TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DIPLOMARBEIT

Fotorealistische Visualisierung mittels Game Engines in der Architektur

Untersuchung der Möglichkeiten von Game Engines anhand einer Echtzeitvisualisierung der Schule am Kinkplatz

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung

Ass.Prof. Dipl.-Ing. Dr.techn. Peter Ferschin

E259.1

Digital Architecture and Planning

eingereicht an der Technischen Universität Wien

Fakultät für Architektur und Raumplanung

von

Peter Steiner

0726926

Wien, am

Abstract

This paper describes the usage of game engines for the creation of photorealistic architectural Visualizations. It examines the functionality and possibilities of such software as well as all of the required steps for the creation of high quality real-time visualizations.

Current game engines are capable of achieving very realistic results, which quality wise are comparable to those of established render engines. The fact that the render process takes places in real-time is very beneficial for the interactivity of the scene. Changes regarding camera angles, materials or lights can be done easily. This allows a very intuitive workflow and offers a lot of new possibilities for presentations.

The first part of the paper includes a short introduction, a description of different available render engines, the advantages or disadvantages that may arise from those differences as well as an overview of current industry standards and established workflows.

In order to clarify the described process the second part is dedicated to a tangible project created with the Unreal Engine. To verify the functionality of the engine the public school building by austrian architect Helmut Richter located on Kinkplatz in Vienna was chosen. With its different material- and light situations and its challenging geometry, the building offers a good example to test the power of current game engines. All of the steps required to achieve the final result are shown, followed by a short description of problematic areas and an outlook into further developments.

Keywords: *Architectural visualization, Game Engine, Realtime, Photorealism, Helmut Richter, Kinkplatz*

Kurzfassung

Diese Arbeit befasst sich mit der Anwendung von Game Engines zur Erstellung von fotorealistischen Visualisierungen in der Architektur. Sie untersucht die Funktionsweise und die Möglichkeiten der Software sowie alle Schritte, die für die Erstellung qualitativ hochwertiger Echtzeitvisualisierungen notwendig sind.

Aktuelle Game Engines sind im Stande, realitätsgetreue Ergebnisse zu erzielen, welche sich qualitativ auf dem Niveau etablierter Render Engines befinden. Die Tatsache, dass die Berechnung in Echtzeit erfolgt, lässt ein hohes Maß an Interaktivität zu – sei es beim Erstellen von Kamerafahrten, dem Tausch von Materialien oder dem Ändern von Lichtsituationen beziehungsweise des virtuellen 3D Modells. Diese Herangehensweise fördert ein sehr experimentelles und intuitives Arbeiten. Zudem eröffnet sich eine Vielzahl neuer Präsentationsmöglichkeiten.

Der erste Teil der Arbeit widmet sich einer kurzen Einführung in das Thema, den Unterschieden zwischen verschiedenen Render Engines und den Vor- bzw. Nachteilen, die sich aus diesen ergeben. Es werden bisherige Industriestandards und bewährte Workflows beschrieben.

Im zweiten Teil werden die zuvor besprochenen Themen anhand eines konkreten Objektes verdeutlicht und praktisch durch die Anwendung der Unreal Engine beschrieben. Als Beispielprojekt wurden einzelne Bereiche der Schule am Kinkplatz von Helmut Richter in Wien gewählt. Mit ihren unterschiedlichen Raum- bzw. Lichtsituationen und dem stark durch Glas bestimmten Erscheinungsbild bietet sie eine angemessene Vorlage zur Überprüfung der Möglichkeiten aktueller Engines. Angefangen beim Modelliervorgang werden alle wesentlichen Schritte in den diversen Programmen bis zur fertigen Visualisierung besprochen. Es folgen eine kurze Auseinandersetzung mit problematischen Bereichen, welche bei der Verwendung der Unreal Engine im Arbeitsalltag auffallen, sowie ein Ausblick auf künftige Entwicklungen.

Schlagwörter: *Architekturvisualisierung, Game Engine, Echtzeit, Fotorealismus, Helmut Richter, Kinkplatz*

1 - Prolog

Visualisierungen sind in den letzten Jahren in der Architektur allgegenwärtig geworden. Bilder – ob stilisierte oder fotorealistische - sind leichter und klarer zu verstehen als Pläne. Gerade in Wettbewerbssituationen können sie oft den Unterschied zwischen Erfolg und Misserfolg ausmachen. Die professionelle Tätigkeit im Bereich der Architekturvisualisierung ist mittlerweile so komplex, dass die meisten Büros diese entweder an externe Auftragnehmer auslagern, oder – sofern es wirtschaftlich möglich erscheint – intern eigene Renderteams aufbauen. Da die Fortschritte rasch erfolgen, wäre es ansonsten für Architekten kaum möglich neben ihrer sonstigen Tätigkeit mitzuhalten und ansprechende Visualisierungen zu erstellen.

„Klassische" Arbeitsweisen bei der Produktion von Architekturvisualisierungen sind aufgrund des verhältnismäßig hohen Zeitaufwands, einer geringen Flexibilität und damit verbundenen höheren Kosten häufig negativer Kritik ausgesetzt. In den meisten Fällen kommen computergenerierte Bilder lediglich nach erfolgter Planung für Präsentationszwecke zum Einsatz. Nur langsam wird der Nutzen von Visualisierungen auch für andere Phasen der Planung erkannt. Dies liegt in erster Linie an der erwähnten Inflexibilität bisheriger Render Engines. Umgestaltungen jeglicher Art – die gerade zu Beginn von Projekten unvermeidbar sind – gehen meist mit viel Aufwand zur Änderung der Visualisierung einher, da die Szene aktualisiert und neu berechnet werden muss. Anpassungen der Kameraperspektive, der Materialien oder der Belichtungssituation bedeuten somit immer einen zusätzlichen Zeitaufwand. Ein Umstand, welchen viele planende Büros möglichst zu vermeiden suchen. Aus diesem Grund erscheint es kaum verwunderlich, dass viele Nutzer in Visualisierungen lediglich „hübsche Bilder" zu Präsentationszwecken oder für Wettbewerbssituationen sehen, wo doch eine Unmenge an Leistungsfähigkeit für zahlreiche andere Anwendungsbereiche in ihnen steckt.

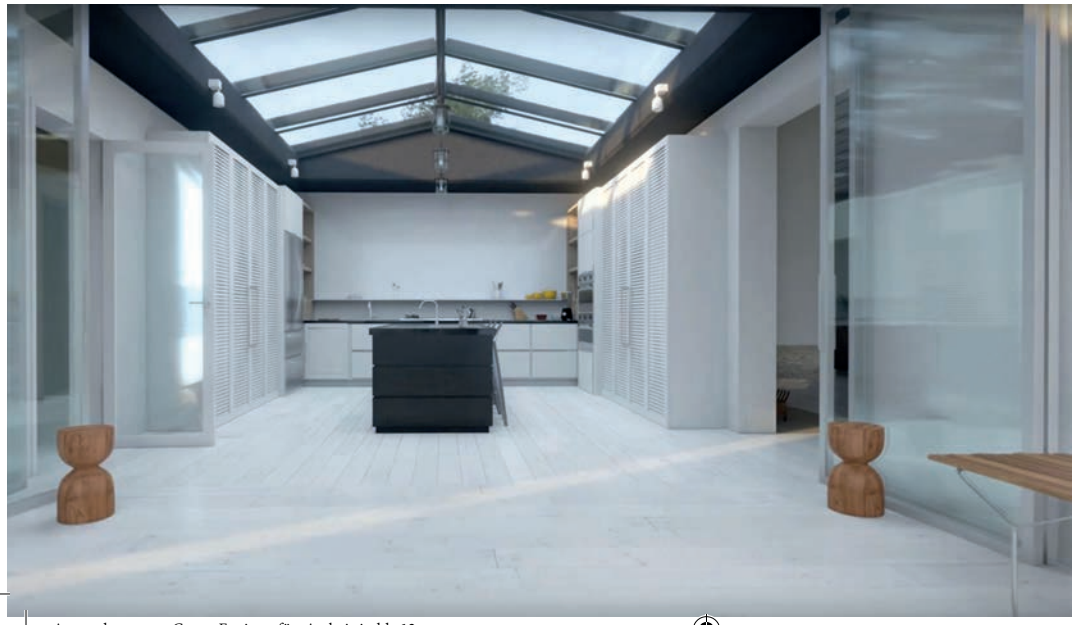
Abb. 1.0: In der Unreal Engine 4 erstellte Visualisierung (Rafael Reis)

Durch die gegenwärtigen technologischen Entwicklungen könnte bald eine Vielzahl an neuen Möglichkeiten vorhanden sein, da mit aktuellen Game Engines die gleichen Werkzeuge zur Verfügung stehen, wie sie auch für überzeugende Effekte in der Film- und Spielindustrie verwendet werden. Durch ihre Vorteile auf dem Gebiet der Echtzeitdarstellung haben sie das Potential die Art und Weise, wie Visualisierungen momentan in der Architektur gedacht werden, von Grund auf zu ändern.

Überlegt man, dass – ähnlich wie in einem Computerspiel – die gesamte virtuelle, fotorealistisch dargestellte Szene in Echtzeit erkundet werden kann, werden einem sehr rasch die großen Vorteile von Echtzeitvisualisierungen bewusst. Beispielsweise könnten Entwurfsvarianten durch die dynamische Änderung vom 3D Modell getestet oder Materialalternativen rasch und intuitiv erprobt werden. Auch Lichtsituationen können innerhalb von Sekundenschnelle geändert und beurteilt werden. Mit klassischen Render Engines ist es zeitaufwendig Animationen oder sogenannte Walkthroughs zu erstellen. Bedenkt man, dass Renderzeiten auf modernen Rechnern zwischen einigen Minuten und mehreren Stunden betragen können, wird klar, dass alleine die benötigte Zeit zum Berechnen der Bilder immens ist. Selbst wenn man von einer verhältnismäßig kurzen Renderzeit von 10 Minuten pro Bild ausgeht, sind ungefähr 40 Stunden notwendig um 10 Sekunden Film zu erstellen. (24Bilder/Sek. * 10min/Bild * 10Sek. = 2400min). Dabei handelt es sich nur um die reine Renderzeit, das Einstellen der Kameras und die Nachbearbeitung erfolgen separat. Hinzu kommt noch, dass solche Animationen oft langweilig und leblos wirken. Game Engines ermöglichen, Animationen live darzustellen und Kamerafahrten in Echtzeit aufzunehmen. Auch klassische Arbeitsschritte - wie die Nachbearbeitung in einem Drittanbieterprogramm – könnten komplett neu gedacht werden.

Mit Hilfe von Game Engines könnten Architekturvisualisierungen viel effektiver und weitreichender als bisher üblich im Planungsprozess eingesetzt werden. Statt lediglich als Präsentationswerkzeug zu dienen sind Anwendungen innerhalb von Entwicklerteams, als Unterstützung beim Entwurf oder zur Überprüfung von Designs, denkbar.

Abb. 1.2: In der Unreal Engine 4 erstellte Visualisierung (Rafael Reis)



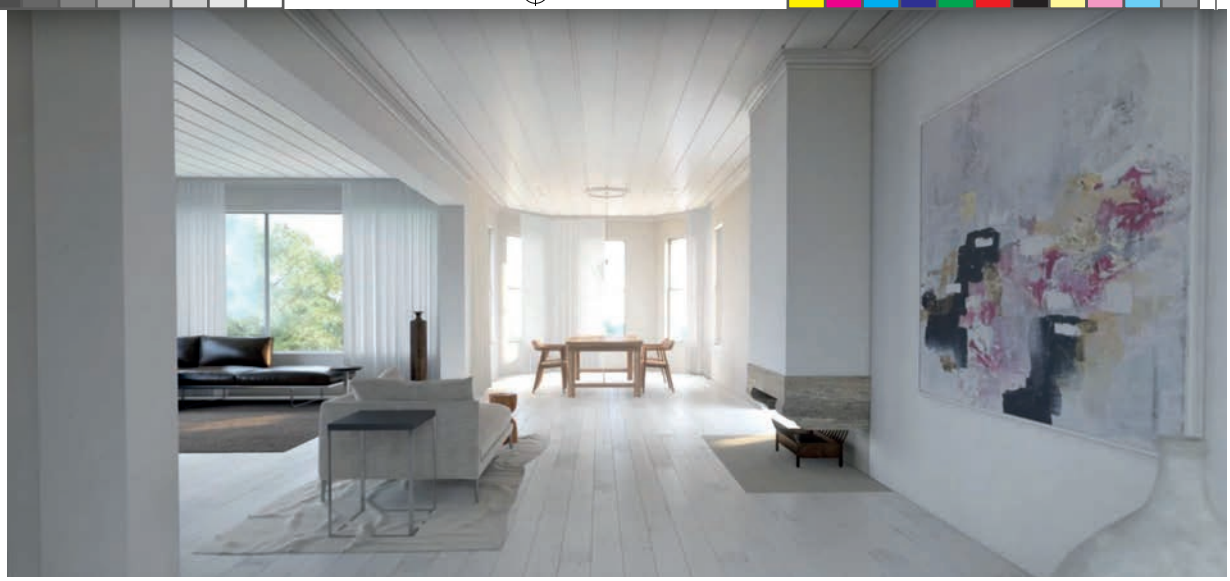


Abb. 1.1: In der Unreal Engine 4 erstellte Visualisierung (Rafael Reis)

Mit der laufenden Entwicklung von Game Engines haben die Grenzen zwischen Architekturvisualisierung und Spiele- beziehungsweise Filmindustrie begonnen zu verschwimmen. Bei einem Interview mit „The Verge“ hat Tim Sweeney, Gründer von Epic (Unreal Engine), gesagt:

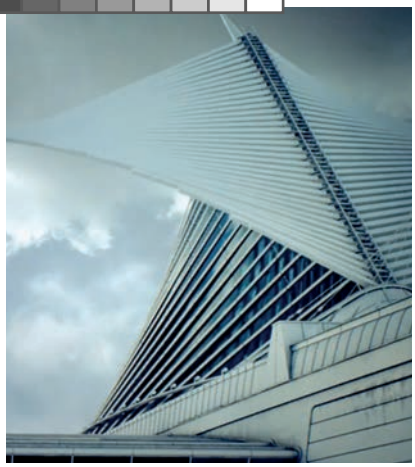
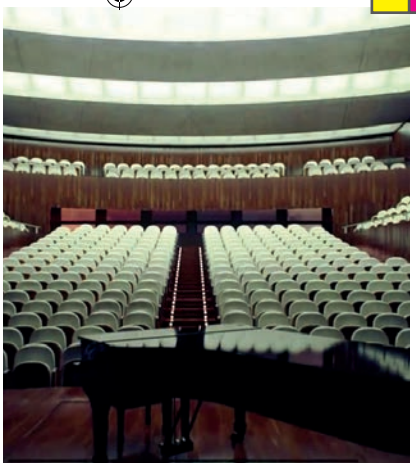
„[...] We're realizing now, that Unreal Engine 4 is a common language between all these fields.“ (Tim Sweeney)

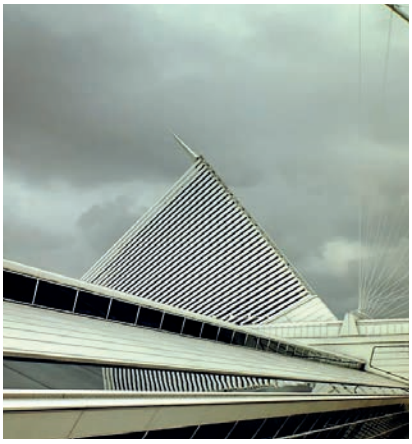
Eine gemeinsame Sprache zwischen scheinbar so unterschiedlichen Feldern wie Architektur, Spielen und Filmen zu erschaffen, bedeutet auch, dass diese Industrien gegenseitig voneinander lernen können. Beispielsweise können Visualisierer die Workflows von Spieleentwicklern verinnerlichen oder aus Filmeffekten lernen, um hochwertigere Visualisierungen für Architekturprojekte zu erstellen [vgl. URL Game Engines 1]. Bezieht man außerdem die qualitativen Ergebnisse, zu denen Game Engines mittlerweile fähig sind, und die oben genannten Möglichkeiten mit ein, drängt sich die Frage auf, warum sich solche Engines noch nicht zum Industriestandard entwickelt haben.

Abb. 1.3: In der Unreal Engine 4 erstellte Visualisierung (koooolalala)

Ein Grund für das verhältnismäßig träge Voranschreiten der Game Engines im Architekturbereich ist sicherlich die Tatsache, dass sie nicht für Architekturvisualisierungen entwickelt worden sind. Für seit Jahren etablierte Render Engines für Architektur- und Produktvisualisierungen gibt es eine Menge an Informationen und Tutorials online. Jeder, der die Grundlagen der 3D Modellierung versteht und bereit ist etwas Arbeit zu investieren, kann sich mit Hilfe von Internet-Tutorials ein grundlegendes Verständnis für die Software aneignen und innerhalb kurzer Zeit erste Ergebnisse erzielen. Natürlich ist dies auch bei aktuellen Game Engines der Fall, allerdings steht hier zum jetzigen Zeitpunkt - gerade im Bereich der Visualisierung von Bauprojekten - nur ein Bruchteil der Information zur Verfügung. Wie in einschlägigen Onlineforen unschwer zu erkennen ist, fällt es selbst Personen, welche seit Jahren im 3D Bereich tätig sind, oft schwer sich das nötige Wissen über die neuen Engines anzueignen, beziehungsweise diese vollkommen zu verstehen. Dies liegt auf einer Seite an den – verglichen mit anderen Engines - teils erheblichen Unterschieden im Workflow. Auf der anderen Seite sind viele Funktionen und Werkzeuge, die bis jetzt in anderer Software verwendet werden, noch nicht in Game Engines implementiert, da diese wie Anfangs erwähnt nicht für die Visualisierung von Architekturprojekten gedacht waren. Gerade bei diesem Aspekt spielt die Community eine große Rolle, da der Austausch zwischen den Entwicklern und den Endnutzern viele Erkenntnisse und neue Funktionen mit sich bringt.

Abb. 1.4: Mittels V-Ray erstellte und mit Photoshop nachbearbeitete Visualisierung (Pixelflakes)



o.: Abb. 1.5
u.: Abb. 1.8

o.: Abb. 1.6
u. Abb. 1.9

o.: Abb. 1.7
u.: Abb 1.10

Mittels V-Ray erstellte und mit Photoshop nachbearbeitete Visualisierungen.
1.5, 1.6, 1.8 + 1.9: Alex Roman
1.7 + 1.10: Bertrand Benoit

Ein weiterer Grund, warum ein Großteil der Visualisierer so zögerlich beim Umstieg agiert, ist die zusätzliche Komplikation im Workflow. Viele anderen Engines stehen als Plug-In für etablierte 3D Software zur Verfügung. Game Engines stellen zurzeit ein eigenes Softwarepaket dar. Dadurch werden zusätzliche Schritte notwendig um zum finalen Ergebnis zu kommen (Abb. 1.11). War es bisher möglich alle Phasen vom Modell bis zum Rendering innerhalb eines Programms zu durchlaufen, muss das Modell für Game Engines zunächst weiter bearbeitet und exportiert werden, um es anschließend in Echtzeit darstellen zu können. Diese zusätzlich benötigten Schritte erfordern natürlich Zeit um sie zu erlernen beziehungsweise durchzuführen. Für viele Büros erscheinen der Mehraufwand und die damit verbundenen Kosten einfach nicht lohnenswert.

Im Zuge der vorliegenden Arbeit werden Game Engines hinsichtlich ihrer Anwendbarkeit für Architekturprojekte erforscht. Es sollen die neuen Möglichkeiten, die sich durch die Anwendung solcher Programme auf dem Gebiet der Architekturvisualisierung bieten, untersucht werden.

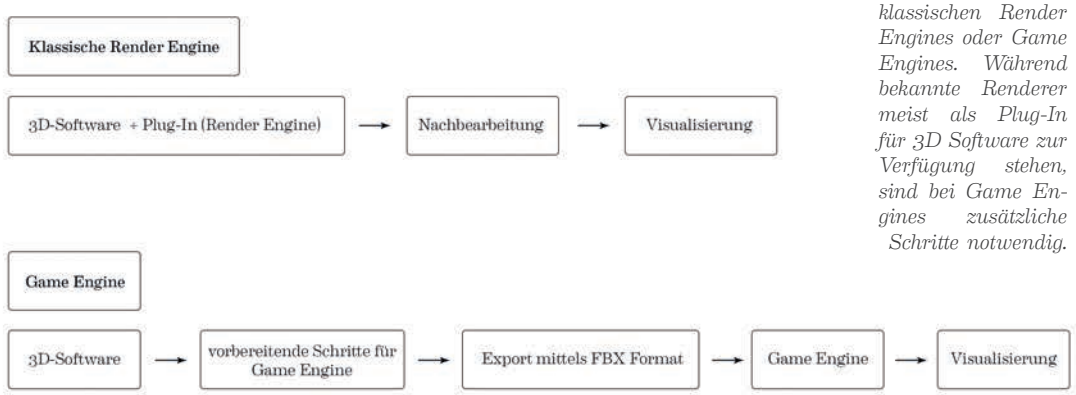


Abb. 1.11: Unterschiede im Workflow bei der Verwendung von klassischen Render Engines oder Game Engines. Während bekannte Renderer meist als Plug-In für 3D Software zur Verfügung stehen, sind bei Game Engines zusätzliche Schritte notwendig.

Abb. 1.12: Mittels V-Ray erstellte und mit Photoshop nachbearbeitete Visualisierung (Peter Steiner)

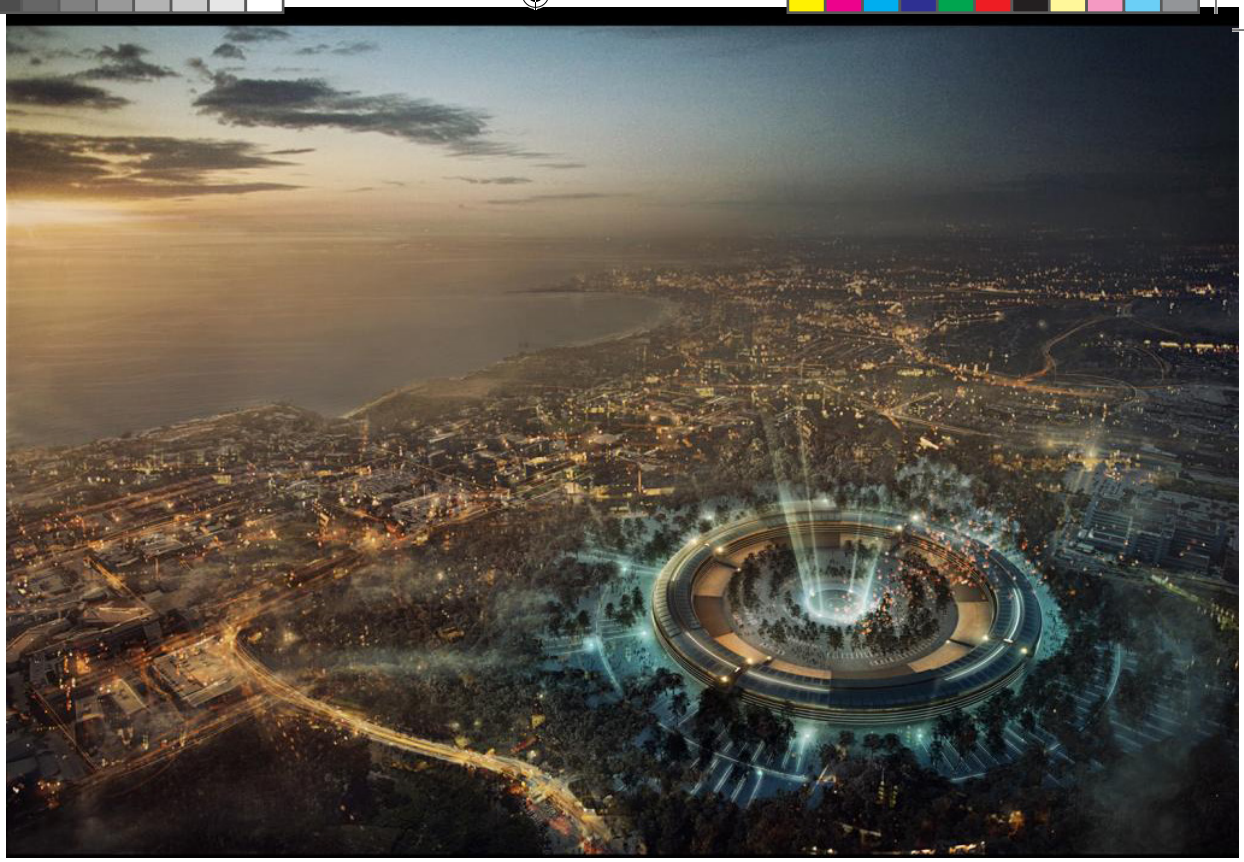


Abb. 1.12.1: Mittels V-Ray erstellte und mit Photoshop nachbearbeitete Visualisierung (Arqui9)

Abb. 1.13: Mittels V-Ray erstellte und mit Photoshop nachbearbeitete Visualisierung (Peter Steiner)

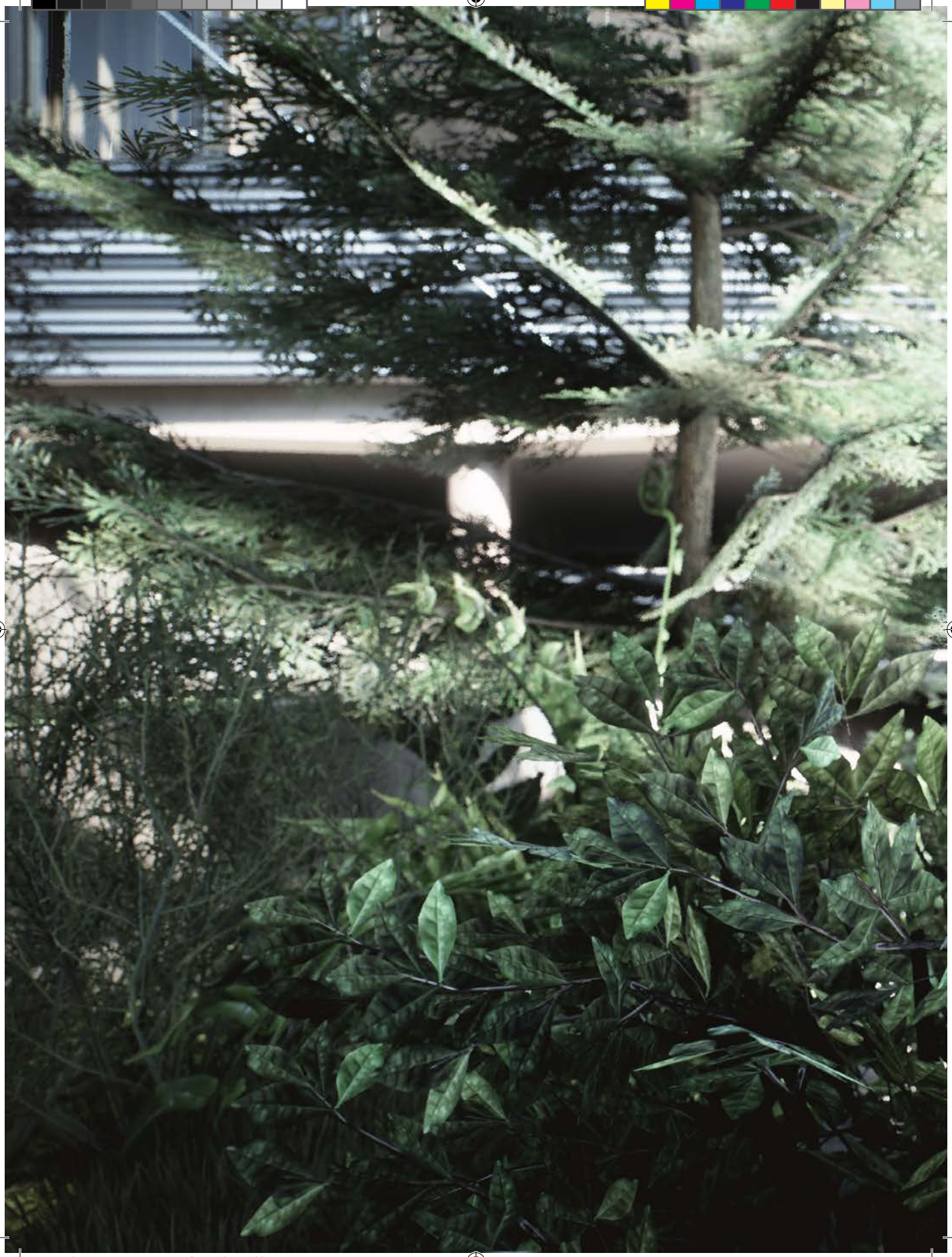


Abb. 0.7

Abb. 1.14: Tennis for two

Abb. 1.15: Spacewar

Abb. 1.16: Sketchpad

Abb. 1.17: Schattierungsmodelle

1.1 - Geschichte

Spricht man von Computergrafik, so sind meistens Bilder, welche mit spezialisierter Hard- und Software digital erstellt worden sind, gemeint. Im Großen und Ganzen beschreibt es „[...] fast alles auf einem Computer, was nicht Text oder Ton ist." [vergl. URL Computer Graphics 3]

Die Entwicklung der Computergrafik hat die Gebiete der Animation, Werbung, Videospiele, Filme und des Grafikdesigns stark beeinflusst. Digital erstelltes Bildmaterial ist heutzutage nahezu allgegenwärtig. Man findet es im Fernsehen oder in Zeitungen bei Statistiken und Graphen, aber auch in dreidimensionaler Form im Bereich der Animation oder Visualisierung für beispielsweise Architektur, Medizin, Biologie oder Meteorologie. Im Folgenden sollen kurz die wichtigsten historischen Ereignisse dargestellt werden, welche zum heutigen Stand der Technik auf diesem Gebiet geführt haben.

Mit Tennis for Two entstand 1958 das erste interaktive Computerspiel (Abb.1.14). William Higinbotham nutzte ein Oszilloskop, auf dem er ein Tennismatch simulierte um Besucher des Brookhaven National Laboratory zu unterhalten. Ein Jahr später gelang es Douglas T. Ross am MIT einen Cartoon Charakter mittels mathematischen Anweisungen digital darzustellen. Im selben Jahr veröffentlichte Ivan Sutherland seine revolutionäre Sketchpad Software, die es erlaubte mittels eines sogenannten „Leuchtstifts" simple Formen auf einem Monitor zu zeichnen (Abb.1.16). Diese konnten sogar gespeichert und später wieder aufgerufen werden. Gleichzeitig begann sich die San Francisco Bay Area (heute auch „Silicon Valley") als Standort für viele Hardwarehersteller zu etablieren. [vgl. URL Computer Graphics 1]

Der Begriff „Computergrafik" entstand tatsächlich erst 1960 und wurde durch William Fetter, einem Grafikdesigner bei Boeing geprägt.

Kurz darauf entwickelte sich das von Steve Russell programmierte Spiel „Spacewar" zu einem sofortigen Erfolg (Abb.1.15). 1963 entstand schließlich der erste computergenerierte Film. E.E. Zajac stellte in diesem die Flugbahn eines Satelliten um die Erde nach.

Es folgten weitere Computerspiele und Mitte der 1960er Jahre veröffentlichte IBM mit dem 2250 Grafik Terminal den ersten kommerziell verfügbaren „Grafikcomputer". 1966 entwickelte Ivan Sutherland das erste „Head Mounted Display", welche es erlaubte Drahtgitterbilder in Stereoskopischem 3D zu betrachten.

