronoi Knoten im Verhältnis zu ihren Freiheiten sehr nah zusammen liegen. In den meisten Fällen wird dieser Fehler schon durch die Fehlerkorrektur des nächsten Kapitel ausgeglichen. Jedoch kann er in bestimmten Situationen beliebig groß werden, was in der Regel jedoch selten vorkommt.

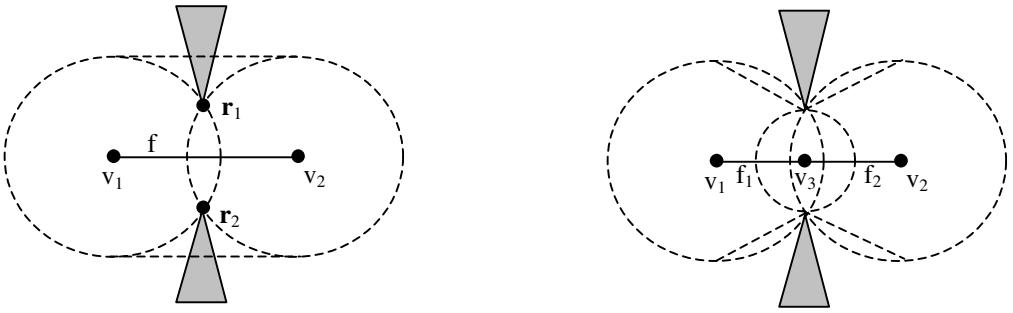


Abbildung 3.9: Fehler der Freiheitsberechnung (l.) und ein passender Lösungsansatz (r.)

Im Folgenden wird dieser Fehler daher nicht weiter beachtet, trotzdem soll an dieser Stelle ein Lösungsvorschlag für zukünftige Implementationen gegeben werden. Die Idee ist es, falls dieser Fehler an einer Kante f zu groß werden würde, einen neuen Hilfsknoten v_3 genau zwischen den beiden Rasterpunkten $\mathbf{r}_1$, $\mathbf{r}_2$ einzufügen. Dieser Hilfsknoten hätte die Freiheit $|\mathbf{r}_1-\mathbf{r}_2|/2$ und die Kante f würde durch die zwei neu Kanten f_1 und f_2 ersetzt werden. Diese Lösung wäre auf $\mathbb{R}^3$ übertragbar und ist relativ einfach umzusetzen.

3.8 Fehlerabschätzung

Ein VDG beschreibt durch die Freiräume um seine Flächen ein gesamtes Freiraumvolumen, welches wie folgt über die Vereinigung der Freiraumvolumen um alle Flächen definiert wird:

$$f \in F : \mathit{Volume}(f) = \{ \mathbf{p} \in \boldsymbol{C} \mid \mathit{Clearance}(\mathbf{p}, f) > 0 \}$$

$$\mathit{Volume}(\mathrm{VDG}) = \bigcup_{f \in F} \mathit{Volume}(f)$$

Für einen generalisierten Voronoi Graphen gilt, dass sein Freiraumvolumen exakt gleich $\boldsymbol{C}_{\mathrm{free}}$ ist. Für eine Näherung eines generalisierten Voronoi Graphen durch einen VDG gilt nur noch, dass ein Teilraum des ursprünglichen Freiraums erfasst wird, also $\mathit{Volume}(\mathrm{VDG}) \subset \boldsymbol{C}_{\mathrm{free}}$. Das entfallene Fehlvolumen ist so gering wie möglich zu halten.

Obwohl bereits alle Knoten des VDGs innerhalb von ***CB*** entfernt wurden, werden Randgebiete von ***CB*** weiterhin von Knoten des VDG als Freiraum beschrieben. Die Freiraumkugel um einen Voronoi Knoten kann durch die Löcher der Rasterung in den Raum ***CB*** hineinreichen. Deshalb müssen die Freiheiten an den Voronoi Knoten um einen Fehlerwert verkleinert werden. Durch die Rasterung wird sichergestellt, dass diese Löcher den maximale Radius m nicht überschreiten, folglich liegen die Rasterpunkte, welche eine Freiraumkugel beschreiben, nicht mehr als 2*m Einheiten auseinander. Falls für $v \in V$ die Freiheit $C(v) > m$ ist, kann der korrigierte Freiraumradius C_{correced} wie folgt berechnet werden:

$$C_{\mathrm{corrected}}(v) = \sqrt{C(v)^2 - m^2}$$

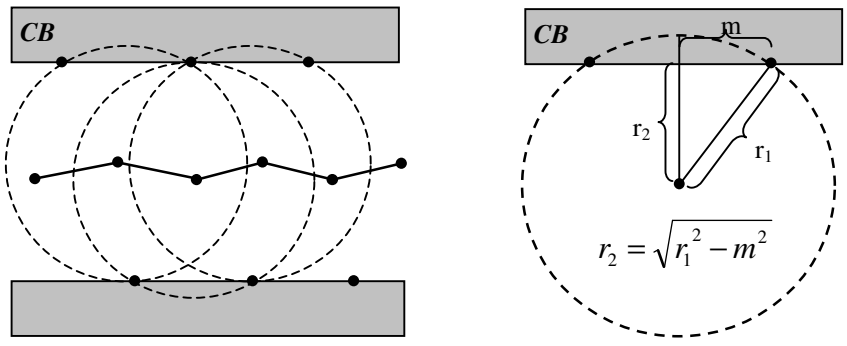


Abbildung 3.10: Fehlerabschätzung der Knoten-Freiheiten

Anderenfalls muss Knoten v aus VDG entfernt werden, da die Bestimmung des Fehlers nicht möglich ist. Diese Methode ist sehr schnell, geht jedoch vom größt möglichen Fehler aus und reduziert gerade kleine Freiheiten teilweise mehr als nötig. Dieser Verlust ist jedoch hinnehm-

bar, da Voronoi Knoten mit kleinen Freiheiten in den Randgebieten eh durch spätere Reduktion entfernt werden.

3.9 Berechnung der Sichtbarkeit

Die Sichtbarkeitskosten f_{vis} (c) vom Kamerastandpunkt c aus sind von entschiedener Bedeutung. Der VDG ermöglicht eine schnelle Überprüfung, ob eine direkte Sichtstrecke von Startpunkt **s** zu Endpunkt **e** frei von Hindernissen ist. Dies ist der Fall, falls gilt

$$\forall \mathbf{p} \in \overrightarrow{se} : \mathbf{p} \in \mathit{Volume}(\mathit{VDG})$$

Da die gerade Strecke **se** stetig ist, existiert für eine freie Sichtstrecke eine Folge benachbarter Flächen f_1 bis f_n:

$$\forall p \in \overrightarrow{se} : \exists f \in \{f_1,..,f_n\} : p \in \mathit{Volume}(f)$$

Diese Folge ist nicht eindeutig, existiert jedoch mindestens eine, ist die Sichtstrecke frei von Hindernissen. Im einfachsten Fall besitzt die Folge nur eine Fläche, in deren Freiraumvolumen sowohl **s** also auch **e** liegt. Für eine algorithmische Berechnung der Sichtbarkeit sei zusätzlich eine Fläche f gegeben, so dass **s** ∈ *Volume*(f) gilt. Liegt **e** nicht in *Volume*(f) wird der Schnittpunkt, an dem die Gerade **se** den Raum *Volume*(f) verlässt, durch die Funktion **s**' = *Trace*(**s**, **e**, f) berechnet.

$$s' = \mathit{Trace}(s,e,f) = \max_{p \in (\mathit{Volume}(f) \cap \overrightarrow{se})} |p - s|$$

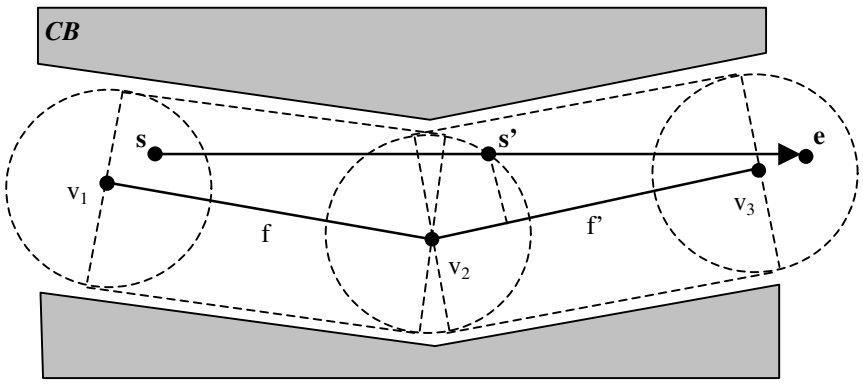


Abbildung 3.11: Verfolgung des Sichtstrahls von Punkt s zu Punkt e

Falls **s'** in einem Freiraumvolumen einer benachbarten Fläche f' liegt wurde damit das Sichtbarkeitsproblem auf die kürzere Strecke **s**'**e** mit Startfläche f' reduziert. Der Algorithmus verläuft entsprechend rekursiv:

$$\mathit{Visible}(\mathbf{s}, \mathbf{e}, f) = \begin{cases} 1, \text{ falls } \mathbf{e} \in \mathit{Volume}(f) \\ 1, \text{ falls } \exists\, f' \in \mathit{Neighbours}(f) : \mathbf{s'} \in \mathit{Volume}(f') \wedge \mathit{Visible}(\mathbf{s'}, \mathbf{e}, f') = 1 \\ 0, \text{ anderenfalls} \end{cases}$$

Die Rekursion ist durch die Bedingung **s'** ∈ *Volume*(f') ist auf einen lokalen Bereich in der Umgebung von **se** beschränkt, so dass sie relativ schnell abgebrochen werden kann. Die Lage des Schnittpunktes **s'** kann geometrisch konstruiert werden, wobei mehrere Fallunterscheidun-

gen auftreten. Eine erweitert Variante des Verfahrens kann über den Freiraum um die Schnittpunkte **s'** Aussagen über die Größe des freien Sichtkegels von **s** nach **e** machen.

Durch die weiter oben beschriebene Verringerung die Freiheiten an den Voronoi Knoten, kann es zu Fehlern bei der Sichtbarkeitsberechnung kommen. Dies geschieht, wenn eine freie Sichtstrecke nahe den Grenzflächen von ***CB*** durch vorher entfernten Freiraum verläuft. Eine mögliche Sichtbarkeit wird dann nicht erkannt. Der Umkehrfall, dass eine nicht freie Sichtstrecke als gültig erkannt wird, tritt nicht auf.

4 Reduktion des Voronoi Graphen

Im letzten Kapitel wurde beschrieben wie der Voroni Distanz Graph aus der Punktrasterung berechnet wird und dass mit ihm alle benötigten Kriterien der Kostenfunktion berechnet werden können. Trotzdem ist die Datenstruktur in dieser Form in der Praxis nicht nutzbar, da Anzahl der Knoten und Flächen für Anwendung in einer zeitkritischen Anwednung zu hoch ist. Deshalb wird über eine Reduktion versucht, möglichst viele Knoten des Graphen zu entfernen, dabei jedoch die Verringerung des beschriebenen Freiraumvolumens möglichst minimal zu halten. Durch die sukzessive Reduktion wird der VDG auf einen Bruchteil seiner ursprünglichen Knotenzahl geschrumpft und ist dann für ressourcenknappe Echtzeitanwendungen einsetzbar.

4.1 Gründe für eine Reduktion

Wie oben erwähnt, sind die Voronoi Graphen enorm groß. Durch die Rasterung ausgedehnter Flächen entstehen sehr viele Rasterpunkte, in der Praxis durchschnittlich zwischen 20 bis 30 Punkte pro Fläche. Ein Voronoi Graph in $\mathbb{R}^d$ hat bei n Eingabepunkte eine maximale Komplexität von $\Theta(n^{d/2})$ [Aur91], dies jedoch nur für sehr speziell konstruierten Eingaben. In der Praxis liegt die Komplexität für Voronoi Diagramme in $\mathbb{R}^3$ bei $O(n)$ [Eri02], so auch für die Voronoi Graphen der Spielkarten (linearer Faktor ≈ 6). Durch die Begrenzung der Voronoi Knoten auf $\boldsymbol{C}_{\text{free}}$ fällt zwar die Hälfte der Knoten wieder weg, jedoch steigt die Anzahl der Rasterpunkte mit kleineren Rasterdichten quadratisch an. In der Praxis darf die Rasterdicht nicht größer 32 Einheiten sein (siehe auch Kapitel 6.1.2) und damit übertreffen die Voronoi Graphen die Komplexität der ursprünglichen Spielkarten um das Hundertfache und sind damit praktisch nicht mehr nutzbar. Durch die beschränkten Speicher- und Rechenkapazitäten während der Laufzeit muss durch die Reduktion wieder eine Komplexität des VDG erreicht werden, die ungefähr der Ausgangskarte entspricht.

Durch den Umweg der Rasterung wurden nicht nur die parabelförmigen Flächen des generalisierten Voronoi Graphen durch kleinere Dreiecksflächen ersetzt, sondern auch die ebenen Flächen. Diese Näherung der Ebenen enthält so wesentlich mehr Dreiecke als nötig ist. Benachbarte Voronoi Flächen, welche auf einer Ebene liegen, können wieder zu größeren Flächen zusammengefasst werden, ohne dass ein Freiraumverlust entstehen würde.

Half-Life Karte	Flächen	Rasterpunkte	Voronoi Knoten
snark_pit.bsp	2788	62831	362483
stalkyard.bsp	2513	64921	341517
rapidcore.bsp	4444	79282	467423
datacore.bsp	4722	88867	522447
frenzy.bsp	2069	89840	472488
lambda_bunker.bsp	4712	109519	631322
subtransit.bsp	5663	154634	893596
undertow.bsp	3175	159077	848123

Abbildung 4.1: Komplexität einiger Karten und deren Voronoi Diagramme

Der VDG beschreibt den gesamten Freiraum $\boldsymbol{C}_{\text{free}}$, von kleinen Reduktionen im Randbereich bedingt durch den Fehler bei der Rasterung abgesehen. Jedoch ist es sinnvoll in den Randgebieten des VDG eventuell noch mehr Freiraum zu entfernen. In einer virtuellen Welt werden meist Objekt oder Akteure gleicher Größenordnung verfolgt. Dies bedeutet, dass die Kamera sich

selbst nur in Teilräumen ähnlicher Größenordnung bewegt. Bei realen Filmaufnahmen ist dies schon durch die physische Ausdehnung einer echten Kamera gegeben. Aufnahmen von Akteuren aus winzigen Teilräumen heraus wirken surreal und sind nur sehr selten erwünscht. Deshalb können Voronoi Knoten entfernt werden, deren Freiheit wesentlich kleiner (<1/10) als die der Größe der Akteure ist. Würde man jedoch Menschen aus der Sicht von Ameisen zeigen wollen, gilt diese Annahme nicht mehr.

Weiterhin wird der VDG bei der späteren Optimierung der Kostenfunktionen nicht nur zur Berechnung von Freiheit und Sichtbarkeit für gegebene Punkte benutzt, sondern dient auch zur Auswahl neuer möglicher Blickpunkte. Die Qualität von nichtlinearen Optimierungsstrategien beruht häufig auf einer guten Wahl von geeigneten Startpunkten. Voronoi Knoten eignen sich in diesem Fall besonders gut, da sie der Mittelpunkt des lokalen Freiraumes sind. Damit haben sie im Vergleich zu anderen Punkten in ihrer Umgebung immer die größtmögliche lokale Freiheit und eine wesentlich höhere Wahrscheinlichkeit, in Sichtverbindung mit einem gewünschten Zielpunkt zu stehen. Da die Optimierung selbst oft zeitaufwendig ist, sollte die Auswahl von Startpunkten möglichst eingeschränkt sein. Daher sollte der VDG eher aus wenigen Voronoi Knoten mit großen Freiheiten und einem geringem Nachbarschaftsgrad bestehen um dieses Verfahren zu beschleunigen.

Zuletzt kann eine Kameralogik die räumliche Lage einer Voronoi Flächen und den Gradienten der drei Freiheiten an den Eckpunkten nutzen, um auf die Symmetrie des umliegenden Raumes zu schließen. Dies funktioniert besonders gut, wenn die Flächen relativ groß im Vergleich zu den Freiheiten der Eckpunkte sind. Liegen die Ecken einer Fläche sehr dicht zusammen und haben große Freiheiten, ist das durch sie beschriebene Freiraumvolumen fast kugelförmig, und man kann keine besondere Aussage über die Symmetrie des umliegende Raumform treffen.

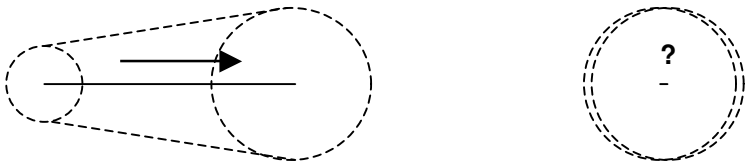


Abbildung 4.2: Abschätzung des umgebenden Raums bei große Flächen (l.) und kleinen Flächen (r.)

4.2 Grundlagen der Reduktion

Ziel der Reduktion ist es, die Anzahl der Voronoi Knoten und damit auch die Anzahl der Flächen zu verringern. Bevor das Verfahren der Reduktion entworfen wird, müssen jedoch grundlegende Bedingungen aufgestellt werden, damit wichtige Eigenschaften des VDG, welche später genutzt werden sollen, durch die Reduktion nicht verletzt werden. Entsteht ein Voronoi Graph durch Entfernen des Knoten k aus VDG, wird dieser mit VDG\k bezeichnet und es müssen folgende drei Bedingungen A, B und C gelten:

A) Der beschriebene Freiraum darf sich nicht vergrößern, also
   $Volume(\ VDG\backslash k\ ) \subseteq Volume(\ VDG\ )$
B) Die Eigenschaft der stetigen Abbildung ρ auf VDG\k muss erhalten bleiben.
C) Zusammenhangskomponenten und Topologie müssen erhalten bleiben.

Grundsätzlich verläuft die Reduktionsstrategie ähnlich dem Verfahren der Reduktion von Netzen, wie z.B. beschrieben von H. Hoppe [Hopp93]. Die Reduktion erfolgt sukzessiv indem jeweils nur ein Knoten nach dem anderen entfernt wird. Für alle Knoten wird ein Kostenbetrag berechnet und jeweils der Knoten mit den geringsten Kosten entfernt. Dies führt zwar nicht zu einem globalen Minimum, ist jedoch sehr schnell und führt in der Regel zu brauchbaren Lösungen.

Die Strategie einen einzelnen Knoten aus einem Graphen zu entfernen besteht darin, durch eine geschickte Reihenfolge von Operationen die umgebenden Kanten so zu verändern, dass der Knoten zum Schluss entfernt werden kann. Die drei elementaren Transformationen (*elementary transformations*) sind: „Kante zusammenziehen“ (*edge collapse*), „Kante teilen“ (*edge split*) und „Kante tauschen“ (*edge swap*), wobei jede Transformation die Topologie des Graphen erhält. Bei der Transformationen „Kante zusammenziehen“ wird die Anzahl der Knoten um eins erniedrigt, bei der Transformation „Kante teilen“ wird ein neuer Knoten hinzugefügt. Die Transformation „Kante tauschen“ verändert die Anzahl der Knoten nicht, änder aber den Kantengrad an den beteiligten Knoten. Es hat sich gezeigt, dass die Transformation „Kanten zusammenziehen“ als einzige Operation ausreicht, um in praktischen Anwendungen ausreichend gute Ergebnisse zu liefern [Hopp93].

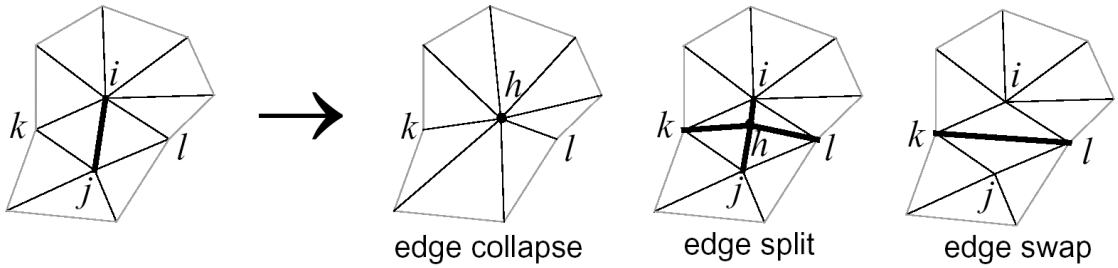


Abbildung 4.3: Die drei elementar Transformationen von Hoppe

Bei der Reduktion der Voronoi Graphen wird nur die Transformation „Kante tauschen“ benutzt um einen Knoten zu entfernen, da in den Voronoi Graphen keinen neuen Knoten eingefügt werden können, welche die Voronoi Bedingungen erfüllen würden. Diese Transformation kann alleine keine Knoten entfernen, jedoch ist es möglich den Nachbarschaftsgrad eines Knoten k bis auf drei zu senken. Dann ist ein trivialer Fall erreicht und ist es möglich, dass der Knoten k entfernt und durch eine neu Fläche zwischen den drei verbleibenden Nachbarn ersetzt werden kann, falls nicht die oben genannte Bedingung A verletzt wird. Diese Transformation erhält ebenso die Topologie des Graphen. Da also keiner der Schritte die Topologie des Graphen verletzt, werden die genannten Bedingungen B und C eingehalten.

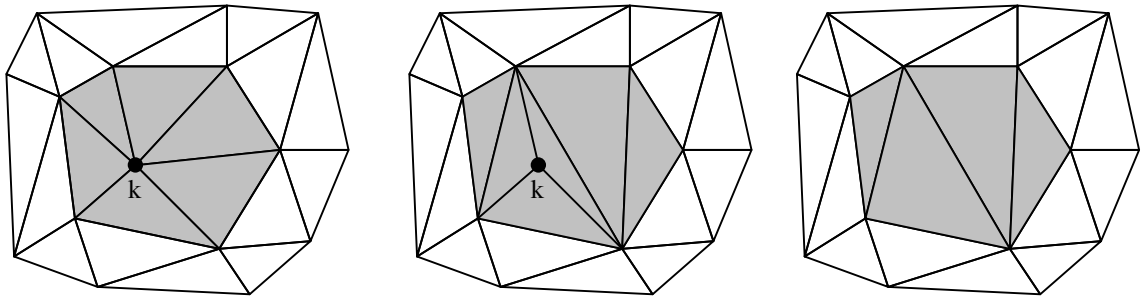


Abbildung 4.4: Reduktion des Graphen (l.) durch Kantentausch um Knoten k (m.) und Entfernen des Knoten (r.)

Falls ein zu entfernender Konten k mehr als drei Kanten hat, wird jeweils die Kante als nächstes getauscht, welche durch den Tausch das geringst Fehlvolumen erzeugt und dabei Bedingung A nicht verletzt. Damit ist das Entfernen eines Knoten eindeutig.

Die Gesamtkosten für das Entfernen eines Knoten k ist die Differenz des Freiraumvolumens zwischen dem ursprünglichen VDG und dem reduzierten VDG\k :

$$\mathit{Costs}(\text{ VDG, k }) = |\mathit{Volume}(\text{ VDG })| - |\mathit{Volume}(\text{ VDG\k })|$$

Da durch das Entfernen eines Knotens sich das Freiraumvolumen nur lokal ändert, braucht in der Praxis nicht jedes Mal die gesamten Volumina der beiden Voronoi Graphen berechnet werden. Es reicht die Differenz der benachbarten Flächen zu berechnen, welche durch einen Reduktion verändert werden könnten. Die Berechnung des Freiraumvolumens einer Voronoi Fläche wird im nächsten Kapitel beschrieben.

4.3 Volumen einer Fläche

Das Volumen des Freiraums, welches durch eine Voronoi Fläche beschrieben wird, wird aus Gründen der Rechenzeit nur nährungsweise berechnet. In der Praxis zeigt sich, dass dieses vereinfachtes Volumenmodel hinreichend genau ist und sich sehr schnell berechnen lässt. Sei die Fläche f beschrieben durch die drei Knoten k_1, k_2 und k_3, welche jeweils eine Freiheit von r_1, r_2 und r_3 haben. Für spätere Definitionen wird eine zusätzliche Notation des Volumens eingeführt, welches durch drei Voronoi Knoten aufgespannt wird:

$$\mathit{Volume}\ (\ (k_1, r_1), (k_2, r_2), (k_3, r_3)\ ) = \mathit{Volume}\ (\ f\ ) \text{ mit } f = (\ k_1, k_2, k_3\ )$$

Das gesamte Freiraumvolumen kann in sieben Komponenten aufgeteilt werden: den Mittelraum senkrecht über und unter der Fläche selbst (halbierter Spat), drei Zylinderhälften an den Kanten zwischen den Knoten und drei Kugelausschnitten an den Eckpunkten.

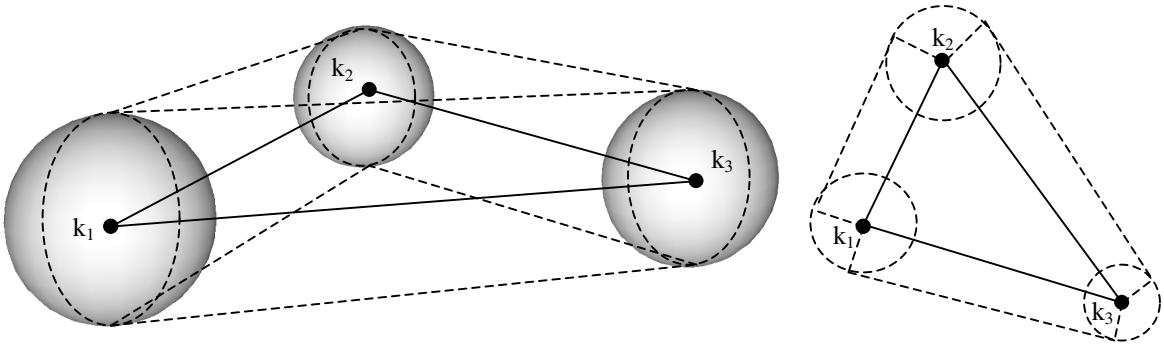


Abbildung 4.5: Das Freiraumvolumen um Fläche (k_1, k_2, k_3)

Der Mittelraum ist das Produkt aus der Grundfläche von f und der gemittelten Freiheit:

$$V_{Mitte} = \left|\overrightarrow{k_1k_2} \times \overrightarrow{k_1k_3}\right| * \frac{(r_1+r_2+r_3)}{3}$$

Das Zylindervolumen zwischen zwei Knoten wird näherungsweise ermittelt. Es ist das Produkt deren Entfernung und der Grundfläche eins Kreises, welcher die gemittelte Freiheit der beiden Knoten als Radius hat:

$$V_{Zylinder} = \left|\overrightarrow{k_1 k_2}\right| * \pi * (\tfrac{r_1+r_2}{2})^2$$

Das Volumen der drei Kugelausschnitte wird an den Ecken durch das Volumen einer einzigen Kugel approximiert unter der Annahme, dass r_1, r_2 und r_3 relativ gleich groß sind:

$$V_{Ecken} = \tfrac{4}{3} * \pi * (\tfrac{r_1+r_2+r_3}{3})^3$$

Das Gesamtvolumen ist die Summe der Teilvolumina. Diese Näherung ist sehr genau, falls die Werte der drei Freiheiten nur wenig voneinander abweichen, was in der Praxis meistens der Fall ist. Vorteilhaft bei diesem Modell ist, dass bei der Berechnung des Gesamtvolumens von zwei benachbarten Flächen, die einzelnen Komponenten, welche jeweils durch beide Flächen beschrieben werden, bei der Rechnung einfach herauszunehmen sind.

4.4 Das Reduktionsverfahren

Das gesamte Reduktionsverfahren besteht aus zwei Phasen. Zuerst werden die Gesamtkosten für jeden Knoten im Graphen ermittelt. Um die Kosten eines Konten zu berechnen wird er aus dem Graphen entfernt, die Volumendifferenz gemessen und anschließend der ursprüngliche Graph wieder hergestellt. Die zweite Phase besteht aus einer Schleife, in deren Durchlauf jeweils ein Knoten aus dem Graphen entfernt wird. Dafür wird zuerst der Knoten mit den geringsten Kosten bestimmt. Für diesen Knoten werden in einer Liste alle Nachbarknoten gespeichert, welche über eine gemeinsame Fläche mit diesem Knoten verfügen. Durch die anschließende Entfernung des Knoten verändert sich auch die Adjazenzlisten der Nachbarn, da Flächen vertauscht oder ersetzt worden sind. Dadurch kann sich auch der Kostenwert der Nachbarknoten ändern, weshalb die Kosten für diese Knoten neu berechnet werden müssen. Diese Schleife terminiert falls entweder eine gewünschte Knotenzahl erreicht wurde oder keine weitere Reduktion mehr möglich ist, da sonst grundlegende Bedingungen verletzt würden. In der Praxis sollte nicht bis zum absoluten Minimum reduziert werden, da gerade die Reduktion der letzen möglichen Knoten teilweise sehr viel Freivolumen entfernt und der VDG dann stark entartet.

Grundsätzlich kann man beim Entfernen von Knoten zwischen zwei Fällen unterscheiden. Entweder der Knoten gehört genau zu einer einzigen Fläche oder zu mehreren. Zuerst wird der einfachste Fall betrachtet, dass nur eine Fläche f an dem zu entfernenden Knoten k_1 hängt und er damit einen Kantengrad 2 hat.

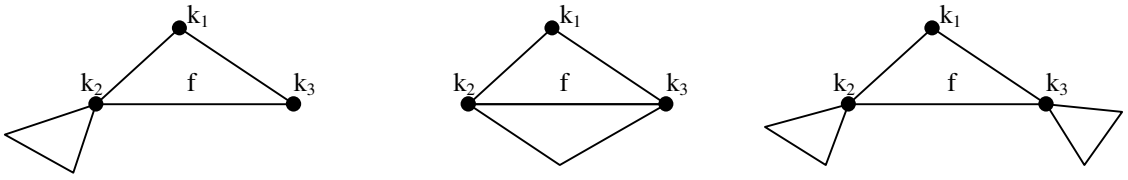


Abbildung 4.6: Knoten k_1 gehört nur zu einer Fläche f, kann aber im letzten Fall (r.) nicht entfernt werden

Haben die beiden anderen Knoten von f auch einen Kantengrad von 2, ist f komplett im Graphen isoliert und die Knoten k_1, k_2, k_3 können entfernt werden, ohne dass ein nutzbarer Freiraum

wegfällt (außer der Graph besteht nur aus dieser einen Fläche). Falls nur für einen der beiden Knoten k_2 oder k_3 gilt, dass er einen Kantengrad von 2 hat (siehe Abbildung 4.6 linker Fall), ist die Fläche über den andern Knoten ($\neq k_1$) mit dem Rest des Graphen verbunden. Dann kann f entfernt werden, ohne dass eine Zusammenhangkomponenten getrennt wird, die Kosten sind dann das Volumen von f. Hängt an beiden Knoten k_2 und k_3 jeweils eine andere Fläche außer f, muss überprüft werden, ob k_2 und k_3 nach dem Entfernen von f immer noch Nachbarn sind. Falls nicht, darf f als Verbindungsfläche nicht entfernt werden.

Im zweiten Fall hat der Knoten k_1 einen Grad größer oder gleich drei und gehört zu mehreren Flächen. Ist der Grad größer drei, muss über iterativen Kantentausch die Gradzahl des Knoten auf drei reduziert werden. Falls der Grad gleich drei ist, kann versucht werden, den Knoten zu entfernen und die drei Nachbarnknoten durch eine neue Fläche zu verbinden. Dabei können zwei Unterfälle auftreten. Entweder an Knoten k_1 liegen zwei Flächen oder drei Flächen, beide Fälle müssen gesondert behandelt werden. Falls zwei Flächen anliegen, kann eine weitere Gradsenkung versucht werden, dann hat der Knoten den Kantengrad zwei und es gelten die bereits bekannten Regeln aus dem letzten Absatz.

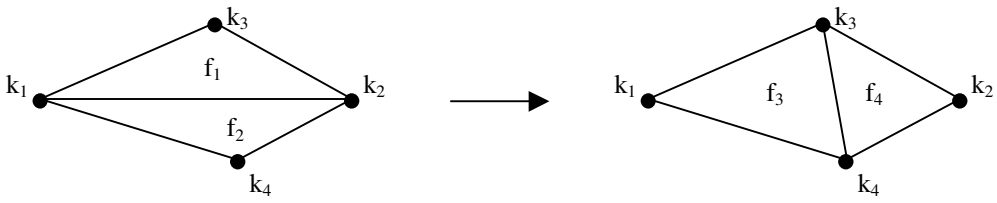


Abbildung 4.7: Kantentausch führt zu Gradsenkung an k_1

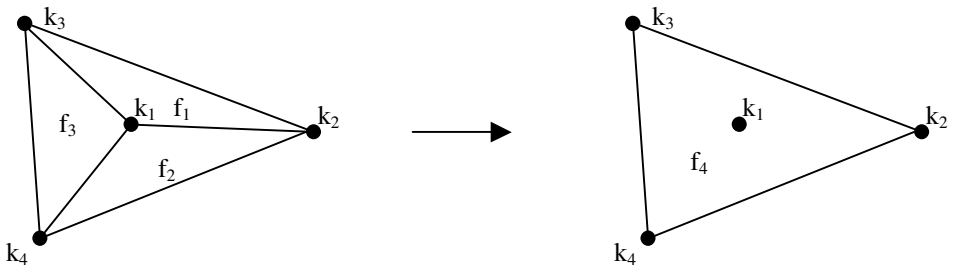


Abbildung 4.8:Entfernung des Knoten k_1 mit Kantengrad 3 und 3 Flächen

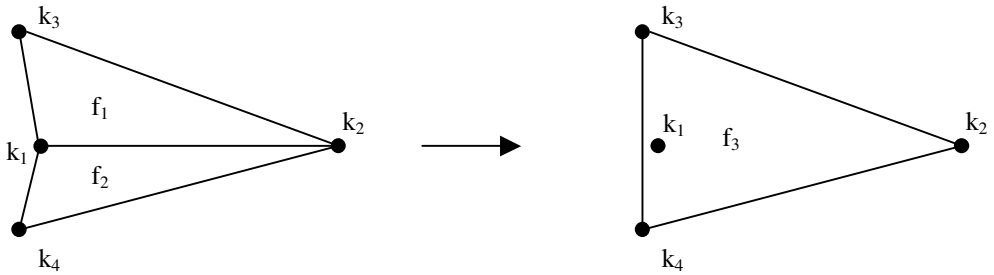


Abbildung 4.9: Entfernung des Knoten k_1 mit Kantengrad 3 und 2 Flächen

```
ReduziereVDG( VDG, Grenze )
{
    FürAlleKnotenVon( VDG, k )
    {
        k.Fehler = KnotenFehler( VDG, k );
    }

    while ( KnotenAnzahl( VDG ) > Grenze )
    {
        k = KnotenMitKleinstenFehler();

        MerkeNachbarn( k, Nachbarn );

        EntferneKnoten( VDG, k );

        FürAlleKnotenIn( Nachbarn, n )
        {
            n.Fehler = Knotenfehler( VDG, n );
        }
    }
}

KnotenFehler( VDG, k )
{
    MerkeZustand( VDG );  // speichere VDG in Kopie
    AltesVolume = BerechneVolumen( VDG );
    EntferneKnoten( VDG, k );
    Fehler = AltesVolume - BerechneVolumen( VDG );
    ZustandWiederherstellen( VDG ); // lade alten Zustand aus Kopie
    return Fehler;
}

EntferneKnoten( VDG, k )
{
    while ( KantenGrad( k ) > 3 )
    {
        KleinsterFehler = MAX_FLOAT;
        BestesPaar = (-1,-1);

        FürAlleFlächenVon( k, f1, f2 )
        {
            Merke( VDG );        // speichere momentanen VDG
            Fehler = TauscheKanteZwischen( f1, f2 );
            Zurücksetzen( VDG ); // stelle alten VDG wieder her

            if( (Fehler >= 0) && (Fehler < KleinsterFehler) )
            {
                KleinsterFehler = Fehler;
                BestesPaar = (f1,f2); // merke bestes Flächenpaar
            }
        }

        if ( BestesPaar == (-1,-1) )
        {
            return -1;    // keine weitere Reduktion möglich
        } else {
            TauscheKanteZwischen (f1,f2);
        }
    }

    ErsetzeFlächenUmKnoten( k ); // trivialer Fall, ersetze Flächen
}
```

4.5 Kanten tauschen

Um einen Knoten k_1 mit einem Kantengrad größer 3 zu entfernen, werden nach benachbarten Flächenpaarem gesucht, die bei einem Tausch ihrer gemeinsamen Kante zu einer Gradsenkung an Knoten k_1 führen. Wenn der Knoten k_1 n Flächen $f_1,...,f_n$ hat, dann muss für jedes Flächenpaar (f_i, f_j) geprüft werden ob ein Kantentausch möglich ist. Grundsätzlich gibt es n über 2 Flächenpaare an Knoten k_1:

$$\text{Anzahl Flächenpaare } (f_i, f_j) = \frac{n*(n-1)}{2} = \binom{n}{2}$$

Die erste Vorraussetzung für einen Kantentausch ist, dass ein Flächenpaar (f_i, f_j) eine gemeinsame Kante haben muss, welche von k_1 zu einem beliebigen Konten k_2 führt. Diese Kante (k_1, k_2) darf nicht Bestandteil einer andern Fläche außer f_i und f_j sein, da sonst ein Entfernen dieser Kante die topologische Konsistenz verletzen würde. Sind diese Bedingungen erfüllt, werden die zwei anderen Knoten mit k_3 und k_4 bezeichnet, so dass f_i=(k_1, k_2, k_3) und f_j=(k_1, k_2, k_4) gilt. Ziel des Kantentausch ist es, die Kante (k_1,k_2) zu entfernen und durch eine neue Kante (k_3,k_2) zu ersetzt, so dass die Freiraumvolumen, welche von f_i und f_j beschrieben werden, durch die neu entstanden Flächen möglichst ohne Verluste ersetzt werden.

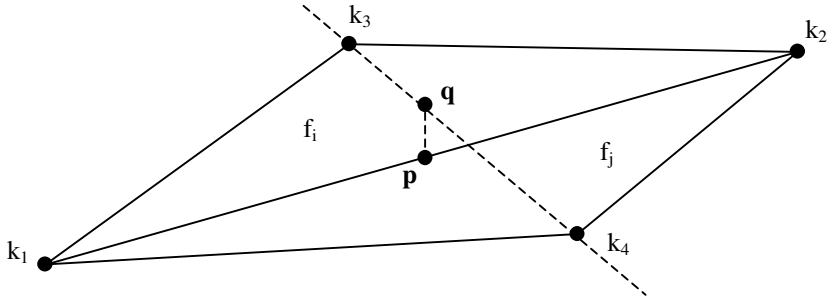


Abbildung 4.10: Berechnung der Lotfußpunkte p und q

Bei dem Kantentausch muss sichergestellt werden, dass kein zusätzliches Freitraumvolumen beschrieben wird. Um dies zu überprüfen, wird das Lot von Kante(k_1, k_2) auf Kante (k_3, k_4) gefällt und die beiden Fußpunkte **p** und **q** berechnet. Für **p** und **q** gilt :

$$\overrightarrow{\mathbf{pq}} \times \vec{n} = \vec{0} \text{ mit } \vec{n} = \overrightarrow{k_1 k_2} \times \overrightarrow{k_3 k_4}$$

Mit den Faktoren t_p und t_q werden die Position von **p** und **q** auf den Geraden beschrieben, welche die beiden Kanten definieren:

$$\mathbf{p} = P(k_1) + t_p * \overrightarrow{P(k_1)P(k_2)} \qquad \mathbf{q} = P(k_3) + t_q * \overrightarrow{P(k_3)P(k_4)}$$

Grundsätzlich muss für einen Kantentausch gelten, dass t_p und t_q innerhalb von [0,1] liegt. Dies ist wichtig für die Abschätzung der Freiheiten um den Punkte **p** und **q**. Liegen die Werte außerhalb dieses Bereiches, können die Freiheiten nicht über lineare Interpolation bestimmt werden.

Entfernt man die Kante (k_1, k_2) und fügt die Kante (k_3, k_4) hinzu, werde die beide Flächen f_i und f_j aus dem Graphen entfernt und zwei neue Flächen f_k =(k_1, k_3, k_4) und f_1 = (k_2, k_3, k_4) entstehen. Entsprechend der Grundbedingung A ist dieser Tausch nur möglich, falls gilt:

$$\{\ Volume(\ f_k\ ) \cap Volume(\ f_l\ )\ \} \subseteq \{\ Volume(\ f_i\ ) \cap Volume(\ f_j\ )\ \}$$

Die Freiheiten an den vier beteiligten Knoten sind jeweils $C(k_1)$, $C(k_2)$, $C(k_3)$ und $C(k_4)$. Die Kanten (k_1, k_3), (k_1, k_4), (k_2, k_3) und (k_2, k_4) bleiben unverändert, folglich muss nur für die neue Kante (k_3, k_4) gezeigt werden, dass sie komplett durch Freiraum von f_i oder f_j führt. An den beiden Eckpunkten k_3 und k_4 liegt diese Kante auf jeden Fall im Freiraum. Durch die Kante (k_1, k_2) und den entsprechenden Freiheiten $C(k_1)$ und $C(k_2)$ wird ein Freiraumzylinder um diese Kante beschrieben.

Die Idee besteht darin, zu zeigen, dass die neue Kante (k_3, k_4) durch diesen Zylinder verläuft, dann liegen auch die Flächen f_k und f_l im beschriebenen Freiraum von f_i und f_j. Um diesen Schnitt mit dem Freiraumzylinder zu berechnen, werden die beiden Punkte bestimmt, an denen sich der Abstand der beiden Kanten (k_1, k_2) und (k_3, k_4) minimiert. Dies geschieht genau an den Punkten **p** und **q**, wie sie bereits oben mit dem Lot zwischen den Kanten bestimmt wurden. Es wird die Distanz d zwischen **p** und **q** sowie die Freiheit c_p um Punkt **p** durch Interpolation bestimmt:

$$c_p = C(k_1) + t_p * (C(k_2) - C(k_1)) \qquad d = \left|\overrightarrow{\mathbf{pq}}\right|$$

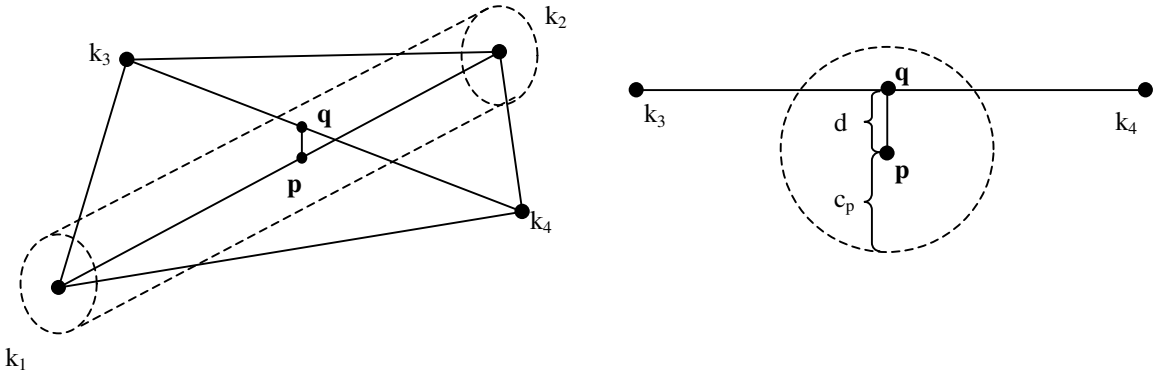


Abbildung 4.11: Kantentausch Fall $t_p \in]0,1[$

Ist $d < c_p$ schneidet die Kante (k_3, k_4) den Freiraumzylinder der Kante (k_1, k_2). Damit ist das letzte Kriterium für den Kantentausch erfüllt und die Kanten können ausgetauscht werden. Jedoch muss eventuell eine noch Korrektur der Freiheiten an k_3 und k_4 vorgenommen werden, um sicherzustellen, dass kein zusätzlicher Freiraum erzeugt wird.

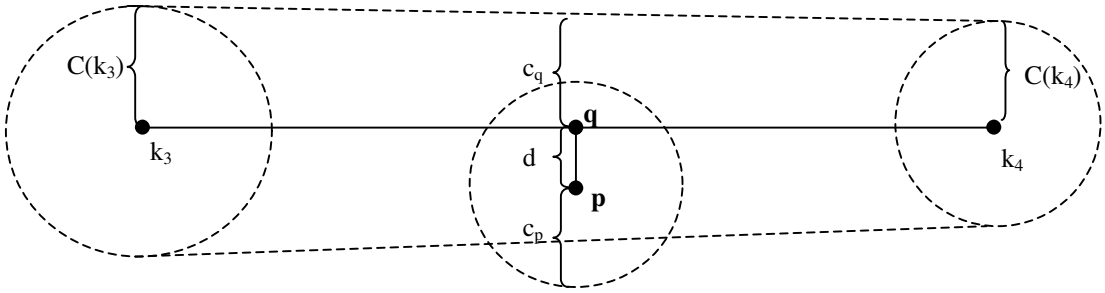


Abbildung 4.12: Freiheiten um Knoten beim Kantentausch

Durch die neue Kante (k_3, k_4) wird auch ein neuer Freiraumzylinder um diese Kante beschrieben, für den wiederum auch gelten muss, dass er komplett durch den alten Freiraumzylinder von Kante (k_1, k_2) verlaufen muss. Ist dies nicht der Fall, müssen die Freiheiten an k_3 und k_4 so verringert werden, dass der neue Zylinder schmaler wird. Die Freiheit des neuen Zylinders an Punkt **q** wird mit c_q bezeichnet:

$$c_q = C(k_3) + t_q * (C(k_4) - C(k_3))$$

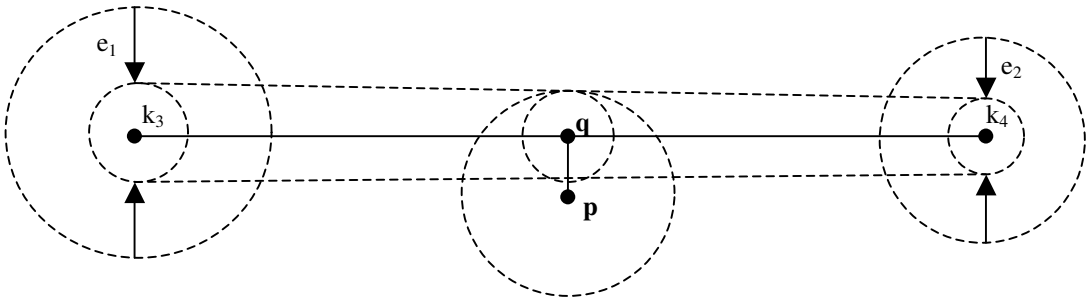


Abbildung 4.13: Reduktion der Freiheiten an Knoten k3,k4

Falls gilt $c_q > (c_p - d)$, beschreibt die neue Kante am Punkt **q** mehr Freiraum als sie darf. Dann müssen die Freiheiten um k_3 und k_4 um jeweils zwei Beträge e_1 und e_2 verringert werden, so dass gilt $c_q = (c_p - d)$. Die Verringerung durch e_1 und e_2 ist nicht eindeutig und kann aber dahin gehend optimiert werden, dass die absoluten Werte von e_1 und e_2 möglichst klein bleiben. Sollte $c_q \leq (c_p - d)$ sein, ist keine Korrektur nötig, denn es wird nur weniger Freiraum als vorher beschrieben. Im optimalen Fall ist d=0 und $c_p=c_q$, dann bewirkt ein Kantentausch keinen Freiraumverlust.

Die Kosten für einen Kantentausch berechnen sich über die entstandene Volumendifferenz. Wurden die Freiheiten an den Knoten k_3 und k_4 nicht verringert, brauchen nur die Flächen f_i, f_j und f_k, f_l beachtet werden.

$$\text{Kosten eine Kantentausches} = (\mathit{Volume}(f_i) + \mathit{Volume}(f_j)) - (\mathit{Volume}(f_k) + \mathit{Volume}(f_l))$$

Wurden die Freiheiten r_3 und r_4 reduziert, müssen die Volumen für der Flächen f_i und f_j jeweils mit den ursprünglichen Freiheiten und die Volumen von f_k und f_l mit den korrigierten Freiheiten berechnet werden. Die Reduzierung einer Freiheit wirkt sich zusätzlich auf alle anderen benachbarten Flächen der jeweiligen Knoten aus und erzeugt zusätzlich Kosten. Für jede an k_3 und k_4 liegende Fläche f mit $f \neq f_i$ und $f \neq f_j$ wird die Volumendifferenz berechnet, welche sich

durch die verringerte Freiheit ergibt. Alle diese Kosten werden dann zu den Gesamtkosteen aufsummiert. Damit wäre der Tauschvorgang abgeschlossen.