Ende der 1960er Jahre wurde die „Special Interest Group on Graphics" (SIGGRAPH) gebildet. Diese spezialisierte sich auf die Organisation von Konferenzen, Grafikstandards und Publikationen im Bereich der Computergrafik. Die erste Konferenz wurde 1974 in Boulder, Colorado abgehalten. Die Besucherzahl betrug damals 600 Teilnehmer. Bei der letzten Veranstaltung in Los Angeles im August 2015 zählte man bereits 14.800 Besucher. [vgl. URL Computer Graphics 1]

Anfang der 70er Jahre hatte die Universität von Utah – nicht zuletzt aufgrund der Anwesenheit von Evans und Sutherland – eine Reputation im Bereich der Computergrafik erlangt. Aus diesem Grund beschloss der Physikstudent Edwin Catmull dort Animation zu studieren. Sein Ziel war es, seine Leidenschaft für Animation in einem Spielfilm zu verwirklichen. Später war er einer der Gründer von Pixar. Auch viele andere – im Bereich der Computergrafik erfolgreiche – Personen studierten an der Universität von Utah. Unter ihnen sind zum Beispiel John Warnock (Gründer von Adobe Systems) und Jim Clark (Gründer von Silicon Graphics) zu finden.

Abb. 1.18: Apple Macintosh

Abb. 1.18.1: Commodore Amiga

Abb. 1.19: Money for Nothing

Abb. 1.20: Toy Story

Henri Gouraud, Jim Blinn und Bui Tuong Phong entwickelten unterschiedliche Schattierungsmodelle, die es erlaubten Tiefe in 3D Modellen besser darzustellen (Abb.1.17). Diese finden auch heutzutage Anwendung in 3D Software. Außerdem wurde durch Jim Blinn das Bump mapping erfunden, welches Unebenheiten einer Oberfläche mittels Texturen simuliert. [vgl. URL Computer Graphics 2]

Ausgestattet mit Spielen wie „Pong", „Speed Race", „Gun Fight" und „Space Invaders" entwickelten sich bis Ende der 70er Jahre die Vorgänger heutiger Spielhallen. Während der 80er Jahre erkannte man das Potential von Computern für den Privatgebrauch, was die Nachfrage enorm ansteigen ließ. Die Entwicklung verlagerte sich von großen, zentralen Mainframes und Minicomputern zu selbständigen (Standalone) Workstations wie dem Orca 1000 (Orcatech, Ottawa). Grafikdesigner und Künstler begannen die Vorteile bei der Verwendung von Computern zu verstehen. Vor allem der Apple Macintosh (Abb. 1.18, heute „Mac") und der Commodore Amiga (Abb. 1.18.1) etablierten sich als ernst zu nehmendes Designwerkzeug.

1982 entwickelten Forscher der Universität von Osaka den LINKS-1 Supercomputer. Dieser bediente sich bis zu 257 Mikroprozessoren und war in der Lage, Bilder mittels Ray Tracing zu berechnen. Mit Hilfe des LINKS-1 wurde das erste computergenerierte Video des gesamten Sternenhimmels realisiert. Lucasfilm und Industrial Light & Magic erlangten große Bekanntheit im Bereich der Computergrafik. Animierte Musikvideos (Dire Straits – „Money for Nothing", Abb. 1.19) sowie CG Charaktere in Filmen (Young Sherlock Holmes, Star Wars) rückten diese - zuvor rein akademische - Disziplin ins Rampenlicht der Öffentlichkeit.

Neben dem Vormarsch der Computergrafik im Video- und Filmbereich explodierte außerdem die Spielebranche förmlich. Firmen wie Atari, Nintendo und Sega verkauften Millionen ihrer Systeme, Echtzeitrendering entwickelte sich weiter und erste Vorläufer der heutigen GPUs (Graphics Processing Unit) wurden geschaffen.

Computer für den Privatgebrauch hatten genug Leistung um Aufgaben zu übernehmen, welche vorher teuren Workstations vorbehalten waren. 3D Modellier-Software gewann an Aufmerksamkeit und erste nahezu fotorealistische Renderings wurden produziert. In Frankreich setzten sich animierte Fernsehserien durch (La Vie des bêtes, Les Fable Géométriques, Quarxs) und Pixar veröffentlichte 1995 seinen ersten außerordentlich erfolgreichen Filmhit „Toy Story" (Abb. 1.20). [vgl. URL Computer Graphics 1]

Im Bereich der Videospiele verhalf der Erfolg von Doom und Quake (Abb. 1.21 + 1.22) den neu entwickelten Game Engines zu großer Beliebtheit und Spielkonsolen wie die Sony Playstation oder Nintendo 64 wurden zu Millionen verkauft.

Ende der 90er hatten sich Nvidia und AMD als Platzhirsche im Grafikhardwarebereich durchgesetzt und trieben durch erste GPUs die Entwicklung im 3D Bereich weiter voran.

Die Entwicklung, die sich Ende der 90er abzeichnete, setzte sich in den 2000er Jahren in beschleunigter Form fort. GPUs in PCs zählten (dank günstigen low-end Alternativen) selbst im niedrigeren Segment zur Standardausstattung. Die Computergrafik in Spielen und Filmen entwickelte sich stark Richtung immer größerem Realismus. Während animierte Filme wie „Ice Age", „Madagascar" oder „Finding Nemo" große Erfolge einspielen konnten, stellten sich andere wie beispielsweise „Final Fantasy: The Spirits Within" (Abb. 1.23) oder „Der Polarexpress" als Misserfolg heraus. Die allgemein anerkannte Begründung dafür ist das Phänomen des uncanny valley (zu Deutsch: unheimliches Tal). [vgl. URL Computer Graphics 1]

Abb. 1.21: Doom

Abb. 1.22: Quake

Abb. 1.23: Final Fantasy

Uncanny Valley

Dieses Phänomen beschreibt eine Akzeptanzlücke, welche ab einem gewissen Realitätsgrad bei der Betrachtung von Robotern oder animierten Charakteren entsteht. Die Vertraulichkeit in einen Charakter wird zunächst mit steigendem Realitätsgrad größer. Ab einem gewissen Punkt fällt diese jedoch rapide ab, da die Ähnlichkeit zum echten Menschen bereits sehr hoch ist, aber trotzdem Unterschiede bestehen, welche für Menschen leicht wahrnehmbar sind. Die Akzeptanz steigt erst ab einem sehr hohen Realitätsgrad wieder an. Heutzutage werden als Hauptfiguren in animierten Filmen oft abstrakte oder absichtlich comichaft überzeichnete Charaktere verwendet, um den uncanny valley zu vermeiden. [vgl. URL Computer Graphics 1]

Im Bereich der Videospiele erfreuten sich sowohl Konsolen (Playstation, Xbox, Gamecube,...) als auch Windows PCs großer Beliebtheit. Titel wie Grand Theft Auto, Assasin's Creed oder Bioshock trieben die Entwicklung im Grafikbereich weiter voran. Der Realismus vorberechneter Animationen für Filme hat mittlerweile ein nahezu vollkommen fotorealistisches Niveau erreicht. Unterschiede zu echten Charakteren sind für das menschliche Auge praktisch nicht mehr erkennbar. Selbst bei Echtzeitdarstellungen ist der Realismus auf einem hohen Level angekommen. Mit entsprechender Hardware ist es möglich täuschend echte Szenen mittels Game Engines zu visualisieren. Render Engines sowie Shadermodelle und Effekte werden laufend weiterentwickelt, um die möglichen Ergebnisse weiter zu verfeinern. Jährlich kommt eine Vielzahl an animierten Filmen in die Kinos. Um den uncanny valley zu umgehen, setzt man immer noch vermehrt auf abstrakte Charaktere. Im Bereich der Videospiele beherrschen bisher die großen Hersteller Sony (Playstation 4), Microsoft (Xbox One) und Nintendo (Wii) den Markt. Auch der Windows PC stellt immer noch eine der beliebtesten Spiele Plattformen dar. [vgl. URL Computer Graphics 2]

Abb. 0.1

1.2 - Was ist eine Game Engine

Eine Game Engine ist eine Softwareumgebung (Framework) zur Entwicklung von Computerspielen. Sie besteht in der Regel aus mehreren Komponenten, die spezielle Aufgaben übernehmen. Der wichtigste Bestandteil ist die Render Engine, welche für die 2D beziehungsweise 3D Darstellung am Monitor verantwortlich ist. Außerdem sind oftmals weitere Komponenten für Physik, Ton, Animation, künstliche Intelligenz usw. vorhanden (Abb. 1.24) [vgl. URL Game Engines 2].

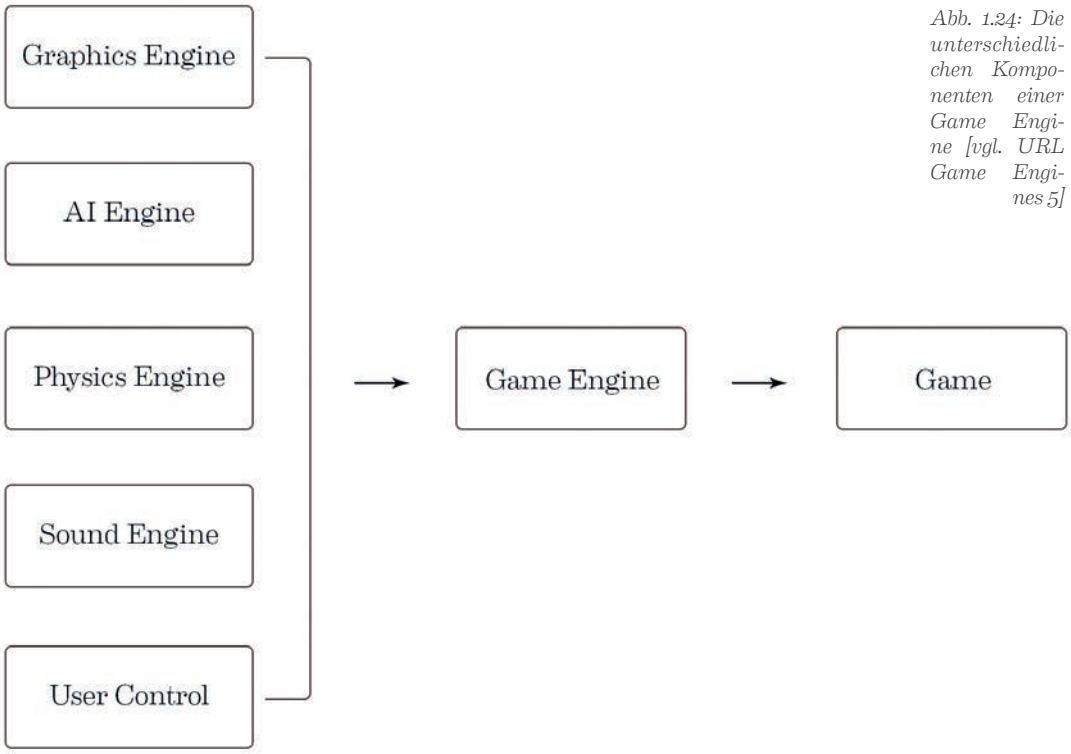


Abb. 1.24: Die unterschiedlichen Komponenten einer Game Engine [vgl. URL Game Engines 5]

Game Engines wurden in den 80er und 90er Jahren mit dem Ziel entwickelt, die Erstellung von Spielen ökonomischer zu gestalten. Anstatt für jedes einzelne Spiel den kompletten Quellcode neu schreiben zu müssen, konnte man sich bereits vorhandener Komponenten bedienen und mit einer einzigen Engine viele verschiedene Spiele kreieren. Obwohl Game Engines ihren Durchbruch erst Anfang der 90er erlebten, gab es bereits in den 80ern sogenannte „game creation systems", die in ihrer Funktionsweise stark den späteren Engines ähneln.

Zu den bekannteren zählen z.B. Pinball Construction Set (1983), ASCII's War Game Construction Kit (1983), Arcade Game Construction Kit (1988) und ASCII's RPG Maker (1988) (Abb. 125). Der Begriff der Game Engine entstand im Jahr 1991 und wurde stark durch ID Softwares Doom und Quake geprägt. Durch den großen Erfolg dieser beiden Titel beschlossen andere Entwickler Lizenzen für Teile der verwendeten Engine zu erwerben und für ihre eigenen Spiele wiederzuverwenden. Sie erstellten lediglich eigene Inhalte wie Grafiken, Charaktere, Waffen oder Levels. Durch diese Separation von Engine und Spielinhalt wurde eine viel stärkere Spezialisierung möglich. Heute ist es üblich, dass ein Entwicklerteam viel mehr Künstler als Programmierer beinhaltet. [vgl. URL Game Engines 3&4]

Oft werden auch einzelne Komponenten individuell kombiniert um die bestmögliche Lösung zu schaffen. Manche dieser Komponenten (auch „Middleware" oder „Software Development Kits" genannt) sind nur für eine einzige Sache programmiert, um eine besonders hohe Qualität auf diesem Bereich zu garantieren. Zu den bekanntesten Paketen gehören Bink (Video Rendering), FMOD (Audio Bibliothek), Havok (Physik Simulation, Abb 1.26) und GFx (Flash User Interface). Havok beispielsweise ist ein - von der zu Microsoft gehörenden irischen Firma Havok entwickeltes - Physik-Software-Development Kit. Die Hauptfunktionen bestehen aus Charakter Animation mit sogenannten Ragdoll Effekten, Simulation von Fahrzeugbewegungen und der Simulation physikalischer Effekte in der Umwelt (z.B. umfallende Kisten). [vgl. URL Game Engines 2]

Abb. 1.25: Game Creation Systems

Abb. 1.26: Anwendung der Havok Engine in Spielen.

1.3 - Was ist eine Echtzeit Engine

Die Begriffe Game Engine und Echtzeit Engine (auch Realtime Renderer oder GPU Renderer genannt) werden oft synonym gebraucht, da der Rendervorgang bei Game Engines in Echtzeit erfolgt. Tatsächlich handelt es sich aber bei Echtzeit Renderern um Software, welche meist zur Erstellung von virtuellen Szenen für Architektur- oder Produktvisualisierung verwendet wird. Einige Beispiele sind Octane (Otoy), Showcase (Autodesk) oder Vray RT (Chaosgroup) [vgl. URL Echtzeit Engine]. Sie sind anders aufgebaut als „echte" Game Engines, da ihr Fokus eben auf der Erstellung von Einzelbildern bzw. Animationen statt auf Applikationen liegt. Die Oberfläche und Bedienung ist mit der von klassischen Renderern vergleichbar. Ihr Name rührt daher, dass die Aktualisierung des Renderfensters sofort und mit mehreren Bildern pro Sekunde geschieht. Die Bildwiederholrate ist stark von der Komplexität der Szene und Leistungsfähigkeit der verwendeten Hardware abhängig, jedoch ist sie für eine flüssige Bewegung durch die dreidimensionale Welt für gewöhnlich zu niedrig. Um einen solchen vergleichsweise trotzdem schnellen Rendervorgang zu ermöglichen, bedienen sich Echtzeit Renderer – genau wie Game Engines - der Leistung der Grafikkarte (GPU – Graphics Processing Unit). Die grundlegenden Unterschiede zwischen GPU und CPU (Central Processing Unit) Rendering werden im nächsten Absatz besprochen. Da sich diese Arbeit mit der Verwendung von Game Engines für Architekturvisualisierung auseinandersetzt, sollen Echtzeit Render Engines hier nur am Rande erwähnt und nicht im Detail ausgeführt werden.

Abb. 1.27: Mit einer Echtzeit-Engine (Octane) erstellte Visualisierung.

1.4 - CPU vs. GPU

Die Hardware zur Erstellung von Visualisierung für Architektur, Film oder andere Medien hat sich in den vergangenen Jahrzehnten stark verändert. Heute unterscheidet man grundsätzlich zwei verschiedene Technologien zur Erstellung virtueller Bilder basierend auf Ray Tracing: die Berechnung mittels der CPU (der Prozessor, Central Processing Unit) oder mittels der GPU (des Grafikchips, Graphics Processing Unit). Grafikprozessoren haben sich erst in den letzten Jahren stark entwickelt. Beide sind unverzichtbare Bauteile eines Computers und bieten sowohl Vor- als auch Nachteile beim Rendern. Diese werden im folgenden Kapitel näher besprochen.

Entwicklung

Die frühesten Vorgänger heutiger GPUs entstehen bereits Ende der 60er Jahre bei Bell Labs. Es handelt sich dabei um sogenannte „Framebuffer", die es erstmals ermöglichen Code in Pixelform auf einem Monitor darzustellen. Anfang der 90er hat sich die Spieleindustrie bereits stark weiterentwickelt und es entstehen Schnittstellen (APIs, „Application Programming Interface") wie OpenGL oder DirectX, welche die Standardisierung stark vorantreiben. Anfang der 2000er Jahre haben sich zwei große Grafikprozessorhersteller etabliert: NVIDIA und ATi (heute AMD). Es folgen erste Algorithmen, die Raytracing auf GPUs ermöglichen, und später Renderer, welche sich der Leistung der Grafikkarte bedienen.

Optimierungsmöglichkeiten der Leistung von Prozessoren

Grundsätzlich gibt es drei Möglichkeiten die Leistung eines Prozessors zu steigern: Die Geschwindigkeit, die Effizienz und die Anzahl an Prozessoren (Abb.1.28).

Geschwindigkeit:

Die Geschwindigkeit (Taktfrequenz) von Prozessoren ist maßgeblich von der Größe und Anzahl der Transistoren auf der Platine abhängig. Aufgrund von physikalischen Einschränkungen beim Stromverbrauch und der Kühlung haben Geschwindigkeiten im Jahr 2005 ihr aktuelles Limit (~4GHz) erreicht. Aus diesem Grund bedarf es anderer Optimierungen um eine Leistungssteigerung zu ermöglichen.

Effizienz:

Die Effizienz von Prozessoren ist hauptsächlich vom Cache abhängig. Beim Cache handelt es sich um einen kleinen Speicher, der direkt am Prozessor liegt. In ihm werden Daten temporär gespeichert, bevor sie weiter bearbeitet werden. Jedoch wurden auch beim Cache mittlerweile die Limits erreicht, sodass eine Vergrößerung keine weitere Leistungssteigerung mit sich bringt.

Anzahl:

Die einzig verbleibende Methode zur Leistungssteigerung ist die Anzahl an Prozessoren, die parallel arbeiten, zu erhöhen. Um ein entsprechendes Ergebnis zu erzielen, ist es notwendig, dass Software für die Nutzung von mehreren Prozessorkernen

auch programmiert ist. Manche Aufgaben eignen sich besser für die parallele Berechnung auf mehreren Prozessoren, als andere.

Abb. 1.28: Schnellere, effektivere und parallele Arbeitsweise.

Unterschiede in der Arbeitsweise von CPUs und GPUs

Die zwei wichtigsten Einflussgrößen bei der Verarbeitung von Prozessen sind die Latenzzeit und der Durchsatz. Die Latenzzeit beschreibt die Dauer, die bis zum Erhalt bestimmter Information verstreicht. Der Durchsatz hingegen bestimmt die Menge an Information, welche in einer bestimmten Zeit transportiert werden kann. Eine Möglichkeit den Durchsatz zu erhöhen liegt demnach darin, die Latenzzeit zu verkürzen, damit Information schneller zur Verfügung steht. Ist das nicht möglich, kann man alternativ mehr Information gleichzeitig (parallel) transportieren (Abb.1.29).

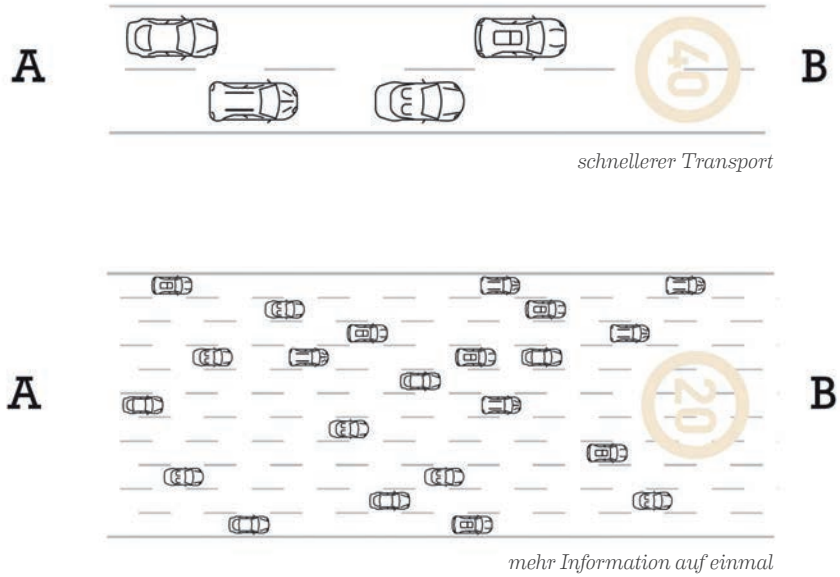


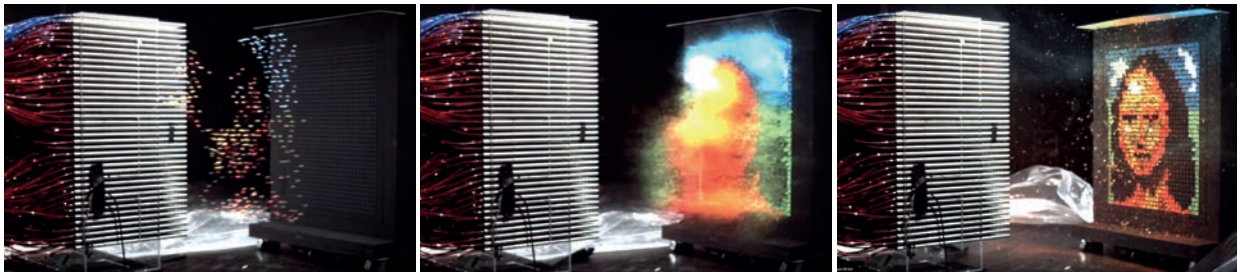
Abb. 1.29

CPUs sind für sehr kurze Latenzzeiten und einzeln nacheinander ausgeführte Aufgaben optimiert. GPUs hingegen sind für die parallele Berechnung vieler - voneinander unabhängiger – Prozesse entwickelt. Ihre Leistung ist, verglichen mit denen von CPUs, relativ gering, allerdings sind sie durch die hohe Bandbreite in der Lage, sehr viele Berechnungen gleichzeitig durchzuführen. Das macht sie perfekt für Aufgaben wie z.B. Ray Tracing, wo viele unabhängige Strahlen zur selben Zeit berechnet werden müssen.

Abb. 1.30: Abstrakte Darstellung der Berechnungsvorgänge bei GPUs. Viele einzelne Prozesse laufen gleichzeitig ab und bilden ein Gesamtergebnis

Abb. 1.32 + 1.33: Mittels GPUs berechnete Visualisierungen

Abb. 1.30



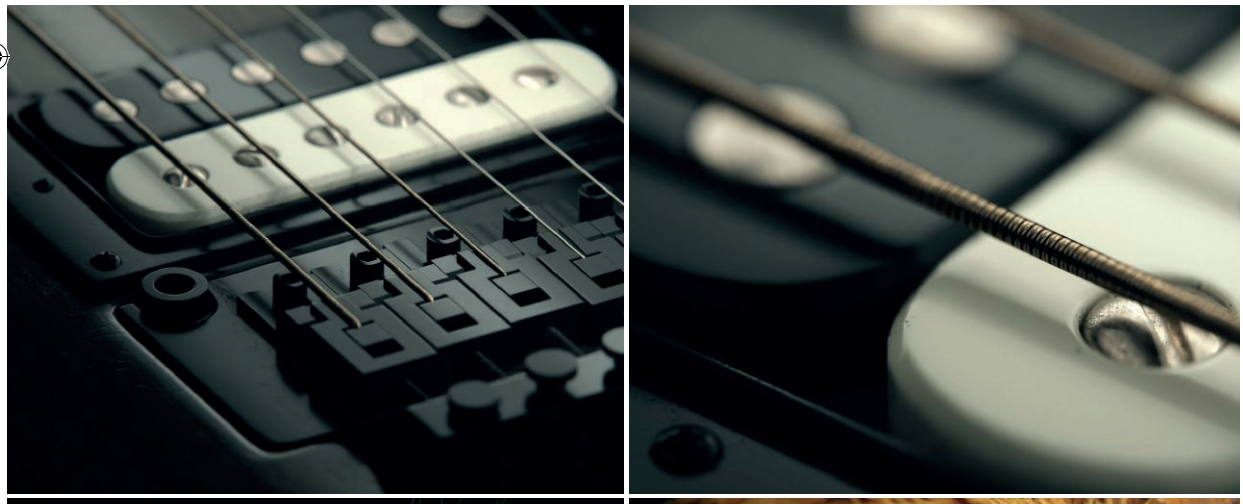


Abb. 1.32

Abb. 1.33

Transistorgrößen

Die Größe der Transistoren beeinflusst maßgeblich die Leistungsfähigkeit der Prozessoren. Kleinere Transistoren sind nicht nur schneller, sondern benötigen auch weniger Energie. Außerdem ist es möglich größere Mengen auf einem Chip unterzubringen. In den letzten Jahren hat sich das 28nm Verfahren bewährt. Ein Umstieg auf 16 oder 14nm wird wahrscheinlich in naher Zukunft erfolgen.

Vor- und Nachteile

Vorteile beim Rendern mittels CPU

- CPU Renderer sind meist weiter entwickelt als GPU Renderer.
- Größere Flexibilität beim Ressourcen-Management. CPUs haben direkten Zugriff auf den Arbeitsspeicher. Dieser ist kostengünstig erweiterbar.
- Oftmals laufen CPU Renderer stabiler, da sie lange erprobt und weit entwickelt sind

Nachteile beim Rendern mittels CPU

- Eine Erhöhung der Prozessoranzahl ist oft schwierig und teuer, da die meisten Motherboards nur einen oder zwei Steckplätze aufweisen. Um eine Renderfarm zusammenzustellen, bedarf es in der Regel mehrerer Einzelrechner. Im Gegensatz dazu bieten alle modernen Mainboards Platz für mehrere Grafikkarten.
- CPUs verwenden oft viel Leistung, um kurze Latenzzeiten zu ermöglichen. Diese sind aber bei parallel laufenden Berechnungen (z.B. Raytracing) nicht notwendig und verschwenden dadurch Ressourcen.

Vorteile beim Rendern mittels GPU

- Eine Erhöhung der Prozessoranzahl ist einfacher, da auf modernen Motherboards ausreichend Steckplätze vorhanden sind. Außerdem erfolgt die Skalierung linear. 2 Grafikkarten liefern also doppelt so schnelle Ergebnisse, 8 Grafikkarten nahezu 8-mal so schnelle.
- GPU Renderer arbeiten oft um ein vielfaches schneller als CPU Renderer. Vergleicht man zwei preislich ähnliche Systeme, so kann ein GPU Renderer bis zu 15-mal schnellere Ergebnisse liefern.

Nachteile beim Rendern mittels GPU

- Im Gegensatz zu CPUs, die auf den Arbeitsspeicher zugreifen können, müssen GPUs mit ihrem eigenen Speicher auskommen. Dieser ist momentan noch vergleichsweise teuer und limitiert.
- Grafikkarten sind Treiber abhängig. Diese werden oftmals in kurzen Abständen aktualisiert, was zeitweise zu Problemen führen kann.

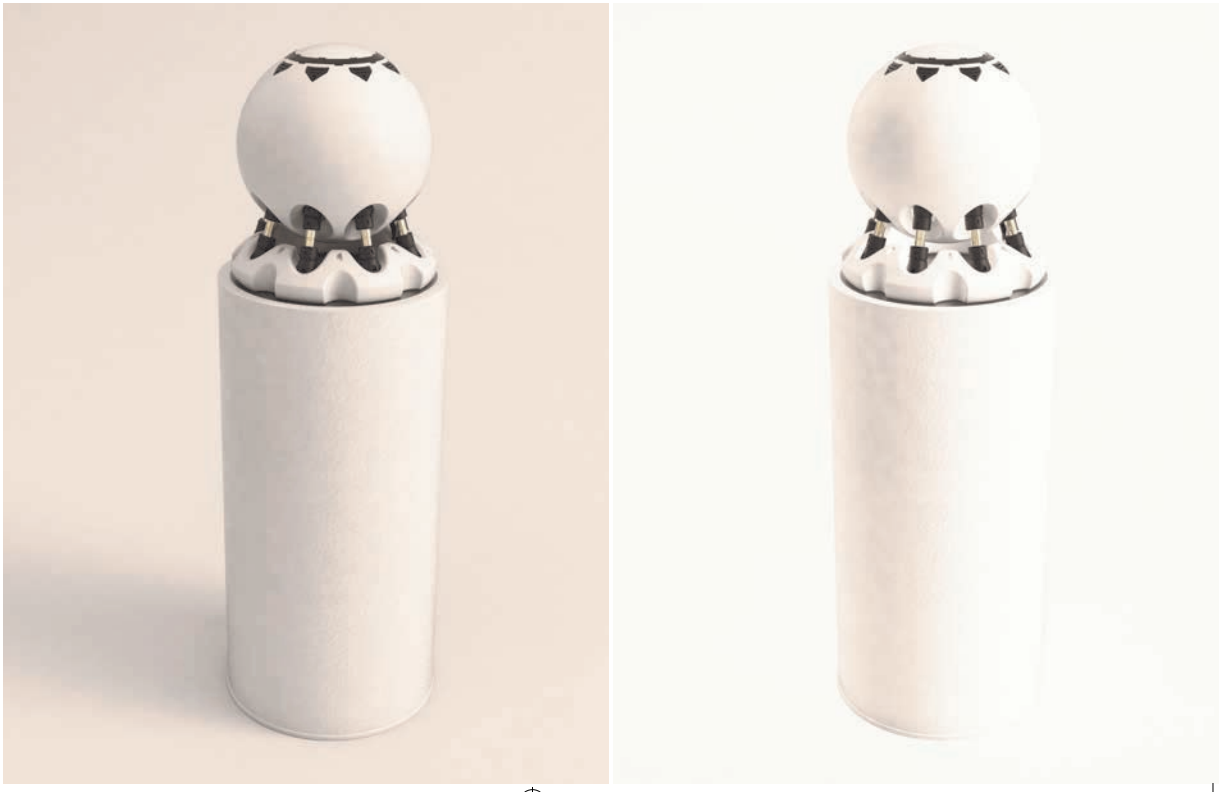
GPUs haben in den vergangenen Jahren stark an Bedeutung für Visualisierungen gewonnen. Dieser Trend wird sich voraussichtlich auch weiter entwickeln. Obwohl in vielen Fällen momentan immer noch die CPU die bessere Wahl darstellt, bieten moderne GPU Renderer durchaus viel Potential und Vorteile gegenüber traditionellen CPU Renderern [vgl. Chaosgroup „Guide to Gpu“ 2016]

1.5 - Status Quo und neue Möglichkeiten

Klassische Render Engines – jene bei denen Berechnungen nicht in Echtzeit erfolgen - haben sich im Bereich der Architekturvisualisierung in den letzten zwei Jahrzehnten zum Industriestandard entwickelt. Es handelt sich dabei meist um Render-Plug-Ins, die in bestehende 3D Software, wie zum Beispiel Cinema 4D (Maxon), 3DsMax (Autodesk) oder Rhinoceros3D (McNeel & Associates), implementiert werden und fotorealistische Visualisierungen ermöglichen.