4.6 Knoten entfernen

Ein Knoten kann entfernt werden, falls sein Kantengrad zwei oder drei ist (ein Kantengrad von eins kann es im Voronoi Graphen nicht geben). Ist der Kantengrad gleich zwei, hängt nur eine einzelne Fläche an dem Knoten und das Entfernen ist trivial (siehe Kapitel 4.4). Ist der Kantengrad gleich drei, können entweder zwei oder drei Flächen an dem Knoten liegen. Beide Fälle müssen gesondert behandelt werden.

4.6.1 Knoten mit zwei Flächen

Der Fall, dass zwei Flächen an Knoten k_1 anliegen wird in drei weitere Teilfälle unterscheiden. Die Flächen werden analog zum Kantentausch wieder mit f_i und f_j benannt, mit der gemeinsamen Kante (k_1, k_2). Des weiteren wird wieder das Lot zwischen (k_1, k_2) und (k_3, k_4) gefällt und die Lotfußpunkte **p** und **q**, bzw. t_p und t_q berechnet. Beim Entfernen des Knotens k_1 mit zwei Flächen wird in die Teilfälle $t_p \in]0,1[$ oder $t_p < 0$ oder $t_p > 1$ unterschieden.

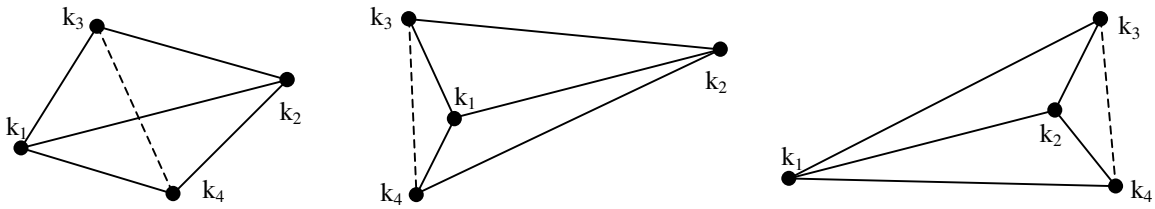


Abbildung 4.14: Die 3 Fälle für $t_p \in]0,1[$ (l.), $t_p \leq 0$ (m.), $t_p \geq 1$ (r.)

Im ersten Fall $t_p \in]0,1[$ wird versucht die Kante (k_1, k_2) mit der Kante (k_3, k_4) wieder zu Tauschen, so dass der Kantengrad an k_1 auf zwei sinkt. Dann ist das Entfernen von k_1 wieder trivial, da nur noch eine Fläche zu beachten ist.

Komplizierter ist der zweite Fall $t_p \leq 0$, wo der Knoten k_1 eine konkave Ecke der beiden Flächen bildet. Hier wird versucht, die beiden Flächen f_i und f_j durch eine neue Fläche $f_l = (k_2, k_3, k_4)$ zu ersetzen. Die Kanten (k_1, k_2), (k1, k_3) und (k_1, k_4) können für diesen Zweck sicher entfernt werden, da sie nur von den beiden Flächen f_i und f_j benutzt werden, anderenfalls wäre der Grad von k_1 größer drei.

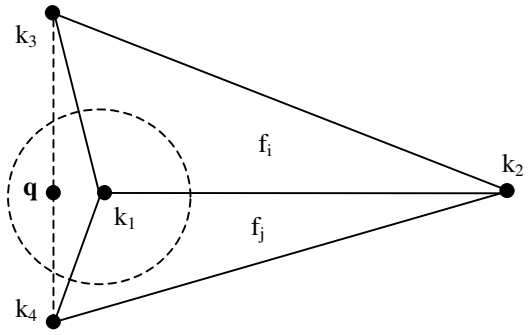


Abbildung 4.15: Knoten mit zwei Flächen entfernen, Fall $t_p < 0$

Weiter muss wieder gezeigt werden, dass die neue Fläche f_l im gültigen Freiraum liegt. Für die Kanten (k_2, k_3) und (k_2, k_4) ist dies sicher, jedoch nicht für die neue Kante (k_3, k_4). Schneidet diese Kante allerdings den Freiraum um Knoten k_1 gegeben durch Freiheit $C(k_1)$, liegt auch diese Kante ganz in den Freiraumzylindern um (k_1, k_3) oder (k_1, k_4). Der Punkt **q**, wie vorher berechnet, liegt auf der Kante (k_3, k_4) und ist dem Knoten k_1 am nächsten. Falls die Entfernung $d = |\mathbf{q}\text{-}P(k_1)| < C(k_1)$ ist, schneidet die neue Kante folglich die Freiraumkugel um k_1. Damit liegt auch die neue Fläche f_l im gültigen Freiraum und die beiden ursprünglichen Flächen können durch diese Fläche ersetzt werden. Der Knoten k_1 ist damit entfernt und die Nachbarschaftsbeziehungen der umliegenden Knoten ist erhalten geblieben. Allerdings muss, wie schon beim Kantentausch, sichergestellt werden, dass das beschriebene Freiraumvolumen der neuen Fläche nicht über das Freiraumvolumen der alten Flächen hinausgeht. Deshalb muss eventuell wieder eine Korrektur der Freiheiten der Knoten k_3 und k_4 erfolgen (siehe Kapitel 4.5).

Der dritte mögliche Fall $t_p \geq 1$ verläuft analog zum Fall $t_p \leq 0$, nur mit dem Unterschied dass k_1 und k_2 vertauscht sind. Um den Algorithmus möglichst einfach zu halten, wird in diesem Fall angenommen, dass der Knoten k_1 nicht zu entfernen ist. Die Reduktion wird dann zu einem anderen Zeitpunkt über den Knoten k_2 geschehen.

4.6.2 Knoten mit drei Flächen

Falls drei Flächen um den Knoten k_1 liegen, ist er von diesen eingeschlossen. Dabei bilden die Flächen im optimalen Fall eine Ebene, im ungünstigen Fall jedoch einen steilen Tetrader. In beiden Fällen wird versucht, diese drei Flächen f_i, f_j und f_k durch eine neue Fläche $f_l = (k_2, k_3, k_4)$ zu ersetzen und so k_1 zu entfernen. Sollte diese Fläche bereits existieren, brauchen keine weiteren Bedingungen überprüft werden und die Topologie bleibt erhalten. Natürlich kann ein Verlust an Freiraumvolumen entstehen, was sich auf die Kosten der Entfernung des Knotens auswirken wird.

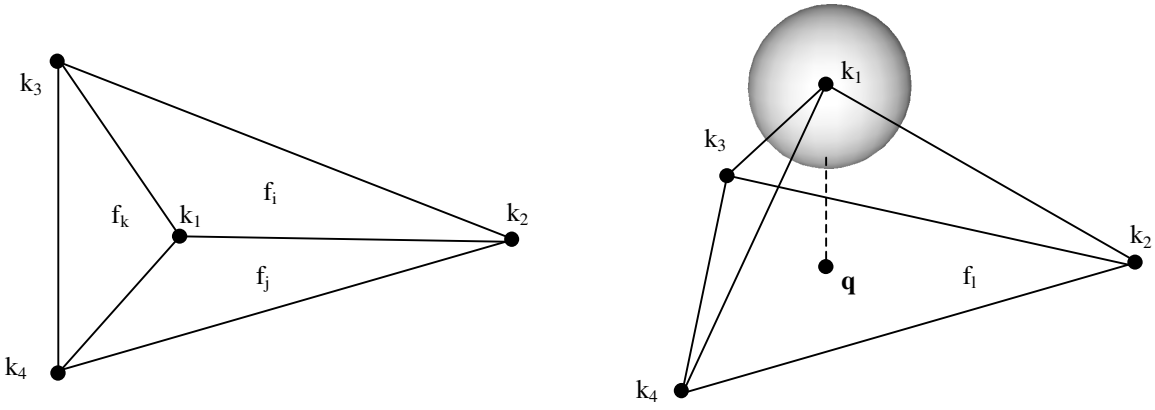


Abbildung 4.16: Knoten k_1 hat drei Kanten und drei Flächen

Wird die Fläche $f_l = (k_2, k_3, k_4)$ neu in den Graphen eingefügt, muss wieder sichergestellt werden, dass sie nur durch vorhandenen Freiraum verläuft. Dafür wird das Lot von Knoten k_1 auf die Fläche f_l gefällt und der Lotfußpunkt **q** bestimmt. Erstreckt sich dann die Freiheit des Knoten k_1 über diesen Punkt **q** hinaus, liegt die gesamte Fläche im Freiraum, welcher zwischen den vier beteiligten Knoten aufgespannt wird. Damit können die drei Flächen durch eine neue Fläche ersetzt werden, jedoch muss wieder überprüft werden, ob die Freiheiten an den Knoten k_2, k_3 und k_4 eventuell wieder reduziert werden müssen.

Der vorher gültige Freiraumradius um **q** ist $c_q = C(k_1) - |\mathbf{q}\text{-}P(k_1)|$. Nach dem Entfernen des Knoten k_1 wird die Freiheit des selben Punktes durch *Clearance* (**q**, f_l) beschrieben. Dabei muss gelten, dass $c_q \geq$ *Clearance* (**q**, f_l) ist, also darf durch die neue Fläche f_i an diesem Punkt nicht mehr Freiraum definiert werden als vorher durch Knoten k_1. Falls *Clearance* (**q**, f_l) > c_q müssen die Freiheiten von k_2, k_3, k_4 entsprechend reduziert werden. Wieder ist diese Reduktion nicht eindeutig und sollte dahingehend optimiert werden, dass die drei Freiheiten möglichst wenig verringert werden, um den Freiraumverlust so gering wie möglich zu halten.

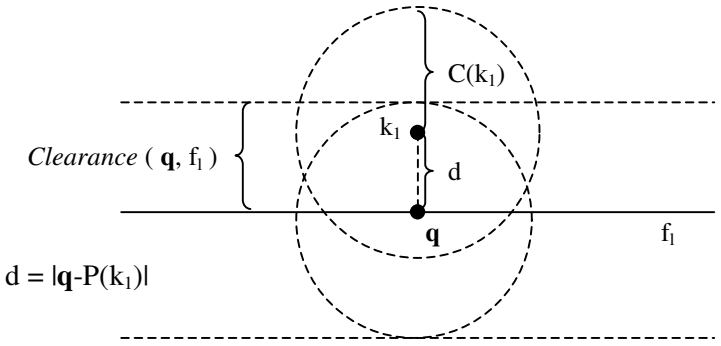


Abbildung 4.17: Überprüfung, ob q im Freiraum von k_1 liegt

5 Kamera Management

Schon bei der Entwicklung des Verfahrens wurde für das Kamera Management verschiedene Kriterien und Faktoren festgelegt, welche die Kamerabewegungen steuern sollen. Dies sind relative Position zum Objekt, Sichtbarkeit, Trägheit und relative Freiheit im Raum. Die Berechnung der einzelnen Faktor kann für einen Raum komplett durch die VDG Datenstruktur abgedeckt werden. Die einzelnen Kostenfunktionen wurden jedoch nur grob umrissen und nicht genau definiert. Weiterhin ist die Frage zu klären, wie für eine aktuelle Kamerakonfiguration eine Nachfolgekonfiguration gefunden werden kann, welche möglichst wenig Kosten verursacht. Und dies in zweierlei Hinsicht, zum einen in Bezug auf die Kostenfunktion, aber auch auf die benötigte Rechenzeit.

5.1 Kostenfunktionen und Gewichte

Da die Gesamtgewichtsfunktion eine Summe ist, kann man nicht die Differenzen von Entfernungen, Winkeln oder Geschwindigkeiten einfach addieren, da sie in ihrer Einheit nicht vergleichbar sind. Um trotzdem die verschiedene Gewichtungen vergleichen zu können, müssen die unterschiedlichen Kostenfunktionen in eine einheitslose Form gebracht werden, welche dann vergleichbar ist. Dies wird durch eine Normierung erreicht, wobei jede Kostenfunktion ein Verhältnis des aktuellen Wert zu dem gewünschten Optimalwert μ und einer erlaubten Toleranzgrenze σ angibt.

Für eine Kostenfunktion $f : \mathbb{R} \rightarrow \mathbb{R}^+$ muss für einen Eingabewert $x \in \mathbb{R}$ in Bezug auf ein Optimum μ und eine Toleranz σ folgendes gelten:

$$\begin{array}{lll} x = \mu & \rightarrow & f(x) = 0 \\ |x - \mu| < \sigma & \rightarrow & f(x) < 1 \\ |x - \mu| = \sigma & \rightarrow & f(x) = 1 \\ |x - \mu| > \sigma & \rightarrow & f(x) > 1 \end{array}$$

Da die Funktionen ein Verhältnis zwischen Eingabe- und Optimalwert ausdrücken sind sie einheitslos. Lineare Gewichtungsfaktoren, wie z.B. in Kapitel 2.5 verwendet, sind nicht direkter Bestandteil der Funktionen. Die Gewichtung von Kostenfunktionen geschieht über die Wahl der Toleranzgrenze σ, je kleiner die Toleranz, desto größer die Funktionskosten. Die meisten Kostenfunktionen sind Variationen der Grundfunktionen f_0:

$$f_0(x) = \left(\frac{x - \mu}{\sigma} \right)^2$$

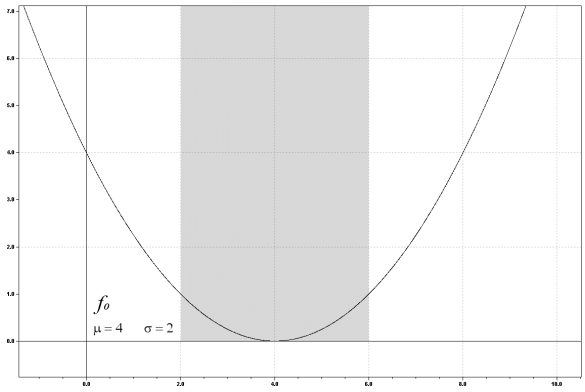


Abbildung 5.1: Graph der Kostenfunktion f_0 mit $\mu = 4$ und $\sigma = 2$

Der Vorteil dieser quadratische Grundfunktionen f_0 ist, dass sie innerhalb der Toleranzgrenzen sich sehr träge verhält und bei geringen Abweichungen vom Optimum fast überhaupt keine Kosten verursacht. Überschreiten die Eingabewerte jedoch den Toleranzbereich, steigen die Kosten sehr schnell an und selbst geringe Unterschiede werden im Gesamtkostenanteil deutlich.

Bei der Kostenfunktion der Distanz wird die optimal euklidische Distanz mit d_μ und die normale Toleranz mit d_σ beschrieben:

$$f_{dist}(c) = \left(\frac{\left| Position(t) - Position(c) \right| - d_\mu}{d_\sigma} \right)^2$$

Für die Kosten des Winkels zwischen optimalem und gegebenen Winkel wird zuerst der Winkel $\measuredangle$ zwischen zwei Vektoren v_1 und v_2 definiert durch:

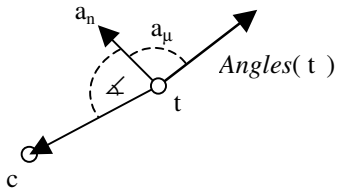
$$\measuredangle(v_1, v_2) = \text{acos}\left(\frac{v_1 \cdot v_2}{|v_1| * |v_2|} \right)$$

Der optimale Winkel a_μ wird in Polardarstellung gegeben und auf den Richtungswinkel des Zielobjektes *Angles*(t) zu einem Gesamtwinkel $a_t = (\alpha, \beta)$ addiert. Aus diesem Gesamtwinkel wird ein Vektor $a_n = Vector(\alpha, \beta, 1)$ berechnet, der die Länge 1 hat. Die Hilfsfunktion *Vector*(θ, ϕ,d) berechnet aus der Polardarstellung eines Punktes (θ, ϕ, d) die kartesischen Koordinaten (x, y, z) im Raum $\mathbb{R}^3$. Dies geschieht indem der Vektor (d,0,0) um θ Grad um die y-Achse und dann um ϕ Grad um die z-Achse rotiert wird:

$$Vector(\theta, \phi, d) = \left(\begin{bmatrix} d \\ 0 \\ 0 \end{bmatrix} * \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \right)^T * \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Die Varianz des Winkels wird gegeben mit $a_\sigma \in\ ]0,\pi]$:

$$f_{angle}(c) = \left(\frac{\measuredangle(\overrightarrow{a_n}, \overrightarrow{tc})}{a_\sigma} \right)^2$$



Die Kostenfunktion der Freiheit eines Punktes hat kein echten Optimalwert, da das Potential der Kehrwert der Freiheit ist und der optimale Potentialwert 0 eine unendlich große Freiheit besitzen würde. Trotzdem kann eine Kostenfunktion basierend auf f_0 benutzt werden, welche ein Verhältnis der Freiheit in Bezug auf eine Toleranz p_σ und der aktuellen Geschwindigkeit angibt :

$$f_{clear}(c) = \left(\frac{Velocity(c) * p_\sigma}{Clearance(c)} \right)^2$$

Die Funktion ist folgendermaßen zu interpretieren. Der Faktor *Velocity*(c)*p_σ gibt eine bestimmte Strecke pro Sekunde an. Ist die Freiheit um Konfiguration c größer als diese Strecke, kann sich die Kamera bei gleicher Geschwindigkeit mindesten p_σ Sekunden noch beliebig im Raum bewegen. In diesem Fall bleiben die Kosten für die Freiheit unter eins. Ist die Freiheit jedoch kleiner dieser Strecke, ist nicht sichergestellt, dass die Kamera sich in den nächsten Sekunden des Toleranzbereichs frei bewegen kann, die Kosten steigen dann sehr schnell an. Je größer man p_σ wählt, desto mehr zwingt man die Kamera, sich an den Mittelpunkten des umgebenden Raums zu orientieren. Bleibt der Spieler stehen, wird die Kamera langsamer und der Zähler der Funktion geht gegen null. So kann sich die Kamera auch den Wänden und Ecken nähern, falls eine ruhige Situation vorliegt.

Für die Trägheitsfunktion muss die Position c_t berechnen werden, welche die Kamera nur durch die Bewegungsträgheit hätte. Dafür muss auf *Position*(c_i) ein Bewegungsvektor addiert werden, der sich aus der Bewegungsrichtung und der Geschwindigkeit mal verstrichener Zeit ergibt:

$$c_t = Position(c_i) + Vector(\ Angles(c_i), Velocity(c_i) * \Delta time\ )$$

Die Kosten der Trägheitsfunktion beschreiben die Differenz zwischen *Position*(c) und c_t im Verhältnis zur Kamerageschwindigkeit wie schon bei f_{clear} geschehen. Der Optimalwert der Trägheit ist ein Abstand von 0 zwischen den beiden Positionen und erscheint folglich nicht in der Funktion:

$$f_{inertia}(c) = \left(\frac{\left| Position(c) - c_t \right|}{\left| Position(c_i) - c_t \right| * i_\sigma} \right)^2$$

Der Trägheitsfaktor i_σ bestimmt die normal erlaubte Varianz der Trägheit. Ein Trägheitsfaktor von 0,5 bedeutet, dass eine Abweichung von der Trägheitsposition um 50% die Kosten von 1 verursachen. Je kleiner i_σ ist, desto höher sind die Trägheitskosten.

Die Kostenfunktion der Sichtbarkeit unterscheidet sich grundsätzlich von den anderen Kostenfunktionen. Sichtbarkeit ist eine Eigenschaft, welche sich nur auf die Existenz eines geradlinigen, freien Weges zwischen zwei Punkten bezieht, also nur wahr oder falsch seien kann. Man könnte also Punkten im Raum, von den aus der Zielpunkt sichtbar ist, die Kosten von null geben und allen anderen Punkten die Kosten von eins. Eine solche einfache Bewertung hilft jedoch nicht, Punkte mit Kosten von eins derart zu klassifizieren, ob sie eher auf dem Weg zu einem Punkt mit Sichtbarkeit zu t liegen oder nicht. Deshalb kann man von folgender Annahme ausgehen: je kürzer der minimale, nicht geradlinige, freie Weg von einem Punkt zum Zielpunkt t ist, desto eher erreicht man über diesen Punkt einen gradlinigen, freien Weg zum Zielpunkt. Dies leitet sich aus der Tatsache ab, dass geradlinige, freie Wege immer auch die Kürzesten sind.

Für diesen Zweck definieren wird neben der euklidischen Distanz noch die Voronoi Distanz für zwei Punkte $\mathbf{p}_{start}$ und $\mathbf{p}_{goal}$ im Raum $\boldsymbol{C}_{free}$ eingeführt. Erst werden die zwei nächstgelegenen Punkte $\mathbf{p}'_{start} = \rho(\ \mathbf{p}_{start}\ )$ und $\mathbf{p}'_{goal} = \rho(\ \mathbf{p}_{goal}\ )$ im VDG bestimmt sowie die entsprechenden Flächen f_{start} und f_{goal} in denen diese beiden Punkte liegen. Sind die Flächen identisch, also $f_{start} = f_{goal}$, ist die Voronoi Distanz gleich der euklidischen Distanz. Ansonsten wird ein kürzester Weg $(v_{start},...,v_{goal}\ )$ mit $v_{start} \in f_{start}$ und $v_{goal} \in f_{goal}$ im VDG gesucht, so dass die Gesamtlänge *vd* minimiert wird:

$$vd = \left| p_{start} - P(v_{start}) \right| + \sum_{t=start}^{goal-1} \left| P(v_i) - P(v_{i+1}) \right| + \left| P(v_{goal}) - p_{goal} \right|$$

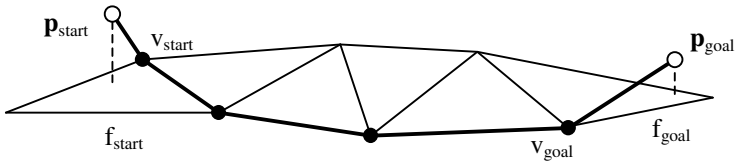


Abbildung 5.2: Kürzester Weg von $\mathbf{p}_{start}$ nach $\mathbf{p}_{end}$ durch den VDG

Der Weg mit der kürzesten Länge *vd* beschreibt die Voronoi Distanz und wird notiert mit *VDistance* ($\mathbf{p}_{start}$, $\mathbf{p}_{goal}$). Zu beachten ist, dass v_{start} und v_{goal} nicht unbedingt die euklidisch nächsten Knoten zu den Punkten $\mathbf{p}_{start}$ und $\mathbf{p}_{goal}$ sind.

Die Kostenfunktion ist eine Fallunterscheidung entsprechend der Sichtbarkeit:

$$f_{vis}(c) = \begin{cases} 0, \text{ falls } \textit{Visible}(c, t) \\ \text{sonst } 1 + \textit{VDistance}(c, t) \end{cases}$$

In der Praxis wird eine Heuristik verwendet werden, welche die Gesamtsichtbarkeit des Zielobjektes als relativen Anteil als einen Wert zwischen null und eins beschreibt. So können Punkte von denen das Zielobjekt vollständig oder nur teilweise aus sichtbar ist noch weiter unterschieden werden. In der Theorie sind unsere Zielobjekte jedoch körperlos.

Für die Gesamtkostenfunktion beschreibt der Faktor f_{novis} die Gewichtung der Sichtbarkeitskosten zu allen anderen Anteilen. Wie lange das Zielobjekt nicht sichtbar gewesen ist wird durch die Zeitspanne Δnvis beschrieben, v_σ beschreibt die erlaubte Toleranz, das Optimum ist 0:

$$f_{nvis} = \left(\frac{v_\sigma}{v_\sigma + \Delta nvis} \right)^2$$

Die Gesamtkostenfunktion ist die Summe der Einzelkosten geteilt durch 5. In Durchschnitt bedeuten also Gesamtkosten um den Wert 1, dass alle Teilkosten ungefähr im normalen Toleranzbereich liegen.

$$f_{total}(c) = \frac{f_{vis}(c) + f_{nvis} * (f_{dist}(c) + f_{angle}(c) + f_{inertia}(c) + f_{clear}(c))}{5}$$

5.2 Berechnung der Nachfolgekonfiguration

Mit dem Beginn einer neuen Szene, muss eine Startkonfiguration c_0 der Kamera bestimmt werden, der Einfachheit halber wählt man in diesem Fall c_0 = t. Für alle weiteren Konfigurationen muss jeweils nach einem kurzen Zeitintervall Δ*time* ein Nachfolgekonfiguration c_{i+1} berechnet werden, basierend auf der aktuellen Konfiguration c_i. In dem Zeitintervall Δ*time* hat sich eventuell die Position, Richtung oder Geschwindigkeit des Zielobjektes t geändert.

Jeder Kamerakonfiguration c_i wird eine Voronoi Fläche f_i zugeordnet für die gilt, dass ρ(*Position*(c_i)) $\in$ f_i ist und die Fläche f_i des Voronoi Graphen die Freiheit *Clearance*(*Position*(c_i),f_i)

maximiert. Für die Fläche f_0 der Startkonfiguration sind alle Flächen im VDG mögliche Kandidaten, welche untersucht werden müssen. Eine räumliche Sortierung der Flächen in einem kd-Baum ist für diese Suche sehr nützlich, da so eine schnelle Aussortierung von möglichen Flächen erfolgen kann. Für jede Nachfolgekonfiguration c_{i+1} muss die Fläche f_{i+1} neu bestimmt werden. Dabei kann angenommen werden, dass die Entfernung zwischen den beiden Konfigurationen c_i und c_{i+1} hinreichend klein ist, da bei normalen Bildfrequenzen von 30 – 50 Hz die Zeitspanne $\Delta time$ sehr kurz ist (0,02-0,03 ms) und zum anderen die Voronoi Flächen durch die Reduktion möglichst groß sind.

Dann gilt für die Fläche f_{i+1} aufgrund der Stetigkeit von ρ (siehe Kapitel 3.3), dass sie in einer lokalen Menge von Flächen $f_i \cup Neighbours(f_i)$ zu finden ist und damit die Bestimmung von f_{i+1} auf einen geringen Aufwand beschränkt wird. Die Anzahl der Nachbarflächen in einem Voronoi Graphen wird nicht durch die Gesamtgröße der Eingabemenge bestimmt, sondern durch die jeweilige lokale Geometrie und Symmetrie der Eingabewelt.

Die exakte Bestimmung der Nachfolgekonfiguration c_{i+1} mit minimalen Kosten ist eine kaum lösbare Aufgabe, wie bereits Drucker feststellte (siehe Kapitel 2.2). Es ist ein klassisches Problem der nichtlinearen Optimierung, wobei der unendliche große Lösungsraum $\boldsymbol{C}_{free}$ durch die zwingende Nebenbedingung $Position(c_{i+1}) \cap \boldsymbol{CB} = \varnothing$ beliebig zufällig unterbrochen wird. Eine systematische Konstruktion der Nachfolgekonfiguration ist also nicht möglich. Dies wäre in einem hindernisfreien Raum für die Kriterien Abstand, Winkel und Trägheit ohne weiteres möglich. Die Strategie, mit der nach einer guten Nachfolgekonfiguration gesucht wird, besteht aus drei Punkten:

1. Reduktion des Lösungsraums
2. Wahl günstiger Startpunkte nahe der Optima der verschiedenen Kriterien
3. Rekombination von bereits gefundenen Lösungen

Der Lösungsraum für mögliche Konfigurationen wird auf den Teilraum $Volume(f_i)$ beschränkt, wodurch sich viele Vorteile ergeben. Da dieser Raum konvex und frei von Hindernissen ist, existiert für alle Konfigurationen c in diesem Raum ein freier, gerader Weg von c_i nach c, da c_i selbst in $Volume(f_i)$ liegt. Weiterhin gilt für alle Konfigurationen c, dass falls sie die Nachfolgekonfiguration c_{i+1} werden, ihre entsprechende Voronoi Fläche f_{i+1} nicht außerhalb der Nachbarschaft von f_i liegt. Der entscheidende Vorteil ist jedoch, dass bei der Rekombination von bereits gefundenen Konfigurationen durch lineare Interpolation zu neuen Konfigurationen, diese immer wieder in $Volume(f_i)$ liegen und damit gültig sind. Durch diese Vorgehensweise wird das Problem der Hindernisse bei der Suche nach dem Optimum ausgeblendet, da der Lösungsraum stetig und zusammenhängend ist. Die Dreiecksfläche f_i mit den Freiheitsradien an den Eckpunkten beschreibt diesen Raum in sehr einfacher Art, wobei die Eigenschaften der Voronoi Graphen sicherstellen, dass möglichst viel vom umgebenen Freiraum wiedergegeben wird.

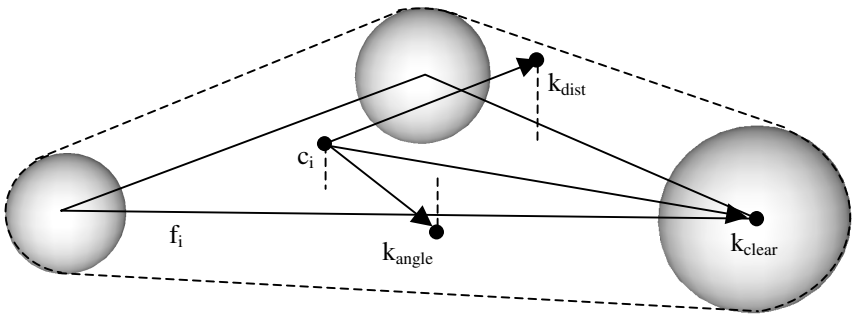


Abbildung 5.3: Kandidaten im Freiraumvolumen um Fläche f_i

Bei der Suche nach guten Positionen für die Nachfolgekonfiguration wird für jedes Kriterium ein Punkt im Raum konstruiert, welcher möglichst nah an dessen Optimum liegt, aber innerhalb des Lösungsraums. Diese Punkte in $\mathbb{R}^3$ werden mit der Menge Kandidaten K bezeichnet, welche zu Beginn sechs Elemente hat $\mathbf{K} = \{ \mathbf{k}_{dist}, \mathbf{k}_{angle}, \mathbf{k}_{inertia}, \mathbf{k}_{clear}, \mathbf{k}_{vis}, \mathbf{k}_{neighb} \}$.

5.3 Berechnung der Kandidaten

Die ersten drei Kandidaten werden entsprechend ihren Kriterien in Bezug auf *Position*(c_i) und der aktuellen Konfiguration t des Zielobjektes konstruiert. Kandidat $\mathbf{k}_{dist}$ beschreibt eine möglichst gute Position für den optimalen Abstand und entsteht aus Verschiebung von $\mathbf{c}_i$ senkrecht zu t, so dass der Abstand |Position(t)- $\mathbf{k}_{dist}$| = d_μ ist. Der Kandidat $\mathbf{k}_{angle}$ entsteht aus einer Rotation von c_i um t, wobei der aktuelle Abstand gehalten wird. Jedoch wird nicht um den gesamten Winkel β rotiert, welcher zur optimalen Winkelposition führen würde, sondern nur um ¼ β. Dadurch rotiert $\mathbf{k}_{angle}$ auch bei sehr großem Winkel $\beta \cong \pi$ um t und der Weg von *Position*(c_i) nach $\mathbf{k}_{angle}$ liegt nahe dem gedachten Rotationskreis. Zum anderen wird so eine gedämpfte Annäherung an den Optimalwinkel erreicht und ein abrupter Rotationsstop vermieden. Bei der Trägheit spielt das Zielobjekt t keine Rolle und $\mathbf{k}_{inertia}$ berechnet sich aus Richtung und Geschwindigkeit von c_i.

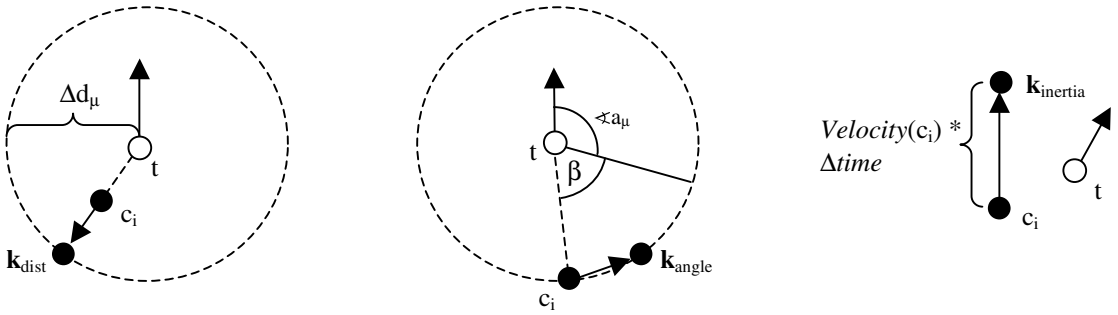


Abbildung 5.4: Konstruktion der Kandidaten $\mathbf{k}_{dist}$ (l.), $\mathbf{k}_{angle}$ (m.), $\mathbf{k}_{inertia}$ (r.)

Die Kandidaten $\mathbf{k}_{clear}$ und $\mathbf{k}_{vis}$ können nicht wie die anderen von c_i aus konstruiert werden. Deshalb werden lokale Voronoi Knoten dahingehend untersucht, ob sie gute Orte für diese Kandidaten sind. Für den Kandidaten des Freiheitskriterium $\mathbf{k}_{clear}$ macht dies eindeutig Sinn, da Voronoi Knoten immer Punkte in der lokalen Umgebung sind, welche eine maximale Freiheit haben. Bei der Sichtbarkeit ist dies nicht ganz so klar, jedoch liegen Voronoi Knoten immer in den geometrischen Mittelpunkten der Ungebung und damit auch möglichst weit entfernt von Hindernissen, welche die Sichtgerade blockieren können. Man kann daher annehmen, dass die durchschnittliche Sichtbarkeit eines beliebigen Punktes von einem Voronoi.Knoten aus höher ist, als von anderen Punkten in seiner Umgebung.

Welche Voronoi Knoten in der direkten Umgebung um einen Punkt $p \in$ *Volume*(f) liegen, wird durch die Menge *Reachable*(p,f) beschrieben. In dieser Menge sind alle lokalen Voronoi Knoten, die vom Punkt p aus auf einem geradem Weg zu erreichen sind. Dies sind auf jeden Fall die drei Voronoi Knoten (v_1, v_2, v_3) der Fläche f, aber je nach Lage von p auch deren Nachbarknoten.

$$\textit{Reachable}(p, f) = \{ v \in \{ \textit{Neighbours}(v_1) \cup \textit{Neighbours}(v_1) \cup \textit{Neighbours}(v_1) \} \mid \textit{Line}(P(v), p) \subset \textit{Volume}\{ f \cup \textit{Neighbours}(f) \} \text{ mit } f = (v_1, v_2, v_3) \}$$

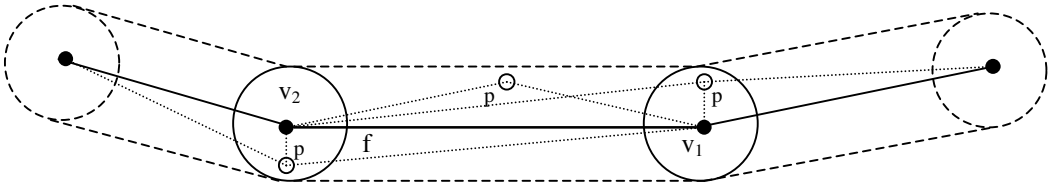


Abbildung 5.5: Erreichbare Knoten von Punkt p aus

Folglich wird der Kandidat $\mathbf{k}_{clear}$ durch den Knoten aus *Reachable*(*Position*(c_i), f_i) beschrieben, der die geringsten Kosten f_{clear} erzeugt. Für $\mathbf{k}_{vis}$ gilt dasselbe entsprechend den Kosten f_{vis}.

$$\mathbf{k}_{clear} = P(v_{clear}) \qquad \text{mit } v_{clear} = \underset{v \in Reachable(Position(c_i), f_i)}{\arg\min} f_{clear}(P(v))$$

$$\mathbf{k}_{vis} = P(v_{vis}) \qquad \text{mit } v_{vis} = \underset{v \in Reachable(Position(c_i), f_i)}{\arg\min} f_{vis}(P(v))$$

Aus der Menge der lokalen Voronoi Knoten wird der letzte Kandidat $\mathbf{k}_{neighb}$ durch den Knoten mit den niedrigsten Gesamtkosten bestimmt.

$$\mathbf{k}_{neighb} = P(v_{total}) \qquad \text{mit } v_{total} = \underset{v \in Reachable(Position(c_i), f_i)}{\arg\min} f_{total}(P(v))$$

Bei der Bestimmung dieser Kandidaten wurde nicht die Einschränkung des Lösungsraums beachtet. Deshalb muss in einem zweiten Schritt für alle Konfigurationen sichergestellt werden, dass sie im Raum *Volume*(f_i) liegen. Gilt für einen Kandidaten $\mathbf{k} \cap Volume(f_i) = \varnothing$, so liegt er außerhalb und muss entsprechend in Richtung *Position*(c_i) zurückgesetzt werden. Dafür wird die Funktion *Trace* benutzt, die schon bei der Berechnung der Sichtbarkeit (Kapitel 3.9) verwendet wurde:

$$\mathbf{k}' = Trace(Position(c_i), \mathbf{k}, f_i)$$

Nach dieser Beschränkung liegen alle Kandidaten in *Volume*(f_i) und sie können kombiniert werden. Die Idee ist es, einen eventuell besseren Kandidaten zu finden, indem man einen gewichteten Mittelpunkt zwischen den bereits bekannten Kandidaten errechnet. Für jedem Kandidat k wird ein Gewicht w(**k**) aus dem Kehrwert der Kosten f_{total} berechnet und die Summe aller Gewichte mit w_{total} bezeichnet:

$$w_{total} = \sum_{\mathbf{k} \in \mathbf{K}} w(\mathbf{k}) \qquad \text{mit} \qquad w(\mathbf{k}) = \frac{1}{f_{total}(\mathbf{k})}$$

Sollte für einen der Kandidaten die Kosten gering genug sein, also $f_{total}(\mathbf{k}) < 0 + \varepsilon$, kann die Suche abgebrochen werden. Ansonsten kann ein neuer Kandidat $\mathbf{k}_{comb}$ aus den bereits vorhandenen erzeugt werden:

$$\mathbf{k}_{comb} = \sum_{k \in K} k * \frac{w(k)}{w_{total}}$$

Dieses Kombinationsverfahren kann beliebig häufig oder mit unterschiedlichen Teilmengen von K wiederholt werden um neue Kandidaten zu produzieren. Hier wird der Einfachheit halber nur

von einer einzigen Rekombination ausgegangen und anschliessend der beste Kandidat $\mathbf{k}_{best}$ mit den kleinsten Kosten f_{total} aus der Menge aller Kandidaten bestimmt..

Durch den Punkt $\mathbf{k}_{best}$ wird die endgültige Position der Nachfolgekonfiguration allerdings noch nicht festgelegt. Zwar ist $\mathbf{k}_{best}$ ein Punkt in der Umgebung, den es zu erreichen gilt, jedoch kann er unter Umständen sehr weit entfernt sein. Sind andere Kostenfaktoren momentan wichtiger, z.B. Abstand oder Sichtbarkeit, fließt der Trägheitsfaktor nur wenig in $\mathbf{k}_{best}$ ein und die Kamera würde von Optimum zu Optimum springen. Deshalb muss eine Schrittweite s bestimmt werden, mit welcher sich die Kamera von *Position*(c_i) in Richtung $\mathbf{k}_{best}$ bewegt und damit auch die neue Geschwindigkeit bestimmt. Die Kamerageschwindigkeit sollte allgemein träge sein, bei Annäherung an die optimale Position langsamer werden und in kritischen Situationen notfalls stark beschleunigen können. Folgende Funktion für s zeigt dieses Verhalten:

$$s = \left((1 - frac) * Velocity(c_i) + frac * (1 + Velocity(c_i)) * \sqrt{f_{total}(Position(c_i))}\right) * \Delta time$$

Die Funktion berechnet eine neue Geschwindigkeit, welche sich erhöht, falls die aktuellen Kosten außerhalb des Toleranzbereiches sind (f_{total} >1). Ist man bereits nah am Optimum (f_{total}<1) sinkt die neue Geschwindigkeit, die Wurzel gleicht das quadratische Verhalten der Kostenfunktion wieder aus. Die entgültige Geschwindigkeit ist ein Mittel zwischen alter und neuer Geschwindigkeit, welche eventuell auf eine erlaubte Maximalgeschwindigkeit begrenzt wird. Die Schrittweite s ergibt sich dann aus Geschwindigkeit mal verstrichener Zeit $\Delta time$.

Abbildung 5.6: Konstruktion der Nachfolgekonfiguration c_{i+1}

6 Implementation

Um die beschriebenen Eigenschaften des Voronoi Distanz Graphen und des darauf basierenden Kamera-Management in der Praxis zu überprüfen, wurde das gesamte System für das 3D-Videospiel *Half-Life* implementiert und ausgewertet. Dafür sprachen hauptsächlich zwei Gründe. Zum einen ist das Spiel *Half-Life* im Genre der 3D-Aktionspiele eine der erfolgreichsten Umsetzungen und seit über 4 Jahren mit verschiedenen Nachfolgeprodukten auf dem Markt (Team Fortress Classic, Opposing Forces, Blue-Shift, Counter-Strike). Addiert man die momentan Spielerzahlen im Internet aller Variationen zusammen, wird *Half-Life* mehr gespielt als alle anderen Spiele dieser Art zusammen [Gam03]. Damit ist *Half-Life* eine Spielplattform, welche als eine realistische Referenz für moderne 3D-Echtzeitspiele in Bezug auf Leistungen und Komplexität herangezogen werden kann. Zum anderen ist es möglich, *Half-Life* einfach und kostengünstig als Basis für eigene Modifikationen zu nutzen.

Vom Hersteller Valve Software wird ein umfangreiches Entwicklerpaket *Half-Life Software Developer Kit* [SDK03] zur freien Verfügung gestellt, welches Ressourcen, Quellcode und Werkzeuge enthält, um das Spiel nach belieben zu verändern. Damit wird es nichtkommerziellen Spielentwicklern ermöglicht, die grundlegende Technologien für eigenen Inhalten und Konzepten zu nutzen, ohne kostspielige Lizenzverträge einzugehen. Diese Konzept der für den Endbenutzer frei veränderbaren Spielplattform wurde mit dem Spiel *Quake* von id Software 1996 eingeführt und hat sich als richtungsweisend in diesem Bereich erwiesen. Für den Entwickler bedeutet dies eine systematische Trennung von Basistechnologien und Spielinhalten. Als Game-Engine werden Komponenten wie Video- und Tonausgabe, Organisation und physikalische Simulation der Spielwelt und Netzwerkkommunikation bezeichnet. Der eigentliche individuelle Spielinhalt gliedert sich in Spiellogik, Modellen, Grafiken, Ton und Texten. Die grundlegende Architektur ist eine Client-Server Umgebung, sowohl im Einzelspieler als auch im Mehrspielermodus. Die Game-Engine unterteilt sich in einen Server-Prozess (hlds.exe) und einen Client-Prozess (hl.exe), welche über eine paketorientiert Netzwerkverbindung (UDP/IP) miteinander kommunizieren. Beide Prozesse laden dabei die jeweils gewünschte Spiellogik als dynamische Laufzeitbibliothek (hl.dll & client.dll). Sowohl die Spiellogik des Server als auch des Clients sind durch das SDK frei programmierbar. Die Spiellogiken bestimmen dann, welche sonstigen Spielressourcen (Texturen, Sounds, Models) geladen werden.

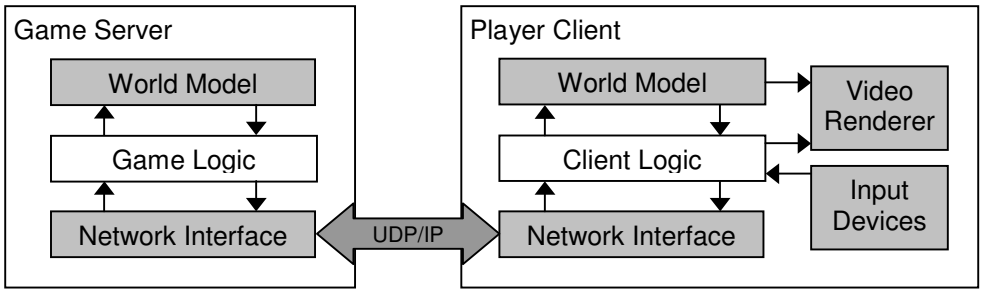


Abbildung 6.1: Half-Life Client-Server Architektur

Dieses Systemdesign ermöglicht eine wesentlich saubere Entwicklung und erleichtert die Lizenzierung der eigenen Technologien an Fremdhersteller. Weiterhin bedeutet es einen Mehrwert für den Endkunden, welcher sich als Hobby-Spielerentwickler betätigen und das Spiel nach seinen eigenen Wünschen modifizieren kann. Diese Veränderungen (*Modifications, Mods*) können relativ einfach sein, z.B. eine geänderte Textur oder eine leichte Änderungen der Spielregeln. Manche Nutzer schließen sich zu privaten Entwicklerteams zusammen und verändern *Half-Life*

so stark, dass vom ursprünglichen Spiel nichts mehr zu erkennen ist und sie erreichen damit teilweise professionelles Niveau (*Total Conversions*). Diese „neuen“ Spiele benötigen weiterhin eine gültige *Half-Life* Installation, müssen jedoch laut SDK Lizenz kostenlos weitergegeben werden [5], wodurch wiederum der Mehrwert des ursprünglichen Spiels gesteigert wird.

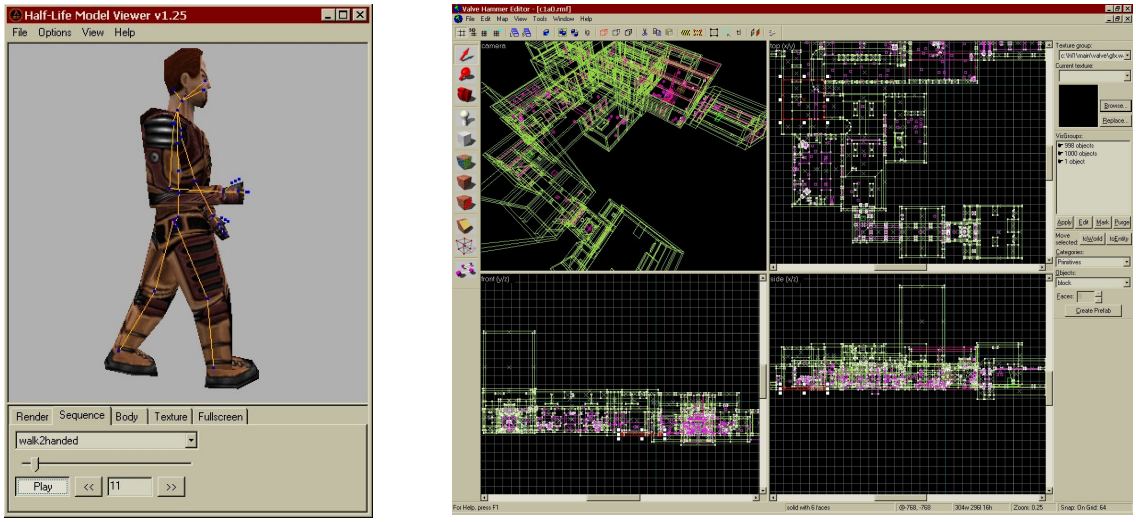


Abbildung 6.2: Charakter Animator (l.), Karten Editor (r.)