Durch ihre physikalisch korrekten und präzisen Berechnungen bieten sie eine überragende Bildqualität. Für einen entsprechend erfahrenen Visualisierer ist es möglich überzeugende Bilder zu erstellen, welche von Fotos kaum beziehungsweiße überhaupt nicht mehr zu unterscheiden sind. Ihre Werkzeuge und Workflows sind ausgereift, vielfach getestet und gut dokumentiert, was den Einstieg für unerfahrene Nutzer wesentlich erleichtert. Weil viele Funktionen und Einstellungen an die reale Welt angelehnt sind, produzieren solche Engines weitgehend vorhersehbare Ergebnisse, welche vom menschlichen Auge erwartet werden, weil wir sie aus unserer tatsächlichen Umgebung kennen. Diese uns vertrauten und als „echt“ empfundenen Simulationen kommen dem Fotorealismus eines Bildes natürlich stark zugute. Gute Beispiele hierfür sind die von der Distanz zur Quelle abhängige Lichtstärke oder der Kontaktschatten zwischen zwei Objekten. Weichen solche Berechnungen von der Art und Weise ab, wie sie der Mensch aus der Realität gewohnt ist, wirkt das Bild sofort unecht und die Illusion des Fotorealismus geht verloren.

Abb. 1.35: Das Objekt im linken Bild ist richtig belichtet. Im rechten Bild fehlen die Schatten und der physikalisch korrekte Lichtabfall



Ein weiterer Vorteil von Render Engines ist die Vielzahl an unterschiedlichen Programmen, die heutzutage vorhanden sind. Viele spielen ihre Stärken in unterschiedlichen Bereichen aus und 3D-Künstler beziehungsweise Architekturbüros haben die Möglichkeit, die für ihre Bedürfnisse am besten geeignete Engine zu wählen. Wie erwähnt sind sie im Regelfall als Plug-In für bestehende 3D Software verfügbar, was den Umgang stark erleichtert, da man sich im gewohnten Umfeld bewegt. Das ermöglicht eine kontinuierliche Arbeitsweise innerhalb eines Programmes und macht den teilweise problematischen und zeitaufwendigen Austausch zwischen verschiedenen Programmen überflüssig. Zusätzliche Geometrie kann schnell modelliert beziehungsweise bestehende angepasst werden. Außerdem stehen zusätzliche Plug-Ins und Skripten zur Erleichterung bestimmter meist repetitiver Aufgaben zur Verfügung (Abb.1.36).

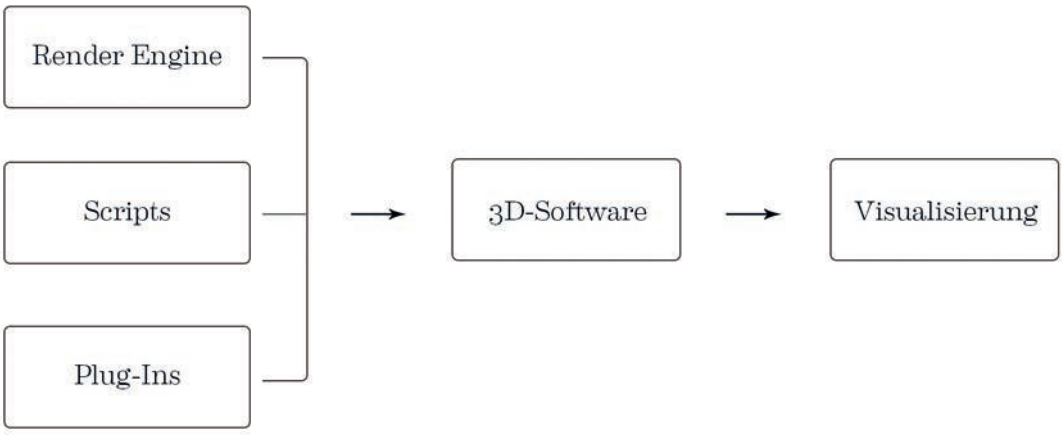


Abb. 1.36: Implementierung von Render-Engines, Scripts und weiteren Plug-Ins in bestehende 3D-Software

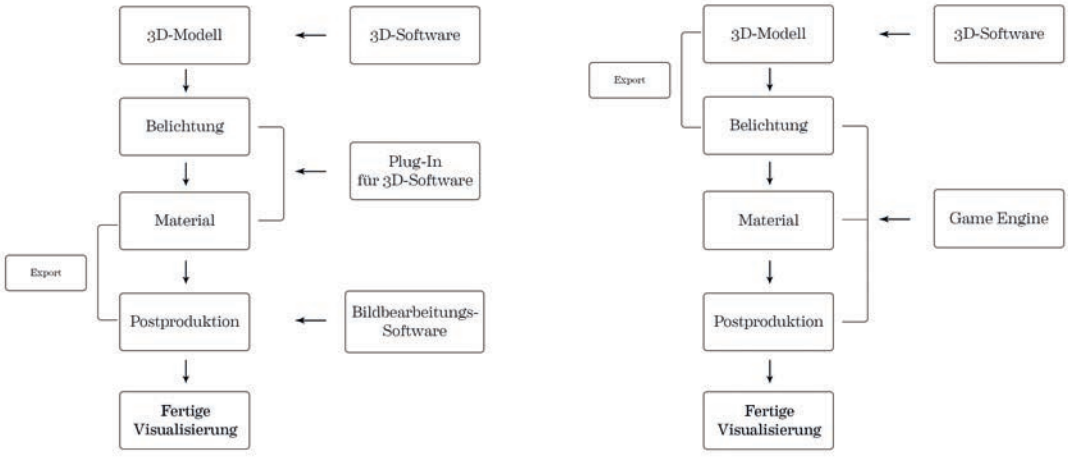


Abb. 1.37: Die einzelnen Schritte im Workflow erfolgen bei der Arbeit mit klassischen Render Engines oder Game Engines jeweils in unterschiedlichen Programmen

So gut etablierte Engines auch funktionieren, unterliegen sie trotzdem bestimmten Einschränkungen, welche Überlegungen zu alternativen Optionen nachvollziehbar machen. Als wichtigstes Kriterium sind hier die verhältnismäßig langen Renderzeiten und die damit einhergehende eingeschränkte Flexibilität zu nennen. Pauschale Aussagen über Berechnungszeiten sind seriös kaum möglich. Diese Schwanken stark in Abhängigkeit von Komplexität der Szene, Berechnungsqualität und verfügbaren Ressourcen. Trotzdem ist davon auszugehen, dass für die Berechnung eines einigermaßen umfangreichen Bildes in angemessener Qualität auf aktuell verfügbarer, für Architekturbüros rentabler Hardware mehrere Stunden benötigt werden. Diese Zeitspanne kann zu Vorschauzwecken durch Verringern der Qualität und Auflösung auf wenige Minuten oder gar Sekunden reduziert werden. Trotzdem ist es beim Einrichten einer Szene unumgänglich, immer wieder die Neuberechnung abzuwarten, um entsprechend auf das Ergebnis reagieren zu können. Dies trifft auf viele wichtige Teilbereiche einer Visualisierung, wie zum Beispiel Licht, Material oder Perspektive, zu. Dieser Umstand behindert oftmals flüssiges Arbeiten und schränkt solche Engines in ihrer Nutzbarkeit als Entwurfswerkzeug stark ein.

Diese Problematik ist – wenn auch nicht alleine – sicherlich mitverantwortlich für die deutlich seltenere Verwendung von Animationen zu Präsentationszwecken im Architekturbereich. Da für eine Sekunde Film eine Vielzahl an Einzelbildern benötigt wird, ist es in den meisten Fällen wirtschaftlich und zeitlich unmöglich, bewegte Szenen mit den vorhandenen Ressourcen zu erstellen. Etwas Abhilfe schaffen hier sogenannte Renderfarmen, welche den Zugriff auf die benötigte Rechenleistung ermöglichen. Allerdings ist dies mit weiteren Komplikationen im Workflow und nicht zuletzt auch mit Kosten verbunden.

Abb. 1.38: Um die Lichtsituation anzupassen ist mit gewöhnlichen Render Engines eine Neuberechnung notwendig. Bei Game Engines kann die Einstellung dynamisch (also in Echtzeit) erfolgen.

Die neuen Möglichkeiten, die Game Engines bieten

In diesem Bereich kommen die Vorzüge von Game Engines am stärksten zur Geltung. Ihr hohes Maß an Flexibilität macht sie sehr gut als Entwurfs und Präsentationswerkzeug nutzbar. Viele wichtige Änderungen – sei es an Licht, Material oder Geometrie – können sofort sichtbar gemacht werden, ohne die Szene neu berechnen zu müssen. Einstellungen an der Lichtstärke oder Lichtfarbe können in Echtzeit erfolgen und Visualisierer können das Ergebnis sofort beurteilen beziehungsweise ohne Verzögerung darauf reagieren (Abb. 1.39 + 1.40). In gleicher Weise ist es möglich Materialeigenschaften wie Farbe oder Spiegelung zu justieren und die Auswirkungen gleichzeitig in der Szene zu verfolgen. Unter gewissen Voraussetzungen können sogar Objekte interaktiv verändert oder neu positioniert werden. Zum Beispiel könnten vom Benutzer ganze Raumkonfigurationen innerhalb Sekundenschnelle ausgetauscht werden, um unterschiedliche Varianten betreffend Material, Licht oder Einrichtung zu testen.

Neben den verschiedenen Möglichkeiten die Szene in Echtzeit einzurichten und anzupassen, bieten Game Engines große Vorzüge bei der Navigation durch den virtuellen Raum. Nachdem das Projekt einmal vorberechnet wurde, kann es als Gesamtes erkundet werden. Räumliche Qualitäten werden dadurch besser erlebbar und Entwurfsentscheidungen können schneller und flexibler getroffen werden. Auch die eingestellten Materialien und die Lichtwirkung kann besser beurteilt werden, wenn man sich – wie in der realen Welt – frei im Raum bewegen kann. Im Vergleich dazu bieten Render Engines nur eine sehr eingeschränkte Editorvorschau, die eine präzise Beurteilung der Situation oft schwierig macht. Grundsätzlich ist durch das unmittelbare Feedback bei Game Engines eine intuitivere und schnellere Arbeitsweise möglich (Abb. 1.41 + 1.42).

Abb. 1.39: Mit Sonnenlicht

Abb. 1.40: Ohne Sonnenlicht

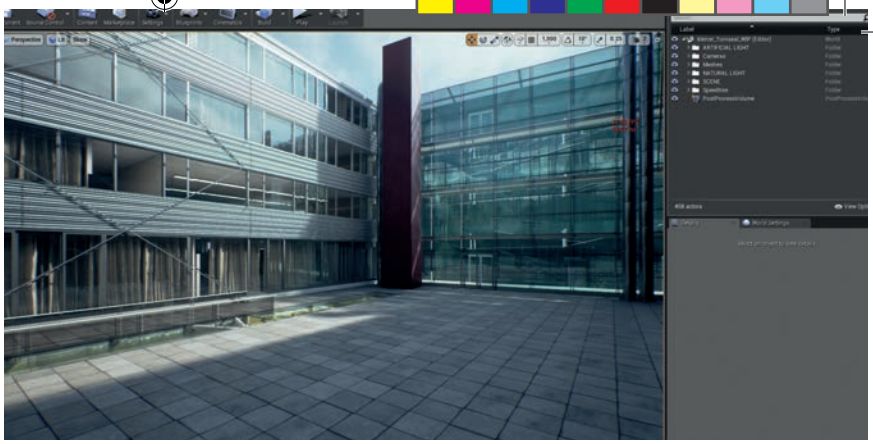


Abb. 1.41: Viewport in der Unreal Engine

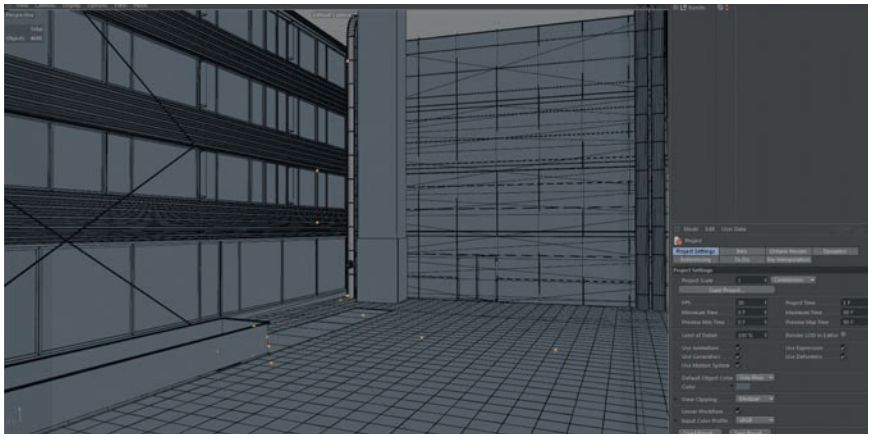


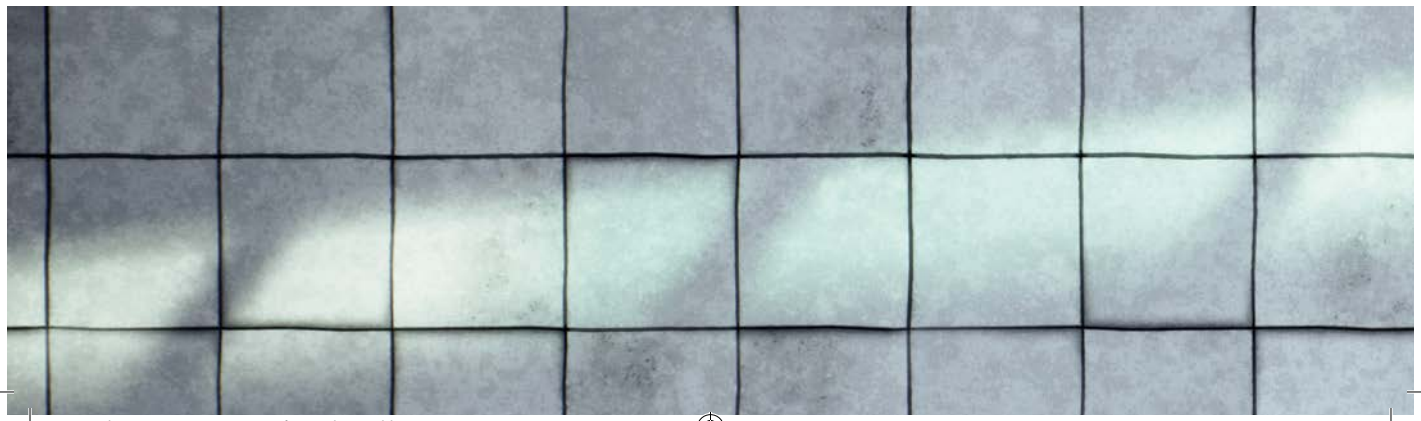
Abb. 1.42: Viewport in Cinema 4D

Gleich wie klassische Renderer unterliegen Game Engines – wenn auch in anderer Art und Weise – gewissen Einschränkungen, welche bedacht werden müssen. In erster Linie ist hier jedenfalls die zurzeit erreichbare fotografische Qualität der Visualisierungen zu nennen. Bedingt durch ihr eigentliches Einsatzgebiet der Spielerstellung sind Game Engines grundsätzlich immer auf einen Kompromiss aus physikalisch korrekten Berechnungen und guter Leistung ausgelegt. Um eine flüssige Darstellung mit 60 Bildern/Sekunde oder mehr zu erreichen, ist eine intelligente Nutzung der vorhandenen Ressourcen unbedingt notwendig. Unterzieht man einzelne Teilbereiche wie Spiegelungen oder Schattendarstellung einer mittels einer Game Engine produzierten Visualisierung einem direkten Vergleich mit jenen aus einer mittels einer normalen Engine erstellten Visualisierung, so zeigt sich, dass letztere genauere und realitätsgetreuere Ergebnisse produzieren kann. Nun setzt sich allerdings ein fertiges Bild aus einer Vielzahl solcher Einzelteile zusammen. Erst wenn sämtliche Einstellungen bei Materialien, Lichtquellen, bewegten und statischen Objekten, Kamerafahrten, Effekten und die Postproduktion abgeschlossen sind, ist eine sinnvolle Beurteilung möglich. Vergleicht man also zwei fertige Projekte in ihrer Gesamtheit, schrumpft der Qualitätsunterschied deutlich. Je nach Arbeitsaufwand und Erfahrungsniveau des Nutzers können die mit Game Engines erzeugten Visualisierungen jene aus bewährten Renderern in ihrem Realitätsgrad auch deutlich übertreffen.

Um sich frei im Projekt bewegen zu können, ist es bei Game Engines notwendig zunächst eine Vorberechnung der Global Illumination (Umgebungsbeleuchtung + indirekte Beleuchtung) durchzuführen. Diese erfolgt prinzipiell immer für das gesamte Modell. Eine Vorberechnung zu Testzwecken von lediglich einer Perspektive oder bestimmten Teilbereichen, wie es bei gewöhnlichen Engines gebräuchlich ist, ist

nicht möglich. Es ist also notwendig immer die Berechnung des gesamten Projektes abzuwarten, unabhängig von dem Ausschnitt an dem gerade gearbeitet wird. Je nach Projektgröße kann dies natürlich zu erheblichem Zeitverlust führen. Um dieser Problematik entgegenzuwirken, gibt es grundsätzlich die Möglichkeit die Bildqualität zu Vorschauzwecken stark zu reduzieren. Dadurch werden zwar Schatten und Licht weniger akkurat berechnet, allerdings ist dies für Testzwecke vollkommen ausreichend und die Berechnungszeit wird auf einen Bruchteil reduziert. Eine weitere effektive Methode stellen so genannte „importance volumes“ dar. Mit ihnen ist es möglich die Genauigkeit der Lichtberechnung innerhalb eines Projektes zu steuern. Während außerhalb der durch den Nutzer frei definierten Bereiche nur eine sehr oberflächliche und schnelle Lichtkalkulation erfolgt, werden die wichtigen Teile innerhalb dieser Volumen mit einer höheren Qualität berechnet. Durch die beschriebene Vorgehensweise lässt sich die benötigte Zeit für Vorschaubilder auf ein Minimum (je nach Szene bis auf wenige Minuten) reduzieren. Zwar ist es momentan nicht möglich schnelle Testbilder, wie mit anderen Render Engines, innerhalb von wenigen Sekunden zu erstellen, allerdings können viele Einstellungen nach einmaliger Vorberechnung in Echtzeit durchgeführt werden und machen ein neuerliches Rendern überflüssig.

Die Tatsache, dass Game Engines nicht als Plug-In, sondern mit einer eigenständigen Benutzeroberfläche verfügbar sind, bietet Vorteile in der Bedienung und ist für viele Anwendungsbereiche notwendig und sinnvoll. Jedoch werden dadurch zusätzliche Schritte im Workflow nötig, welche zu Komplikationen oder Fehlern führen können. Beispielsweise sind für die Belichtungsberechnung die Texturkoordinaten von Geometrie ausschlaggebend. Werden diese zuvor in einer externen 3D Anwendung nicht korrekt ausgelegt, erscheinen die Objekte später in der Game Engine vollkommen schwarz, da sie kein Licht reflektieren. Auch beim Export und Import zwischen den unterschiedlichen Programmen kann es zu Fehlern kommen, wenn nicht auf die korrekte Skalierung und Position des 3D Modells geachtet wird. Zusätzlich sind verschobene Achsen oder Nullpunkte häufig Ursachen von Problemen. In Anbetracht dieser Tatsachen ist klar, dass jede zusätzliche Anwendung, die im Workflow verwendet wird, diesen komplexer und fehleranfälliger gestaltet. Glücklicherweise sind viele Werkzeuge und Skripte vorhanden, welche notwendige Schritte vereinfachen und das Fehlerpotenzial stark reduzieren. Für erfahrene Nutzer ist somit durchaus eine flüssige und vorhersehbare Arbeitsweise mit Games Engines im Bereich der Architekturvisualisierung möglich. Die erstmalige Einrichtung der Szenen nimmt zwar etwas mehr Zeit in Anspruch, allerdings wird diese im späteren Arbeitsverlauf durch die beschriebenen Stärken solcher Renderer mit Leichtigkeit wieder aufgewogen.



1.6 - Anwendungsszenarien von Game Engines

Bedenkt man die Vielzahl an verschiedenen Navigationsarten, wird auch die Bedeutung von Game Engines für Präsentationszwecke im Architekturbereich deutlich. Wie bei der klassischen Architekturvisualisierung mit normalen Render Engines können jederzeit Einzelbilder aus der vorhandenen Szene generiert werden. Der große Vorteil ist, dass diese dank der Möglichkeiten aktueller Grafikkarten innerhalb von Sekunden und in beliebiger Auflösung erstellt werden. Natürlich ist die Nachbearbeitung solcher Bilder auch in gängigen Bildbearbeitungsprogrammen möglich.

Neben dem klassischen Standbild können auch Kamerafahrten schnell und intuitiv in Echtzeit aufgezeichnet werden. Die Kameras dazu werden als Objekte positioniert beziehungsweise animiert – sämtliche Änderungen an der Szene werden in Echtzeit in allen eingestellten Perspektiven sichtbar. Die einzelnen Clips können mittels integrierter Werkzeuge zu längeren Sequenzen zusammengefügt und bearbeitet werden. Ein weiterer Export in Drittanwendungen ist somit nicht notwendig.

Alternativ sind die freie Bewegung oder die Erkundung mittels eines Charakters möglich. Je nach Einstellung kann die Kamera frei durch das gesamte Projekt und die Geometrie bewegt werden, oder bestimmte Geometrieelemente (Wände, Decken, Türen...) werden als Kollisionsobjekt definiert und ein Durchgang somit verhindert. Diese Navigationsarten eignen sich besonders als Entwurfswerkzeug, da die Umgebung schnell und frei betrachtet werden kann. Für Präsentationen sind sie nur mit Einschränkungen geeignet, da externe Personen schnell in Bereiche des 3D Modells vordringen können, die nicht detailliert genug ausgearbeitet sind, um überzeugende Darstellungen zu bieten. Alternativ können mit Hilfe so genannter „points-of-interest" die möglichen Standpunkte innerhalb einer Umgebung präzise definiert werden, während dem Kunden lediglich die Bedienung der Blickrichtung überlassen wird. Da die Fähigkeit die eigene Bewegung innerhalb einer virtuellen Welt zu kontrollieren starken Einfluss auf den Nutzer haben kann, sind solche Navigationsarten fallweise durchaus für Präsentationszwecke überlegenswert.

Unabhängig von der Präsentationsart kann das Erlebnis durch die Anwendung von Virtual Reality Headsets stark intensiviert werden. Bislang kommen Versuche dieser Art meist aus Richtung der Unterhaltungsindustrie. Auch in der Auto- oder Möbelbranche finden sich vereinzelt Beispiele für die Nutzung von Game Engines in Kombination mit Virtual Reality Brillen. Außerdem sind zukünftig auch Anwendungen für Ausstellungen, Messen oder im Bereich der Werbung und des Marketing denkbar. Eine tiefer gehende Beschreibung der erwähnten Möglichkeiten soll zu einem späteren Zeitpunkt innerhalb dieser Arbeit erfolgen.

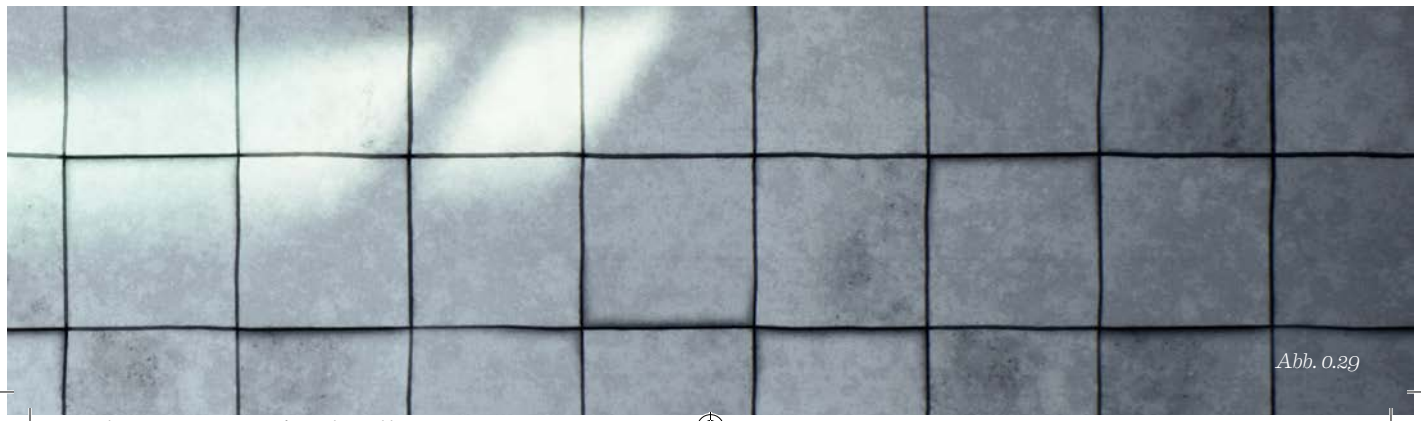


Abb. 0.29

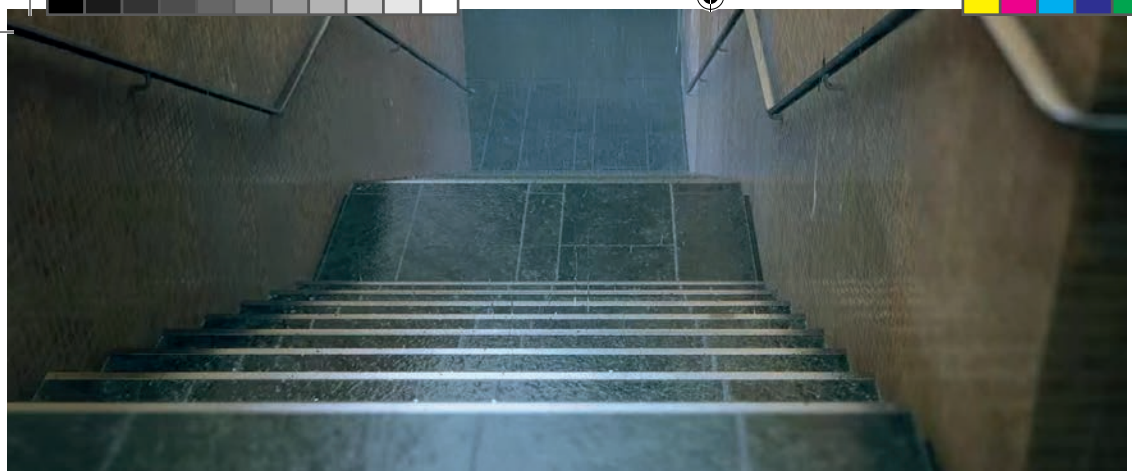


Abb. 1.43 - 1.46: In einer Game-Engine verwirklichte Projekte.

Abb. 1.43

Abb. 1.44

Abb. 1.45

Abb. 1.46

1.7 - Bisher mit Game Engines im Bereich der Architektur verwirklichte Projekte

„A buyer walks into your office. He says he's interested in a particular neighborhood. Instead of driving him through that area, you sit down with him at your computer. You press a few buttons, and a map of the neighborhood appears. [...] The buyer points to one of the homes on the map, and you press another button. Suddenly, the screen shows a close-up of the front of that house. ‚Here you go,' you say, handing the computer mouse to the buyer. ‚Take a walk through the house.'„ (Warren Berger, 1994)

Bereits in den frühen 90er Jahren - kurz nach Vorstellung der ersten Game Engines – wurde dieses zukünftige Szenario von Warren Berger prophezeit. Fünf Jahre später untersucht Vito Miliano in seiner 1999 veröffentlichten Arbeit die Möglichkeiten zur Anwendung von Game Engines in der Immobilienbranche. Als konkrete Problemstellung der damals aktuellen Situation nennt er in erster Linie den hohen Zeit- und damit verbundenen Kostenaufwand zur Erstellung von Visualisierungen. Obwohl die Vorteile der Nutzung von virtuellen 3D Modellen und Szenen in allen Bereichen – vom Entwurf bis zur Präsentation – bereits damals dokumentiert sind, wären lange Renderzeiten und die unflexible Arbeitsweise von Render Engines die Hauptgründe für die geringe Akzeptanz dieser Werkzeuge im Umfeld der Architektur. Gerade in frühen Phasen – wenn mit zahlreichen und weitreichenden Änderungen zu rechnen ist – würden deshalb hauptsächlich konzeptuelle Skizzen oder physische 3D Modelle verwendet werden. Die Lösung sieht Miliano in der Anwendung von Software, die damals seit fast einem Jahrzehnt in der Spielindustrie verwendet wird, nämlich Game Engines. Er nennt vier Hauptgründe, welche für die Nutzung solcher Programme sprechen: die zuverlässige (weil vielfach erprobte) Arbeitsweise, die Aktualisierung der Szenen in Echtzeit, die Leistungsfähigkeit aktueller Engines und die neuen Möglichkeiten, welche sich für Präsentation ergeben.

Vito Miliano beschreibt in seiner Arbeit weiters das Projekt „Unrealty" – eine Anwendung, welche Immobilienbüros erlaubt virtuelle Rundgänge durch die zum Verkauf stehenden Immobilien anzubieten. Kunden können über Einblendungen mit Informationen versorgt werden, die umgebende Landschaft wird (für damalige Verhältnisse) realistisch dargestellt und dynamische Tag/Nacht-Wechsel sind möglich. Das Programm basiert auf der ersten Unreal Engine und erlaubt auf einem Mittelklasse PC flüssige Darstellung. Abschließend stellt der Autor fest, dass die Anwendung einer Game Engine und die damit einhergehende Flexibilität in der Arbeits- und Präsentationsweise einen immensen Mehrwert gegenüber „klassischen" CAD Programmen bietet. Ergänzt man die Software zusätzlich mit – auf den jeweiligen Nutzer zugeschnittenen – Funktionen, heben sich solche Werkzeuge noch deutlicher von allen bis zu diesem Zeitpunkt verfügbaren Systemen ab [vgl. URL Vito Miliano].

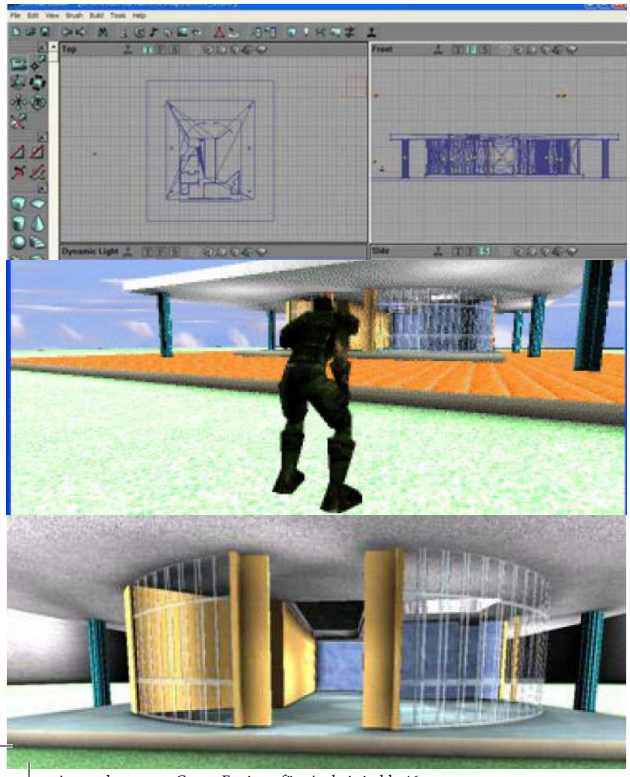
Abb.: 1.47: Die Anwendung „Unrealty" erlaubt Immobilienbüros virtuelle Rundgänge durch Immobilien anzubieten

Drei Jahre später widmen sich Fairuz Shiratuddin und Walid Thabet vom Department of Building Construction an der Virginia Tech ebenfalls der Anwendung von Game Engines im Bereich des Bauwesens. Im Gegensatz zu Miliano, der den Fokus auf Verkaufspräsentation legt, konzentrieren sie sich auf die Möglichkeiten, welche solche Anwendungen während der Entwurfsphase eines Projektes bieten. Anhand zur Planung stehender Büroräumlichkeiten erläutern sie die Problematik in der ausschließlichen Nutzung „traditioneller" Pläne. Sie wären oft nicht eindeutig interpretierbar oder lesbar, viele Beteiligte hätten nicht die entsprechende Erfahrung um konkrete Vorstellungen aus den zweidimensionalen Zeichnungen zu entwickeln. Visualisierungen in einer Echtzeitumgebung würden die interdisziplinäre Arbeit fördern und allen Mitwirkenden Hilfestellung bei Entscheidungen bieten, welche die Projektziele betreffen.