Für die Implementation des VDG muss an zwei Punkten der Spielentwicklung eingegriffen werden. Zum einen muss der reduzierte VDG in einer Vorderarbeitsphase aus der statischen Spielwelt berechnet werden. Zum anderen wird das Kamera Management in den Laufzeitcode der Client-Spiellogik eingebunden. Der Quellcode im SDK für Entwicklungswerkzeuge und Spiellogik ist in C/C++ umgesetzt, womit eine gute Balance zwischen Ausführungsgeschwindigkeit, Portabilität und Programmierkomfort erreicht wird. Als Entwicklungsumgebung dient Microsoft Developer Studio unter Windows 2000, wobei jedoch keine Windows spezifischen Funktionen genutzt wurden. Die hier benutzte Hardware war ein Intel Celeron 800 MHz mit 384 MB RAM und einer nVidia Geforce2 Grafikkarte, also ein für aktuelle Maßstäbe schon unterdurchschnittliches System. Zur Visualisierung der Voronoi Graphen und deren Reduzierungsprozesses wurde während der Entwicklung ein Hilfswerkzeug (VDGView) erstellt, welches die OpenGL Bibliothek für die graphische 3D-Ausgabe nutzt. So sind alle Komponenten einfach auf andere Betriebssysteme, z.B. Linux portierbar.

Grundsätzlich folgt die Implementation den bereits beschriebenen Verfahren und Datenstrukturen. Es werden hier die grundlegenden Abläufe des Programmcodes beschrieben und nur vereinzelt auf spezielle Details eingegangen, sofern es nötig erscheint. Dies ist häufig dann der Fall, wenn eine direkte Umsetzung der theoretischen Algorithmen nicht effizient in Bezug auf Rechenzeit oder Speicherbedarf wäre. Meistens werden die Methoden durch Hilfsstrukturen erweitert, wie z.B. kd-Bäume, Bounding Boxen und sortierte Indexlisten, um Zugriffe auf die immensen geometrischen Datenmengen zu beschleunigen. Im Zweifelsfall sollte der eigentliche Quellcode als Referenz genutzt werden, welcher zusätzliche Dokumentation und Kommentare enthält (in Englisch).

[5] ähnlich der *GNU General Public License*

6.1 Vorberechnung des VDG

Die Berechnung des reduzierten VDG wird einmalig für jede Karte (Spielwelt) vom Werkzeug BSP2VDG ausgeführt und gliedert sich in folgende Schritte:

1. Einlesen der Spielwelt im .BSP Format
2. Rasterung der Wandflächen-Polygone
3. Berechnung des Voronoi Graphen
4. Reduktion der Voronoi Knoten
5. Speicherung als .VDG Datei

Die Konsolenapplikation BSP2VDG lässt sich über Eingabeparameter steuern (siehe Anhang E) und nahtlos in den Entwicklungsprozess einer Spielkarte einfügen. Bevor die Umsetzung der einzelnen Schritte beschrieben wird, soll eine kurze Erklärung der manuellen Erstellung und automatische Weiterverarbeitung einer Spielkarte helfen, den Gesamtprozess zu verstehen.

6.1.1 Das .BSP Karten-Format

Das .BSP Kartenformat ist eine Datenstruktur, welche die virtuelle Welt, in der sich der Spieler bewegt, speichert und für die graphische Darstellung und physikalische Kollisionsberechnung genutzt wird. Dieses Format eignet sich besonders gut für geschlossene und halboffene Räume und ist mit allen Variationen das älteste und am meisten verbreitete Format für diesen Zweck im Spielebereich. Die erste Version wurde für das Videospiel *Doom* von id Software entwickelt und ist eigentlich für eine 2½D Welt ausgelegt. *Doom* hat zwar eine 3-dimensionale Welt, es gab jedoch keine übereinander gelegen, getrennte Räume, womit das Sichtbarkeitsproblem auf zwei Dimensionen reduziert wurde. Der Nachfolger *Quake* war der erste echte 3D Ego-Shooter, welche keine Restriktionen in der Lage der Räume in der geometrischen Welt hat. Das Spiel *Half-Life* basiert auf lizenzierter und weiterentwickelter *Quake* Technologie und nutzt mit einigen Veränderungen das selbe Format. Die aktuellste .BSP Version des Spiels *Quake 3 Arena* bleibt dem basierenden Format treu und hat neue Elemente hinzugefügt (z.B. Portale und zur laufzeitberechnete gekrümmte Flächen).

Die Extension .BSP leitet sich von den Term „Binary Space Partitioning-Tree“ ab und beschreibt die Organisation der Flächen in der Datenstruktur. Diese allgemein bekannte Struktur ist z.B. beschrieben in [Fol90], wird heute jedoch nicht mehr wie bei *Doom* für die Tiefensortierung der Flächen benutzt. Für die Sichtbarkeitsberechnung wird der Gesamtraum in konvexe Teilräume (Cluster) unterteilt und in einer Vorverarbeitungsphase wird für jeden Teilraum bestimmt, welche anderen Teilräume von dort aus zu sehen sind. Diese Menge (Potential Visible Set) wird für jeden einzelnen Teilraum in komprimierter Form (RLE) gespeichert und zur Laufzeit entpacket, wenn sich der Spieler in diesem Teilraum befindet. Die endgültige Sichtbarkeitsüberprüfung der einzelnen Flächen findet im Z-Buffer der genutzt Graphikbibliothek statt (z.B. OpenGL oder Direct3D).

Für die Kollisionsberechnung wird ein dynamisches Objekt durch eine achsenparallele Boundig-Box ersetzt und die Räume um die Größe dieser Begrenzung geschrumpft. Diese geschrumpften Räume heißen Hüllen, wobei eine Kollision auftritt, wenn der Schwerpunkt des bewegten Objektes diese Hülle berührt. Die Hüllen werden in einer Vorverarbeitungsphase für vier typischerweise genutzte Bounding-Box Größen (stehenden Spieler, duckenden Spieler, Punktgröße und Monstergröße) vorberechnet und im .BSP gespeichert. Weiterhin werden im .BSP Textur- und Beleuchtungsdaten (*lighmaps*) für Flächen, Positionen von Gegenständen

(*entities*) und Ereignisschalter (*event triggers*) gespeichert. Eine genaue Beschreibung des .BSP Formats findet sich bei [Gui00].

Zum Erstellen von Spielkarten benötigt man einen BSP Editor (Valve Hammer, Q3Radiant, Quark usw.), welcher einem CAD Werkzeug ähnelt (siehe Abbildung 6.2). Der Designer modelliert dabei nach der Constructive Solid Geometry Methode und erstellt die Welt aus einer Menge von Grundkörpern, welche miteinander kombiniert und verändert werden. Der Editor erlaubt es Texturen auf den Flächen zu positionieren oder Lichtquellen und Gegenstände einzufügen. Ein solches Kartenprojekt wird in einer .RMF (Rich Map Format) Datei gespeichert. Um eine Karte in *Half-Life* zu laden, muss sie erst ins .BSP Format kompiliert werden. Dies geschieht in einer Folge von Arbeitsschritten, wobei von mehreren Werkzeugen die Karte Schritt für Schritt modifiziert wird.

Zuerst exportiert der Editor die CSG Objekte, welche dann durch das CSG Programm weiter in simple, konvexe CSG Objekte zerlegt werden. Aus diesen Objekten werden vom BSP Werkzeug die Flächen und teilenden Halbräume und ihre Baumstruktur berechnet. Eine Einteilung in konvexe Cluster und die Berechnung deren Sichtbarkeiten erfolgt durch das VIS Programm. Anhand der eingefügten Lichtquellen berechnet RAD dann für jede Fläche die Beleuchtungstextur. Danach kann die Karte im .BSP von *Half-Life* eingelesen und benutzt werden.

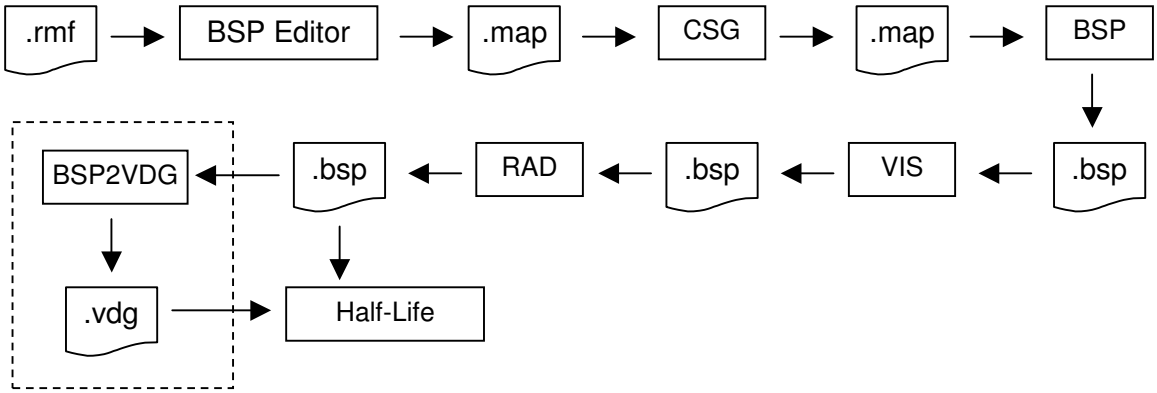


Abbildung 6.3: Kompilierung einer BSP Karte

Alle diese Vorverarbeitungsschritte sind durch einzelne Konsolenapplikationen realisiert, welche über Kommandozeilen-Optionen gesteuert werden. Die Eingabedaten werden als Datei erwartet und ihr Ergebnis wird wieder als Datei abspeichert. Folglich wird der VDG nach gleichem Schema durch das Programm BSP2VDG berechnet und in den Kompilierungsprozess eingebunden. Jedoch können die Daten nicht mit in die .BSP Datei kompiliert werden, da dies eine Veränderung des .BSP Formates bedeuten würde. Das Einlesen und Verarbeiten von .BSP Karten gehört aber zur Kerntechnologie der *Half-Life* Engine und lässt sich durch den SDK Quellcode nicht entsprechend anpassen. Folglich muss der Voronoi Graph in einer separaten Datei gespeichert werden, welche ein proprietäres Format und die Endung .VDG hat.

6.1.2 Rasterung der Grenzflächen

Bei der Rasterung der Begrenzflächen der Welt ist die Bestimmung des maximalen Abstandes m zweier Rasterpunkte einer Fläche von entscheidender Bedeutung (siehe Kapitel 3.5 und 3.8). Durch die Fehlerkorrektur der Freiheiten an den Voronoi Knoten müssen alle Knoten entfernt werden, deren Freiheit * 2 < m beträgt. Folglich ist die Rasterung möglichst klein zu wählen, um möglichst wenig durch den VDG beschriebenen Freiraum zu verlieren. Andererseits steigt

die Anzahl der Rasterpunkte quadratisch je kleiner m wird. Um eine zufriedenstellende Größe für m zu finden, müssen die Ausdehnung der Welt und die der Spieler betrachtet werden.

Räume und Objekte in einer *Half-Life* Karte können in einem Kubus mit den maximalen Koordinaten (-4096, -4096, -4096) und (4096, 4096, 4096) liegen, wobei Eckpunkte von Raumflächen auf ganze Beträge beschränkt sind. Ein stehender *Half-Life* Spieler hat eine Ausdehnung von 32×32×72 Einheiten, in geduckter Haltung 32×32×36. Das bedeutet, dass eine Öffnung in der Welt mindesten 32×32 Einheiten groß seien muss, um von einem Spieler passierbar zu sein[6]. Da die Kamera den Spieler auf allen Wegen folgen können muss, dürfen Voronoi Knoten mit einem Freiheitsradius größer 16 nicht durch die Fehlerkorrektur entfernt werden. Daraus folgt eine maximale Größe von 16*2 = 32 für den maximalen Rasterpunktabstand m. Der Wert für m kann kleiner als 32 gewählt werden um Verluste des Freiraums durch die Fehlerkorrektur zu verringern, dies ist nur durch die gewünschte Gesamtlaufzeit des Programms BSP2VDG und den verfügbaren Hauptspeicher beschränkt. Beim benutzten Testsystem hat sich m = 24 für die meisten Karten als geeignet erwiesen, für Karten mit großen Außenwelten musste m = 28 gewählt werden. Im BSP2VDG Quellcode wird m durch die globale Variable `g_SampleDensity` beschrieben.

Die Rasterung der Flächen folgt der rekursiven Unterteilung aus Kapitel 3.5 und wird in Funktion `AddFaceToSamplePoints` realisiert. Die Rasterpunkte sind in einem Octree [Berg00] organisiert, da dieselben Punkte auf Kanten von benachbarten Flächen mehrfach erstellt werden und so Duplikate schnell zu eliminieren sind. Für die Distanzmessungen der Rasterpunkte werden jeweils nur die Quadrate der Entfernungen verglichen, um teure `sqrt()` Funktionsaufrufe zu sparen.

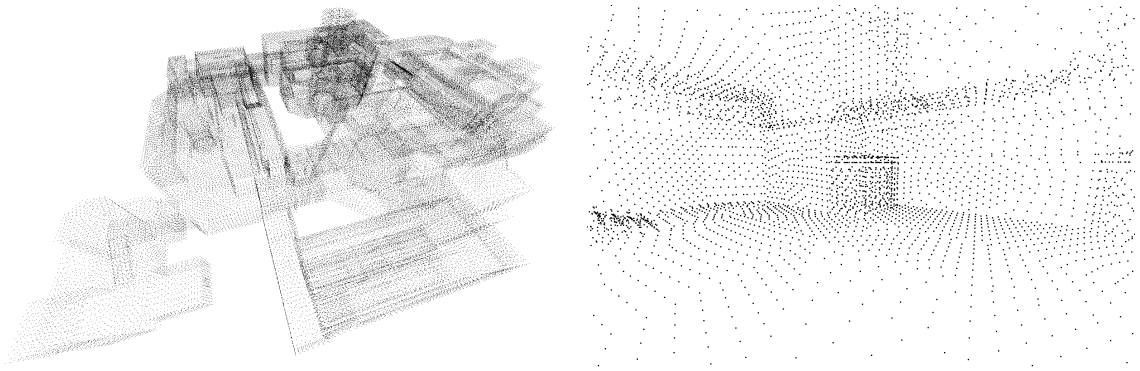


Abbildung 6.4: Rasterpunkte der Karte datacore von aussen (l.), von innen (r.)

Es werden nicht alle Flächen der Welt gerastert, da *Half-Life* viele unterschiedliche Flächentypen kennt. Neben den soliden Flächen für normale Hindernisse gibt es noch halb-transparente Flächen für Glas und Wasseroberflächen oder unsichtbare Hilfsflächen, welche zur Einschränkung der Bewegungsfreiheit oder für Event-Trigger benutzt werden. Transparente oder unsichtbare Flächen werden nicht gerastert mit einer Ausnahme. Karten in *Half-Life* könne geschlossen oder halb-offen sein, wobei der Innenraum von geschlossene Karten komplett durch solide Flächen begrenzt wird, z.B. eine Tiefgarage. Bei halb-offenen Karten ist die Welt nach Oben oder zur Seite nicht durch Wände begrenzt und eine perspektivische Skybox wird um die gesamte Szene gezeichnet [Gre86]. Die Skybox ist nicht Teil der .BSP Darstellung der Welt, welche jedoch fordert, dass alle Räume von soliden Objekten umgeben seien müssen. Deshalb werden unsichtbare Hilfsflächen eingefügt um die Anforderungen der .BSP Struktur wieder zu erfüllen.

[6] In der Praxis sind es 40×40 Einheiten, da es sonst die Bewegung sehr schwer fällt

Vergisst der Designer den Raum mit diesen Hilfsflächen zu schließen, kommt es zu Fehldarstellungen (leaks). Die Hilfsflächen werden besonders gekennzeichnet (`SURF_DRAWSKY`) und liegen außerhalb der vom Spieler erreichbaren Gegenden, da er sonst gegen unsichtbare Wände stoßen würde. Diese Flächen werden beim Zeichnen ausgelassen, so dass die dahinter liegende Skybox erscheint.

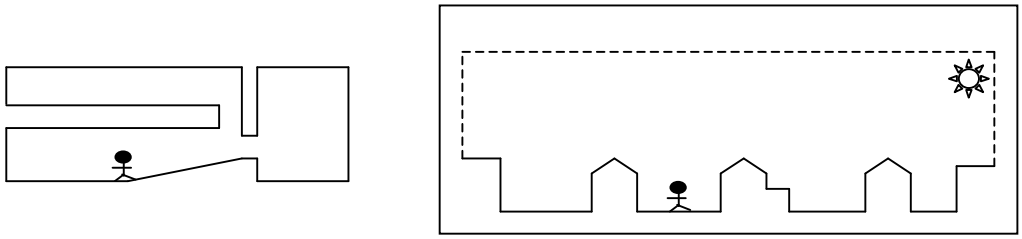


Abbildung 6.5: Geschlossener BSP (l.), halb-offener BSP mit Skybox (r.)

Diese unsichtbaren Hilfsflächen sind in halb-offenen Karten z.T. sehr ausgedehnt und erzeugen bei der Rasterung sehr viele Punkte. Sie können jedoch nicht ignoriert werden, da auch für die Erstellung des VDG gefordert wird, dass der Freiraum $\boldsymbol{C}_{\text{free}}$ komplett vom Hindernisraum $\boldsymbol{CB}$ eingeschlossen wird. Jedoch muss bei diesen Flächen nicht der maximale Mindestabstand der Rasterpunkte eingehalten werden, da sie nicht Freiraum von echten Hindernissen trennen. Voronoi Knoten, deren Freiheitsradius durch Löcher in den Rasterungen diese Hilfsflächen ragen, erzeugen an diesen Stellen also keinen Fehler. In der Implementation werden deshalb für Flächen, welche mit `SURF_DRAWSKY` markiert sind, der doppelte Mindestabstand m benutzt und so die Anzahl der Rasterpunkte dieser Flächen auf einviertel reduziert.

6.1.3 Berechnung des Voronoi Graphen

Für die Berechnung des Voronoi Graphen aus den Rasterpunkten wird die frei verfügbare Qhull Bibliothek verwendet [Qhull02]. Qhull wird seit 1996 an der University of Minnesota entwickelt und nun als Open-Source Projekt weitergeführt, die aktuelle Version ist von August 2002. Ursprünglich wurde Qhull entworfen um konvexe Hüllen für Punkte in $\mathbb{R}^n$ zu berechnen, später wurden Optionen hinzugefügt um auch Delaunay Triangulationen und Voronoi Graphen zu berechnen (über Projektion auf einen Paraboloid in $\mathbb{R}^{n+1}$) . Qhull als Konsolenapplikation kann über Startparameter gesteuert werden und erwartet Eingaben als formatierte Textdateien. Ausgaben werden ebenfalls als Textdatei in verschiedenen Formaten angeboten, z.B. für Geomview. Alle gängigen Betriebssysteme Windows, Linux, Unix, Macintosh und DOS werden unterstützt.

Der Qhull Quellcode ist in C geschrieben und eigentlich keine Software-Bibliothek mit fest definierten Schnittstellen. Das Design ist entsprechend funktional und stark mit globalen Variabeln und Compiler-Makros durchsetzt. Durch die gute Dokumentation fällt es jedoch leicht, die entsprechenden Codefragmente in eigene Projekte einzubinden und die Ein- und Ausgabe der Textdateien durch eigene Routinen zu ersetzen. Der Ablauf der Qhull Funktionsaufrufe in der BSP2VDG `main` Funktion ist simpel, Qhull Funktionen sind durch ein `qh_` Präfix im Namen gekennzeichnet:

```
qh_init_A (stdin, stdout, stderr, NULL, NULL );
qh DELAUNAY= True;
qh VORONOI= True;
points= qh_readpoints (&numpoints, &dim, &ismalloc);
qh_init_B (points, numpoints, dim, ismalloc);
qh_qhull();
qh_check_output();
qh_produce_output();
qh_freeqhull( False );
```

Abbildung 6.6: Qhull Aufrufe in BSP2VDG

Zuerst werden Grundeinstellungen mit `qh_init_A` gesetzt und die Optionen für Delaunay und Voronoi gesetzt, wie es auch über Kommandozeilenparameter geschehen würde. Danach werden die Punkte der Eingabemenge durch `qh_readpoints` eingelesen und in der Menge `points` gespeichert. Diese Funktion ist so überschrieben worden, dass anstatt Punkte aus einer Eingabedatei, die Rasterpunkte der .BSP Welt gelesen werden. Diese Punkte werden in `qh_readpoints` in den Raum $\mathbb{R}^4$ projiziert und die Eingabedimension `dim` nimmt den Wert 4 an. In `qh_init_B` wird benötigter Speicher reserviert und entsprechend der gewünschten Dimension 4 bestimmte Strukturen initialisiert, welche bei der Berechnung der konvexen Hülle benötigt werden. Die eigentliche Berechnung geschieht durch den Aufruf von `qh_qhull`.

Die konvexe Hülle wird anschließend durch `qh_check_output` auf Konsistenz und Korrektheit geprüft. Die Ausgabe der Hülle erfolgt in der Funktion `qh_produce_output`, wobei hier die entscheidende Rücktransformation von der 4-dimensionalen Hülle zum 3-dimensionalen Voronoi Graphen geschieht. Dies muss später weiter erläutert werden, da hier die Daten in das weiterhin zentrale Objekt `CVoronoiGraph` übertragen und bereits reduziert werden. Danach werden alle Speicherreservierungen von Qhull mittels `qh_freeqhull` wieder freigegeben. Insgesamt herrscht während `qh_produce_output` der größte Speicherbedarf, da sowohl die Eingabepunkte, die Qhull Daten und auch das neue `CVoronoiGraph` Objekt gleichzeitig benötigt werden. Bei sehr großen Karten steigt der Gesamtbedarf auf über 512MB, was zu langsamen Speicherauslagerungen durch das Betriebssystems führt. Da die Laufzeit von `qh_produce_output` generell jedoch sehr kurz ist (wenige Sekunden) ist dieser Nebeneffekt akzeptabel.

6.1.4 Die Klasse CVoronoiGraph

Der Voronoi Graph wird in einem Objekt `g_VDG` der Klasse `CVoronoiGraph` gespeichert. Dieses Objekt kapselt alle Daten und Funktionen des Voronoi Graphen, welche für die Reduktion, das Laden und Speichern und vom Kamera Management benötigt werden. Die Basiselemente des Voronoi Graphen sind die Knoten und Flächen, welche durch die Strukturen `vertex_t` und `face_t` beschrieben werden.

```
typedef struct
{
  int               index;
  vec3_t            position;

  CIndexList        faces;
  CIndexList        neighbours;

  float             error;
  float             clearance;
  float *           distances;
} vertex_t;
```

```
typedef struct
{
  int               index;
  int               vertices[3];
  CIndexList        neighbours;
  float             distance;
} face_t;

typedef float vec3_t[3];
```

Abbildung 6.7: Struktur vertex_t (l.), Struktur face_t (r.)

Ein Voronoi Knoten `vertex_t` hat einen eindeutigen `index` und eine `position`, die Menge `faces` beischreibt alle Voronoi Flächen, welche diesen Knoten als Eckpunkt haben. Durch `neighbours` wird die Menge der benachbarte Voronoi Knoten beschrieben. Der Wert `error` beschreibt das bei der Reduktion erwartete Fehlvolumen, welches durch das Entfernen dieses Knoten entstehen würde. Der Freiraumradius um den Knoten wird mit `clearance` angegeben, der Zeiger `distances` wird für die schnelle Berechnung von Pfadlängen im Graphen benötigt.

Eine Voronoi Fläche `face_t` ist eindeutig indiziert und hat drei `vertices` als Eckpunkte. Benachbarte Voronoi Flächen werden in der Menge `neighbours` gespeichert. Die Variable `distance` hilft bei der schnellen Berechnung von Sichtbarkeiten.

Die wichtigsten Datenelemente der Klasse `CVoronoiGraph` sind eine Liste der Knoten, eine Liste der Flächen und einem kd-Baum [Berg00] ähnliche hierarchische Struktur `BBoxNode`, in welcher die Flächen räumlich organisiert sind. Mit Hilfe dieses kd-Baums ist es möglich für einen beliebigen Punkt schnell jene Voronoi Fläche zu finden, welche diesem Punk am nächsten liegt. Dies ist nötig, falls die Kamera sprunghaft ihre Position wechseln muss. Diese Baumstruktur wird nach der Reduktion einmalig erstellt. Für die Reduktion werden die Knoten in einer nach den Kosten des Fehlervolumens sortierten Liste `m_ErrorList` verwaltet.

```
class CVoronoiGraph
{
      int           m_NumVertices;
      vertex_t *    m_Vertices;
      int           m_NumFaces;
      face_t*       m_Faces;
      PrioList      m_ErrorList;
      BBoxNode      m_BBoxRoot;
      ...
}
```

```
struct BBoxNode
{
      float         middle;
      BBoxNode      *left;
      BBoxNode      *right;
      int           numFaces;
      int           *faces;
}
```

Abbildung 6.8: Klasse CVoronoiGraph (l.), Struktur BBoxNode (r.)

Der Objekt `g_VDG` wird, wie oben erwähnt, durch die Funktion `qh_produce_output` gefüllt. Dafür werden die Eingabepunkte aus $\mathbb{R}^4$ wieder nach $\mathbb{R}^3$ zurück projiziert, indem die vierte Komponente entfernt wird. Die Qhull Ausgabe einer konvexen Hülle in $\mathbb{R}^4$ sind Kanten zwischen den Eingabepunkten auf der konvexen Hülle, welche Facetten (facets) auf dieser Hülle bilden. Wegen der Projektion auf den Paraboloid liegen alle Eingabepunkte auf dieser Hülle. In $\mathbb{R}^3$ zerteilen die Kanten der Hülle den Raum in Tetraeder, welche die Delaunay Bedienung erfüllen. Jeder Facette entspricht einem dieser Tetraeder, aus welchem ein Voronoi Knoten entsteht. Der Knoten ist der Mittelpunkt der Kugel, welche die durch die vier Eckpunkte des Tetraeders

beschrieben wird. Die Funktion `GetCenter` berechnet für jede Facette diesen Mittelpunkt und den Radius der Kugel, welcher die Freiheit dieses Knoten ist.

Für die Berechnung der Voronoi Flächen werden alle Hüllenkanten zwischen zwei Rasterpunkten betarchtet. Senkrecht wird jede dieser Kanten durch eine Voronoi Fläche geschnitten, welche durch Voronoi Knoten begrenzt wird. Dies sind mindestens drei oder mehr Knoten und gehören zu den Tetraedern, welche um diese Kante liegen (sie also enthalten). Qhull ermöglicht eine Suche nach diesen benachbarten Voronoi Knoten mittels der Funktion `qh_printvdiagram2`. Bei der Erstellung der Voronoi Flächen werden allerdings nur die Flächen dem Graphen hinzugefügt, welche komplett im Freiraum der BSP Karte liegen. Dafür wird überprüft, ob die Gerade zwischen allen Knotenpaaren der Fläche durch Freiraum verläuft. Das Objekt der BSP Karte bietet dafür die Funktion `TraceLine`. So werden nur Voronoi Flächen hinzugefügt, welche die innere Medialachse der Karte beschreiben. Hat eine Voronoi Fläche mehr als drei Eckpunkte, wird sie wiederum mittels Delaunay trianguliert. Sind alle möglichen Flächen gefunden, werden Voronoi Knoten, welche keine dieser Flächen beschreiben, wieder entfernt. Als letzter Schritt wird in `qh_produce_output` für jeden Voronoi Knoten ein Fehlbetrag mit `CalcSampleError` berechnet und von dem Freiheitsradius abgezogen, wie in Kapitel 3.8 beschrieben.

6.1.5 Reduktion der Voronoi Knoten

Die Reduktion des Voronoi Graphen ist Kern der BSP2VDG Anwendung und besitzt den größten Laufzeit- und Quellcodeanteil (den Qhull Code nicht mit einbezogen). Die Implementierung folgt dem Verfahren aus Kapitel 4.3. Die benutzten Funktionen sind Methoden der Klasse `CVoronoiGraph` oder globale Funktionen definiert in `mathlib.h`. Es werden hier nur einige der wichtigsten Funktionen und Abläufe kurz beschrieben, für Details ist der Quellcode einzusehen.

Die Funktion `CalcVolumeErrors` berechnet für alle Knoten den Volumenfehler, welcher beim Entfernen entstehen würde. Der Fehler eines einzelnen Knoten `v` wird durch `CalcError(v)` berechnet. Ist dieser Wert gleich -1, bedeutet dies, dass der Knoten nicht entfernt werden kann. Für die Fehlerberechnung eines Knoten v wird mit `SaveBackup` seine Nachbarschaft gespeichert (Knoten, Flächen, Freiheiten), dann der Knoten mit `RemoveNode(v)` möglichst optimal entfernt und der dabei entstandene Volumenverlust gemessen. War ein Entfernen nicht möglich, liefert `RemoveNode(v)` eine -1 als Rückgabewert. Danach wird der Graph wieder mit `RestoreBackup` in seinen vorherigen Zustand versetzt und alle Änderungen am Knoten und seinen Nachbarn rückgängig gemacht. Zuletzt wird der Knoten entsprechend seinem Fehlerwert in `m_ErrorList` einfügt.

In `RemoveNode(v)` werden alle Flächenpaare `f1,f2` versucht mit Aufruf von `FlipFaces` auszutauschen, solange bis `v` keine Flächen mehr besitzt. `FlipFaces` kann Flächen tauschen und auch Flächen entfernen da beide Verfahren sehr ähnlich ablaufen. Es liefert bei erfolgreichem Flächentausch das entstandene Fehlvolumen zurück oder –1, falls ein Tausch nicht möglich ist. Die Funktion `FlipFace` ist der Kern des gesamten Algorithmus und entsprechend umfangreich, da alle Teilfälle, wie in Kapitel 4.5 und 4.6 beschrieben, berücksichtigt werden müssen. Das Bestimmen von t_p und t_q erfolgt über `SolveLES` eine sehr häufig benutzt Funktion zur Lösung von linearen Gleichungssystemen mit drei Unbekannten (Ax = b, x = $((A_{adj})^T * b)/\det(A)$). In sehr selten auftretenden Fällen (<0,1%), überlappen sich beide Flächen und das LGS ist nicht lösbar, dann wird einer der beteiligten Knoten um einen winzigen Betrag durch `JerkPoint` „geschüttelt“, um so weitere Fallunterscheidungen zu vermeiden. `IsSingleEdge` überprüft für eine Kante, ob sie nur durch die zwei Flächen `f1,f2` benutzt wird. Mit `FaceVolume` werden die Volumina der alten und neuen Flächen berechnet, wie in Kapitel 4.4 beschrieben.

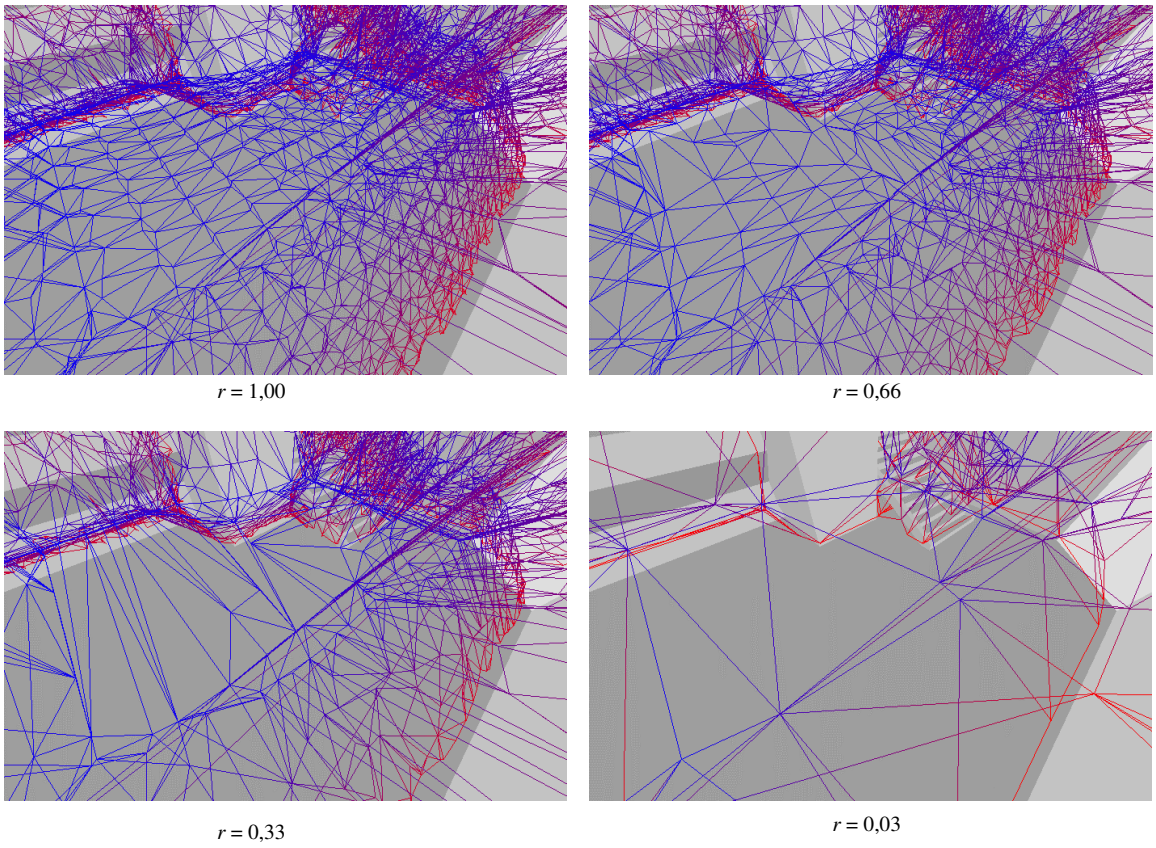


Abbildung 6.9: Reduktion des VDG in 4 Stufen

Falls bei einem Kantentausch die Freiheiten der beteiligten Knoten verringert werden müssen, werden mit `OptimalLinearReduction` die beiden Fehlbeträge `e1` und `e2` berechnet (siehe 4.5.1). Das Problem wird so abstrahiert, dass zwei Punkte $(0,h_1)$ und $(1,h_2)$ gegeben werden sowie ein Zielpunkt (f,h), wobei $f \in [0,1]$ und dieser Zielpunkt unterhalb der Geraden g zwischen $(0,h_1)$ und $(0,h_2)$ liegt. Die Höhen h_1, h_2 und h entsprechen den Freiheiten der beteiligten Knoten. Nun ist eine Gerade e gesucht, welche durch (f,h) verläuft und unterhalb der Geraden g und über 0 verläuft. Diese Lösung ist nicht eindeutig und wird dermaßen optimiert, dass die Flächen unterhalb der neuen Gerade zwischen [0,1] maximal ist. Dadurch wird erreicht, dass möglichst viel Volumen erhalten bleibt. Wird die Freiheit eines Knotens verringert, wirkt sich

dieses auf das beschriebene Volumen alle benachbarter Flächen aus und muss im Gesamtvolumenverlust berücksichtigt werden (mit `CalcNodeReduceDistanceError`).

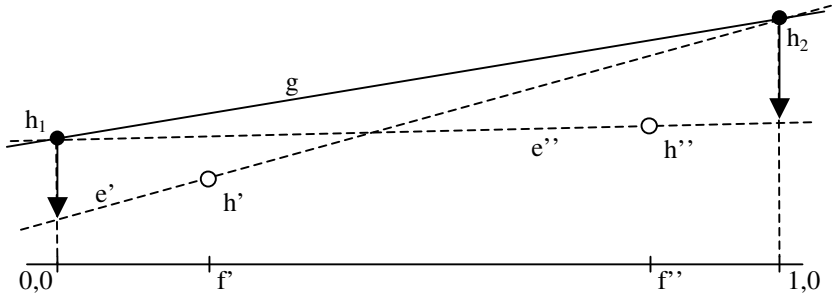


Abbildung 6.10: Optimierung der Geraden g durch e' oder e"

Mit `RemoveNextNode` wird bei der Reduktion jeweils ein Voronoi Knoten entfernt. Dafür wird der erste Knoten `v = m_ErrorList.RemoveFirst` aus der Prioritätsliste geholt. In einer Liste werden die aktuellen Nachbarknoten von `v` mit `GetVertexNeighbours` gespeichert. Nachdem der Knoten `v` entfernt wurde, werden die Fehlerwerte der alten Nachbarknoten mit `CalcError` neu berechnet und ihre Position in der Prioritätsliste aktualisiert.

6.1.6 Speicherung als .VDG Datei

Das Speichern der reduzierten VDG Struktur ist eigentlich eine direkte Speicherkopie in einer Datei. Dies hat den Vorteil das der Ladevorgang kaum Zeit benötigt. Bevor der Graph jedoch mit `g_VDG.Save` gespeichert wird, werden durch `BuildBBoxTree` die Voronoi Flächen in einer kd-Baum ähnlichen Struktur `m_BBoxRoot` einsortiert. Die Blätter dieses Binärbaums sind Teilräume von minimal 256×256×256 Einheiten Ausdehnung. In jedem Blatt sind die Indizes aller Voronoi Flächen gespeichert, deren beschriebenes Freiraumvolumen den Teilraum des Blattes schneidet. Die verwendete Struktur `BBoxNode` ist dieselbe wie schon bei der Rasterung der Eingabeflächen. Der kd-Baum wird durch rekursiven Aufruf von `WriteBBoxRecrusive` in die Datei geschrieben. Die .VDG Datei hat bis auf die Extension den gleichen Namen wie die .BSP Quelldatei und wird von *Half-Life* Client im Unterverzeichnis `\valve\maps\graphs` erwartet.

Blockname	Bytes	Anzahl	Inhalt
Header	4	1	„VDG1" Identifikations Marker
NumVertices	4	1	Anzahl Voronoi Knoten = m
NumFaces	4	1	Anzahl Voronoi Flächen = n
Vertices	17	M	Position, Freiheit und Anzahl Nachbarn eins Knoten
Faces	6	N	Indizes der 3 Knoten pro Flächen
BBoxMin	12	1	(x,y,z) Untergrenze des kd-Baum Raumes
BBoxMax	12	1	(x,y,z) Obergrenze des kd-Baum Raumes
BBoxTree	variabel	variabel	Rekursive Struktur des kd-Baums (Knoten, Blätter)

Abbildung 6.11 - VDG Datei Struktur

6.2 Kamera-Management in HL

Durch die Implementation des Kamera-Management in *Half-Life* wird die neue Kameranavigation getestet und nur Veränderungen an dem client-seitigem Code vorgenommen. Die Kameraposition und Einstellung eines Spielers im Zuschauermodus sind beliebig wählbar und nicht von Bedeutung für das restliche Spiel an sich. Der Server weiß nur welcher Spieler von einem Zuschauer verfolgt wird, aber nicht die genau Position und Blickwinkel des Zuschauers. Zwar wird ein Direktor Modul implementiert, jedoch ist es nur ein Platzhalter für künftige Erweiterungen. Es analysiert nicht den Spielverlauf und bestimmt keine eigenen Kameraeinstellungen oder Zielobjekte. Daher kann die Client-Server Architektur hier vernachlässigt werden und ein einfacherer Aufbau der Game-Engine angenommen werden.

Die *Half-Life* Engine arbeitet in festen Zyklen (oder Frames), welche in der Hauptschleife des Prozesses durchlaufen werden. Die Reihenfolge der Hauptfunktionen in einem Zyklus ist fest und an bestimmten Stellen werden Funktionen der Client-Logik über die Schnittstelle `cldll_func_t` aufgerufen. Ein Zyklus beginnt mit dem Auslesen der Eingabegeräte (Maus, Joystick, Tastatur), deren Daten durch die Client-Logik interpretiert und in entsprechende Ereignisse umgesetzt werden. Mit jedem Zyklus ist die Zeit etwas vorangeschritten und die Spielwelt wird entsprechend den Spielregeln, der Spielphysik und den neuen Ereignissen verändert. Nachdem die Welt vollständig aktualisiert wurde, wird über die Schnittstellenfunktion `V_CalcRefdef` der Client-Logik die aktuelle Blickrichtung und Position ermittelt. In diesen Funktionsaufruf wird sich das neue Kamera-Management einklinken und diese Berechnungen im Zuschauermodus[7] übernehmen. Danach kann mit diesen Daten eine Liste von sichtbaren (und hörbaren) Objekten erstellt werden (siehe BSP in Kapitel 6.1.1), welche an die Graphik- und Tonausgabe zur Darstellung weitergegeben werden. Dann hat sich der Kreis „Eingabe-Verarbeitung-Ausgabe“ geschlossen und ein neuer Zyklus wird begonnen. Dies alles geschieht mit einer für interaktive, virtuelle Systeme typischen Frequenz von 25 bis 100 Hz, je nach Rechenleistung und Komplexität der Spielwelt.

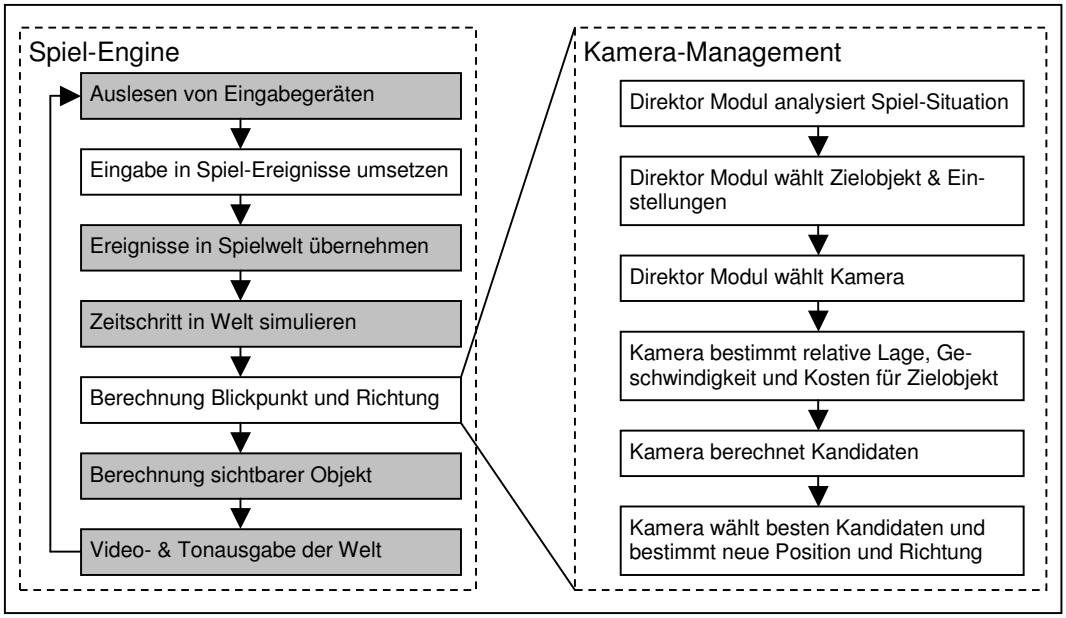


Abbildung 6.12 - Architektur Engine & Kamera-Management

[7] Der Zuschauer-Modus wird in der HL Konsole mittels `v_diretcormode 1` eingeschaltet

Das Kamera-Management besteht aus zwei Klassen: `CDirector` und `CCamera`, beide Klassendefinitionen sind vom Rest des Quellcodes durch den Namensraum `namespace Voronoi` getrennt um Namens- und Definitionsüberschneidungen mit bereits existierenden Funktionen zu verhindern. Von der Klasse `CDirector` existiert eine einzige, globale Instanz `v_Director`, welche die Schnittstelle zwischen der Funktion `V_CalcRefdef` und dem Kamera-Management bildet. Das Objekt `v_Director` kann neue Kameras als Instanzen der Klasse `CCamera` erstellen, verwaltet sie in der Liste `m_Cameras` und bestimmt welche die momentan aktive Kamera ist. Es können beliebig viele Kameras gleichzeitig mehrere Zielobjekte mit unterschiedlichen Einstellungen verfolgen. Jeder Kamera können individuelle Parameter (Winkel, Distanz, usw) mit `CCamera::SetConstraints` übergeben werden. Das `v_Director` Objekt lädt den Voronoi Graphen der aktuellen Spielkarte aus der entsprechenden .VDG Datei in das Objekt `m_VDG` der Klasse `CVoronoiGraph`. Diese Klasse ist dieselbe, wie sie auch von BSP2VDG verwendet wird und das Objekt `m_VDG` wird von allen Kameras gemeinsam benutzt. Eine Kamera besitzt eine Position `m_Position` und eine Bewegungsrichtung `m_Motion` als Vektor, dessen Länge gleich der momentanen Geschwindigkeit ist. Die jeweils aktuelle Voronoi Fläche, in deren Freiraumvolumen sich die Kamera gerade befindet, wird über `m_Face` referenziert.

```
class  CDirector
{
  void SetGlobalConstraints(angle, dis
         tance, visibility, mobility);

  bool        ActivateCamera(camID );
  CCamera *   CreateCamera(camID );
  CCamera *   GetCamera( camID );
  void        Think();

  bool        GetViewOrigin(origin);
  bool        GetViewAngles(angles);

  ObjectList      m_Cameras;
  CCamera *       m_CurrentCamera
  CVoronoiGraph   m_VDG;
  ...
}
```

```
class CCamera
{
  void  SetConstraints(angle, distance,
          visibility, mobility);
  void  Think();
  void  DoCut();
  bool  AnalyseScene();
  void  ApplyConstrains();
  void  MoveTo(pos, angles);

  void  GetViewOrigin(origin);
  void  GetViewAngle(angle);

  vec3_t m_Position ;
  vec3_t m_Motion;
  face_t * m_Face;
  ...
}
```

Abbildung 6.13 - Klasse CDirector (l.), Klasse CCamera(r.)

Ruft die Game-Engine die Funktion `V_CalcRefdef` auf und die Client-Logik befindet sich im Zuschauer Modus, wird der Aufruf an `V_CalcDirectorRefdef` weitergeleitet. Hier wird zuerst die Funktion `v_Director.Think` ausgeführt, welche pro Zyklus genau einmal aufgerufen werden muss. In `v_Director.Think` wird überprüft, ob für die aktuelle Spielkarte die richtigen VDG Daten geladen sind. Falls dies nicht der Fall ist, müssen die entsprechende .VDG Datei mit `CDirector::ChangeLevel` neu geladen und initialisiert werden. Dann wird vom Direktor für jede Kamera in `m_Cameras` die Funktion `CCamera::Think` aufgerufen und jede Kamera berechnet einen neuen Schritt in diesem Zyklus. Nachdem alle Kameras aktualisiert worden sind, wird mit `v_Director.GetViewOrigin` und `v_Director.GetViewAngles` die Position und Blickrichtung der aktiven Kamera abgefragt und an die Game-Engine zurückgegeben.

```
::V_CalcRefdef( pparams )
  ::V_CalcDirectorRefdef ( pparams )
    CDirector::Think( pparams )
      CCamera::Think()
        ThinkChaseMode()
           AnalyseScene()
           ApplyConstrains()

    CDirector::GetViewOrigin( pparams->vieworg )
      CCamera::GetViewOrigin( origin )

    CDirector::GetViewAngles( pparams->viewangles )
      CCamera::GetViewAngle( angles )
```

```
typedef struct candidate_s
{
        vec3_t   pos;
        face_t * face;
        float    costs;
        float    visibility;
        float    distance;
        float    angle;
        float    freedom;
        float    inertia;
} candidate_t;
```

Abbildung 6.14 Callstack von `V_CalcRefdef (l.)`, `Struktur candidate_t (r.)`

Die eigentliche Arbeit geschieht in Funktion `CCamera::Think`. Hier wird zuerst entschieden, ob ein Kameraschnitt nötig ist, z.B. wenn das Zielobjekt lange nicht mehr sichtbar war. Bei einem Schnitt (`CCamera::DoCut`) wird die Kamera an die Position des Zielobjektes gesetzt. Die neue Voronoi Fläche für diese Position muss in diesem Fall mit `CVoronoiGraph::FindNearestFace` über den kb-Baum des VDG gesucht werden. Dann wird in `CCamera::AnalyseScene` die momentane Position und Richtung des Zielobjektes ermittelt und die VDG Struktur für die Berechnungen der Voronoi Distanz über den A* Algorithmus vorbereitet, da sich alle folgenden Voronoi Distanzen auf die Position dieses Zielobjektes beziehen. Die Berechnung und Bewertung der möglichen Kandidaten und die entgültige Bestimmung der neuen Kamerakonfiguration findet in `CCamera::ApplyConstrains` statt.