Auch Shiratuddin und Thabet sehen die Möglichkeit eines Feedbacks in Echtzeit, sowie die Leistungsfähigkeit und den erreichbaren Realitätsgrad als größte Vorteile. Weiters nennen sie auch die Kollisionsdetektion und Interaktivität von Game Engines als herausragendes Merkmal.

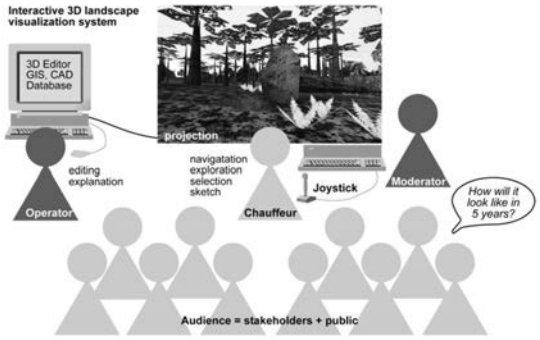
„Game engines and its design concept has [sic] *the capability to represent a realistic virtual environment in real-time. In the construction industry alone, it can generate real-time VR applications that can represent architectural walkthroughs, 4D planning, virtual pre-construction planning processes and many more. The 3D game engine also offer low-cost VR solution* [sic] *with very outstanding built-in features such as multi-participant capabilities, collision detection, higher frame rates per second and still only requires entry level hardware"* (Shiratuddin/Thabet, 2002)

Sie erkennen in der Anwendung von Game Engines für Bauprojekte immense Vorteile für alle Beteiligten, seien es Architekten, Ingenieure, Baufirmen oder -herren. Auch sollten Entwicklungen in der Spielindustrie von der Bauindustrie nicht außer Acht gelassen werden, da beide Branchen voneinander lernen und „fremde" Werkzeuge zu ihren Gunsten nutzen könnten [vgl. Shiratuddin/Thabet, „Virtual Office Walkthrough Using a 3D Game Engine", 2002].



v.o.n.u.:
Abb. 1.48: Import des 3D Modells in den Unreal Editor
Abb. 1.49: Testen von 3D Charakteren zur Erkundung der Szene
Abb. 1.50: Fertig belichtetes Projekt

Abb. 1.51: Von Herwig & Paar getestete Arbeitsgruppensituation



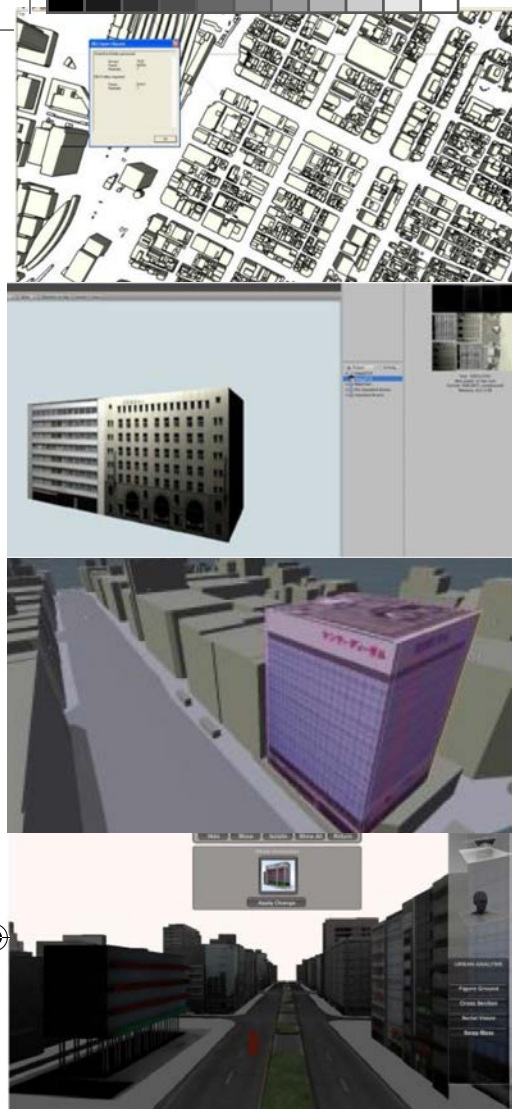
Dieses Argument machen sich auch Adrian Herwig und Philip Paar zu nutze, als sie 2004 ihre Arbeit über die Nutzung von Game Engines für Landschaftsplanung veröffentlichen.

„The fast growing market of computer games forces the development of constantly improved software and increasingly powerful hardware. Meanwhile many of the computer games can simulate virtual environments, e.g. synthetic landscapes, extremely close to reality in real-time on PCs or game consoles. This rapid development in computer game technology is almost unnoticed by the users of professional CAD-, GIS-, and illustration software." (Paar/Herwig, 2004)

Die Autoren gehen der Frage nach, ob die Nutzung von Software aus dem Bereich der Spielindustrie für Landschaftsplaner – zum Beispiel bei der interdisziplinären Planung - von Vorteil sein könnte. Sie beleuchten, wie die Planer bzw. deren Prozesse von solchen Anwendungen profitieren und ob die Technologie in bestehende Arbeitsweisen eingebunden werden kann. Dies geschieht anhand eines konkreten Projektes in Strausberg, östlich von Berlin. Eine Machbarkeitsstudie soll die Anwendbarkeit von Game Engines in diesem Bereich belegen. Strausberg soll ökologische Eingriffe erfahren, welche den durch Bautätigkeit bedingten Verlust von natürlichem Lebensraum kompensieren sollen. Herwig und Paar beschreiben, wie die unterschiedlichen Entwürfe in einer Arbeitsgruppe präsentiert und diskutiert wurden. Neben „klassischen" Präsentationswerkzeugen, wie Plänen, Skizzen und Fotos, kamen mittels einer Game Engine erstellte Visualisierungen zum Einsatz. Bei einer anschließenden Befragung aller Beteiligten wurde eine hohe Akzeptanz, sowie großer praktischer Nutzen der virtuellen Darstellungen deutlich. Die Autoren sehen in Game Engines eine interessante und kostensparende Anwendung im Bereich der Landschaftsplanung. Lediglich das Fehlen von Funktionen, wie eines GIS- oder Datenbankinterface, die in professioneller Planungssoftware zu finden sind, werden von ihnen kritisiert. Auch der zur damaligen Zeit erreichbare Realismus in der Darstellung der Vegetation wird als Schwachpunkt beschrieben [vgl. Paar/Herwig, „Game Engines: Tools for Landscape Visualization and Planning?", 2004].

Einen weiteren Versuch Game Engines in die Entwicklung von Architekturprojekten zu implementieren, unternehmen 2009 Aswin Indraprastha und Michihiko Shinozaki vom Shibaura Institute of Technology in Japan. Sie untersuchen unterschiedliche Anwendungen und realisieren eine städtebauliche Studie mit Hilfe einer Game Engine.

„Design visualization is needed, not only as the culmination of design development, but as the input for design strategy as well." (Indraprastha/Shinozaki, 2009)



v.o.n.u.:
Abb. 1.52: Erstellung des 3D Modells
Abb. 1.53: Texturierung der Gebäude
Abb. 1.54+Abb. 1.55: unterschiedliche Perspektiven, die vom Nutzer eingenommen werden können

Sie untersuchen die Vor- und Nachteile der Software hinsichtlich ihres Einsatzes während der Entwicklungsphase von Projekten.

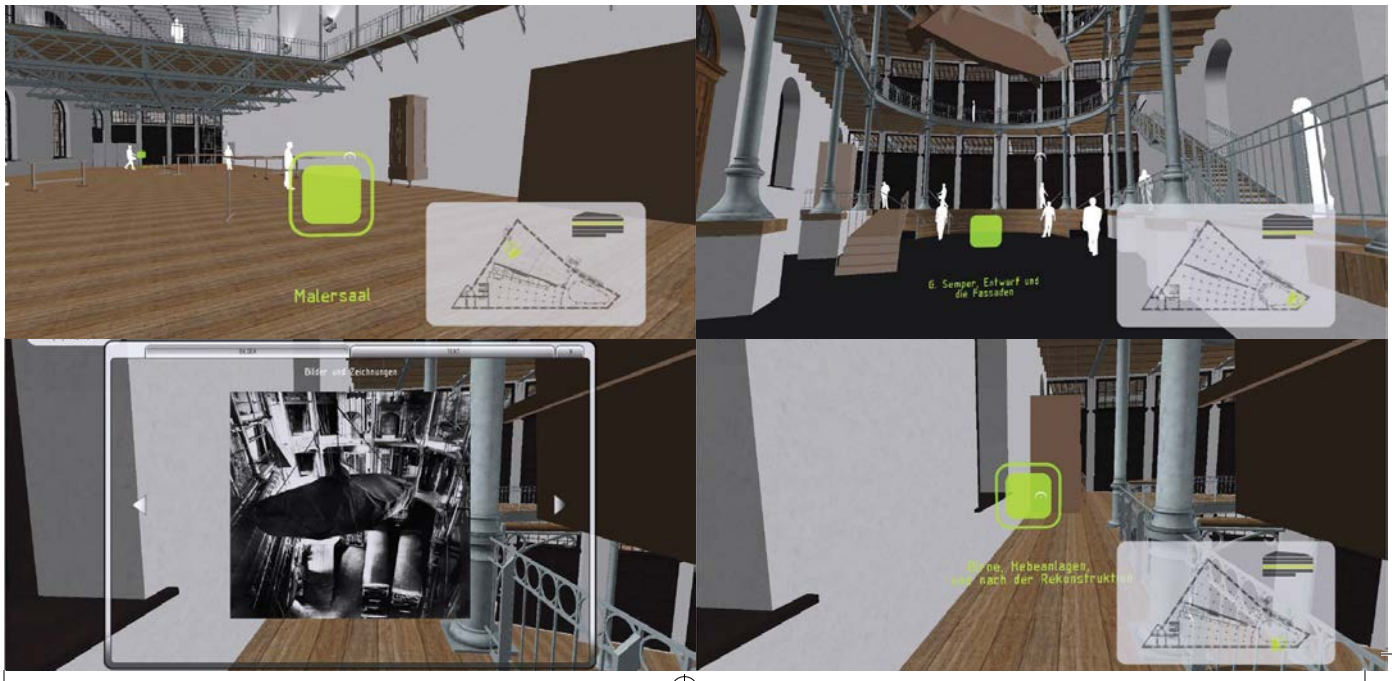
„Game Engine based environments on a particular urban area, could be a tool to assist 3D visual simulation for policy makers and every stakeholder in a design study and process. [...] Modern game engines offer high speed and rendering quality, interactivity and multi-user support, that are difficult to obtain using any existing visualization tools."

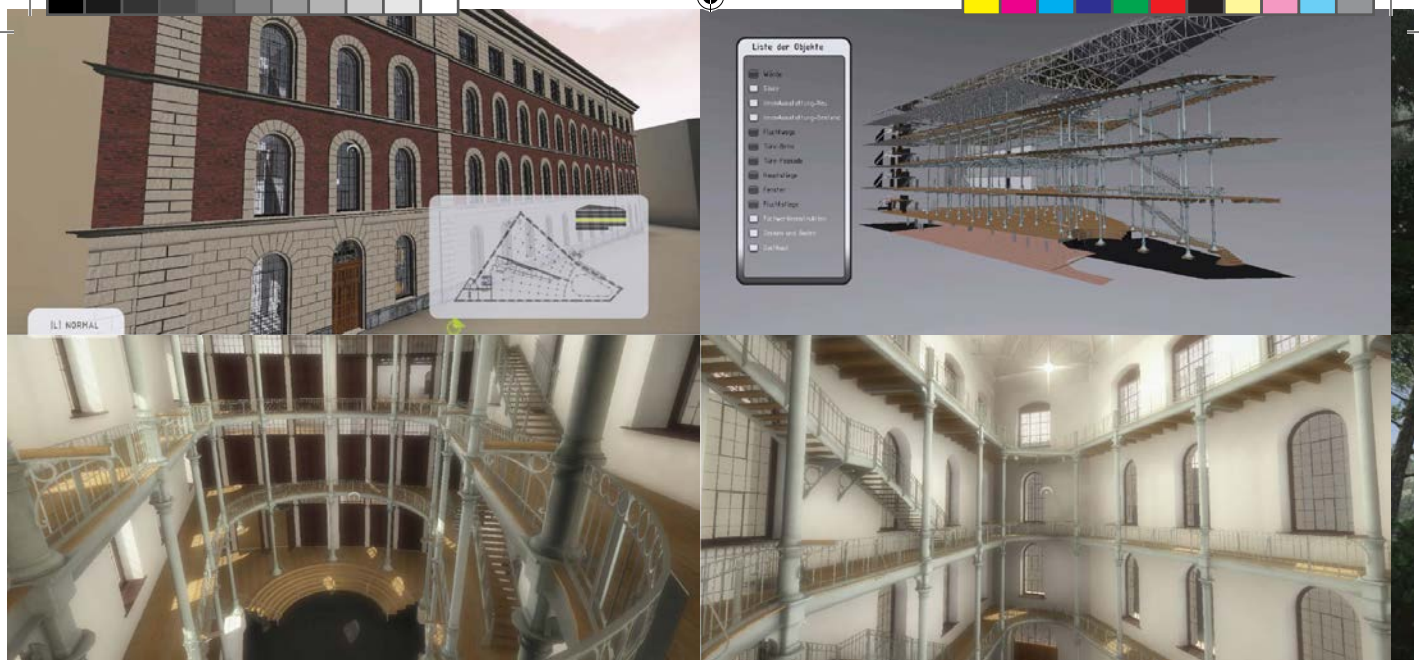
Die neuen Navigations- und Interaktionsmöglichkeiten werden von ihnen als ausschlaggebendes Kriterium für die Anwendung solcher Programme beschrieben. Anhand einer städtebaulichen Studie im Zentrum Tokyos (Bezirk Yeasu, ˜220.000m2 Fläche) kommen sie zu dem Schluss, dass Game Engines hohes Potential hinsichtlich der möglichen Interaktivität auf dem Gebiet des Städtebaus bieten. Als limitierende Faktoren werden hauptsächlich die fehlende Implementierung von CAD-Werkzeugen, wie sie aus 3D Modellierprogrammen bekannt sind, genannt [vgl. Indraprastha/Shinozaki, „The Investigation on using Unity3D Game Engine in Urban Design Study", 2009].

2012 verfasst Michal Rontsinsky eine Arbeit über Echtzeit Rendering-Techniken für die Planung und Visualisierung von Architektur-Projekten. Er untersucht die Anwendungsmöglichkeiten von Echtzeitvisualisierungen im Bereich der Architektur. Mit einer interaktiven Applikation des Semper Depots in Wien zeigt er die Einsatzmöglichkeiten von Ga-

me-Engines für Architekten auf. Neben einer theoretischen Auseinandersetzung mit Echtzeitvisualisierung, beschreibt er in seiner Arbeit den angewandten Workflow von der Erstellung des Modells in 3DsMax über den Export in die Unity Game Engine bis hin zur fertigen Applikation. Diese soll eine interaktive virtuelle Rekonstruktion des Semper Depots schaffen, die dem Betrachter Informationen auf mehreren Ebenen zur Verfügung stellen soll. Die Themen reichen von der Geschichte über die Konstruktion, Gestaltung und Raumordnung bis zur Nutzung und Veränderung des Gebäudes. Um die Vielzahl an Informationen transportieren zu können, ist die Anwendung in mehrere Bereiche gegliedert. Über das Hauptmenü können die unterschiedlichen Funktionen aufgerufen werden. Mittels der „Walkthrough-Funktion" kann der Nutzer das virtuelle Modell aus einer „First-Person-Perspektive" frei erkunden. Ein Übersichtsplan bietet Informationen über den aktuellen Standpunkt und über eine Teleportfunktion kann man schnell an vordefinierte Orte im Gebäude springen. An bestimmten durch Symbole markierten Stellen können zusätzliche Informationen und Bilder zum Semper Depot aufgerufen werden. Als weitere Möglichkeit das Objekt zu erkunden, stellt der Autor eine „Fly-Through" Funktion zur Verfügung. Über diese ist es möglich die Kamera frei zu steuern und das Modell aus Perspektiven zu betrachten, welche für gewöhnlich nicht einnehmbar wären. Neben der Bewegung und Rotation der Kamera kann auch deren Geschwindigkeit über drei Stufen reguliert werden. Die Anwendung bietet außerdem die Möglichkeit Objekte über ein Menü aus- und einzublenden. Eine Liste zeigt den aktuellen Status aller Elemente an. Dadurch will Michal Rontsinsky eine nähere und verständlichere Auseinandersetzung mit dem 3D-Modell ermöglichen. Die Darstellung des Modells erfolgt bei den beschriebenen Funktionen der Übersichtlichkeit halber abstrakt. Mittels einer „Showroom-Funktion" zeigt der Autor die Möglichkeiten zur fotorealistischen Bildberechnung der verwendeten Unity Engine. Durch Schattenberechnung, Umgebungsverdeckung, Kameraeffekte und Nachbearbeitung wird ein deutlich höherer Realitätsgrad erreicht. Die Interaktivität, welche bei den anderen Funktionen verfügbar ist, ist in diesem Modus beschränkt, da der Fokus hier auf der Qualität der Darstellung liegt.

Abb. 1.56 - 1.59: verschiedene Bereiche des virtuell dargestellten Semper Depots mit einblendeten Zusatzinformationen





oben: Abb. 1.60 + 1.61: Außenansicht des Semper Depots. Links als komplettes Modell, rechts mit ausgeblendeter Fassade
unten: Abb. 1.62 + 1.63: Innenraum des Semper Depots, dargestellt mittels der „Showroom" Funktion

Abschließend stellt Michal Rontsinsky fest, dass die schnelle Bildberechnung und der damit verbundene intuitive Workflow eine große Stärke in der Arbeit mit Game Engines darstellt. Er betrachtet die aktuellen Möglichkeiten von Echtzeitvisualisierung als sinnvolles Werkzeug für die Aufgaben von Architekten. Die im Rahmen seiner Arbeit entwickelte Anwendung soll als Framework verstanden werden und kann durch Ersetzen der Inhalte auch für andere Projekte verwendet werden [vgl. Rontsinsky, „Echtzeit-Rendering-Techniken für die Planung und Visualisierung von Architektur-Projekten", 2012].

Neben Unity stellt die Unreal Engine eine sehr beliebte Game Engine für Architekturvisualisierung dar. Bereits mit der Unreal Engine 3 (auch „UDK" oder „Unreal Development Kit" genannt) wurden zahlreiche Visualisierungen im Bereich der Architektur entwickelt.

2011 nutzt Ben Prince die Unreal Engine 3 während seines Studiums, um die Möglichkeiten von Echtzeitvisualisierung zu erkunden. Er kommt zu dem Schluss, dass Game Engines neben fotorealistischer Visualisierung großes Potenzial für virtuelle In-

Abb. 1.64+1.65: In der Unreal Engine 3 erstellte Echtzeitvisualisierung von Ben Prince. Verglichen mit Projekten aus früheren Unreal Versionen, ist ein deutlich gesteigerter Realitätsgrad in der Licht & Schattenberechnung erkennbar.

Abb. 1.66.1: Fallingwater von Frank Lloyd Wright, visualisiert in der Unreal Engine 3. Auch hier ist verglichen mit älteren Engineversionen (vorallem bei der Darstellung der Vegetation) ein höherer Realitätsgrad erkennbar.

teraktionen zwischen Projektbeteiligten bieten. Als Beispiel nennt er virtuell abgehaltene Besprechungen in geplanten, beziehungsweise gerade im Bau befindlichen Projekten [vgl. URL Ben Prince].

Eines der bekanntesten Beispiele für Architekturvisualisierung in der Unreal Engine 3 ist die Darstellung des Kaufmann-House (auch „Fallingwater", Frank-Lloyd Wright) von Maarten Breedveld im Jahr 2013. Er beschreibt das Projekt als Experiment, um die Möglichkeiten von Echtzeitvisualisierungen für Architektur zu erkunden [vgl. URL Falling Water].

Ebenfalls 2013 experimentieren Bryan Mock und Lauren Deshler von der Iowa State University mit der Unreal Engine 3, um Umgebungsgeräusche an den Betrachter ihres Entwurfs vermitteln zu können. Das Projekt hat zum Ziel, die Umgebungsgeräusche des Bauplatzes in Boston möglichst gut abzuschirmen. Um ihren Entwurf zu überprüfen, bauen die Autoren diesen in der Unreal Engine nach und kombinieren das Modell mit der aus der realen Umgebung aufgenommenen Geräuschkulisse [vgl. URL Bryan Mock].

Abb. 1.66+1.67: Echtzeitvisualisierung von Bryan Mock und Lauren Deshler. Der Fokus liegt hier auf der implementierung von Geräuschen anstatt einer möglichst fotorealistischen Darstellung.

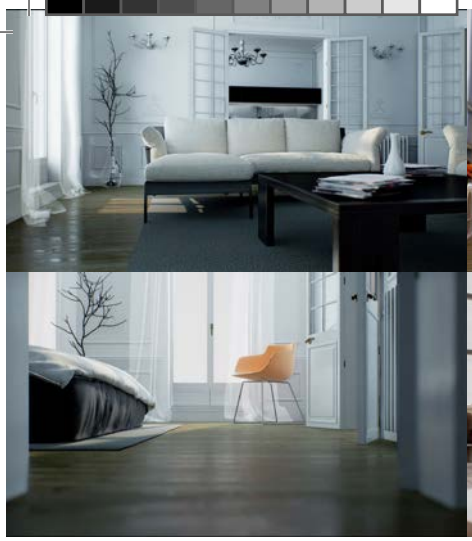


Abb. 1.68: „Unreal Paris“

Abb. 1.69: „Sun and Tales“

Abb. 1.70: „Lucid Arch“

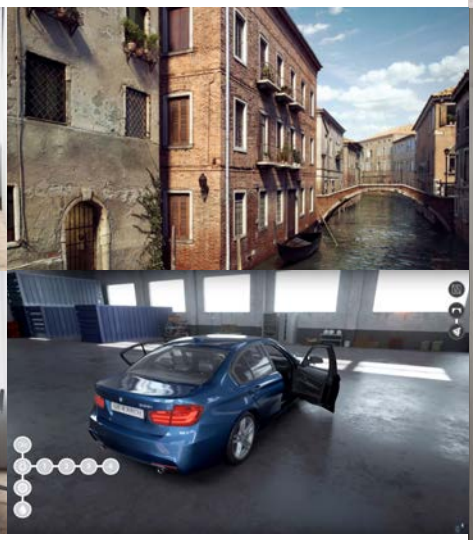
Seit der Veröffentlichung der vierten Generation der Unreal Engine im März 2014 wurden mit ihr zahlreiche Projekte auf dem Gebiet der Architekturvisualisierung verwirklicht. Eines der ersten und bekanntesten Projekte stellt „Unreal Paris“ von dem französischen Künstler Dereau Benoit dar. Aufgrund der hohen visuellen und ästhetischen Qualität erfuhr es schnell große Bekanntheit. Mit seinen zwei darauf folgenden Projekten „Lucid Arch“ und „Sun and Tales“ unternimmt Derau Benoit den Versuch die fotorealistische Qualität von Echtzeitvisualisierungen noch weiter zu steigern [vgl. URL Dereau Benoit].

Ein weiteres gutes Beispiel für die Verwendung der Unreal Engine 4 für Architekturvisualisierungen sind die Projekte des Brasilianers Rafael Reis Saliba. Er sieht in den Möglichkeiten der Belichtung, Materialerstellung und Interaktion von Game Engines großes Potenzial. Die Berechnung der Globalen Illumination beschreibt er als stark verbesserungswürdig, da es oft schwierig ist, zufriedenstellende Ergebnisse zu erhalten. Eine seiner bekanntesten Arbeiten ist das Projekt „Brazilian Old Kitchen“. Es zeigt einen traditionellen Küchenraum in Minas Gerais in Brasilien. Um möglichst detaillierte und natürliche Texturen zu erhalten, erstellt Rafael Reis Saliba diese in den meisten Fällen händisch in einer Bildbearbeitungssoftware. Auch in seinen anderen Projekten (u.a. Venice, London Apartment, Barcelona Pavillion) sind die überzeugenden Ergebnisse seiner Arbeitsweise sichtbar. Mit seinem „Car Customizer“ stellt er Anfang 2016 außerdem eine weitere Möglichkeit der Nutzung von Game Engines außerhalb der Architektur vor [vgl. URL Rafreis].

Abb. 1.71: „Brazilian Old Kitchen“

Abb. 1.72: „London Appartement“

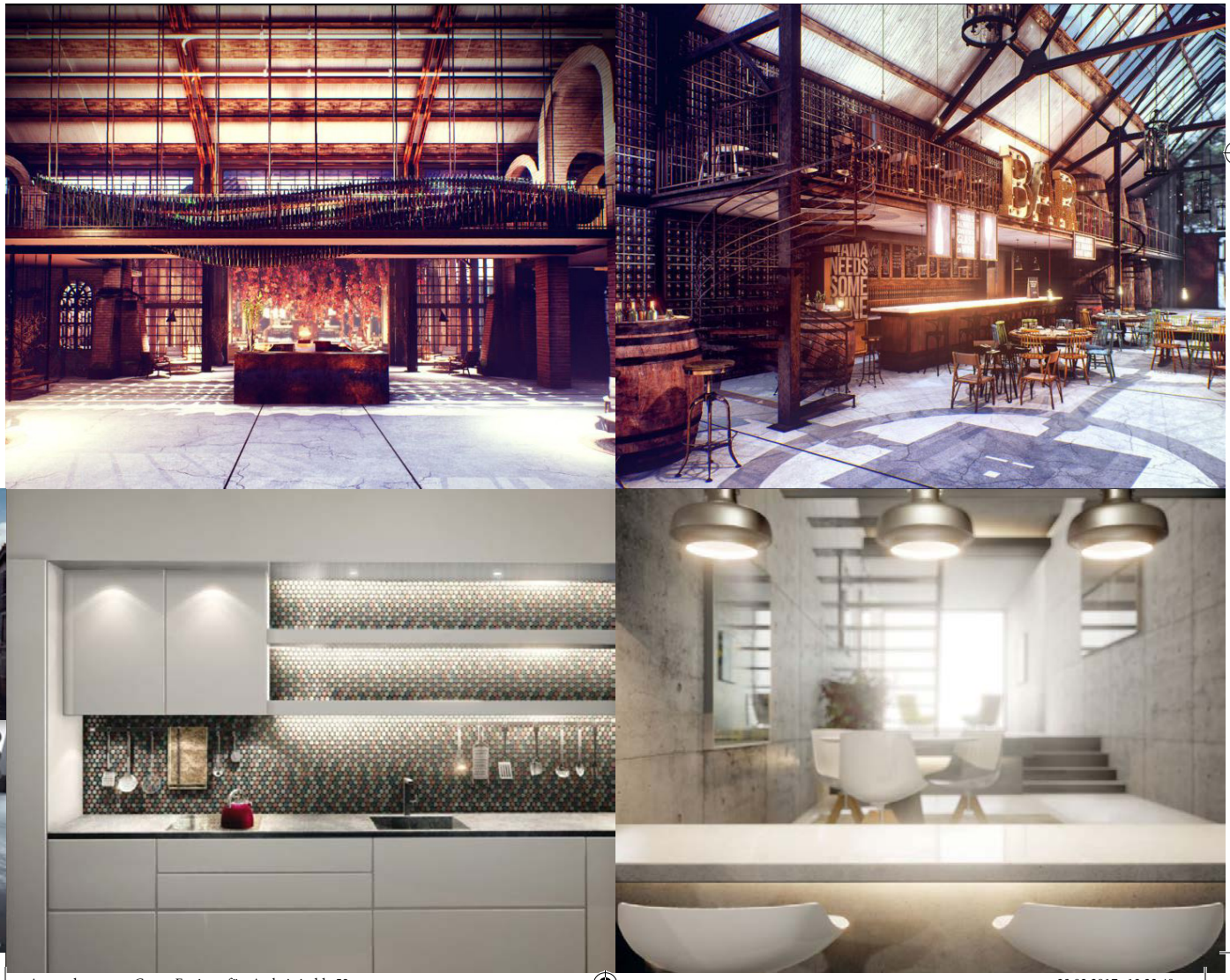
o.: Abb. 1.73: „Venice“
u.: Abb. 1.74: „Car Customizer“

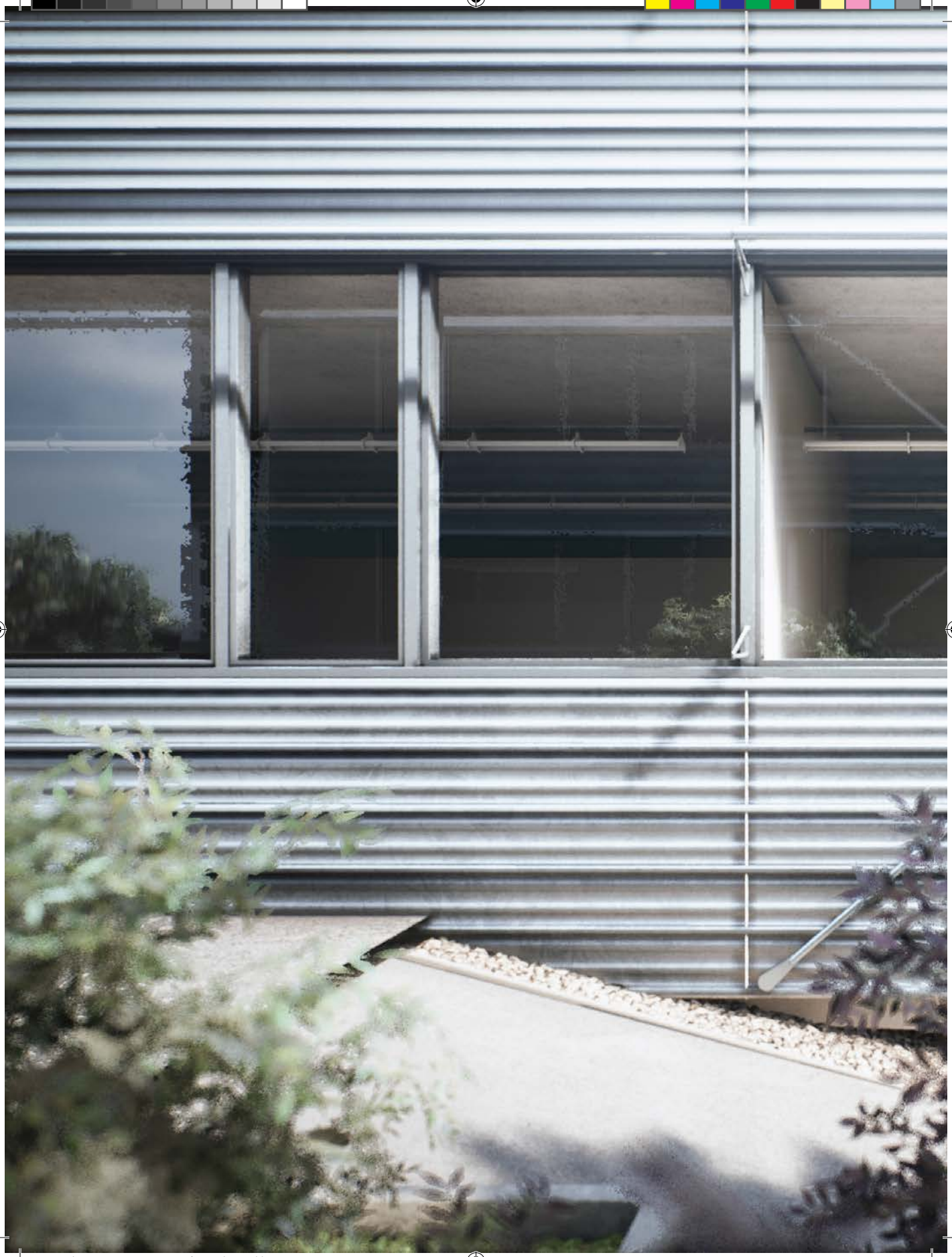


Im Sommer 2016 veranstaltet Epic (Unreal Engine) gemeinsam mit den Betreibern des Architekturvisualisierungsblogs www.ronenbekerman.com die „Vineyard Challenge“ - einen Visualisierungswettbewerb mit dem Ziel die Vorteile von Game Engines einer breiten Masse zugänglich zu machen. Die Wettbewerbsteilnehmer sind dazu aufgefordert neue Methoden auszuprobieren und werden von eine Fachjury in den Kategorien Licht, Design und Interaktivität bewertet [vgl. URL Vineyard Challenge].