Die Kandidaten sind vom Typ `candidate_t` und beinhalten die Position, die nächste Voronoi Fläche und die Kosten für diesen Punkt. Die einzelnen Kandidaten werden entsprechend Kapitel 5.3 konstruiert und mit der Funktion `CCamera::CalcCostForPoint` bewertet. Die Voronoi Distanz eines Kandidaten zum Zielobjekt wird über die `CVoronoiGraph::GetTargetDistance` berechnet, wobei die Menge *Reachable*(p,f) durch `CVoronoiGraph::FindReachableNeighbours` realisiert wird. Die Sichtbarkeit wird mit der Funktion `CVoronoiGraph::Visible entsprechend` der rekursiven Definition *Visible*(**s**, **e**, **f**) aus Kapitel 3.9 berechnet. Der Schnittpunk s' = *Trace*(s, e, f) wird mit `CVoronoiGraph::Trace` ermittelt. Bereits bekannt Ergebnisse rekursive Teilwege (*Visible*(**s'**, **e**, **f**)) werden zwischengespeichert um doppelte Berechnungen durch die Rekursion zu vermeiden. Die Freiheit um einen Punkt wird mit `CVoronoiGraph::InterpolatePointAndDistance` berechnet, wie in Kapitel 3.7 beschrieben.

Wenn der best (günstigste) Kandidat ermittelt wurde, wird eine passende Schrittweite (siehe Kapitel 5.3) berechnet und die Kamera um diesen Betrag mit `CCamera::MoveTo` bewegt, wobei Geschwindigkeit und Bewegungsrichtung der Kamera aktualisiert werden. Die Voronoi Fläche `m_Face` muss entsprechend der neuen Position durch `CVoronoiGraph::GetClosestFace` aktualisiert werden. Dafür werden alle benachbarten Flächen untersucht, ob sie für die neuen Position ein größeres Freiraumvolumen beschreiben.

6.3 Hilfswerkzeug VDGView

Um während der Entwicklung der verschiedenen Funktionen der Klasse `CVoronoiGraph` zu testen, wurde das Werkzeug VDGView zur Visualisierung des Voronoi Graphen in seiner BSP Karte entwickelt. Das Programm VDGView startet als eine Win32 Konsolenapplikation und öffnet dann eine GUI mit einem OpenGL Fenster. Das Programm kann Eingabedateien vom Typ .BSP, .VDG und .TXT über das „File“ Menü öffnen, alle weiteren Einstellungen werden über Konsolenbefehle (z.B. `load <Dateiname>`) vorgenommen.

Die Bewegung innerhalb des Raumes findet über die Tasten A, D, W, S (links, rechts, vorwärts, rückwärts) statt, die Blickrichtung kann mit der Maus und gedrückter linker Maustaste verändert werden. Mit dem Befehl `gopos <x> <y> <z>` kann zu beliebigen Orten im Raum gesprungen werden, derselbe Befehl ohne Parameter zeigt die aktuelle Position als (x, y, z) Koordinate. Wird die Variable `noclip` auf Null gesetzt, kann nur innerhalb des vom VDG beschriebenen Freiraumvolumen navigiert werden. Diese Funktion eignet sich sehr gut zum Testen von reduzierten Voronoi Graphen und allen Funktionen, wie sie vom Kamera-Management benutzt werden.

Die Wände der BSP Karte werden als Flächen in Graustufen dargestellt, die Voronoi Flächen des VDGs können entweder als Drahtgittermodel (`rendermode 1`) oder als gefüllte Dreiecke (`rendermode 3`) dargestellt werden. In beiden Fällen signalisiert die Farbe eines Punkte die maximale Freiheit um diesen Punkt: blau bedeutet große Freiheit, rot hingegen eine kleine Freiheit. Falls BSP2VDG mit der `-savesamples` Option gestartet wird, werden die Rasterpunkte der Karte in einer .TXT Datei im Textformat gespeichert. Diese Textdatei kann von VDGView eingelesen werden und die Rasterpunkte werden dann als schwarze Punkte dargestellt.

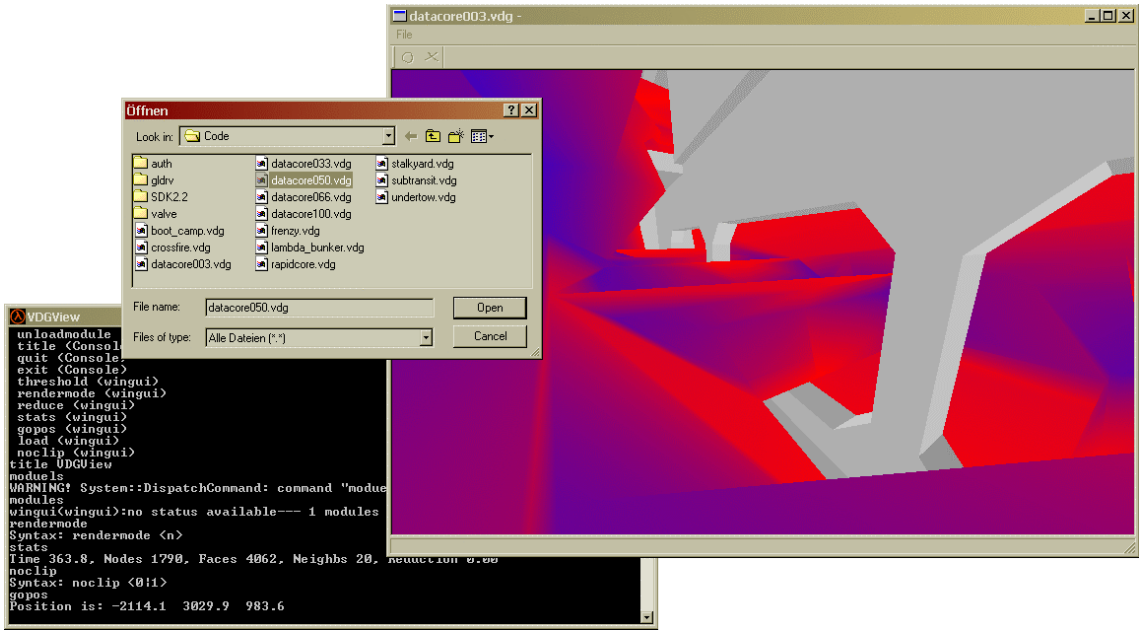


Abbildung 6.15: VDGView im Rendermodus 3

Die aktuelle Anzahl von Konten und Flächen des VDG kann mit dem Befehl `stats` angezeigt werden. Wurde beim Ausführen von BSP2VDG keine Reduktion erzwungen (`-rl 1.0`), kann der VDG innerhalb von VDGView bei ständiger graphischer Ausgabe reduziert werden. Dies ist zwar wesentlich langsamer als mit der Konsolenapplikation BSP2VDG, jedoch lassen sich

Fehlverhalten des Reduktionsalgorithmus so wesentlich schneller erkennen. Um eine bestimmte Anzahl n von Knoten zu entfernen, muss der Befehl `reduce n` aufgerufen werden. Um zu sehen, wie die einzelnen Konten für die Reduktionsreihenfolge bewertet werden, gibt es den speziellen Drahtgittermodus `rendermode 2`. Knoten, welche nicht entfernt werden können (`CalcError == -1`), werden rot gezeichnet, alle anderen Knoten haben einen Farbwert zwischen grün und schwarz. Je ‚grüner' ein Knoten ist, desto weniger Fehlvolumen würde durch seine Entfernung entstehen. Wichtig ist, dass nachdem eine Reduktion mit VDGView vorgenommen worden ist, arbeitet die `noclip 0` Funktion nicht mehr korrekt, da der kd-Baum nicht mehr stimmt. Der reduziert Voronoi Graph kann mit VDGView nicht abgespeichert werden.

7 Ergebnisse

Bei der Auswertung der Ergebnisse wurden beide Implementation betrachtet. Zum einen die Erstellung und Reduktion des Voronoi Graphen durch die Anwendung BSP2VDG, zum anderen die Nutzung des Graphen beim Kamera Management im *Half-Life* Client. Für beide Implementationen ist gültig, dass sie stabil laufen und mit jeder beliebigen *Half-Life* Karte verwendet werden können. Testläufe wurden mit allen 11 mitgelieferten Mehrspieler-Karten erfolgreich durchgeführt.

7.1 Bewertung von BSP2VDG

Das Testen der Anwendung BSP2VDG begann schon während der Entwicklung und führte nach und nach zu immer schnelleren und weniger speicherintensiven Versionen. Die gewonnene Rechenzeit und Speicherressourcen wurden meist wieder zur Erhöhung der Qualität, also kleineren Rasterdichten, verwendet. Der aktueller Standardwerte 24 für die Rasterdichte ist so gewählt, dass für normal große Karten die Reduktion mit den 384MB Hauptspeicher des Testsystems ohne Auslagerungen durchlief. Die Gesamtzahl der Rasterpunkte durfte 250000 nicht überschreiten, da sonst der Qhull Algorithmus mit Speicherauslagerungen beginnt und praktisch nicht mehr weiter rechnet. Deshalb musste für sehr große, halb-offene Karten (*boot_camp*, *bounce*) eine größere Rasterdichte manuell bestimmt werden [8].

Karte	Flächen	Raster-dichte	Raster-punkte	innere Knoten	innere Flächen	Zeit Qhull	red. Knoten	red. Flächen	Gesamt-zeit	VDG Grösse
snark_pit	2883	24	63855	185218	264405	55 s	4475	10862	00:19h	260
stalkyard	2551	24	66146	194002	308705	324 s	3931	8671	00:27h	429
rapidcore	4444	24	79282	247034	335243	59 s	4842	11404	00:24h	400
frenzy	2089	24	88780	268231	156808	2039 s	5725	11416	01:02h	527
datacore	4722	24	88867	271038	365963	55 s	5421	12519	00:26h	393
lambda_bunker	4712	24	123609	344390	504466	106 s	6931	17480	00:42h	472
undertow	3233	24	155110	400408	628650	417 s	12446	37874	00:56h	2115
subtransit	5690	24	165275	496520	734902	168 s	15962	44833	01:04h	2001
crossfire	4385	24	180015	502073	844358	444 s	24479	79504	01:22h	3345
bounce	4207	26	240674	617050	1260003	668 s	35375	126365	01:36h	6752
boot_camp	9926	32	248363	758664	1276013	884 s	16128	38129	01:50h	5455

Die Laufzeit und das Ergebnis von Qhull ist mitunter sehr interessant. Das Verhältnis zwischen Rasterpunkten und Voronoi Knoten ist ungefähr 1:6, demnach kommen auf jeden Punkt zirka sechs Tetraeder (siehe Kapitel 4.1). Die Anzahl der inneren Voronoi Knoten (im Inneren des BSP) ist ungefähr die Hälft aller Voronoi Knoten. Die Laufzeit von Qhull zeigt quadratisches Verhalten, was normal für die Berechnung konvexer Hüllen in $\mathbb{R}^4$ ist. In [Rou98] wird für diese Berechnung ein Laufzeitverhalten von $O(n \log n + n^{d/2})$ oder besser mit $O(ndF)$ angegeben, mit n Eingabepunkten in $\mathbb{R}^d$, wobei F die Anzahl der Facetten auf der Hülle ist. In unserem Fall ist die Anzahl der Facetten gleich der Anzahl der Voronoi Knoten und damit linear zu n. Jedoch gibt es bei den Laufzeiten von Qhull ein paar Ausreißer, im besonderen die Karte *frenzy*. Obwohl *frenzy* eine fast gleiche Anzahl von Eingabepunkten und die Ausgabe eine ähnliche An-

[8] siehe Anhang BSP2VDG Startparameter

zahl von Voronoi Knoten wie z.B. von *datacore* hat, dauert die Qhull Rechenzeit 37 mal länger. Die anschließende Reduktion dauert wieder vergleichbar lang und liefert ähnlich gute Reduktionsraten. Die Karte *frenzy* ist insofern besonders, da sie von allen Karten die wenigsten Flächen besitzt, welche jedoch einzeln sehr groß sind. Scheinbar zeigt Qhull ein Worst-Case Verhalten, falls viele der Eingabepunkte in $\mathbb{R}^3$ in einer Ebene liegen, bevor sie auf den Paraboloid in $\mathbb{R}^4$ projiziert werden. Dieses Phänomen bleibt hier unerklärt, da wiederum die Karte *snark_pit* mit ähnlichen Flächengrößen dieses Verhalten nicht zeigt.

Bei der eigentlichen Reduktion des Voronoi Graphen zeigt sich ein erwartetes $O(n \log n)$ Laufzeitverhalt. Im Grunde wird bei jedem Schritt ein Knoten entfernt, dessen Aufwand mit der Anzahl m der benachbarten Knoten quadratisch steigt, da (m*(m-1))/2 mögliche Kombinationen beim Kantentausch auftreten können. Die durchschnittliche Anzahl der Nachbarn pro Knoten steigt während der Reduktion jedoch nur wenig und wird daher als konstant angenommen (die maximale Nachbarnzahl sinkt sogar). Das Entfernen eines Knoten und damit auch die Fehlereinschätzung eines Knoten sind damit zwar sehr teure Operationen, jedoch im Ganzen nicht von der Gesamtzahl der Voronoi Knoten abhängig. Die Verwaltung der Prioritätsliste der Knotenkosten allerdings schon, woraus sich das $\log n$ ableitet. Die Reduktionsrate der Voronoi Knoten ist mit 0,020 bis 0,025 relativ konstant und liefert hinreichend gute Reduktionen. Trotzdem können die Größen der resultierenden .VDG Datei mit ähnlichen Knoten und Flächen Zahlen schwanken. Dies hängt mit dem zusätzlichem kd-Baum zusammen. Je mehr sich die Flächen über einen Raum verteilen, desto größer ist die Baumstruktur. Liegen die Voronoi Flächen nahe beieinander, werden entsprechend weniger Blätter und Pfade im Baum benötigt.

7.2 Bewertung des Kamera Managements

In Kapitel 3.1 wurden die gewünschten Laufzeiteigenschaften an das Kamera Management aufgestellt. Die wichtigsten Forderungen waren ein kleiner Anteil (<5%) an der Gesamtlaufzeit und eine geringe Varianz der Rechnungszeiten pro Zyklus. Um den Laufzeitanteil des Kamera-Managements zu messen, wurden vor und hinter dem Aufruf von `CDirector::Think` spezielle Funktionen zur Zeitmessung eingefügt. Da diese Zeitmessung im Bereich von Millisekunden genau seien muss, wurde die besonders genaue Win32 Funktion `QueryPerformanceCounter` benutzt, welche die Anzahl der Betriebsystemzyklen seit dem Systemstart angibt. Für die Messung wurde ein Spieler über knapp zwei Minuten mittels des neuen Kamera Managements in der größten Spielkarte *boot_camp* verfolgt, wobei der Spieler quer durch die gesamte Welt gelaufen ist. Es wurden in 110 Sekunden 3500 Spielzyklen durchlaufen, was einer durchschnittlichen 32 Bildern pro Sekunde entspricht. Ein Spielzyklus dauerte im Durchschnitt 114467 Systemzyklen, der durchschnittliche Anteil des Kamera Managements 4934 Systemzyklen, demnach ein Anteil von 4,3% an der Gesamtlaufzeit. Das Kamera Management bleibt damit in seiner durchschnittlichen Rechenzeit unter der geforderten 5% Grenze.

Interessant sind die Schwankungen in der Laufzeit pro Spielzyklus, da sich der Spieler durch unterschiedlich große und detaillierte Räume bewegt. Dies wirkt sich besonders auf den Aufwand der Graphikberechnung aus. Die Messwerte ergeben eine Standardabweichung von 32400 Systemzyklen pro Spielzyklus, dass sind zirka 28% vom Durchschnittswerts. Entgegen der Erwartung beträgt die Standardabweichung des Kamera Managements allerdings 2100 Systemzyklen, was 43% des Durchschnittswerts entspricht. Dies überrascht, da eigentlich erwartet wurde, dass die Berechnung im VDG nur lokal und unabhängig von der Komplexität des umgebenden Raumes sind. Im Detail ist für dieses Verhalten die Berechnung der Sichtbarkeit verantwortlich. Durch die rekursive Definition müssen in großen Räumen sehr viele Teilwege der Sichtstrecke getestet werden. Des weitern schwanken die Rechenzeiten des A* Algorithmus für

die Voronoi Distanz mit der Entfernung zum Zielobjekt. Hier kann aber eventuell durch geschicktere Implementation Rechenzeit gewonnen werden. Insgesamt ist die Laufzeit des Kamera Managements aber zufriedenstellend.

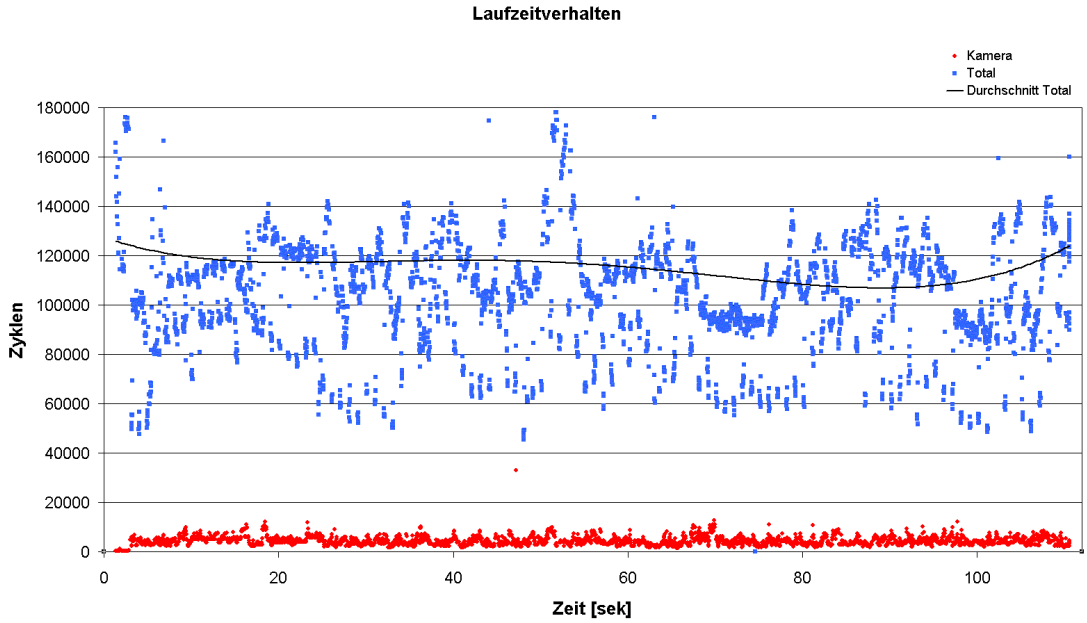


Abbildung 7.1: Laufzeitverhalten des Kamera Managements

Der Speicherbedarf des Kamera Managements zur Laufzeit wird hauptsächlich durch die Größe der jeweiligen VDG Struktur und deren kd-Baum bestimmt. Da beide Strukturen fast als direktes Speicherabbild in den .VDG Dateien gespeichert werden, ist der Speicherbedarf deren jeweiligen Größe equivalent. Bei kleinen, geschlossenen Karten bleibt deren Größe kleiner als 500kB. Bei großen, halb-offenen Karten steigt die Anzahl der Knoten und Flächen des VDG stark an und damit auch der Speicherbedarf bis auf 6MB. Da man bei heutigen PC Systemen von einer Minimalausstattung von 128MB RAM ausgehen kann, sind dies höchstens 5% des Gesamtspeichers. Die Ladezeiten sind vernachlässigbar, da die VDG Daten ohne aufwendige Operationen direkt aus der Datei in den Speicher geladen werden.

Während die Laufzeit und der Speicherverbrauch des Kamera Managements relativ leicht zu Messen ist, kann über die Qualität der Kameraführung nur schwer Aussagen getroffen werden. Allgemein ist das Empfinden der Kameraführung subjektiv und kann von mehreren menschlichen Betrachtern sehr unterschiedlich bewertet werden. Unabhängig vom Verfahren sind auch die gewählten Optimal- und Toleranzwerte für die verschiedenen Kamerakriterien sehr entscheidend für die Qualität des Kamera Managements, wobei deren Feinabstimmungen ausgesprochen lange dauern können. Hilfreich ist es dabei, ein Spiel aufzuzeichnen und es dann mit verschiedenen Parameterwerten abzuspielen. So kann man für immer gleichbleibende Ausgangswerte mehrerer Variationen ausprobieren.

Das Kamera-Management zeigt beim Testen viele der gewünschten Eigenschaften (siehe auch Videos auf CD-ROM). Die Kamera bewegt sich träge und auf harmonischen Wegen. Bei höheren Geschwindigkeiten bleibt die Kamera in der Mitte von Gängen oder Schächten. Bewegt sich der Spieler langsame oder bleibt stehen, nähert sich die Kamera auch Wänden und versucht Winkel und Abstand zu optimieren. Die Kamera verfolgt den Spieler stetig über die gesamte Zeit, ohne zu springen. Zwar verliert sie einen schnellen Spieler z.B. an Ecken oder Senken, folgt ihm aber auch ohne Sichtkontakt weiter und holt in nach kurzer Zeit wieder ein.

Schwierig ist es, einen optimalen Wert für die Trägheit zu bestimmen. Bei einer zu großen Trägheit kommt es beim abrupten Abbremsen des Spielers dazu, dass die Kamera über die optimale Einstellung hinausschießt und anschließen zurück pendelt. Ist die Trägheit zu gering, kommt es bei der Verfolgungen von schnellen Spielern dazu, dass die Kamera zu stark den Voronoi Flächen folgt und sich auf einem Zick-Zack Kurs zwischen den Voronoi Knoten bewegt.

Eine weitere Eigenart der Voronoi Kamera ist es, dass beim schnellen Durchqueren von weiten Räumen die Kamera stark von den Voronoi Knoten mit großen Freiheiten angezogen wird. Dies ist besonders bei halb-offenen Räume ohne Decken der Fall, wo die Kamera sich automatisch nach oben bewegt. Andererseits entsteht dadurch eine relativ übersichtliche Ansicht.

Eine andere Möglichkeit die Qualität des Kamera Managements zu messen, ist es die Geschwindigkeit der Kamera über einen längeren Zeitraum zu messen und deren Änderungsraten zu analysieren. Eine träge, stetige Kamera sollte keine sprunghaften Geschwindigkeitsänderungen vollziehen und ähnliche Geschwindigkeiten wie der verfolge Spieler haben. Dafür wurde einen Spiel aufgezeichnet, in dem der verfolgte Spieler einmal quer durch die Karte *datacore* läuft. Dann wird beim Abspielen der Aufzeichnung über 60 Sekunden die Geschwindigkeit der Kamera pro Zyklus in einer Textdatei gespeichert. Eine graphische Darstellung der Messdaten ist in Abbildung 7.2 zu sehen. Die Laufgeschwindigkeit eines Spieler beträgt 265 Einheiten pro Sekunde, beim geduckten Schleichen sinkt die Geschwindigkeit auf 96 Einheiten pro Sekunde. Es zeigt sich besonders beim gemittelten Durchschnitt, dass die Voronoi Kamera sich in diesen beiden Geschwindigkeitsbereichen aufhält.