“The opportunity to bring real-time gaming technology to the world of architectural visualization highlights not only the versatility of Unreal Engine, but also opens the door for a new concept of visualization, where the use of interactive animations and immersive VR will be increasingly relevant. I encourage all followers of the blog, as well as newcomers, to take part in this Challenge. Great prizes aside, this is an amazing opportunity to explore new territory and shape the way arch viz is going to be done in the near future.” (Ronen Bekerman, 2016)

Abb. 1.75: Die beiden Siegerprojekte der „Vineyard Challenge“ in den Kategorien „Solo“ (Rostislav Niokolaev, oben) und „Team“ (Factory Fifteen, unten)





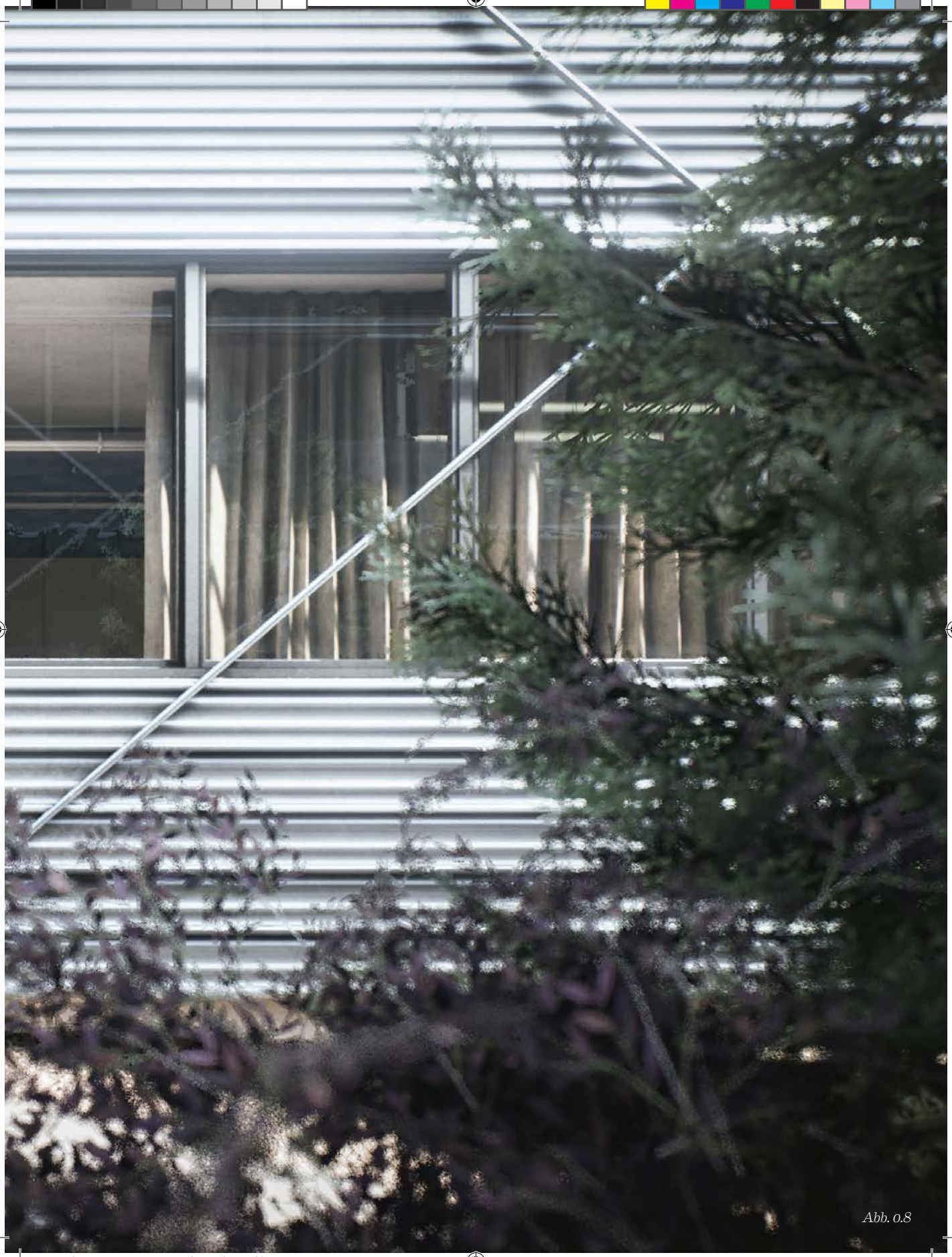


Abb. 0.8

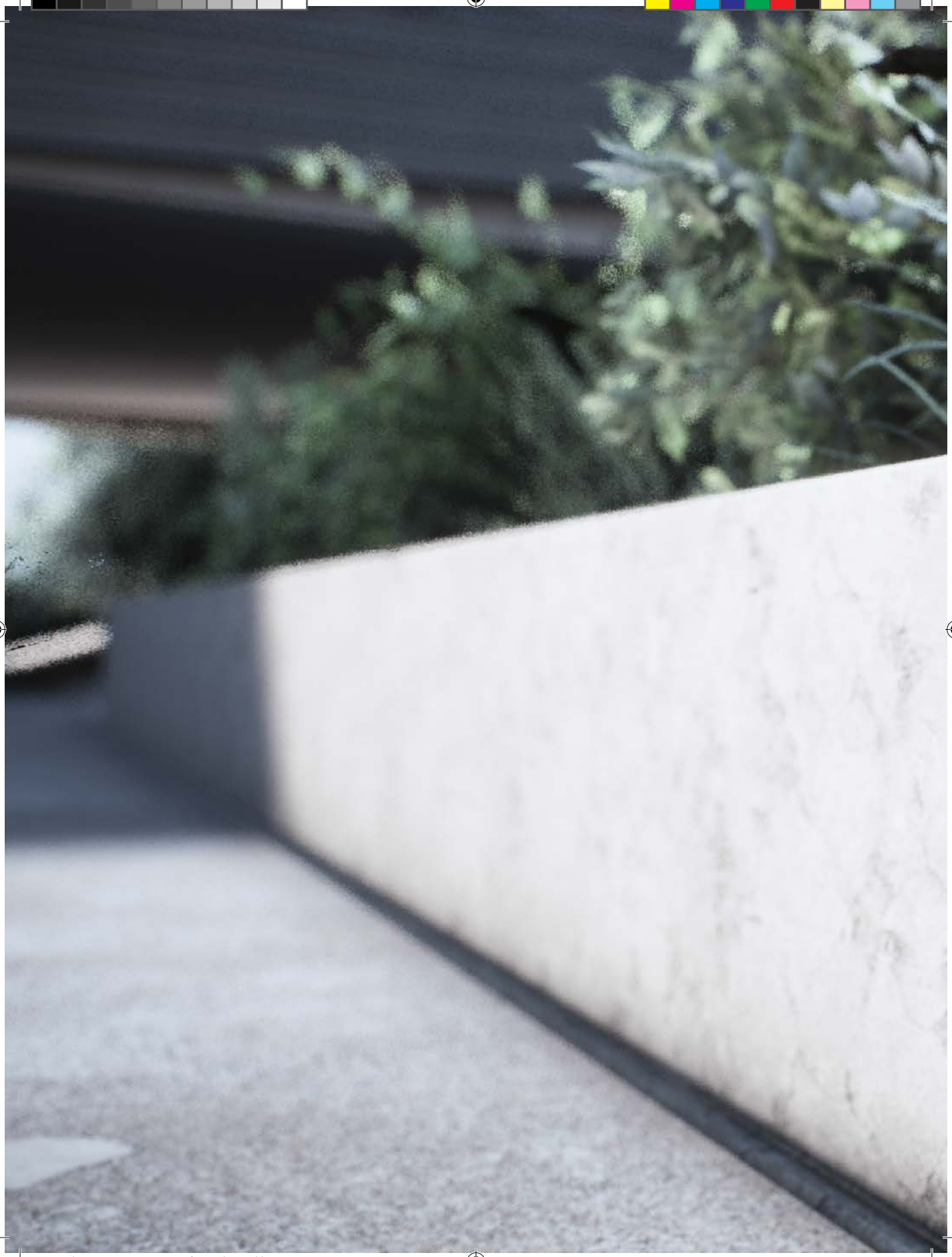


Abb. 0.9

2 - Workflow

Das menschliche Auge ist beeindruckend gut darin, kleinste Unstimmigkeiten in einem virtuellen Bild zu erkennen und dieses von der Realität zu unterscheiden. Um eine wahrlich fotorealistische Visualisierung zu erstellen, ist es deshalb notwendig, ein möglichst exaktes Abbild der echten Umwelt – inklusive aller kleinen Fehler – zu kreieren und das Auge damit zu täuschen. Grundsätzlich gibt es drei Faktoren, welche maßgeblich am Realismus mitwirken: das 3D Modell, die Belichtung und die Materialität. Im Folgenden sollen diese Teilbereiche näher erklärt werden.

2.1 - Modell

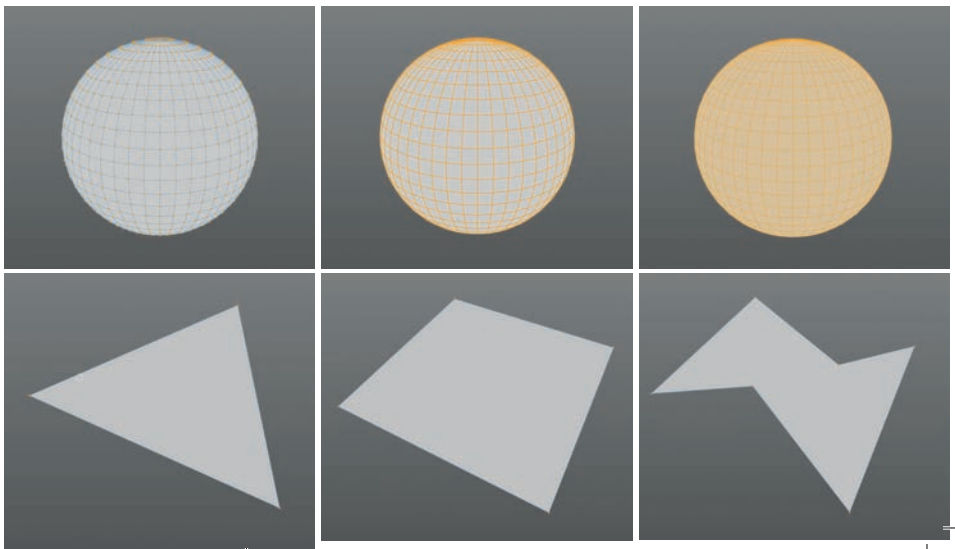
Die 3D Modellierung ist ein Teilbereich der Computergrafik. Man versteht darunter das Erstellen eines virtuellen dreidimensionalen Objektes mittels spezieller Modellier-Software. Diese 3D Objekte können später für zweidimensionale Renderings oder Animationen verwendet werden. Der Modelliervorgang kann entweder manuell (Schritt für Schritt „per Hand“) oder automatisiert (mit Algorithmen, „parametrisches Modellieren“) erfolgen. Der folgende Absatz soll einen Überblick über den Aufbau und die Bearbeitung von 3D Modellen geben. Außerdem werden wichtige Techniken für die Erstellung von 3D Modellen für Game Engines beschrieben.

Aufbau eines 3D Modells:

Ein 3D Modell besteht aus Punkten (Vertex bzw. Vertices) im dreidimensionalen Raum, welche durch Linien (Kanten oder Edges) miteinander verbunden sind (Abb. 2.1). Verbindet man 3 Punkte miteinander, so entsteht ein Triangel – die einfachste Form eines Polygons. Bei vier Punkten spricht man von einem Quad. Bei Polygonen mit 5 Punkten und mehr spricht man oft auch von „n-Gons“ (Abb.2.2). Verbindet man mehrere Polygone miteinander, so ist die Rede von einem Element. Ein 3D-Objekt ist oft aus mehreren Elementen zusammengesetzt.

Polygone, Kanten und Punkte können separat ausgewählt werden, um sie zu bearbeiten und damit das gesamte Element beziehungsweise Objekt zu manipulieren. Grundsätzlich gibt es zwei verschiedene Workflows, um ein Modell manuell zu erstellen: Das Polygon- und das Spline-Modeling [vgl. Asanger, „Cinema 4D 13“, 2012, s. 99 ff].

o.: Abb. 2.1: Punkte, Kanten und Polygone
u.: Abb. 2.2: Triangel, Quad und n-Gon



Polygon-Modeling

Beim Polygon-Modeling erstellt und verformt man einzelne Polygone. Theoretisch wäre es möglich einzelne Punkte im Raum zu definieren, diese mit Kanten zu verbinden und damit Polygone zu erstellen. Man beginnt meistens aber mit einem sogenannten Grundkörper (Würfel, Kugel, Zylinder...) und verformt diesen Schritt für Schritt, um sich dem gewünschten Ergebnis immer stärker anzunähern. Dieser Vorgang nennt sich auch Box-Modeling (Abb.2.3). Es können Polygone, Punkte und Kanten hinzugefügt, gelöscht oder transformiert werden. Außerdem steht eine Vielzahl an verschiedenen Modellierwerkzeugen zur Verfügung, mit denen das Objekt weiter detailliert werden kann. Grundsätzlich arbeitet man sich immer von großen, groben Formen zu kleineren, detaillierteren Bereichen vor. Das Polygon-Modeling ist eine der am weitesten verbreitete Formen und ein Großteil aller 3D Modelle entsteht mit dieser Technik. Vorteile sind die große Flexibilität und die schnellen Renderzeiten. Ein Nachteil besteht darin, dass gekrümmte Flächen mit vielen einzelnen, flachen Polygonen angenähert werden müssen. Ein wichtiger Anwendungsbereich des Polygon-Modelings liegt zum Beispiel in der Modellierung von Fahrzeugen [vgl. Asanger, „Cinema 4D 13“, 2012, s. 119 ff].

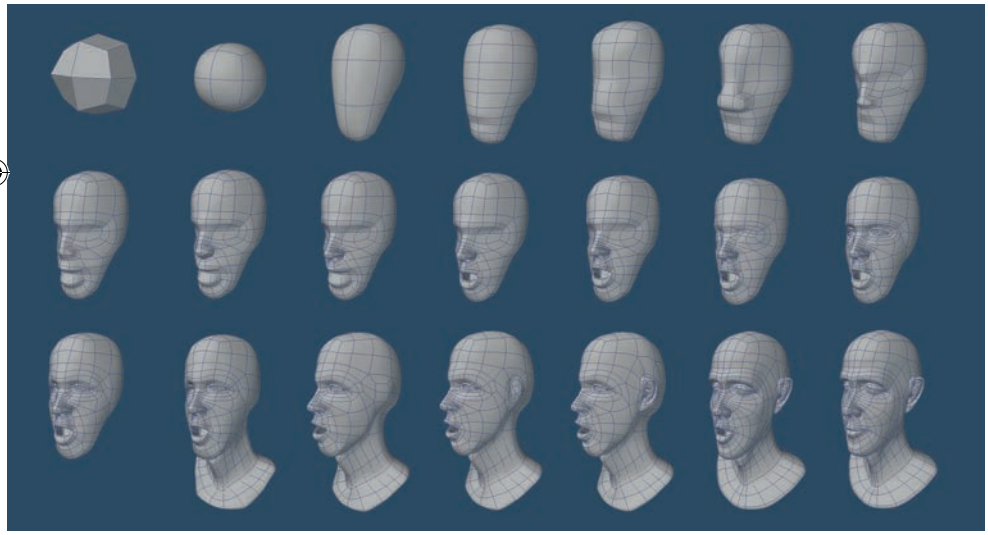


Abb. 2.3: Box-Modeling

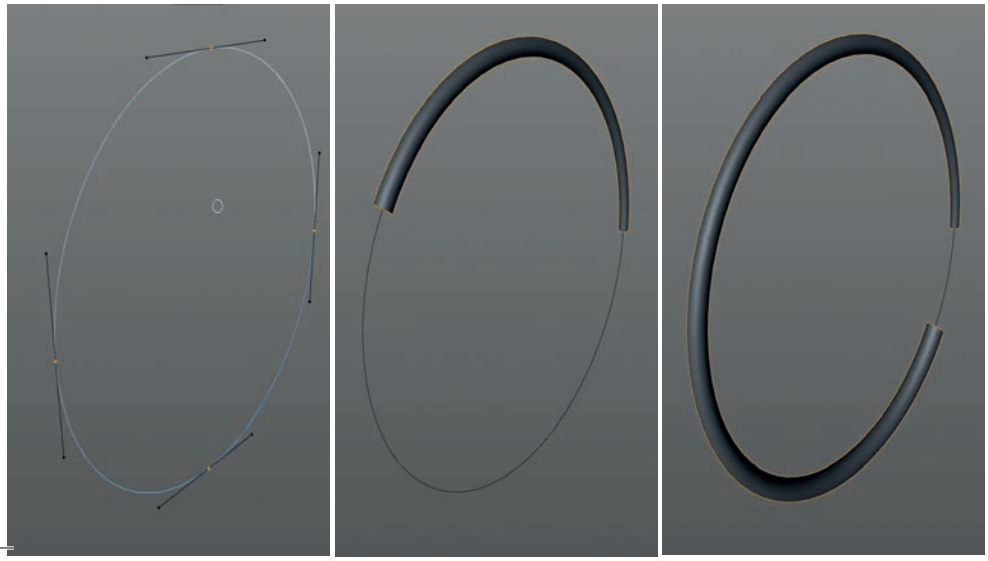


Abb. 2.4: Spline-Modeling

Spline-Modeling

Beim Spline-Modeling werden unter anderem NURBS (nicht-uniforme rationale B-Splines) verwendet um 3D Objekte zu erstellen. NURBS sind mathematisch definierte Kurven oder Flächen, die über Kontrollpunkte sehr intuitiv und vorhersehbar gesteuert werden können.

In den 50er Jahren wurden mathematisch exakte Beschreibungen von Freiformflächen für den Automobil- und Schiffbau benötigt. Der Ingenieur Pierre Étienne Bézier begann mit der Entwicklung der nach ihm benannten Bézierkurve. Dabei handelt es sich um eine geschwungene Kurve, welche durch wenige Kontrollpunkte gesteuert wird, die nicht direkt auf der Kurve selbst liegen. Um beispielsweise einen Kreis exakt darzustellen war die Bézierkurve allerdings noch nicht ausreichend. Daraufhin folgte die Entwicklung der B-Splines, bei denen einzelne Kontrollpunkte unterschiedlich stark auf den Kurvenlauf einwirken (nicht-uniform). Später wurden sowohl die Bézierkurve als auch B-Splines in Form von NURBS verallgemeinert. Damit wurde es möglich jede Art von Objekt mathematisch exakt abzubilden.

Für das Modellieren mit Splines steht heutzutage in fast jedem 3D Programm eine große Anzahl an Werkzeugen zur Verfügung. Wichtige Beispiele sind das Loft-, Lathe- und Swipe-Werkzeug (Abb.2.4). Mit diesen ist es möglich, einen zuvor mittels Splines erstellten Umriss durch Verbinden, Rotieren oder Verschieben in Polygon-Geometrie umzuwandeln. Meist kann die Kurve später angepasst werden, um das Polygon-Objekt zu verändern [vgl. Asanger, „Cinema 4D 13“, 2012, s. 107 ff].

Parametrisches Modellieren

Eine vollkommen andere Art zur Erstellung von virtuellen Objekten stellt das parametrische Modellieren dar. Dabei wird das Modell nicht manuell bearbeitet, sondern Schritt für Schritt mit verschiedenen aufeinander folgenden Befehlen aufgebaut. Ein bekanntes Beispiel ist das Grasshopper-Plugin für das Programm Rhinoceros 3D (Abb. 2.5 + 2.6). Anstatt Änderungen direkt am Modell vorzunehmen, werden verschiedene Komponenten (auch „Nodes“) miteinander verbunden, die nach und nach das Objekt aufbauen. Der große Vorteil liegt darin, dass Änderungen, welche an einzelnen Komponenten durchgeführt werden, automatisch und sehr schnell auf das gesamte Modell angewandt werden. Dadurch ist es möglich in sehr kurzer Zeit eine Vielzahl an Variationen zu erstellen. Stellt man sich beispielsweise ein Hochhaus vor, dessen Grundriss durch eine geschlossene Kurve definiert wird, so reicht es aus diese Kurve zu verformen um eine andere Form zu erhalten. Alle anderen Teile des Hochhauses, die auf dieser Basiskurve aufbauen, werden automatisch aktualisiert. Im Vergleich dazu wäre es beim klassischen Polygon-Modeling notwendig das gesamte Gebäude neu aufzubauen. Aufgrund der Funktionsweise solcher 3D-Modeller ist ein gewisses mathematisches Grundverständnis (z.B. der Vektormathematik) notwendig. Außerdem gestalten sich spätere Änderungen an einzelnen, bestimmten Bereichen des Objektes schwieriger als bei klassischen Modelliermethoden [vgl. Asanger, „Cinema 4D 13“, 2012, s. 145 ff].

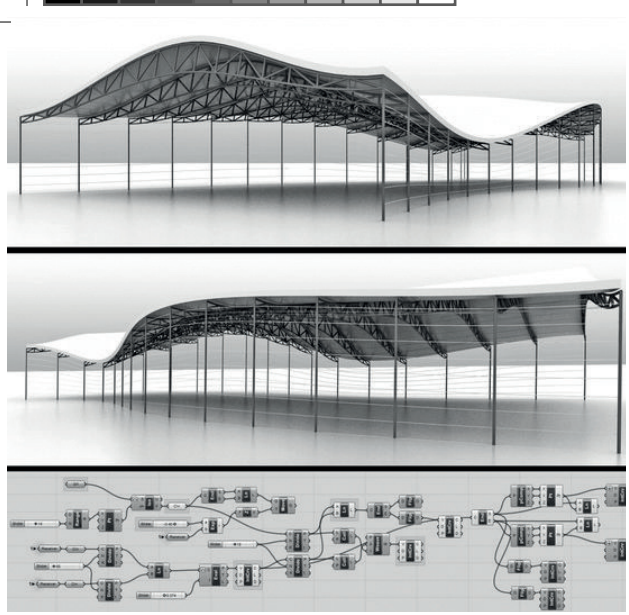


Abb. 2.5: Grasshopper

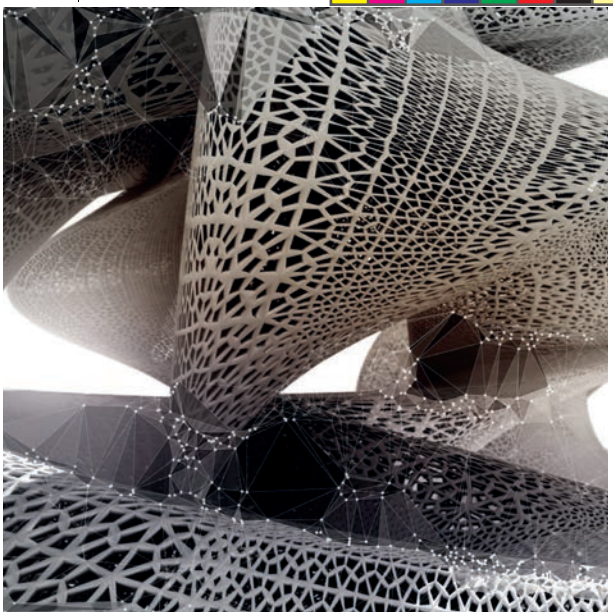


Abb. 2.6: Grasshopper

Wichtige Techniken beim Erstellen von 3D Modellen für Game Engines

Um 3D Objekte in einer virtuellen Szene täuschend echt aussehen zu lassen, sollte man bereits beim Modellieren auf einige grundlegende Dinge achten. Nachfolgend sind wichtige Aspekte beschrieben, welche sowohl für die visuelle Qualität, als auch für einen angemessenen Ressourcenverbrauch beim Visualisieren ausschlaggebend sind.

Polygonzahl

Beim Modellieren von 3D Objekten für die Verwendung in Visualisierungen und Animationen spielt die Polygonanzahl eine wichtige Rolle für die Bedienbarkeit und Renderzeit. Es handelt sich dabei um die Gesamtanzahl aller Polygone in einer Szene, welche benötigt wird, um diese als Bild darzustellen. 3D-Modelle mit höherer Polygonzahl sind optisch ansprechender, weil sie detaillierter aussehen und gekrümmte Oberflächen glatter erscheinen (Abb.2.8). Allerdings steigen mit der Anzahl der Polygone auch der Ressourcenverbrauch beziehungsweise die Anforderungen an die Hardware. Bei Verwendung von klassischen Render Engines wird dadurch bei der Navigation im Vorschaufenster die Grafikkarte stärker beansprucht, was eine effektive Darstellung der Szene unmöglich machen kann. Beim Rendern selbst kann es auch zu Problemen kommen, da der Arbeitsspeicher schnell an seine Grenzen stoßen kann, wenn sich zu viele, sehr hoch auflösende Modelle in der Szene befinden. Doch insbesondere bei der Visualisierung mit Game Engines sind möglichst kurze Renderzeiten ausschlaggebend, um die erforderliche Anzahl der Bilder pro Sekunde für eine flüssige Darstellung zu erreichen.

Aus diesen Gründen ist es wichtig, die Polygonzahl der Modelle immer im Auge zu behalten und sich an gewissen Grundregeln zu orientieren. Die Leistung beim Visualisieren wird von vielen Faktoren beeinträchtigt und gut modellierte 3D Objekte

Abb. 2.7: Anhand der Silhouette lassen sich schnell die wichtigsten Bestandteile eines Objektes ausmachen, die das Meiste zu dessen Form beitragen

Abb. 2.8: Unterschiedlich stark detailierte Modelle

alleine bilden noch keine Garantie für niedrige Renderzeiten. Trotzdem sind Fehler in diesem Bereich oft Ursache für zu hohen Ressourcenverbrauch.

Notwendiger Detailgrad

Nachträgliche Änderungen an der Polygonanzahl bei bereits bestehenden Objekten sind grundsätzlich immer schwierig. Besser ist es gleich von Beginn an auf eine korrekte Auflösung zu achten. Dabei ist zu bedenken, was vom Modell sichtbar sein wird, sowie welche Teile wichtiger und welche weniger wichtig sind. Eine gute Methode dafür ist, die Silhouette des zu modellierenden Objektes zu betrachten. Das erlaubt dem Visualisierer schnell die charakteristischen Teile, die das Objekt ausmachen, zu erkennen (Abb.2.7). Auf diese sollte beim Modellieren besonders Wert gelegt werden, beziehungsweise ist in diesen Bereichen eine höhere Polygonzahl vertretbar. Andere Bereiche kommen mit einer weitaus niedrigeren Auflösung aus, außerdem können auch viele Details mit Texturen abgedeckt werden. Gerade bei der Erstellung von kleinen Oberflächenunebenheiten sollten mittels Texturen erstellte Effekte bevorzugt werden, da diese um ein vielfaches ressourcenschonender arbeiten. [vgl URL 3D Modeling].

Parametrische Kontrolle über Polygondichte

Die Kontrolle über die Polygonanzahl sollte auf eine Art und Weise erfolgen, welche es erlaubt die Auflösung zu einem späteren Zeitpunkt gegebenenfalls anzupassen. Hierfür sind in den meisten 3D Anwendungen verschiedene Werkzeuge vorhanden („modifier“) mit denen eine parametrische Anpassung möglich ist (Abb.2.9). Sie ermöglichen dem Nutzer zunächst an einem groben Modell zu arbeiten, das später innerhalb weniger Schritte automatisch verfeinert werden kann.

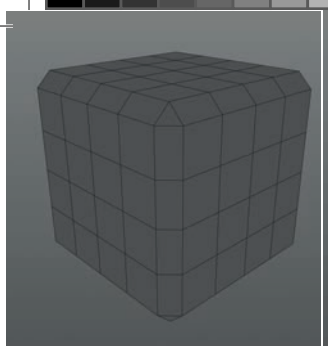


Abb. 2.9: Subdivision

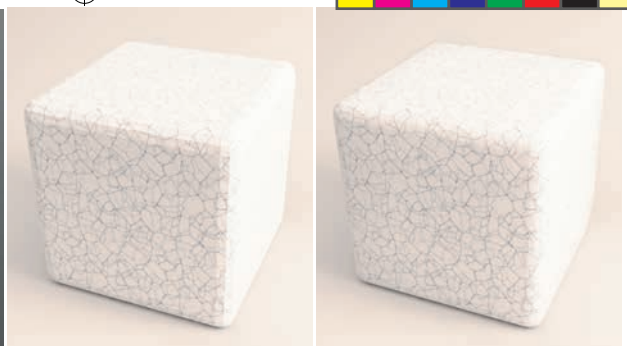


Abb. 2.10: Kantenglättung

Glättung

Gekrümmte Oberflächen (z.B. abgerundete Kanten) sind oft für überhöhte Polygonzahlen bei 3D Modellen verantwortlich. Sie können grundsätzlich durch viele einzelne flache Polygone angenähert werden und erscheinen (bei entsprechend hoher Unterteilung) für das freie Auge vollkommen glatt. Allerdings ist dies aufgrund der zuvor genannten Einschränkungen nicht empfehlenswert. Aus diesem Grund entwickelte der Wissenschaftler Bui Tuong Phong 1973 eine Methode, welche es ermöglicht, Kanten mittels Schattierung glatt erscheinen zu lassen, ohne neue Geometrie hinzuzufügen oder die bestehende Geometrie zu verändern. Um eine bessere Kontrolle über die Glättung zu erhalten, kann diese über eine Winkelbeschränkung oder so genannte Phong-Brakes gesteuert werden. Dabei definiert man bestimmte Winkellimits, bis zu welchen die Kanten geglättet werden sollen. Dadurch ist es möglich eine abgerundete Kante, welche nur grob unterteilt ist, vollkommen glatt erscheinen zu lassen (Abb.2.10).