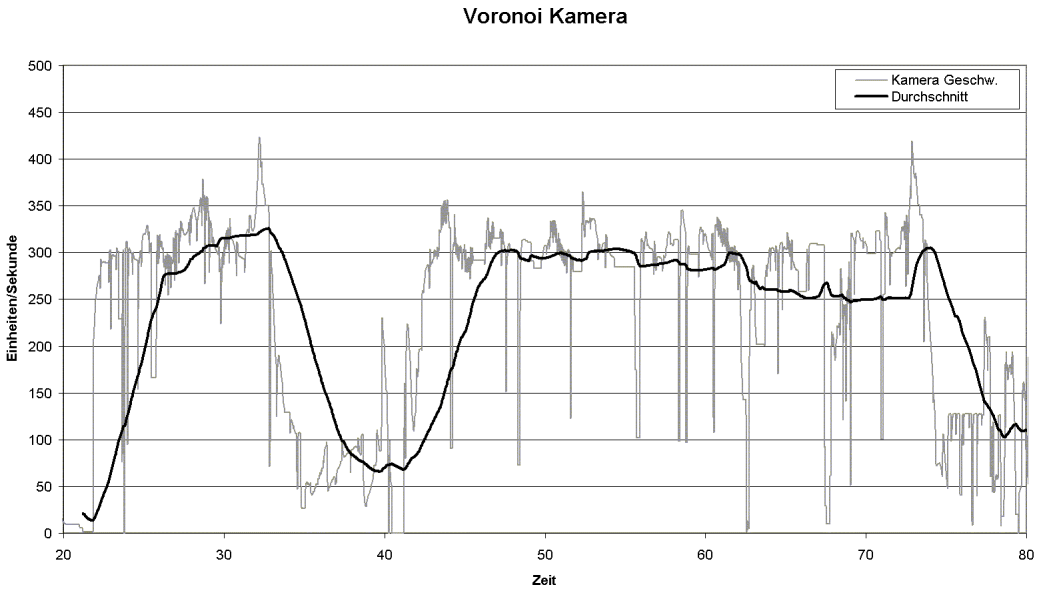


Abbildung 7.2: Geschwindgkeiten der Voronoi Kamera

Um die gemessenen Daten besser Bewerten zu können wurde die Spielaufzeichnung auch im herkömmlichen Zuschauer-Modus der Standard Kamera in *Half-Life* (*locked chase cam*) abgespielt und über den selben Zeitraum die Geschwindigkeiten dieser Kamera aufgezeichnet (Abbildung 7.3). Die Standard Kamera zeigt den Spieler immer aus dem selben, relative Winkel (von hinten) und dem gleichen Abstand. Liegt ein Hindernis zwischen Spieler und Kamera, spring die Kamera vor dieses Hindernis. Dadurch entstehen sehr große, sprunghafte Geschwindigkeitsänderungen, welche Spitzengeschwindigkeiten bis zu 7000 Einheiten pro Sekunde ergeben. Die Standard Kamera kann also nicht als stetig oder träge bezeichnet werden kann.

Im Vergleich zwischen den beiden Kameraverhalten zeigt sich, dass die Standard Kamera über die ganze Zeit sehr stark und häufig springt und damit die durchschnittliche Geschwindigkeit wesentlich höher liegt als die der Voronoi Kamera. Zwar sind im Verlauf ähnliche Tendenz zu erkennen, z.B. die langsame Phase um Sekunde 40, jedoch ist selbst der gemittelte Durchschnitt der Standard Kamera relativ sprunghaft. Dies hängt auch mit der Geometrie der gewählten Spielkarte *datacore* zusammen, welche aus kleinen und verwinkelten Gängen besteht und so die Standard Kamera fast immer hin und her springen zwingt.

Obwohl das Voronoi Management im direkten Vergleich ein wesentlich kontinuierlicheres Bewegungsbild liefert, sind auch im Graphen der Voronoi Kamera kurze aber abrupte Einbrüche der Geschwindigkeit zu beobachten. Diese Stocken der Kamera wird manchmal auch im Spiel beobachtet und scheint durch sehr kleine Voronoi Flächen im VDG zu entstehen. Durch die Forderung nach Stetigkeit der Funktion ρ, ist die Schrittweite auf die Volumenausdehnung der aktuellen Voronoi Fläche begrenzt (siehe Kapitel 5.2). Da pro Spielzyklus nur ein Schritt im Voronoi Graphen ausgeführt wird, kann das bei hohen Geschwindigkeiten und kleinen Voronoi Flächen zu diesem Problem führen, dass dieser einzelne Schritt zu klein ist. Eventuell könnte man diesen Effekt vermeiden, falls in einem Spielzyklus mehrere Schritte der Kamera im VDG erlaubt sind, so das eine gewünschte Gesamtlänge der Schrittweite erreicht ist.

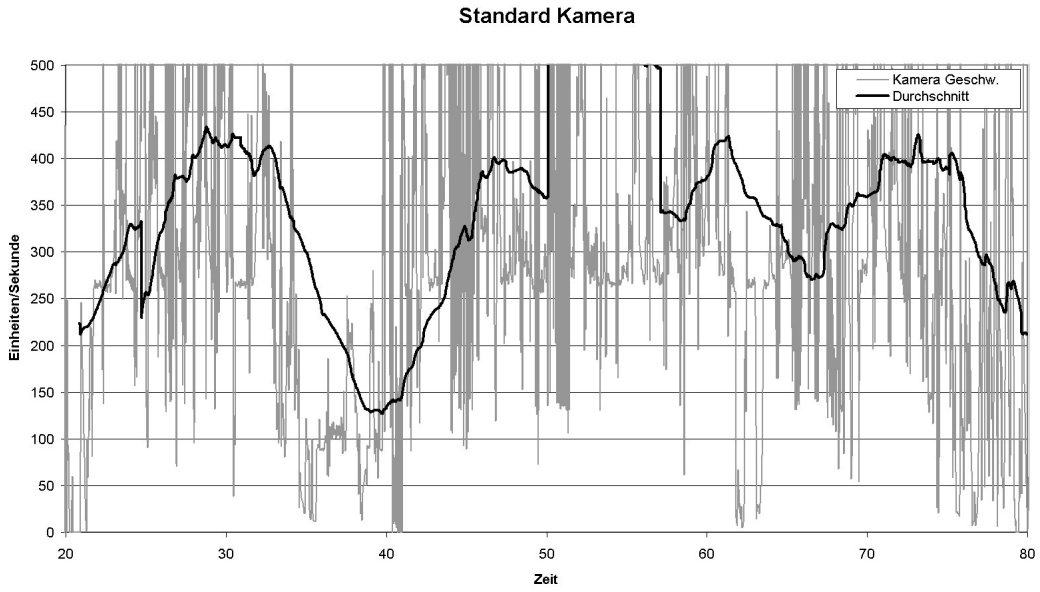


Abbildung 7.3: Geschwindigkeit der Standard Kamera

8 Fazit

Ziel dieser Arbeit war es, eine Datenstruktur und Methoden zu schaffen, welche eine schnelle lokale und globale Kameranavigation unter Berücksichtigung verschiedener Kriterien in einer Echtzeitanwendung ermöglicht. Diese Eigenschaft, global und lokale Navigation zu unterstützen, wurde durch eine Erweiterung der klassischen Voronoi Graphen erzielt. Eine schnelle Ausführung zur Laufzeit wurde durch eine Datenreduktion in einer einmaligen Vorverarbeitungsphase ermöglicht. Durch die Implementation des Verfahren in einer aktuellen Spielumgebung wurde die Praxistauglichkeit unter realistischen Bedingungen unter Beweis gestellt. Die benutzen Datenstrukturen sind klein und die Methoden schnell genug um in Echtzeitsystemen verwendet zu werden. Das über Kostenfunktionen gesteuerte Kamera Management erwies sich als stabil und zuverlässig. Es zeigte die gewünschten Eigenschaften, wie sie auch von konventioneller Kameraführung aus Spielfilmen erwartet werden.

Das Hauptproblem der Arbeit war die Berechnung des Voronoi Graphen für eine 3-dimensionale Welt aus Polyedern. Da die praktische Erforschung neuer Verfahren der Kameraführung das Hauptziel dieser Arbeit war, ist es nicht möglich gewesen, auch eine effizienten und stabilen Algorithmus für die Berechnung der generalisierten Voronoi Graphen aus soliden Hindernissen in $\mathbb{R}^3$ zu implementieren. Dies ist Aufgrund der parabolischen Kanten und Flächen eine sehr komplizierte Aufgabe und es standen auch keine frei nutzbaren Softwarebibliotheken für diesen Zweck zur Verfügung. Deshalb musste der aufwendige Umweg über die Rasterung der Hindernisse gegangen werden, welcher viele Probleme mit sich brachte. Zum einen ist die gleichmäßige Rasterung und deren enormer Speicherbedarf zu lösen. Weiterhin steigt die Datenmenge durch die Rasterung quadratisch an und gerade bei simplen Räumen mit großen Flächen entstehen unnötig viele und redundante Daten. Ein Großteiler der Rechenzeit der anschließenden Reduktion wird nur für das Entfernen dieser redundanten Knoten verwendet. An dieser Stelle ergibt sich daher der beste Ansatz für zukünftige Erweiterungen. Ein direktes Berechnen der Voronoi Graphen aus der polygonalen Welt würde insgesamt eine kürze Rechenzeit bedeuten und den Speicherbedarf enorm senken. Dadurch könnten wesentlich größere Spielkarten unterstützt werden und die reduzierten Voronoi Graphen wären noch kleiner.

Ein weiteres ungelöstes Problem sind die halb-offenen Karten, welche keine natürlichen Grenzen für den Voronoi Graphen bieten. Um zu Verhindern, dass der gesamte ‚Himmel' durch den Graphen beschrieben wird, müssen sinnvolle, künstliche Grenzen eingefügt werden. Diese können entweder vom Kartendesigner definiert werden, wodurch eine Modifizierung des .BSP Karten Format nötig werden würde. Oder diese Grenzen werden durch ein automatisches Verfahren während der Berechnung des Voronoi Graphen eingefügt. In diesem Fall müssten Regeln definiert werden, wo und wann diese Grenzen zu setzen sind.

Insgesamt kann gesagt werden, dass der Aufwand den VDG für eine virtuelle Welt zu Berechnen sehr hoch ist und das hier vorgestellte Verfahren lange nicht optimal. Die Ergebnisse zeigen jedoch, dass mit dem VDG eine effiziente und sehr leistungsfähige Struktur für eine natürliche Navigation gegeben ist, da er die Struktur und Symmetrie der Räume in einer sehr nützlichen Weise repräsentiert. Obwohl hier speziell die Kameraführung als Zielanwendung gewählt wurde, ist der VDG universell für viele Aufgaben auch aus dem Bereich der künstlichen Intelligenz einsetzbar.

Betracht man die aktuelle Entwicklung moderner Computerspiele sind mehrerer Tendenzen klar zu erkennen. Eine 3-dimensionale Welt ist heutzutage Pflicht, selbst für Spiele die vom Wesen her 2-dimensional bleiben (z.B. Echtzeit-Strategie oder Jump-Run Spiele). Zweitens lässt sich nur mit graphischen Effekten im Zeitalter der hardwarebasierten 3D-Graphikbeschleuniger kei-

ne Spitzenprodukte mehr verkaufen. Spieler wollen nicht nur durch eine wunderbare aber statische Welt laufen, sie wollen sie auch erleben. Dazu gehören realistische Physik und eine massive Bevölkerung mit künstlichen Intelligenzen. Entwicklung und Einsatz von leistungsfähigen Physiksimulationen im Spielsektor hat in den letzten Jahren enorm zugenommen und damit gebräuchliche Standards entwickelt (z.B. die *Havok* oder *Karma* Bibliotheken). Im Bereich der KI hat sich allerdings noch nicht viel getan, jeder Hersteller entwirft eigene Konzepte und Lösungen. Vieles des intelligenten Verhalten von computergesteuerten Charakteren (NPCs) wurde von Designer für verschiedene Situationen individuell erstellt. Damit lassen sich kurzfristig zwar erstaunliche Effekte erzielen, bei sehr großen Welten mit viele NPCs ist diese arbeit nicht mehr von Hand zu erledigen. Aktuelle Verfahren zur globalen Navigation von NPCs orientieren sich ähnlich dem VDG Verfahren an Wegegraphen (*way points*), allerdings werden diese Wegegraphen direkt vom Designer erstellt oder durch ‚Training' mit menschlichen Spielern erstellt (ein Spieler läuft durch den Level und hinterlässt dabei einen Wegegraphen). Diese Methoden sind für größere Welten nicht mehr praktikabel und es muss ein automatisierbarer Weg gefunden werden, diese Wegegraphen zu erstellen.

Ein weiter modifizierter VDG könnte ein guter Ansatz sein, eine universell nutzbare Struktur zu schaffen, welche für viele unterschiedliche Arten von intelligenter Navigation im Raum genutzt werden kann.

9 Anhang

A Quellenverzeichnis

[Ari76] D. Arijon, “Grammar of the Film Language”, Silman-James Press, Los Angeles, 1976

[Aur91] F. Aurenhammer, „Voronol Diagrams -A Survey of a Fundamental Geometric Data Structure**”,** *ACM Computing Surveys*, Vol 23, Num 3, 1991

[Bar89] J. Barraquand, B. Langlois, J.C. Latombe, „Numerical Potential Field techniques for Robot Planning“, Stanford University, 1989.

[Berg00] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, “Computational Geometry: Algorithms and Applications”, Seconds Edition, Springer Verlag, Berlin, 2000

[Blu67] H. Blum, “A Transformation for Extracting New Descriptors of Shape”, In *Models for the Perception of Speech and Visual Form*, MIT Press, Cambridge, MA, Seiten 362-380. 1967

[Coh96] M. Cohen, Li-wei He, D. Salesin, „The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing”, Microsoft Research/University of Washington, Seattle, 1996

[Dru92] S. Drucker, T. Galyean, D. Zeltzer, “CINEMA: A System for Procedural Camera Movements“, Massachusetts Institute of Technology Media Lab, 1992.

[Dru94] S. Drucker, “Intelligent Camera Control for Graphical Environments“, Ph.D. Thesis, Massachusetts Institute of Technology Media Lab, 1994.

[Ebe01] D. Eberly, „ 3D Game Engine Design: a practical approach to real-time computer graphics“, Morgan Kaufmann Publishers, 2001

[Eri02] J.Erickson, ”Dense Point Sets Have Sparse Delaunay Triangulations”, Proceedings ACM-SIAM symposium on Discrete algorithms 2002, Pages: 125 – 134, ACM Press, 2002

[Fol90] J. Foley, A. van Dam, S.Feiner, J. Hughes, “Computer Graphics, Principles and Practice”, Second Edition, Addison-Wesley Publishing, 1990

[Gam03] GameSpy.com, „Top Game Servers By Players”, http://www.gamespy.com/stats/ , 2003

[Gre86] N. Greene, ”Environment Mapping and Other Applications of World Projections“, IEEE Computer Graphics and Applications, Vol 6, Nr 11, 1986

[Gui00] Max McGuire, „Quake 2 BSP File Format“, FlipCode Tutorials, http://www.flipcode.com/tutorials/tut_q2levels.shtml ,2000

[Häf99] K. Häfner, „Computerspiele - pädagogisch sinnvoll?“, Skript zur Veranstaltung, AG ITG-L Fachbereich 3, Universität Bremen, 1999

[Halp01] N. Halper, R. Helbing, T. Strothotte, “A Camera Engine for Computer Games: Managing the Trade-Off Between Constraint Satisfaction and Frame Coherence”, University of Otto-von-Guericke, Magdeburg, Germany, in Eurographics Volume 20, 2001

[Hoff99] K.E.Hoff, Tim Culver, John keyser, Ming Lin, Dinesh Manocha, „Fast Computation of Generalized Voronoi Diagrams Using graphics hardware“, Depratment of Computer Science, University of North Carolina at Chapel Hill, 1999

[Hopp93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, “Mesh Optimization”, University of Washington Seattle, in ACM SIGGRAPH 1993, pages 19-26

[IDSA02] Interactive Digital Software Association, “Essential Facts About the Computer And Video Game Industry”, www.idsa.com, 2002.

[Kon02] L.Konrad, „Spielplatz Computer – Kultur, Geschichte und Ästhetik des Computerspiels“, dpunkt Verlag, 2002, http://www.spielplatz-computer.de/

[Lat93] J.C. Latombe, “Robot Motion Planning”, Third Edition, Stanford University, 1993.

[LiYu00] Tsai-Yen Li, Tzong-Hann Yu, Yang-Chuan Shie, “On-Line Planning For An Intelligent Observer In A Virtual Factory”, Computer Science Department National Chengchi University, Taipei, Taiwan, 2000

[Maz96] T. Mazuryk, M. Gervautz, „Virtual Reality History, Applications, Technology and Future”, Technical Report TR-186-2-96-06, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1996

[Mon00] J. Monaco, „How to Read a Film”, Second Edition, Oxford University Press, London/New York, 2000.

[Oka92] A. Okabe, B. Boots, K. Sugihara, “Spatial Tessellations: Concepts and Applications of Voronoi Diagrams”, John Wiley & Sons, Chichester, UK, 1992

[Qhull02] C.B. Barber, D.P. Dobkin, H.T. Huhdanpaa, "The Quickhull algorithm for convex hulls," *ACM Trans. on Mathematical Software*, 1996. http://www.geom.umn.edu/software/qhull

[Rou98] J. Rourke, “Computational Geometry in C”, Second Edition, Cambridge University Press, 1998

[SDK03] “Half-Life Software Developer Kit”, Valve Software, http://www.valve-erc.com , 2003

[Sin99] S. Singhal, M. Zyda, “Networked Virtual Environments: Design and Implementations”, 1. Auflage, ACM Press Books, New York, 1999.

[Suth65] I.Sutherland, “The Ultimate Display”, Proceedings of IFIP Congress 2, 1965.

B Abbildungsverzeichnis

C Abkürzungsverzeichnis

AI	Artificial Intelligence
BSP	Binary Space Partitioning-Tree
CPL	Cyberathletic Professional League
CSG	Constructive Solid Geometry
DFÜ	Datenfernübertragung
DLL	Dynamic Link Library
DOF	Degrees of Freedom
GPU	Graphic Processing Unit
HL	Half-Life
HLTV	Half-Life TV
HMD	Head Mounted Display
KI	Künstliche Intelligenz
LES	Linear Equation System (=LSG)
MAT	Medial Axis Transformation
NPC	Non Player Character
NVE	Networked Virtual Environment
PVS	Potential Visible Set
RLE	Run Length Encoded
SDK	Software Developer Kit
UDP	User Datagram Protocol
VDG	Voronoi Distance Graph
VE	Virtual Environment
VR	Virtual Reality

D CD-ROM Inhalt

Die CD-ROM zu dieser Diplomarbeit enthält mehrere Verzeichnisse in denen sich sämtliches Material und Quellen zu dieser Arbeit befinden. Alle enthaltene Daten und Programme sind freiverfügbar und verwendbar, unter Berücksichtigung der jeweiligen Benutzerlizenzen, z.B. des HL SDK. Bei Fragen oder Anregungen zum Quellcode antworte ich gerne per E-Mail (martin@slipgate.de).

Pfad	Inhalt
\\Documents	Die Diplomarbeit als .PDF Datei
\\Documents\Resources	Verschiedene Arbeiten und Veröffentlichungen zu den hier behandelten Themenbereichen.
\\Half-Life	Alle benötigten Dateien der Half-Life Implementation. Zum Ausführen müssen alle Unterverzeichnisse und Dateien über eine gültige Half-Life Installation Version 1110 kopiert werden.
\\Pictures	Verschiedene Bilder dieser Arbeit als Originale
\\SDK	Quellcode des SDK 2.2 als MS Visual C++ Projekte
\\SDK\bsp2vdg	Quellcode des BSP2VDG Werkzeugs
\\SDK\cl_dll	Quellcode der Client Logik inklusive des Kamera Managements
\\SDK\dlls	Quellcode der Server Logik (unverändert)
\\SDK\qhull	Quellcode der Qhull Bibliothek
\\SDK\shared	Gemeinsam genutzter Quellcode (z.B VoronoiGraph.cpp)
\\SDK\WinGui	Quellcode der VDGView Applikation
\\Tools	Die Programme BSP2VDG.EXE und VDGVIEW.BAT für Microsoft Windows 32 Bit Systeme
\\Videos	Videos des Kamera-Managements in Half-Life und eine Zeitrafferaufnahme des Reduktionsprozesses in VDGVIEW

E BSP2VDG Startparameter

Die Konsolenanwendung BSP2VDG erwartet als ersten Startparameter den Namen der Spielkarte im .BSP Format, danach folgen beliebig viele Steuerparameter:

```
bsp2vdg <Karte.bsp> [-Parameter <Wert>] ...
```

Folgende Parameter sind möglich:

`-bb <n>`	Setzt die minimale Größe der Bounding-Boxen der Blätter im kd-Baum. Standardwert 256.
`-md <n>`	Setzt die minimale Freiheit, die ein Voronoi Knoten haben muss, um nicht entfernt zu werden. Standardwert 20.
`-ms <n>`	Setzt das obere Limit für die Anzahl der Rasterpunkte. Standardwert 250000.
`-rl <f>`	Setzt die gewünschte Reduktionsgrenze. 1,0 = keine Reduktion, 0,0 = maximale Reduktion. Standardwert 0,025.
`-savesamples`	Speichert die berechneten Rasterpunkte in einer Textdatei, welche von VDGView gelesen werden kann.
`-sd <n>`	Setzt die Rasterdichte. Standardwert 24.
`-subset`	Berechnet nur 1/8 der Eingabedaten für kurze Testläufe

Ein typischer BSP2VDG Durchlauf erzeugt folgende Konsolenausgabe (für *datacore*):

```
Loading model: datacore.bsp (1209 kB)
Bounding Box (-3728.0,1244.0,668.0)-(-788.0,3432.0,1404.0)
Sampling from 4722 faces ...done.
Sampled 75407 points from 4541 faces (300248 subfaces)
Projecting points on 4D parabolid ...done.
Running Qhull with 75407 sample points...done.
Building voronoi graph...done.
Voronoi vertices 440256, inner vertices 228077.
Voronoi faces 526870, inner faces 119283, triangulated 339867
Removing unused vertices...done(264594)
Time 53.1, Nodes 175662, Faces 338915, Neighbours 22, Reduction 0.00
Reducing to 4391 (0.025) vertices...
Volume error calc: found 159774, failed 15888
Time 106.3, Nodes 175661, Faces 338914, Neighbours 22, Reduction 0.09
Time 2784.8, Nodes 76828, Faces 147766, Neighbours 20, Reduction 0.13
Time 3485.0, Nodes 38130, Faces 70656, Neighbours 20, Reduction 0.23
Time 3692.1, Nodes 24796, Faces 44712, Neighbours 24, Reduction 0.35
Time 3832.1, Nodes 15394, Faces 28588, Neighbours 28, Reduction 0.54
Time 3927.0, Nodes 7727, Faces 17003, Neighbours 23, Reduction 0.85
Volume error calc: found 1, failed 5852
Volume error calc: found 0, failed 5853
Redcution limit reached.
Calculating Bounding Box ...done (-3722, 1244, 703)(-789, 3431, 1396).
Building BBox Tree...nodes 1107, leafs 839, depth 11 done.
Writing datacore.vdg ...Wrote file, size 447670.
```

F Half-Life

Um das Kamera Management in *Half-Life* zu testen wird eine komplette Installation der aktuellen Version 1110 benötigt. Das Implementation des Kamera Managements befindet sich auf der beliegenden CD in der Datei `\\Half-Life\valve\cl_dlls\client.dll`. Die vorberechneten .VDG Dateien für die Spielkarten liegen im Verzeichnis `\\Half-Life\valve\maps\graphs\*.vdg`. Diese neuen Dateien müssen über die bestehende *Half-Life* Installation kopiert werden. Dann kann *Half-Life* über den Aufruf von „`hl.exe -console`“ gestartet werden, wobei die Textkonsole zur Eingabe von Befehlen aktiviert wird.

Nachdem das Programm gestartet wurde, ist die Konsole über den entsprechende Menüauswahl zu aktivieren. Um z.B. die Spielkarte *datacore* zu Laden, müssen folgenden beide Befehle ausgeführt werden (das > symbolisiert das Konsolenprompt):

```
> deathmatch 1
> map datacore
```

Nun wird ein Serverprozess gestartet, welcher die Karte lädt und der Client verbindet sich automatisch mit diesem Server. Danach befindet man sich direkt im Spiel und sieht die Welt aus der First-Person Perspektive. Um in den neuen Kamera-Modus zu aktivieren, muss die Konsole mit der ^ oder ~ Taste wieder in den Vordergrund gebracht werden. Dann folgenden Befehl absetzten:

```
> v_directormode 1
```

Dann sieht man seinen eigenen Charakter aus der Third-Person Perspektive und die Kamera schwenk langsam in eine Position hinter den Spieler. Um den Spieler zu bewegen können die Cursortasten und die Maus benutz werden. Mit Space kann der Spieler springen, mit Ctrl duckt sich der Spieler und kann in kleinere Räume gelangen. Um den umgebenden Voronoi Flächen anzuzeigen, muss

```
> v_directormode 2
```

eingegeben werden. Das rote Dreieck ist jeweils die aktuelle Voronoi Fläche, die anderen blauen Dreiecke sind die Flächen in dessen Nachbarschaft, welche in die Berechnungen des Kamera Managements berücksichtigt werden.

Um die Toleranzgrenzen der unterschiedlichen Kamerakriterien zu verändern, muss der folgende Befehl mit den entsprechenden fünf Werten aufgerufen werden:

```
>v_constraints <angle> <distance> <visibility> <mobility> <inertia>
```