Bevel (Fase)

Um solche Kanten zu erstellen ist in der Regel ein Bevelwerkzeug vorhanden. Es stellt eines der meist verwendeten Werkzeuge beim 3D Modellieren dar und trägt stark zum Realismus des Modells bei, da es in der Realität praktisch keine Objekte mit vollkommen scharfen 90 Grad Kanten gibt. Erst durch die Abrundung kann Licht auf die entsprechenden Bereiche fallen und diese realitätsnah visualisiert werden (Abb.2.11).

Verdeckte Polygone

Grundsätzlich sollte darauf geachtet werden, keine Polygone zu verschwenden und diese in Bereichen zu verdichten, wo sie auch benötigt werden. Aus diesem Grund sollten logischerweise Bereiche, die von der gewählten Perspektive kaum oder gar nicht sichtbar sind, nur wenige Polygone enthalten, beziehungsweise gänzlich gelöscht werden (Abb.2.12). Gerade bei der Verwendung von Game Engines kann dieser Punkt zu erheblichen Leistungseinsparungen führen, da bei Vorberechnungen von Licht immer die gesamte Szene berechnet wird, unabhängig davon, welche Objekte später tatsächlich im finalen Ausschnitt sichtbar sind. Bei gewöhnlichen Engines ist dies zwar nicht der Fall, allerdings sollte auch hier darauf geachtet werden, nicht sichtbare Details zu löschen, da dadurch die Bildwiederholrate bei der Vorschau und damit die Bedienbarkeit steigt [vgl. URL 3D Modeling].

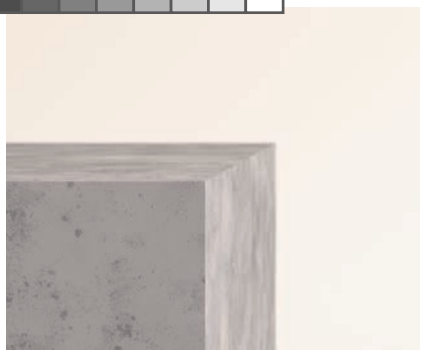
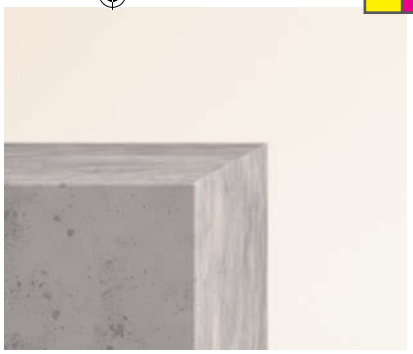


Abb. 2.11: Bevel



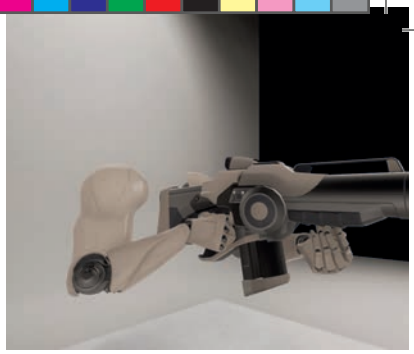


Abb. 2.12: Bei diesem Charakter sind nur die Hände modelliert, da der restliche Körper aus der First Person Perspektive ohnehin nicht sichtbar ist

LOD

Level of Detail (LOD) beschreibt eine Technik, bei der - abhängig von der Entfernung zur Kamera - unterschiedlich detaillierte Modelle, aber auch Texturen zum Einsatz kommen. Sie stammt aus der Spielindustrie, wo sie in nahezu jedem 3D Spiel zur Anwendung gelangt. Ohne diese Technik wäre es unmöglich große Mengen unterschiedlicher bewegter Objekte auf einmal darzustellen. Oft sind drei oder mehr Varianten desselben Modells vorhanden: ein hoch detailliertes für Nahaufnahmen, eines mit reduzierten Details für Aufnahmen von weiter weg und ein stark vereinfachtes für Hintergrundszenen. Entfernt sich die Kamera nun, wird bei definierten Distanzen zwischen den Detailgraden umgeschaltet. Je weiter weg sich der Nutzer befindet, desto weniger Polygone haben die einzelnen Objekte (Abb.2.13). Dies wiederum erlaubt bei gleich bleibender Renderzeit mehr Geometrie auf einmal zu berechnen.

Ein Problem, das sich bei der Verwendung von LOD Modellen ergibt, ist das sichtbare Umschalten zwischen den einzelnen Detailstufen. Betrachtet man einen Baum, der während einer Kamerafahrt immer näher rückt, so erscheinen beim Wechsel plötzlich zusätzliche Äste und Blätter scheinbar aus dem Nichts. Dies schwächt den Effekt einer fotorealistischen Illusion natürlich immens und muss bei der Anwendung für Architekturvisualisierungen vermieden werden. Trotzdem ist der Einsatz von LOD Modellen auch auf diesem Gebiet durchaus von großem Vorteil, da die unterschiedlichen Detailstufen auch manuell vom Nutzer und unabhängig von der Kameradistanz definiert werden können. Dadurch können vom Detailgrad an die Sichtbarkeit angepasste Modelle verwendet werden und es wird vermieden, dass Geometrie berechnet werden muss, welche aufgrund der hohen Distanz zur Kamera mit dem freien Auge sowieso nicht sichtbar ist [vgl. URL 3D Modeling].

Abb. 2.13: Unterschiedliche Detailstufen (Level of Detail) eines Modells

Modelltopologie

Neben einer angemessenen Polygonzahl ist außerdem deren Anordnung (Topologie) von großer Bedeutung für die Qualität des 3D-Modells. Prinzipiell sollte immer darauf geachtet werden „saubere“ Geometrie zu modellieren. Wichtige Punkte sind:

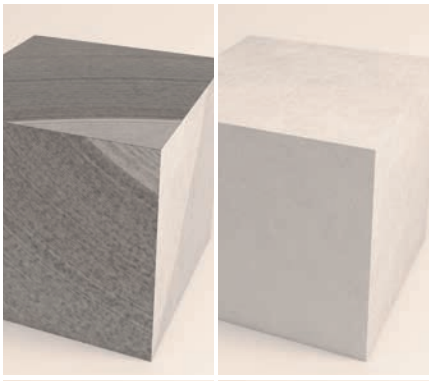


Abb. 2.14: Überlappende Polygone

- *Keine überlappenden Polygone, Punkte oder Kanten*

  Deckungsgleiche Geometrie sollte vermieden werden, da sie nichts zum Objekt beiträgt, zu Fehlern bei der Lichtberechnung führt und außerdem den Ressourcenverbrauch negativ beeinflusst (Abb.2.14).

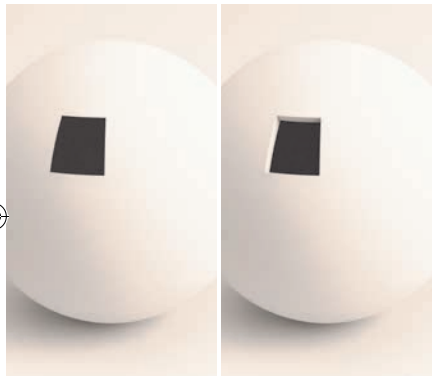


Abb. 2.15: Offenes vs. geschlossenes Mesh

- *Keine „Löcher“ in Objekten, Mesh sollte immer in sich geschlossen sein.*

  Wenn ein „Loch“ bzw. eine Öffnung absichtlich modelliert wird, sollte trotzdem eine „Wandstärke“ vorhanden sein (Abb.2.15). Geometrie sollte niemals aus einzelnen Polygonen bestehen.

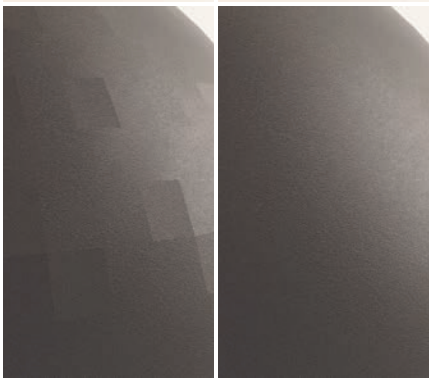


Abb. 2.16: Ausrichtung der Normalen

- *Normalen ausrichten*

  Jedes Polygon hat eine so genannte Oberflächennormale. Es handelt sich dabei um den Normalvektor der Ebene. Die Richtung des Vektors bestimmt die Vorder- bzw. Rückseite des Polygons. Die Ausrichtung der Normalen wirkt sich einerseits auf die Texturierung von Objekten aus, da ein Material entweder auf der Vorder- bzw. Rückseite oder beidseitig dargestellt werden kann. Andererseits hat sie auch Auswirkung auf die Belichtung des Modells. Oft wird die Rückseite von Polygonen gar nicht dargestellt, d.h. es wird überhaupt kein Licht reflektiert und das Polygon erscheint durchsichtig. Oder aber es kann zu Fehlern bei der Darstellung kommen, wenn nicht alle Normalen gleich ausgerichtet sind (Abb.2.16). Aus diesen Gründen lassen sich die Normalen in den meisten 3D Programmen zur Kontrolle darstellen und bei Bedarf mittels verschiedener Werkzeuge ausrichten beziehungsweise umkehren.

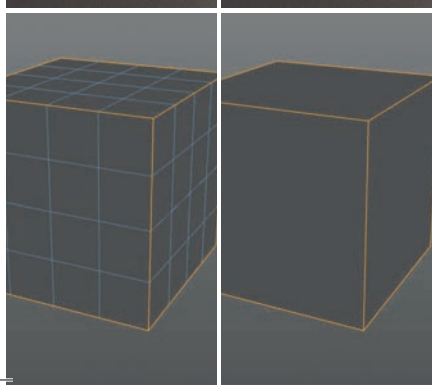


Abb. 2.17: unnötige Polygone

- *Polygone, welche nichts zur Form beitragen, löschen (unnötige Unterteilungen innerhalb einer ebenen Fläche)*

  Ein perfekter Würfel sollte beispielsweise nie aus mehr als sechs viereckigen Polygonen bestehen (Abb.2.17).

- *Grundsätzlich immer mit Quad-Polygonen (Viereckige Polygone) arbeiten (Abb.2.18).*

Der letzte Punkt ist gerade bei der Verwendung von Game Engines im Zusammenhang mit der Triangulation ausschlaggebend:

Triangulation

Die Triangulation beschreibt in der Geometrie die Unterteilung eines Objektes in dreieckige Polygone. Drei Punkte im Raum, welche nicht auf denselben Koordinaten oder einer Linie liegen, beschreiben immer eine flache Ebene. Dies trifft nicht zwingend auch auf komplexere Polygone mit mehr als drei Punkten zu. Die flache Topologie ermöglicht eine sehr leichte und schnelle Berechnung des Normalvektors der Ebene. Dieser wird für die Lichtberechnung beim Raytracing verwendet.

Da Grafikkarten grundsätzlich mit dreieckigen Polygonen arbeiten, werden Objekte, nachdem sie in einer 3D Anwendung modelliert wurden, beim Export/ Import in die Game Engine immer trianguliert. Dies erfolgt automatisiert im Hintergrund und benötigt so lange entsprechend auf die Topologie geachtet wurde nur wenige Ressourcen. Denn für die Umwandlung eines Vierecks in ein Dreieck gibt es nur zwei Möglichkeiten und es müssen lediglich zwei Punkte miteinander verbunden werden (Abb.2.19).

Sobald jedoch Polygone mit mehr als vier Seiten trianguliert werden müssen, steigen die Komplexität und damit der Ressourcenverbrauch stark an (Abb.2.20 + 2.21). Dieses Problem wird mit steigender Polygon- und Seitenzahl immer größer und kann ab einem gewissen Punkt zu starken Leistungseinbußen oder Abstürzen der Engine führen. Aus diesem Grund sollte immer darauf geachtet werden, Modelle möglichst aus viereckigen Polygonen aufzubauen und Sonderformen zu vermeiden oder zumindest auf ein Minimum zu reduzieren [vgl. URL 3D Modeling].

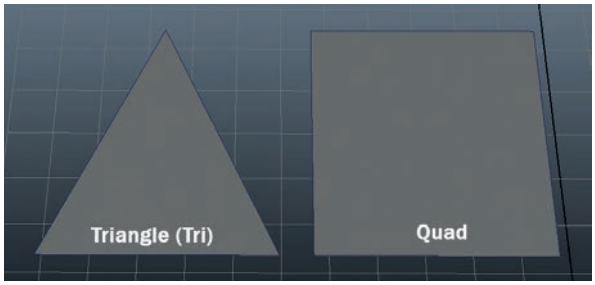


Abb. 2.18

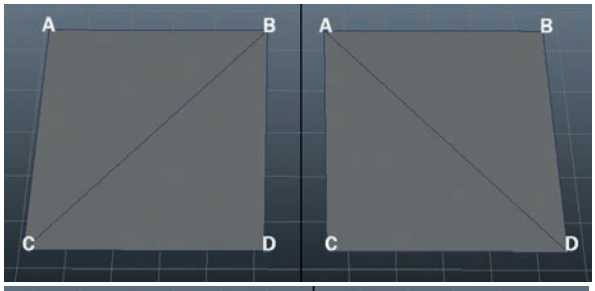


Abb. 2.19

Abb. 2.20

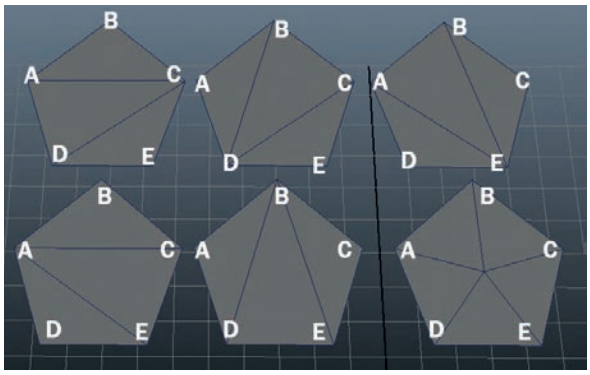
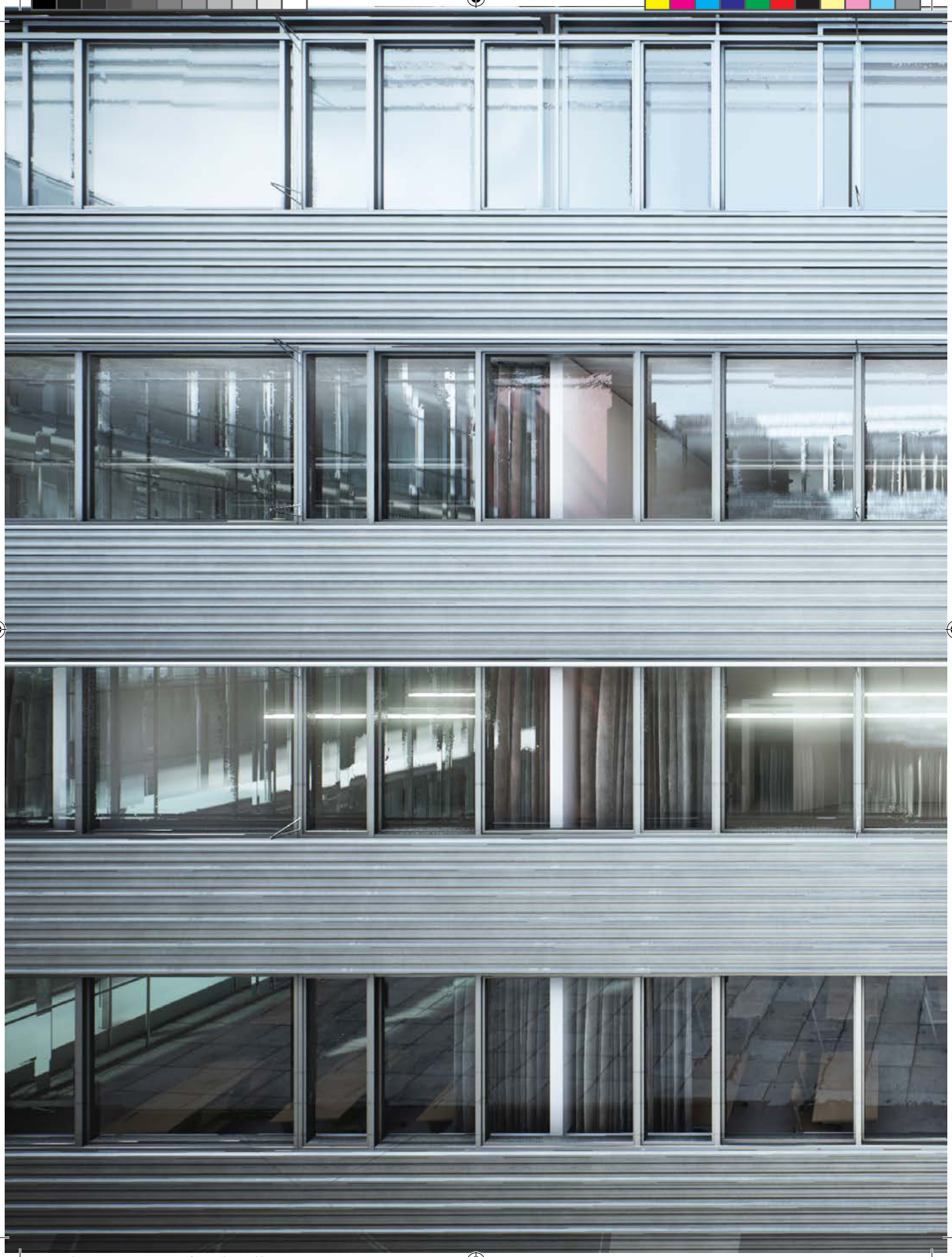


Abb. 2.21

Grundsätzlich gibt es nur zwei verschiedene Möglichkeiten ein Pentagon zu unterteilen (Abb. 2.20). Da die einzelnen Punkte aber von Grafikkarten unterschieden werden, sind tatsächlich sechs Varianten vorhanden (Abb. 2.21)



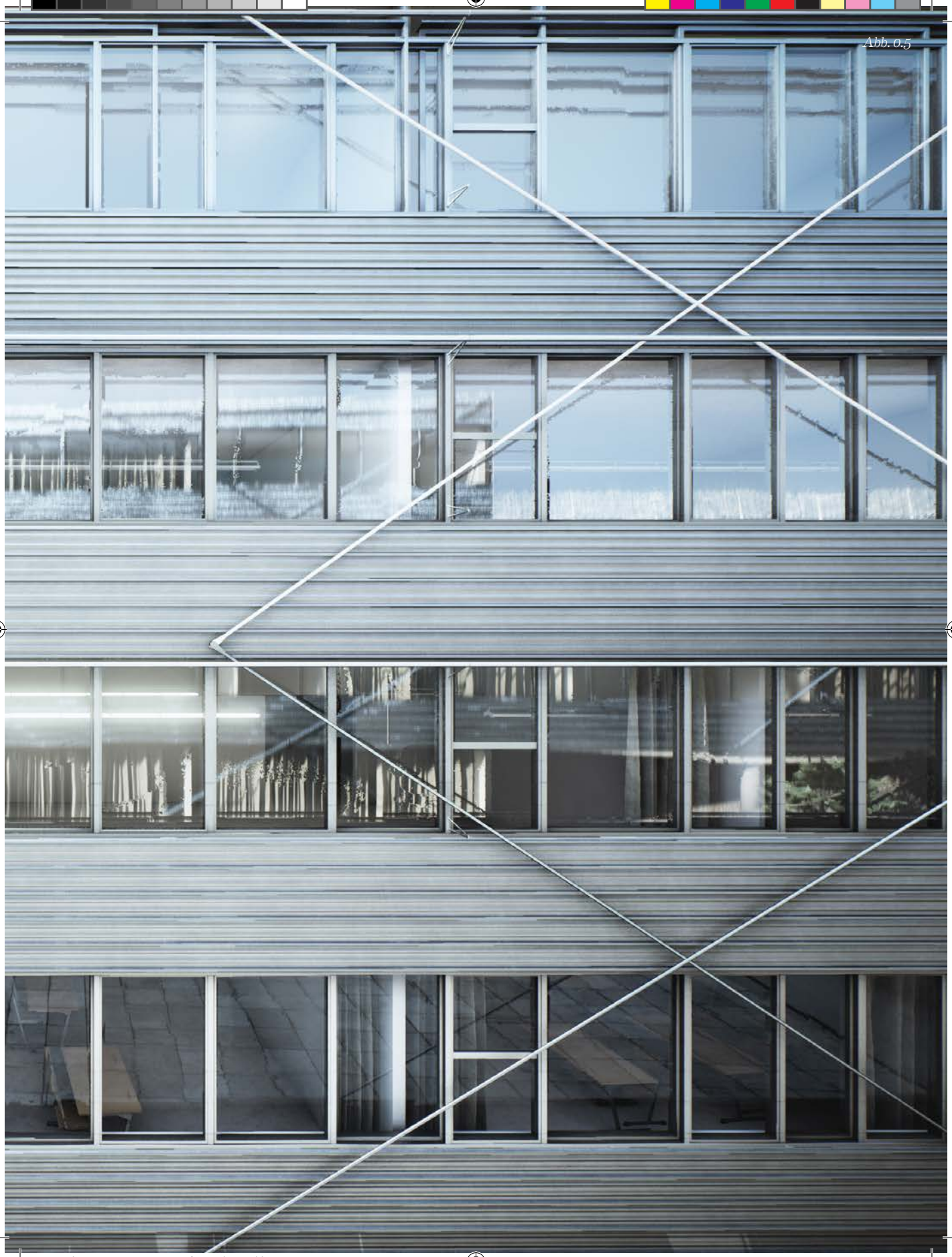


Abb. 0.5

2.2 - Licht

Selbst das beste 3D Modell wird sofort als virtuelle Nachbildung erkannt, wenn es nicht entsprechend belichtet beziehungsweise beleuchtet wird. Aus diesem Grund gibt es verschiedene Ansätze, welche allesamt versuchen, die in der echten Welt vorkommenden Lichtquellen nachzuahmen. Grundsätzlich sollte immer darauf geachtet werden, Lichtquellen da zu setzen, wo sie auch in einer realen Szene vorkommen würden. Zwar kommt es gerade bei Innenraumszenen vor, dass man auf unsichtbare, zusätzliche „Fülllichter" („fill light") zurückgreifen muss, um das gewünschte Ergebnis zu erzielen. Diese Technik sollte jedoch mit Bedacht und großer Sorgfalt eingesetzt werden, um eine unnatürliche Lichtstimmung zu vermeiden. Eines der wichtigsten Kriterien für realistische Ergebnisse ist der Einsatz der globalen Illumination.

Was ist globale Illumination?

Globale Illumination (im folgenden „GI") beschreibt Algorithmen, welche in der Erstellung von Computer Grafiken verwendet werden. Sie bestimmen, wie Licht, das auf bestimmte Oberflächen trifft, absorbiert, reflektiert oder gebrochen wird. Damit sind GI-Systeme ein maßgeblicher Faktor, um natürliche Lichtsimulationen zu erreichen. Als eine der wichtigsten Funktionen ist die Berechnung von indirektem Licht zu nennen. Ohne eine Simulation von indirektem Licht erscheinen alle Bereiche eines Modells, welche nicht direkt von einer Lichtquelle getroffen werden, vollkommen schwarz [vgl. Legrenzi, „Vray the complete Guide, 2010, s 209 ff].

Abb. 4.9.2

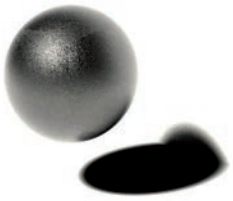
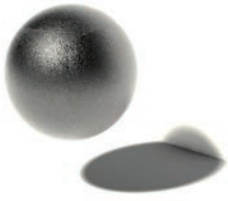


Abb. 4.9.3



Im gezeigten Beispiel wird das Objekt mittels einer unendlichen Lichtquelle (Sonne) belichtet. Auf Abb. 4.9.2 sieht man, dass Bereiche, die keinem direkten Licht ausgesetzt sind, komplett schwarz erscheinen. Durch die Aktivierung der GI in Abb. 4.9.3 wird eine natürlichere Belichtungssituation erreicht. Dies geschieht, weil Lichtstrahlen, die von der Lichtquelle ausgehen, nicht nach dem ersten Auftreffen auf einer Fläche absorbiert, sondern teilweise reflektiert werden. Dadurch werden auch Bereiche, die eigentlich im Schatten liegen, belichtet.

Abb. 4.10: Mit Radiosity erstelltes Bild

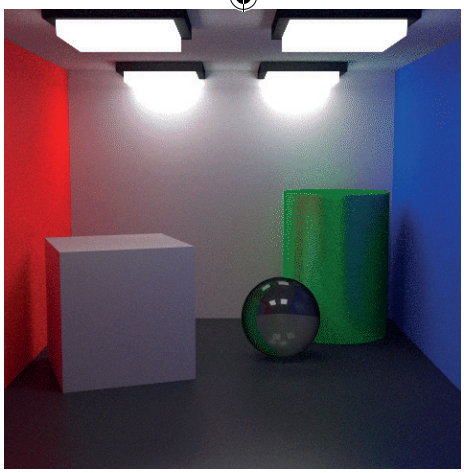


Abb. 4.11: Mit Path Tracing berechnetes Bild

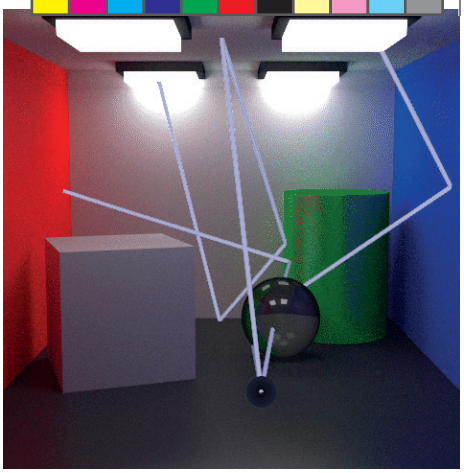


Abb. 4.12: Strahlen, die beim Path Tracing von der Kamera ausgesendet werden

GI ist nicht bloß ein System zur Berechnung von indirektem Licht. Sie ermöglicht auch die Simulation von Licht basierend auf physikalischen Gesetzen, wodurch noch realistischere Ergebnisse möglich werden. Der erste mathematische Algorithmus für eine realistische Lichtsimulation wurde bereits Mitte der 80er Jahre entwickelt.

Radiosity wurde 1985 als erster Algorithmus für die realistische Simulation von Licht von Michael Coben und Donald Greenberg vorgestellt. „Lightscape" war eines der ersten kommerziellen Softwarepakete, welche Radiosity implementierten. Im Vergleich zu heutigen Systemen sind jedoch einige Schwächen zu nennen. Der größte Nachteil liegt wahrscheinlich darin, dass jedes Mesh vor dem Rendern unterteilt werden muss und die Qualität stark von der Anzahl der Unterteilungen abhängt. Dieser Prozess ist natürlich sehr leistungsintensiv. Weiters ist Radiosity nicht in der Lage Effekte wie Reflektionen, Refraktionen oder Transparenz zu erzeugen. Ein Vorteil dieses Algorithmus' ist, dass die Lichtsimulation für die gesamte Szene berechnet wird. Das bedeutet, dass auch nach einem Wechsel der Perspektive die korrekte Lichtberechnung zur Verfügung steht [vgl. Legrenzi, „Vray the complete Guide, 2010, s 209 ff].

Path Tracing wurde 1986 von James T. Kaijya eingeführt. Dieser Algorithmus kann als Erweiterung des Ray Tracing gesehen werden. Die zur Lichtberechnung benötigten Strahlen werden bei Radiosity von der Lichtquelle aus in die Szene gesendet. Im Gegensatz dazu wird das Licht, sowohl beim Path- als auch beim Ray-Tracing von der Kamera ausgesendet. Für jeden Pixel im fertigen zweidimensionalen Bild wird zumindest ein Strahl erstellt. Dies optimiert den Renderprozess, da ein Großteil der Strahlen, die von der Lichtquelle ausgesandt werden, niemals das Auge beziehungsweise die Kamera erreichen und somit unnötig Leistung verschwenden. In der Natur sind Objekte für uns tatsächlich nur deshalb sichtbar, weil von ihnen unendlich viele Lichtstrahlen reflektiert werden, von denen einige unser Auge treffen. Eine Simulation von unendlich vielen Strahlen auf dem Computer wäre aber durch die begrenzte Leistung nicht möglich.

Bei einem Path Tracer werden die Lichtstrahlen von den Objekten in der Szenen so lange reflektiert, bis sie eine Lichtquelle treffen oder bis die eingestellte Maximalzahl an „bounces" erreicht wird. Jedes Mal, wenn ein Strahl von einer Oberfläche abgestoßen wird, wird er von deren Eigenschaften beeinflusst und abgeändert. Trifft er zum Beispiel auf ein farbiges Objekt, nimmt er dessen Farbe auf. Trifft er hingegen auf einen Körper aus matt-spiegelndem Material (z.B. gebürstetem Alu), wird der Strahl in mehrere neue geteilt und in abgeschwächter Form diffus reflektiert. Nachdem die Maximalzahl an „bounces" erreicht wurde, werden alle diese gespeicherten Eigenschaften

addiert und der Pixel entsprechend erstellt. Die Bildberechnung erfolgt durch eine progressive Annäherung. Dadurch wird die Qualität mit jeder Berechnung verbessert und der Rendervorgang kann theoretisch unendlich lang fortgesetzt werden. In der Realität ist jedoch ab einer bestimmten Stufe kein Qualitätsunterschied mehr mit freiem Auge erkennbar.

Im Gegensatz zum beschriebenen Vorgang werden beim verwandten Ray Tracing die Strahlen nicht von den Objekten abgestoßen. Stattdessen werden bei Auftreffen auf einen Körper Strahlen direkt zu den Lichtquellen gesendet und daraus die Belichtung an diesem bestimmten Punkt berechnet. Somit wird beim Ray Tracer nur der direkte Lichteinfall berechnet und eine „echte“ GI ist gar nicht möglich. Die indirekte Belichtung kann mittels einer allgemeinen Belichtung der Szene (durch Interpolation aller Lichtquellen) simuliert werden. Diese Methode wird oft in der VFX oder Spieleindustrie verwendet, um die Leistung zu optimieren [vgl. URL GI 1].

Quasi Monte-Carlo (QMC) beschreibt eine weitere mit dem Path-Tracing verwandte Methode. Genauso wie beim Path-Tracing werden hier ebenfalls Strahlen (Samples) von der Kamera aus in die Szene gesendet, welche auf Objekte auftreffen und deren Oberflächeneigenschaften aufnehmen. Auf diese Weise kann eine Vielzahl an Effekten berechnet werden, die mit Radiosity nicht möglich sind. Im Unterschied zum Path-Tracing erfolgt die Berechnung jedoch mit sogenannten „Buckets“ anstatt durch progressive Annäherung. Somit ist ein festes Ende des Rendervorgangs definiert.

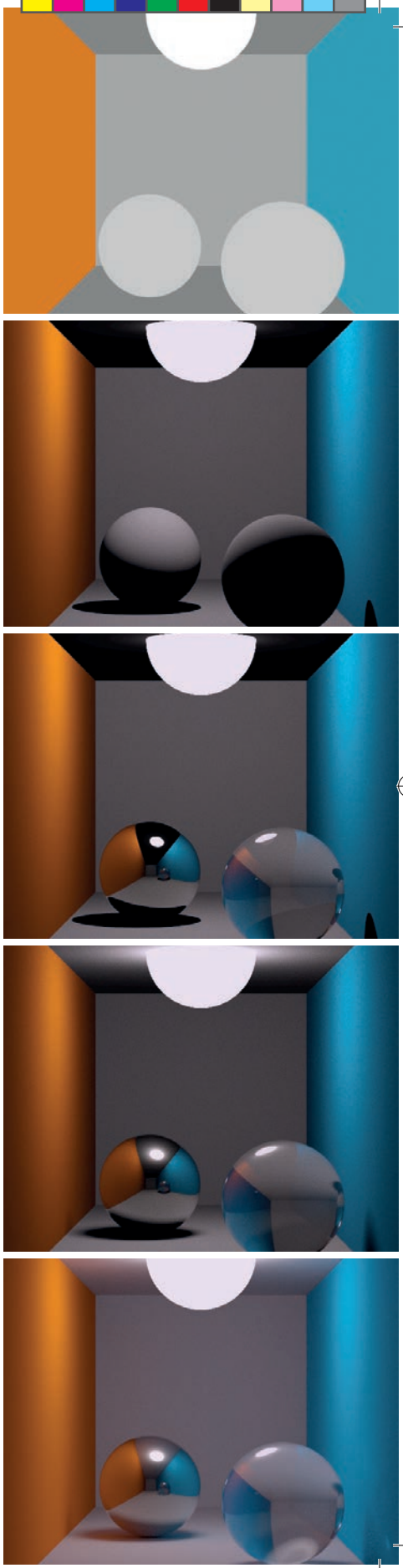


Abb. 4.13 - 4.17: Unterschiedliche Berechnungsvarianten, die mit Hilfe von Ray Tracing Algorithmen möglich sind.

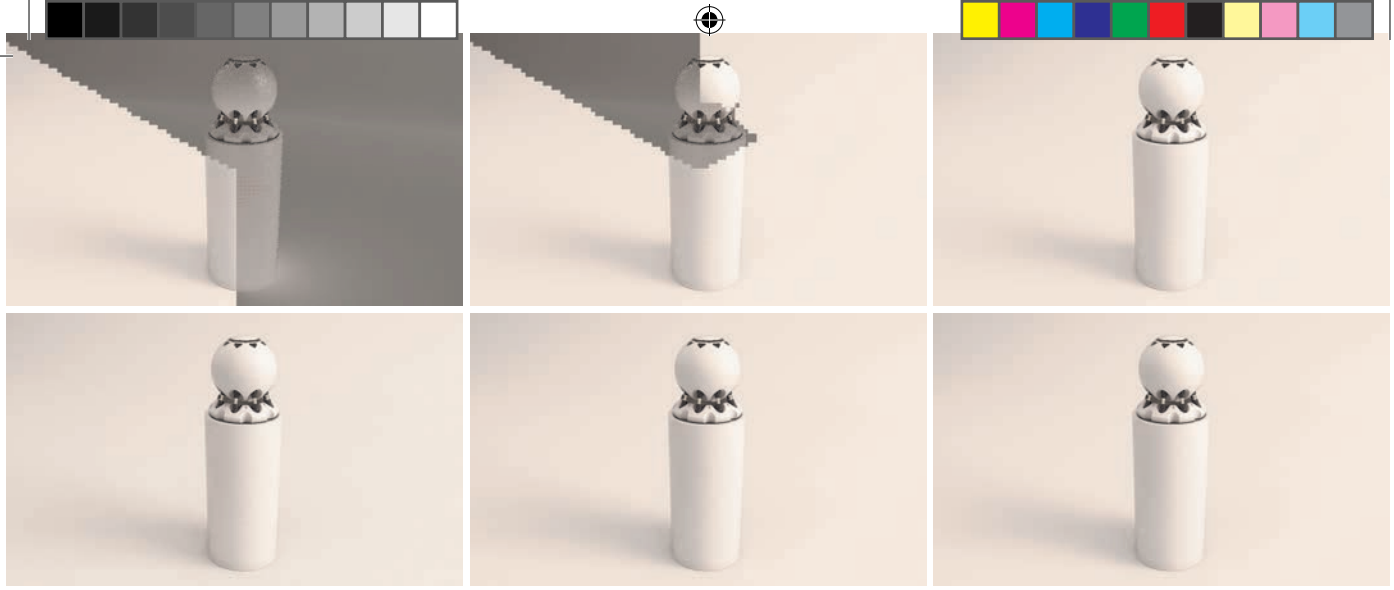


Abb. 4.23: Die obere Bildfolge zeigt eine Berechnung mit „Buckets“, einzelne Bildbereiche werden nach der Reihe gerechnet. Unten ist eine progressive Annäherung dargestellt, die Qualität vom gesamten Bild wird schrittweise verbessert.

Sowohl das Path Tracing als auch QMC haben den Nachteil, dass sie erstens vergleichsweise leistungsintensiv und langsam arbeiten und zweitens für Bildrauschen anfällig sind (Abb. 4.24). Für eine effektive Rauschunterdrückung muss die Sampleanzahl erhöht werden, was im Gegenzug längere Renderzeiten bedeutet.

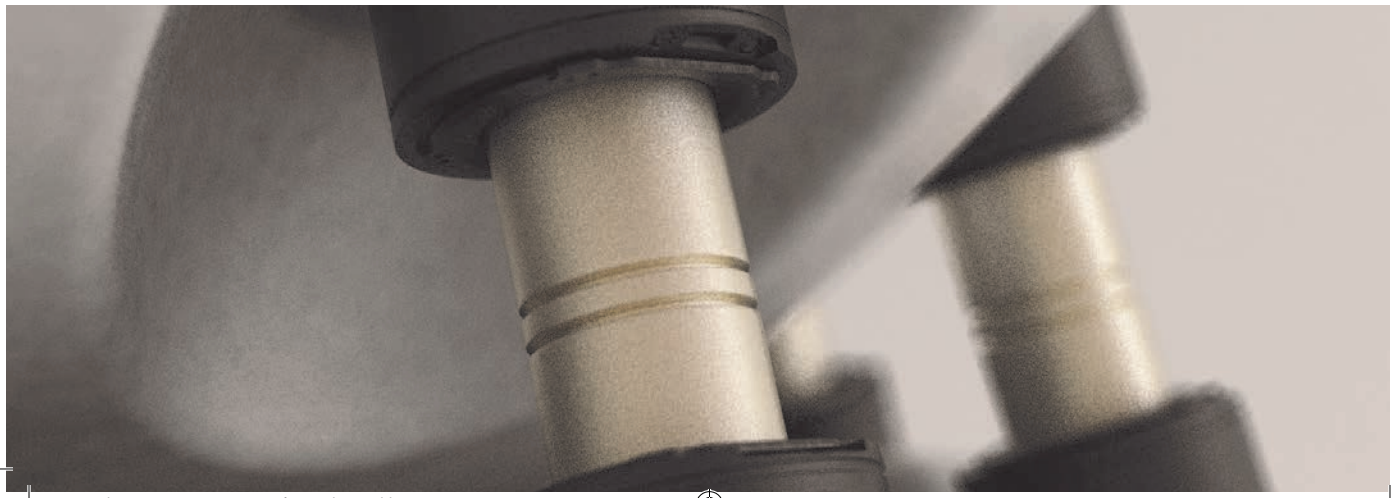
Photon Mapping wurde 1996 von Henrik Wann Jensen vorgestellt, um dem Problem des Bildrauschens und langer Renderzeiten von Ray- bzw. Path Tracing entgegenzuwirken. Es kann als eine Erweiterung dieser beiden Algorithmen verstanden werden.

Die Grundidee besteht darin, Strahlen von der Lichtquelle aus in die Szene zu schicken und - während diese im Raum herumspringen - die Stellen, wo diese auftreffen, in einer Photonenkarte zu speichern. Diese Karte enthält Informationen wie die Position des Auftreffens, die Richtung, aus welcher der Strahl kommt, und dessen Lichtstrom (Lumen).

Um eine glatte und natürliche Rekonstruktion des Lichtes im Raum zu erhalten, ist es notwendig eine sogenannte Dichteschätzung (Technik aus dem Bereich der Statistik) vorzunehmen. Stellt man diese Dichteschätzung direkt dar, so ist sehr viel niederfrequentes Rauschen sichtbar (Abb. 4.26).

Um dieses Problem zu lösen, wird ein „Final Gather Pass“ verwendet. Final Gathering ist ein System, das erlaubt, die Lichtintensität an einem bestimmten Punkt x durch Mittelung der - von diesem Punkt x aus sichtbaren - anderen umgebenden Punkte zu bestimmen (Abb. 4.28) [vgl. Legrenzi, „Vray the complete Guide, 2010, s 209 ff].

Abb. 4.24: Bei diesem Ausschnitt ist deutliches Bildrauschen erkennbar



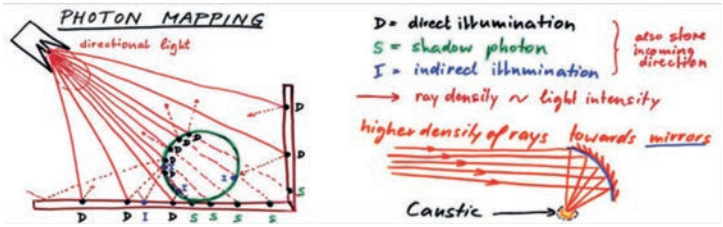


Abb. 4.25

Abb. 4.25: Die Originalskizze von Henrik Wann Jensen zu seinem 1996 vorgestellten Photon Mapping Algorithmus

Abb. 4.26

Abb. 4.26: Dichteschätzung einer mit Photon Mapping berechneten Szene. Das niederfrequente Rauschen ist deutlich sichtbar.

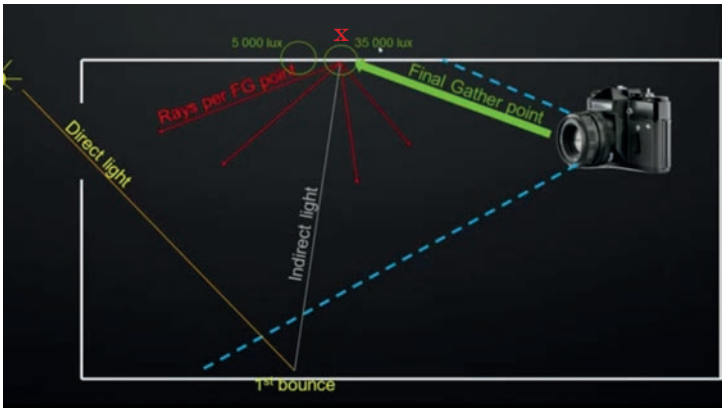


Abb. 4.28

Abb. 4.28: Das Bild zeigt eine vereinfachte Darstellung des Final Gathering mit jeweils nur einem direkten und indirekten Lichtstrahl. Der Punkt x wird in dieser Szene indirekt beleuchtet. An dieser Stelle wird ein Final Gather Punkt gesetzt, aus dessen Hemisphäre Strahlen in Richtung Szene ausgesandt werden. Wo immer diese auftreffen, wird die Lichtstärke gemessen. Anschließend wird der Mittelwert berechnet, woraus sich die Beleuchtungsstärke für den Punkt x ergibt.

Belichtung Außenszenen

Stellt man sich an einem klaren Tag auf ein ebenes Feld und schaut in den Himmel, sieht man über sich eine Halbkugel, welche die gesamte Umgebung belichtet. Das stärkste Licht kommt aus der Richtung der Sonne, dieses bildet recht klar begrenzte, scharfe Schatten. Die restliche Halbkugel strahlt ein gleichmäßiges, diffuses Licht ab, welches auch Bereiche, die nicht im direkten Sonnenlicht liegen, belichtet. Diese natürliche Lichtsituation versucht man bestmöglich mit einem sogenannten HDRI (High Dynamic Range Image, Abb. 2.22) nachzuahmen. HDRIs sind spezielle Himmelspanoramen, welche auf die Innenseite einer unendlich großen Halbkugel („Dome") projiziert werden können und einen sehr hohen Dynamikumfang aufweisen. Dadurch geben sie große Helligkeitsunterschiede (zwischen Sonne und Himmel) detailreich wieder.

Um in einer virtuellen Außenszene natürliche Belichtung zu erzielen, platziert man eine „Dome"-Lichtquelle („Domelight") irgendwo im Raum. Der genaue Punkt ist nicht ausschlaggebend, da diese Halbkugel als unendlich groß angenommen wird und somit auf jeden Fall die gesamte Szene einschließt. Danach ist es nur mehr notwendig dem Domelight ein passendes HDRI als Textur zuzuweisen und die Helligkeit entsprechend anzupassen. Nun übernimmt die Lichtquelle die Informationen aus dem HDRI und belichtet die Szene. Das stärkste Licht entspringt aus dem Bereich, in welchem die Textur die größte Helligkeit aufweist (die Sonne auf dem HDRI), vom restlichen Bereich strahlt ein schwächeres, diffuses Licht ab. Um die Stärke der Sonneneinstrahlung besser kontrollieren zu können, wird oftmals eine zusätzliche, unendliche Lichtquelle im Bereich der Sonne gesetzt. Über die Lichtstärke dieser Lichtquelle ist es möglich die Intensität des Sonnenlichts unabhängig vom restlichen Himmel zu steuern.

Dieses Lichtsetup bietet eine sehr realitätsgetreue Belichtung, da sowohl Lichtstärke als auch die Farbe über ein tatsächliches, qualitativ sehr hochwertiges Foto gesteuert werden. Das hat eine natürliche Auslichtung der Szene, wie sie das menschliche Auge bei den vorherrschenden Wetterbedingungen erwarten würde, zur Folge. Aufgrund der Vorgangsweise wird diese Technik auch als „Image-based Lighting" bezeichnet (Abb. 2.23 +2.24)

Abb. 2.22: Ein HDRI (High Dynamic Range Image), das zur Belichtung von virtuellen Szenen verwendet werden kann.

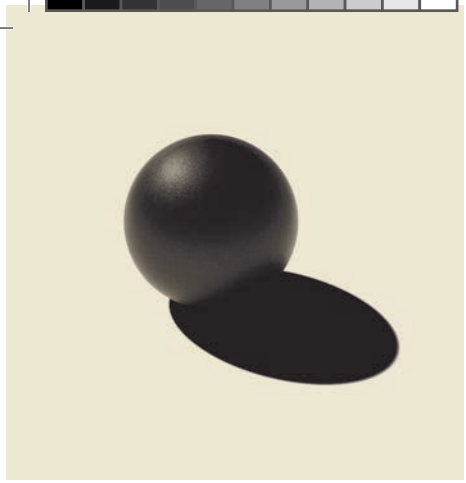


Abb. 2.23: ohne HDRI

Abb. 2.24: natürlichere Belichtung durch HDRI

Abb. 2.25: mit HDRI, geringere Schattenschärfe durch größere Lichtquelle

Belichtung Innenszenen

Bei der Visualisierung von Innenszenen wird naturgemäß verstärkt künstliches Licht eingesetzt. Trotzdem sollten Lichtquellen grundsätzlich immer so platziert werden, wie sie in der Realität auch vorhanden wären, um eine möglichst realistische Ausleuchtung der Szene zu erzielen. Je nach Situation kommen verschiedene Lichtarten zum Einsatz.

„Image-based Ligthing" oder HDRI

Dasselbe Lichtsetup, wie es bei Außenszenen zur Anwendung kommt, wird auch bei Innenbildern verwendet. Es ist ideal geeignet um Tageslichteinfall über Öffnungen in einen Innenraum zu simulieren. Außerdem kann das verwendete HDRI gleichzeitig als Hintergrundbild verwendet werden. Um die gewünschte Lichtstimmung zu erzielen, werden oftmals zusätzlich künstliche Lichtquellen im Inneren platziert.

Flächenlicht

Flächenlichter erzielen eine sehr genaue Abbildung realer Lichtquellen, wie sie verstärkt in Fotostudios eingesetzt werden. Diese „Softboxen" bieten im Vergleich zu Spot- oder Punktlichtquellen ein sehr diffuses Licht, was eine gleichmäßige Ausleuchtung ohne harte Schatten zur Folge hat. Genau wie bei echten Flächenlichtern steigt die Unschärfe der geworfenen Schatten zusammen mit der Größe des Lichts an (Abb.2.25).

Grundsätzlich können Flächenlichter auch beliebige andere Formen aufweisen. Dazu ist es notwendig das gewünschte Mesh zu modellieren und dieses dem Licht zuzuweisen. Auch das zuvor beschriebene Domelight gehört zur Gruppe der Flächenlichter.

Portallichter

In vielen Renderern gibt es die Möglichkeit bei Flächenlichtern die Option „Portallicht" zu aktivieren. Diese wird verwendet, um Innenräume effektiver und ressourcenschonender über Außenlichtquellen zu belichten. Verfügt ein geschlossener Innenraum nur über kleine Öffnungen, fallen dementsprechend nur wenige Lichtstrahlen von außen hinein. Da aber das gesamte kuppelförmige Domelight über der Szene

Licht emittiert, würde ein großer Teil davon umsonst berechnet werden. Außerdem wäre das Ergebnis nicht zufriedenstellend, da der Raum nicht ausreichend belichtet und somit zu dunkel wäre. Portallichter werden in solchen Situationen dazu eingesetzt, der Engine mitzuteilen, wo sich diese kleinen Öffnungen befinden, um an den entsprechenden Stellen gezielt mehr Strahlen auszusenden.

Spotlicht

Spotlichter werden in Innenräumen oft zur gerichteten Beleuchtung (z.B. als Leuchtmittel in Lampen) eingesetzt. Es stehen einige Optionen zur Verfügung, über welche sich der Lichtkegel und der Lichtabfall steuern lassen.

IES Lichter

Diese bilden eine Sonderform von Spotlichtern. Der IES Standard wurde erfunden, um den Austausch photometrischer Daten zu ermöglichen. Ein IES File beinhaltet gemessene Daten über die Intensitätsverteilung bei verschiedenen Lampenmodellen unterschiedlicher Hersteller. Mit Hilfe dieser Daten ist es möglich den tatsächlichen Lichtkegel eines Beleuchtungskörpers präzise in Visualisierungen wiederzugeben. Viele Hersteller stellen IES Daten ihrer Produkte im Internet zur Verfügung. Diese können mit der Lichtquelle im 3D Programm verknüpft und dadurch gerendert werden.

Punktlicht

Punktlichter sind unendlich kleine Punkte, die gleichmäßig in alle Richtungen Licht aussenden. Sie können gut dazu verwendet werden, um zum Beispiel das Licht von Glühbirnen zu simulieren, obwohl diese Vorgangsweise physikalisch nicht korrekt ist, da selbst punktuelle Lichtquellen in der Realität eine gewisse physikalische Ausdehnung aufweisen. Dafür werden aber weitaus weniger Ressourcen verbraucht, als bei einem Flächenlicht, welches einem Mesh zugewiesen ist.

Parallele Lichtquelle

Diese ist gut geeignet, um Sonnenlicht in 3D Szenen zu visualisieren. Theoretisch gesehen treffen die Lichtstrahlen der Sonne nicht parallel auf der Erde ein. Allerdings ist die Distanz zwischen Lichtquelle und Empfänger in diesem Fall so groß, dass praktisch ein paralleles Licht vorliegt. Diese Lichtquelle emittiert parallele Strahlen in eine bestimmte Richtung. Die Option „Lichtabfall“ bewirkt, dass das Licht unabhängig von der Distanz zur Quelle immer gleich hell strahlt.

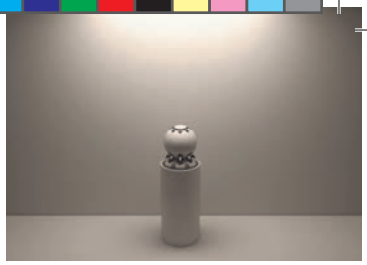


Abb. 2.26: Flächenlicht

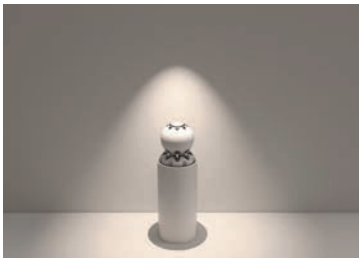


Abb. 2.27: Spotlicht

Abb. 2.28: IES Licht

Abb. 2.29: Punktlicht

Abb. 2.30: Unendliches Licht

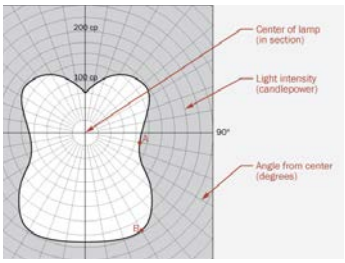


Abb. 2.30.1: Kennlinie eines Leuchtmittels. Sie gibt Auskunft über dessen Intensität und Abstrahlwinkel

Abb. 2.31

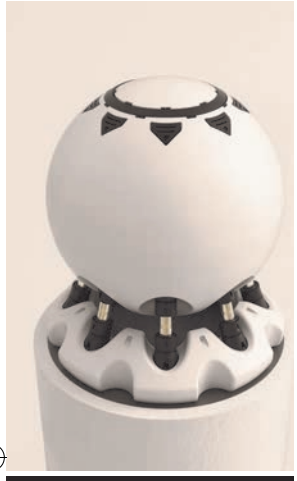


Abb. 2.32

Abb. 2.33

v.o.n.u.:
Schatten deaktiviert
Schatten aktiviert
Intensitätsabfall

Wichtige Techniken bei der Belichtung mit Game Engines

Neben qualitativ hochwertigen 3D Modellen und Texturen bildet das Licht in einer Visualisierung einen der wichtigen Bestandteile, welche für ein realistisches Bild ausschlaggebend sind. Unabhängig von der gewählten Lichtquelle, sollte man grundsätzlich immer einige wichtige Regeln einhalten, um eine naturgetreue Lichtstimmung zu erzielen:

Schatten aktivieren

„Ohne Schatten gibt es kein Licht; man muss auch die Nacht kennen lernen“ (Albert Camus)

Nachdem in der Realität jedes Licht auch Schatten wirft, sollte diese Eigenschaft auch bei allen Lichtquellen in einer virtuellen Szene aktiviert werden. Die Option Schatten zu deaktivieren steht in den meisten Renderern zur Verfügung. Sie sollte aber bei der Erstellung von fotorealistischen Szenen gemieden werden, da dies zu unnatürlichen Lichtsituationen führt, die vom menschlichen Auge sofort als falsch interpretiert werden. Objekte, welche zwar Licht empfangen, von denen aber keine Schatten ausgehen, wirken flach und schwebend.

Intensitätsabfall

Die Intensität des Lichts fällt mit steigender Distanz zur Quelle ab. Konkret ist die Intensität reziprok proportional zum Quadrat der Distanz von der Lichtquelle. Das bedeutet, je weiter weg ein Objekt von der Lichtquelle entfernt ist, desto schwächer wird es von dieser beleuchtet (Abb.2.33). Dieser Lichtabfall kann in den meisten aktuellen Engines aktiviert werden und trägt stark zum Realismus des fertigen Bildes bei.

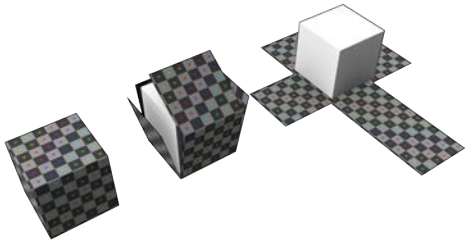
Intensität

Wie bereits erwähnt sollten Lichtquellen in Visualisierungen immer so platziert, werden wie sie auch in der echten Welt vorkommen. Ebenso muss darauf geachtet werden, dass sich die Intensität in einem realistischen Rahmen bewegt. Dafür stehen meist mehrere verschiedene Einheiten zur Auswahl. Um eine gewisse Kontinuität und Kontrollierbarkeit innerhalb einer Szene zu gewährleisten, sollte bei allen Lichtern dieselbe Einheit ausgewählt werden. Dadurch können diese besser aufeinander abgestimmt werden.

Lichtfarbe

Im realen Umfeld gibt es äußerst selten 100% weißes Licht. Selbst wenn eine Lichtquelle weiß erscheint, ist - abhängig vom Leuchtmittel – meist eine leichte Tönung vorhanden. Dies lässt sich beim Rendern über den Parameter Lichtfarbe steuern. Neben einfachen Farben können auch Shader bzw. Texturen verwendet werden, um bestimmte Effekte zu erzielen.

Abb. 2.33.1: UV Mapping bei einem Würfel.



Bedeutung von UV Koordinaten für die Lichtberechnung

Um komplexe 3D Modelle korrekt texturieren zu können, sind sogenannte UV-Koordinaten notwendig. U und V beschreiben Texturkoordinaten, welche bestimmten Punkten (x und y Koordinaten) des Polygonobjekts zugewiesen werden. UV-Mapping ist der geometrische Modellierungsprozess der Herstellung eines 2D-Bilds, welches ein 3D-Modell repräsentiert. Es erlaubt dem Nutzer ein dreidimensionales Modell mit einem zweidimensionalen Bild zu texturieren. Für Grundobjekte (Würfel, Kugel, Zylinder,...) werden die UV-Koordinaten automatisch vom 3D Programm angelegt. Diese können später manuell vom Benutzer angepasst oder neu erstellt werden. Bei simpler Geometrie kann man auch auf einfachere Standard-mappings zurückgreifen (kubisch, zylinderförmig, kugelförmig...)

Multiple UV Kanäle

Bei der Echtzeitvisualisierung werden sogenannte Light-map Texturen für die Lichtberechnung verwendet. Dadurch erspart man sich die Neuberechnung des Lichts bei jedem Frame, was der Leistung natürlich stark zugutekommt. Für diese Vorberechnung sind separate UV-Koordinaten notwendig, welche vor dem Export aus der 3D Software auf einem eigenen UV-Kanal erstellt werden müssen.

Unterschied zwischen vorberechnetem- und dynamischen Licht

Prinzipiell bieten Game Engines ähnliche Funktionen zur Belichtung von 3D Szenen wie klassische Renderer, mit dem wichtigen Unterschied, dass bestimmte Lichtquellen in Echtzeit verändert werden können. Statt viele einzelne Testbilder berechnen zu müssen, um die gewünschte Lichtstimmung einzustellen, können Parameter wie Richtung, Lichtstärke und –Farbe oder die Schärfe von Schatten mit unmittelbarer Rückmeldung im Editor eingestellt werden. Das ermöglicht Visualisierern sofort entsprechend auf die Änderungen zu reagieren.

Grundsätzlich kann der Nutzer zwischen dynamischen (in Echtzeit veränderbaren) und statischen (vorberechneten und nicht veränderbaren) Lichtquellen wählen.
Statische, nicht in Echtzeit veränderbare Lichtquellen
Diese Lichtquellen sind stark mit jenen aus bekannten Engines vergleichbar. Sie werden im Editor platziert und bieten lediglich eine Vorschau, um die Parameter einstellen zu können. Die physikalisch korrekte Belichtung wird erst nach einer Vorberechnung sichtbar. Werden danach Änderungen vorgenommen, muss erneut gerendert werden. Der Ressourcenverbrauch solcher Lichter ist vergleichsweise gering, weshalb sie so oft wie möglich gewählt werden sollten.

Dynamische, in Echtzeit veränderbare Lichtquellen

Diese Lichtquellen werden in Echtzeit berechnet, wodurch ihr Ressourcenverbrauch – verglichen mit statischen Lichtern – stark ansteigt. Je nach Einstellung können entweder nur die Lichtparameter (Intensität, Farbe, Abfall,...), oder auch die Position bzw. Ausrichtung verändert werden. Änderungen werden sofort berechnet und in der Szene sichtbar – eine Neuberechnung, um die Auswirkungen beurteilen zu können, ist nicht notwendig. Um die Vorteile von dynamischen Lichtquellen effizient nutzen zu können, gilt es vor allem den Ressourcenverbrauch im Auge zu behalten. Wie so oft im Bereich der Architekturvisualisierung wird das Limit viel mehr durch die Leistungsfähigkeit der verfügbaren Hardware, als durch die Möglichkeiten der Engine bestimmt. Aus diesem Grund sollten dynamische Lichtquellen mit Bedacht und lediglich in Situationen, wo sie tatsächlich benötigt werden, eingesetzt werden.

Dynamische Objekte bei der Architekturvisualisierung

Aus Sicht der Architekturvisualisierung bieten dynamische Lichtquellen Vorteile in mehreren Bereichen. In erster Linie können sie die Anwendung von Visualisierungen als Entwurfswerkzeug weiter verbessern. Gerade während dieses frühen Stadiums kommt es, Geometrie und Licht betreffend, oft zu weitreichenden Änderungen. Entweder gilt es unterschiedliche Lichtstimmungen zu testen, Materialien anzupassen, oder es werden entwurfsbedingte Änderungen an der Geometrie notwendig. Während mit anderen Engines gerade in dieser Phase oft viel Zeit für Testbilder verwendet wird, ist es dank der Möglichkeiten beweglicher Lichtquellen beziehungsweise Objekte in Game Engines möglich, unterschiedliche Varianten schnell durchzuspielen.

Unterschiedliche Lichteinstellungen

Verschiedene Lichtsetups können gespeichert und in Echtzeit zu Probezwecken ausgetauscht werden. Dies trifft sowohl auf natürliche Belichtung (Sonne, HDRI) als auch auf künstliche (z.B. Spots) zu. Somit kann zum Beispiel die Wirkung von zusätzlicher, künstlicher Belichtung eines Innenraums - verglichen mit rein natürlicher Belichtung - schnell dargestellt werden. Aktuelle Game Engines bieten die Möglichkeit komplette, vorberechnete Lichtsetups abzuspeichern, um diese später wiederzuverwenden. Ein möglicher Anwendungsbereich dafür ist der Wechsel zwischen Tag und Nacht innerhalb einer Szene. Beide Lichtstimmungen können separat adjustiert, vorberechnet und gespeichert werden. Anschließend ist es mit wenigen Schritten möglich zwischen Tag- und Nachtszene zu wechseln, eine neuerliche Vorberechnung entfällt (Abb.2.34) [vgl. URL Unreal Engine 4.14].

Abb. 2.34: Unterschiedliche Lichteinstellungen in einer Game Engine

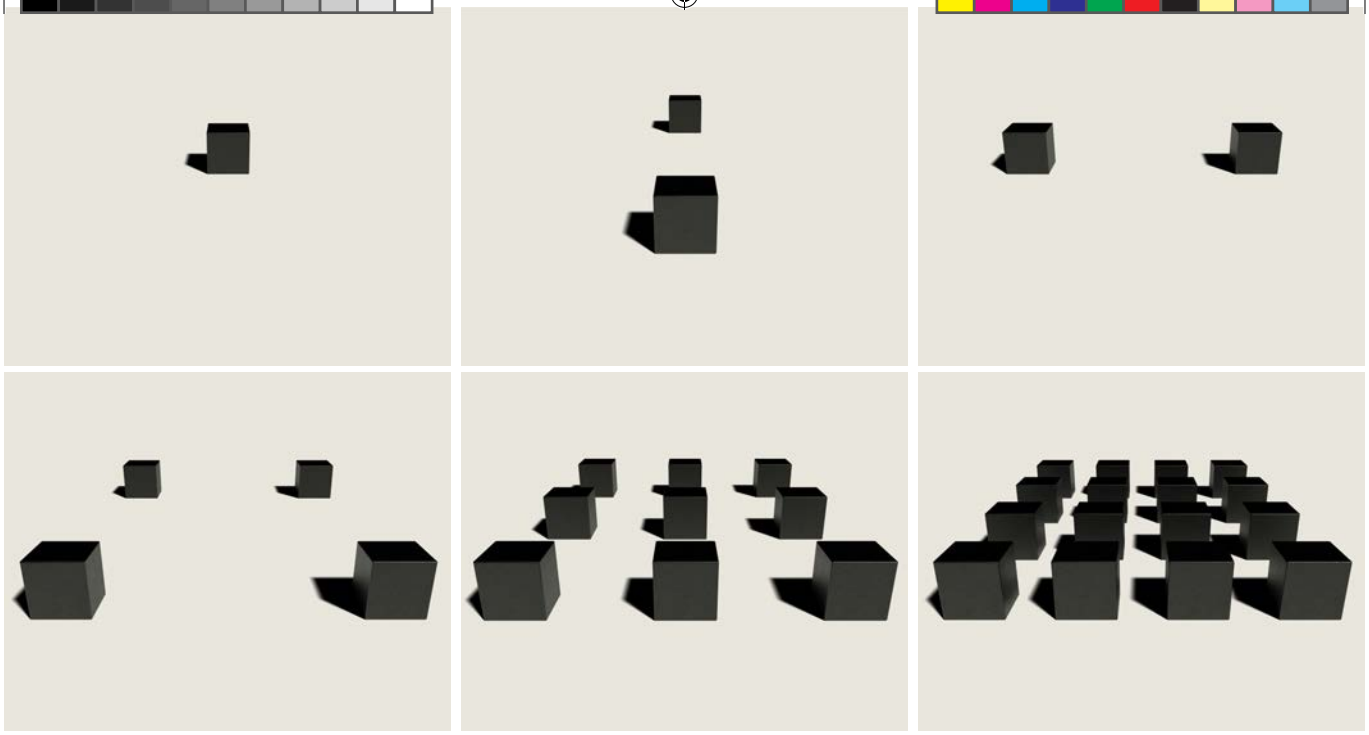


Abb. 2.35: Unterschiedliche Geometriesetups innerhalb einer Szene. Bei Verwendung von dynamischen Objekten in Game Engines entfällt die Neuberechnung.

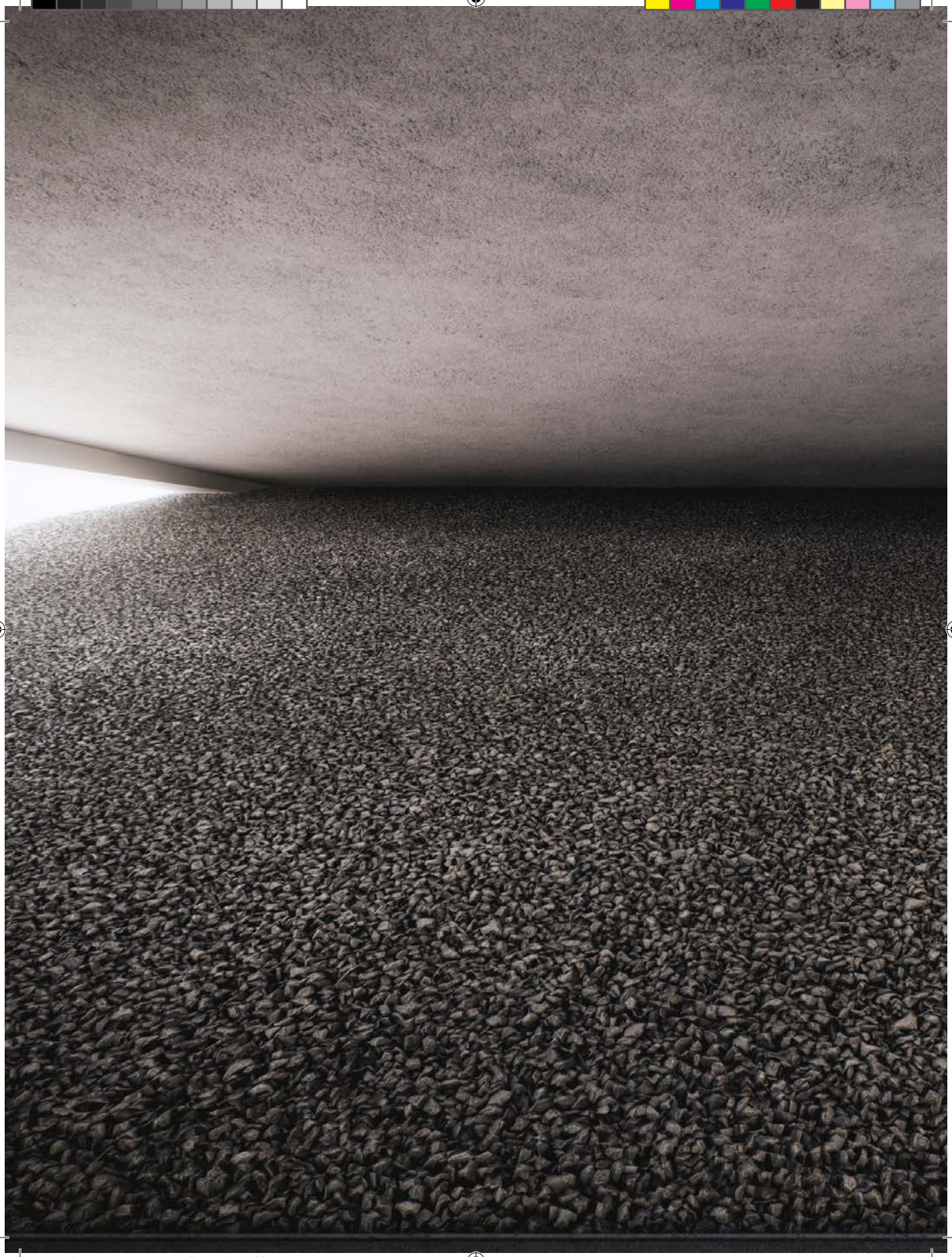
Unterschiedliche Geometrieeinstellungen

Geht es um Änderungen am Entwurf, so werden in den meisten Fällen Anpassungen an der Geometrie notwendig. Neben Lichtquellen ist es deshalb auch möglich bestimmte 3D Objekte als „beweglich" zu definieren. In Kombination mit dynamischen Lichtern werden die Belichtung des Objektes und der daraus resultierende Schatten somit in Echtzeit gerendert. Dies ermöglicht dem Nutzer Modelle innerhalb der Szenen zu verändern (verschieben, rotieren, skalieren), zu löschen oder durch Andere zu ersetzen. Diese Funktion birgt dort großes Potenzial, wo es in der Praxis häufig darum geht verschiedene Raumkonfigurationen zu testen (Interior-Design, Abb. 2.35).

Unterschiedliche Materialeinstellungen

Neben unterschiedlichen Licht- und Geometriesetups können bei Visualisierungen in Game Engines auch Materialvarianten schnell getestet werden. Eine detaillierte Beschreibung der Möglichkeiten in diesem Bereich erfolgt im nächsten Kapitel dieser Arbeit. Jedoch hat die Belichtung einer Szene maßgeblichen Einfluss auf die Wirkung eines Materials. Das - von einer Oberfläche mit bestimmter Materialität reflektierte - Licht wirkt sich auch auf die umgebende Geometrie aus. Dieser Effekt wird als Color-Bleeding bezeichnet und beschreibt den Transfer von Farbe zwischen zwei benachbarten Objekten, welcher aufgrund einer Reflektion des indirekten Lichts stattfindet.

Wird bei einem Material nun in Echtzeit die Farbe geändert, so verändert sich (bei Anwendung von dynamischen Lichtquellen) automatisch die durch Color-Bleeding auf andere Objekte übertragene Farbe. Dies ist dem Realismus stark zuträglich.



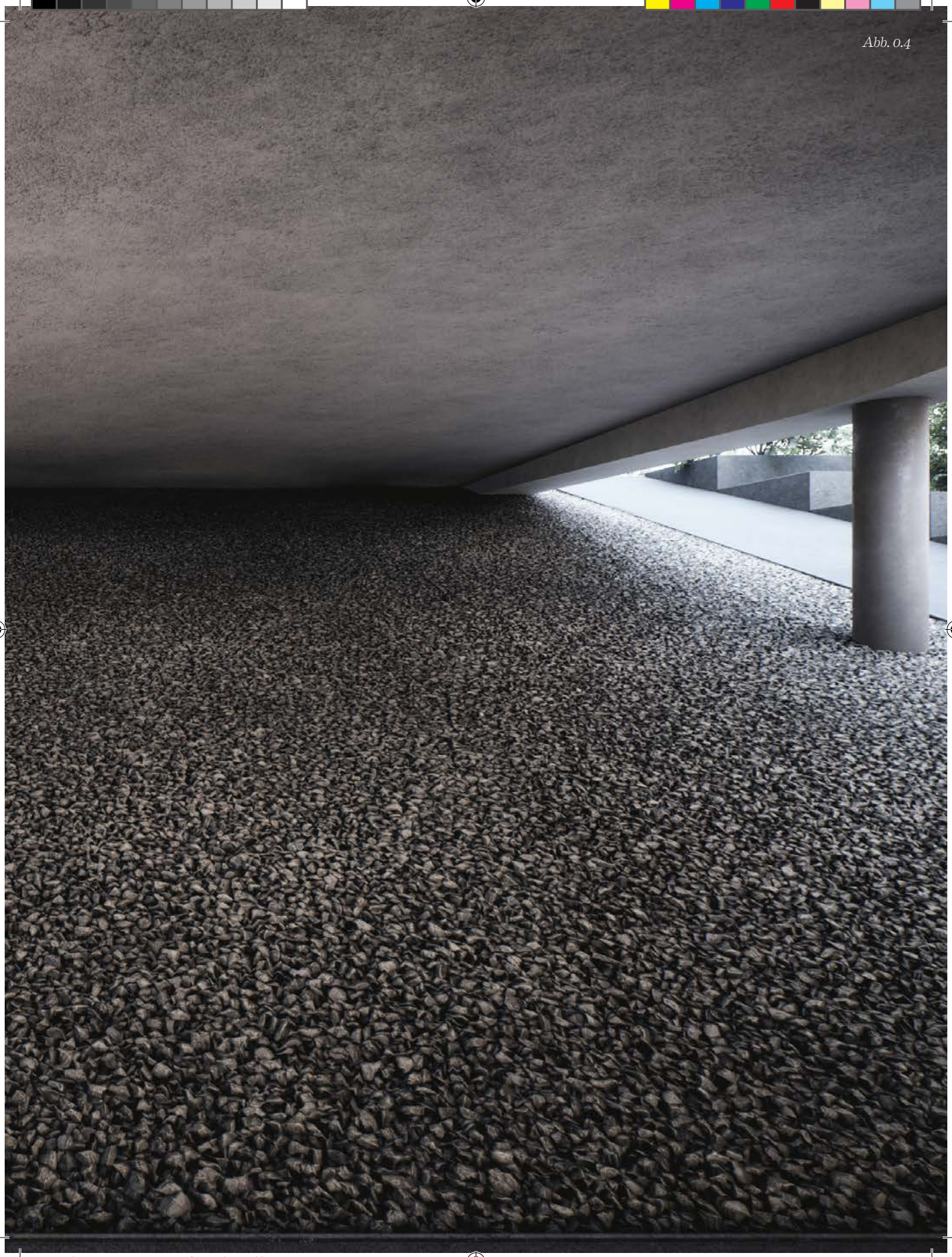


Abb. 0.4

2.3 - Material

Der Vorgang, ein 3D Modell mit einem Oberflächenmaterial zu versehen, wird im Allgemeinen als Texturierung (auch „Textur-Mapping“) bezeichnet. Dieses Material wird heutzutage für gewöhnlich von einem (meist in der Engine integrierten) Materialsystem gesteuert und kann verschiedenste Parameter enthalten. Angefangen bei einer simplen Farbe über Texturen bis hin zu komplexen Shadern zur Simulation von Unebenheiten ist nahezu jede Kombination möglich.

Neben dem 3D Modell und Lichtsetup ist das Material der dritte wichtige Aspekt, um die Basis für eine qualitativ hochwertige Visualisierung zu schaffen. Moderne Materialsysteme erlauben mit ihrer Vielzahl an verschiedenen Shadern praktisch alle erdenklichen Effekte zu kreieren. Materialien setzen sich aus unterschiedlichen Kanälen zusammen, die mit Texturen oder Shadern gefüllt werden. Aus der Gesamtheit aller Kanäle entstehen am Ende fertige Materialien. Diese werden heutzutage mit einem so genannten „pbr-workflow“ (physically based rendering workflow) erstellt. Die verwendeten Texturen und Shader sind entweder aus der realen Welt übernommen, oder sie orientieren sich stark an dieser. Dadurch wird es möglich, Materialien zu erstellen, welche für das menschliche Auge nicht mehr von Fotos zu unterscheiden sind (Abb.2.36).

Da dieser „pbr-workflow“ unabhängig von der verwendeten Anwendung funktioniert, bieten Materialsysteme von Game Engines prinzipiell ähnliche Kanäle wie bekannte Renderer:

Abb. 2.36: Material

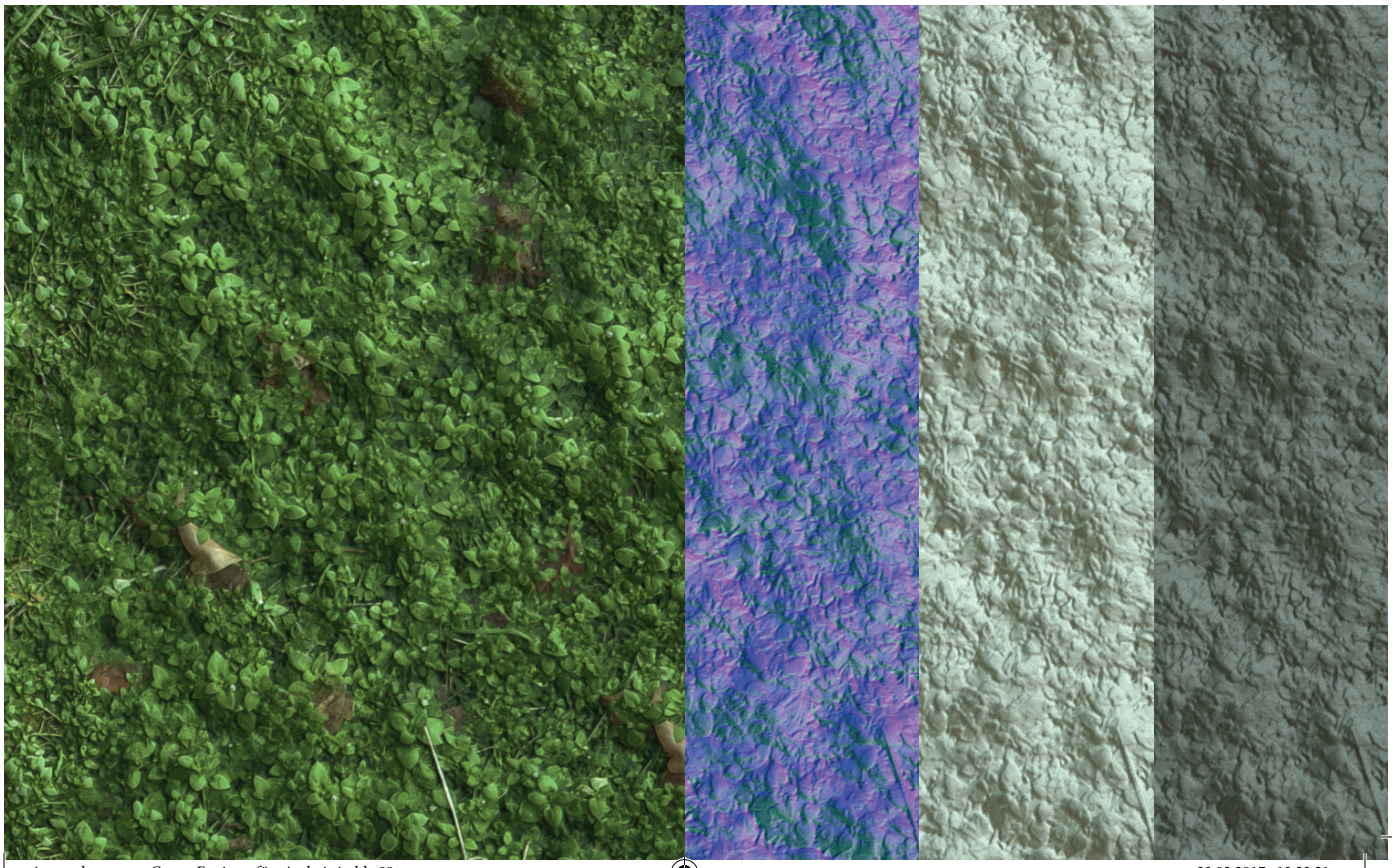


Abb. 2.37: Diffus Kanal

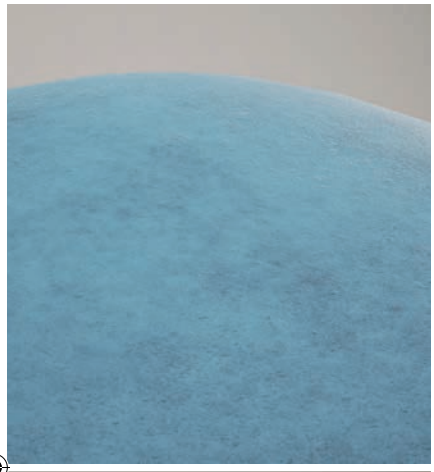


Abb. 2.38: Diffus Kanal

Abb. 2.41: Specular Kanal

Abb. 2.42: Specular Kanal

Diffus Kanal (auch „Albedo" oder „Color")

Hier wird die Farbe oder Textur des Materials festgelegt (Abb. 2.37 + 2.38). Dies kann über eine simple Farbpalette, über ein Bild oder prozedural (z.B. über einen Noise-Shader) erfolgen. Zu beachten ist, dass es in der realen Welt kein Material gibt, das zu 100% gesättigt oder komplett schwarz oder weiß ist. Zu Gunsten eines natürlichen Materials sollte auch beim Rendern auf diese extremen Einstellungen verzichtet werden.

Specular Kanal (auch „Reflection")

In diesem Kanal erfolgt die Einstellung der spiegelnden Eigenschaften (Abb. 2.41 + 2.42). Die wichtigsten Parameter sind die Stärke und die Schärfe der Spiegelung und des Glanzlichts, sowie der Fresnel IOR (Index of Refraction). Die Stärke kann entweder über einen Farbwert oder über eine Textur erfolgen. Hier ist es ebenfalls möglich die Spiegelung einzufärben, was bei bestimmten Materialien (hauptsächlich Metalle) zu realistischen Ergebnissen führt.

Die Schärfe wird ebenfalls über Texturen oder über einen Wert zwischen Null und Eins gesteuert. Im letzteren Fall stellt Null eine komplett matte und Eins eine vollkommen spiegelnde Oberfläche dar. Genau wie im Farbkanal sollten auch hier die beiden Extreme vermieden werden, da es praktisch kein Material gibt, welches vollkommen spiegelnd beziehungsweise matt ist.

Der Fresnel IOR (Index of Refraction) bestimmt den IOR, welcher bei der Berechnung des Fresnel Effekts verwendet wird. Dieser beschreibt die Tatsache, dass die Stärke der Spiegelung eines Materials vom Blickwinkel abhängig ist. Sieht man im 90° Winkel auf eine Glasfläche, so ist die Spiegelung weitaus schwächer, als wenn man diese aus einem flachen Winkel von der Seite betrachtet. Dieser IOR kann gemessen werden und ist für sehr viele Materialien in Tabellen ablesbar. Zum Beispiel besitzt Glas einen IOR zwischen 1,5 – 1,75, Plastik oder poliertes Holz 3-6, während der Wert für Metalle bei 10 beginnt und bis 200 reichen kann.

Bump Kanal

Das Bump mapping wurde in den 70er Jahren durch Jim Blinn erfunden. Es ermöglicht Unebenheiten in der Materialoberfläche mittels Texturen zu simulieren (Abb. 2.39 + 2.40). Diese Details bestehen nicht tat-

sächlich aus Geometrie und sind in der Szene eigentlich nicht vorhanden. Das menschliche Auge wird durch Licht und Schatten getäuscht und nimmt dies als Erhebungen und Absenkungen in der Oberfläche wahr. Das nimmt viel weniger Zeit in Anspruch und ist deutlich ressourcenschonender als modellierte Geometrie. Gesteuert wird dieser Effekt über den Bump Kanal mittels Graustufentexturen oder prozedural erstellten Shadern (z.B. Noise Shader). So gut wie jedes Material weist Unebenheiten in der Oberfläche auf. Diese können so fein sein, dass sie nur aus kleiner Distanz wahrgenommen werden. Trotzdem beeinflussen sie das Erscheinungsbild maßgeblich und tragen stark zum Realismus bei. Der Bump Kanal kann sowohl im sehr großen Maßstab (z.B. die hügelige Beschaffenheit einer Landschaft) als auch für sehr feine Details (Kratzer in Metallen, Poren in Putzoberflächen,...) verwendet werden.

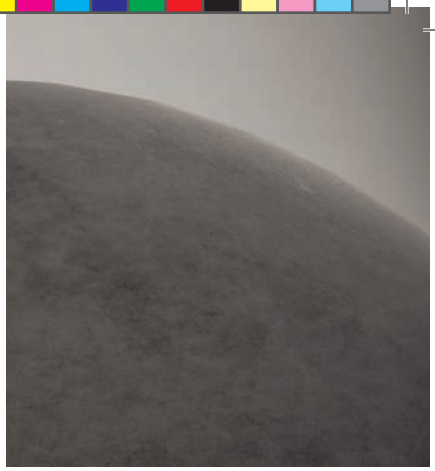


Abb. 2.39: Bump Kanal

Alpha Kanal

Dieser Kanal ermöglicht die Simulation detaillierter Geometrie mittels Texturen. Ähnlich wie beim Bumpmapping existieren die Details nicht tatsächlich in der Szene, sondern werden während des Rendervorgangs berechnet. Das sorgt für eine gute Navigierbarkeit der Szene und kurze Renderzeiten. Gesteuert wird diese Funktion über schwarz weiß Texturen beziehungsweise Shader.

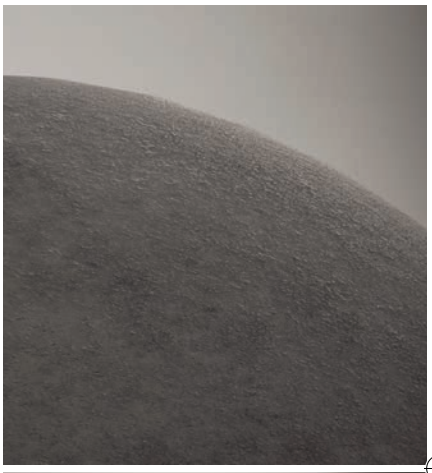


Abb. 2.40: Bump Kanal

Luminosity Kanal (auch „Illumination" oder „self-illumination")

Über diesen Kanal ist es möglich das Material leuchten zu lassen. Das ist unter anderem nützlich, um Materialien für Bildschirme oder Lampen zu erstellen. Gesteuert wird auch dieser Teil des Materials über eine Farbe, Textur oder sonstige Shader. Außerdem kann die Intensität des Lichts eingestellt werden.

Abb. 2.43: Refraktion

Refraktion

Über die Refraktion (Lichtbrechung) werden der natürliche Verlauf und die Brechung des Lichts durch Materialien simuliert. Gute Beispiele hierfür sind Glas und Wasser. Der vermutlich wichtigste Parameter ist auch hier wieder der Index of Refraction (IOR). Dieser kann für die meisten Materialien aus Tabellen abgelesen werden und steuert die Art und Weise, auf welche das Licht gebrochen wird (Abb. 2.43 + 2.44) [vgl. Legrenzi, „Vray the complete Guide, 2010, s 315 ff].

Abb. 2.44: Refraktion

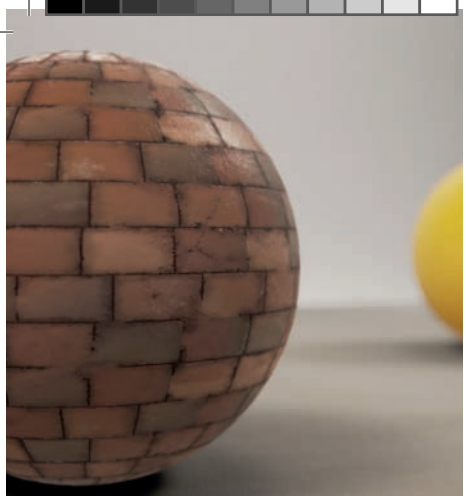


Abb. 2.45

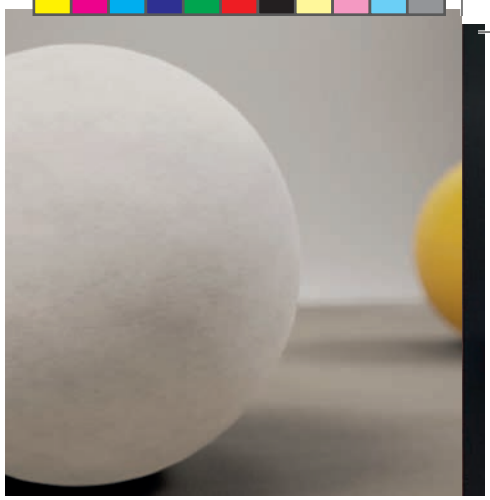


Abb. 2.45 + 2.46: parametrisch erstellte Materialien

Vorteile bei der Materialerstellung in Game Engines

Auch im Bereich der Materialien bieten Game Engines große Vorteile gegenüber bisherigen Renderern. So ist es möglich sämtliche Attribute eines Materials in Echtzeit anzupassen und die Änderungen sofort am zugehörigen Objekt zu beobachten. Außerdem ist es möglich Materialien mit wenigen Schritten zu tauschen, was immenses Potential birgt, wenn man an Änderungen während der Entwurfsphase, oder an unterschiedliche Materialvarianten bei einer Präsentation denkt.

Parametrische Materialien für Architekturvisualisierung

Neben den beschriebenen Kanälen sind weitere vorhanden, welche für spezielle Effekte verwendbar sind. Für gängige Materialien reichen die erwähnten Kanäle jedoch meist aus. Die Qualität ist maßgeblich von den verwendeten Texturen beziehungsweise Shadern abhängig. In vielen Fällen reicht es aus, Bilder (selbst fotografiert, Internet, Texturpakete...) zu verwenden. Es handelt sich dabei meist um fotografierte Oberflächen. Die unterschiedlichen Texturen, die für den jeweiligen Kanal gebraucht werden, können in Bildbearbeitungsprogrammen erstellt werden. In vielen Fällen ist es jedoch schwierig, gewünschte Ergebnisse exakt mit vorhandenem Bildmaterial zu rekonstruieren oder die Materialien wirken zu perfekt, sauber und langweilig, wodurch die Szene schnell als Visualisierung erkannt wird.

Abhilfe schaffen hier Anwendungen, die speziell zur Erstellung von Texturen geschaffen sind und die mit Hilfe von Plug-Ins mit der jeweiligen Engine verbunden werden. Sie ermöglichen die parametrische Erstellung von allen - für ein beliebiges Material benötigten - Texturen. Angefangen mit einem leeren Bild können so nach und nach verschiedene Elemente zu einer fertigen und komplexen Textur zusammengesetzt werden. Die für die unterschiedlichen Kanäle benötigten Texturarten (diffus, specular, bump,...) können mit wenigen Schritten direkt generiert werden, was die zusätzliche Anwendung von Bildbearbeitungsprogrammen überflüssig macht. Spätere Anpassungen am Material sind leicht möglich, da sämtliche Elemente, aus denen sich die Textur zusammensetzt, parametrisch aufgebaut sind und unabhängig voneinander bearbeitet werden können. Vergleichbar mit Ebenen in einer 3D Anwendung können bestimmte Elemente ausgeblendet werden, wenn diese momentan nicht erwünscht sind.

Abb. 2.46: Parametrische Materialien

Gerade in Kombination mit den Möglichkeiten von Game Engines bieten solche Systeme große Vorteile. Mit den vorhandenen Plug-Ins entsteht eine sehr gute Anbindung und Parameter, die während der Texturerstellung freigegeben wurden, können später während der Visualisierung in der Engine angepasst werden. Angefangen bei der Farbe, über Spiegelung oder Oberflächenbeschaffenheit bis zu Alterungserscheinungen bei bestimmten Materialien, gibt es eine nahezu grenzenlose Zahl an Anwendungsformen. Außerdem können mit diesem System einige wenige „Mastermaterialien" erstellt werden, aus denen durch verschiedene Variationen eine Vielzahl an „Submaterialien" entstehen kann. Damit ist es möglich, eine effiziente Materialbibliothek aufzubauen, welche projekt- und engineübergreifend verwendet werden kann.

Abb. 2.48: Aufgrund von Tessellation sehr realistisch wirkendes Material.

Simulation von Oberflächeneffekten durch Material

In der Realität besitzt nahezu jedes Material Unebenheiten an der Oberfläche. Diese können entweder der Materialbeschaffenheit selbst, oder Abnutzungen, die mit der Zeit entstanden sind, geschuldet sein. Es kann sich dabei um feine Details – wie die Poren einer Sichtbetonwand – oder deutlicher erkennbare Vertiefungen – wie die Fugen zwischen einzelnen Fliesen – handeln. Um ein möglichst realistisches Abbild des gewünschten Materials zu erzeugen, ist es notwendig diese Unebenheiten darzustellen. Dies sollte, wenn möglich, allerdings am besten mittels Texturen geschehen, da diese um vieles ressourcenschonender arbeiten als modellierte Geometrie. Würde man versuchen die feinen Unebenheiten einer verputzen Wand mittels Geometrie darzustellen, würde man schnell an die Leistungsgrenzen aktueller Hardware stoßen. Ähnlich, wenn auch in einem größeren Maßstab, verhält es sich mit Holz- oder Fliesenböden. In gewissen Fällen kann es notwendig werden, die einzelnen Elemente einer solchen Oberfläche tatsächlich zu modellieren. Meist reicht jedoch eine Simulation der Unebenheiten mittels entsprechender Texturen vollkommen aus und spart Ressourcen, welche in anderen Bereichen besser verwendet werden können.

Abb. 2.47: Aufgrund von Tessellation sehr realistisch wirkendes Material.

Wie bereits beschrieben bietet die einfachste Variante solche Effekte darzustellen das Bumpmapping. Über Licht & Schatten wird hier die gewünschte Illusion kreiert. Ist höhere Qualität notwendig, kann man auf das so genannte Displacement zurückgreifen (Abb. 2.49). Es funktioniert ähnlich wie Bumpmapping, allerdings wird hier die Geometrie tatsächlich während des Rendervorgangs anhand der vorgegebenen Textur deformiert. Dies erzeugt detaillierte und überzeugende Ergebnisse, wenn auch auf Kosten längerer Renderzeit. Bei Game Engines spricht man oft von Parallax-Mapping (ähnlich wie Bumpmapping – keine „echte" Geometrie wird erstellt) und Tessellation (ähnlich wie Displacement, Geometrie wird erstellt, leistungsintensiver, Abb. 2.47 + 2.48).

Alpha

Ein weiterer nützlicher Effekt, der mittels Material simuliert werden kann, ist das Alphamapping. Es ist vergleichbar mit Ebenenmasken in einem 2D Bildbearbeitungsprogramm. Der Alpha Kanal erlaubt über eine Schwarzweiß-Textur bestimmte Bereiche eines 3D Objekts durchsichtig erscheinen zu lassen. Diese Technik findet beispielsweise bei Blättern von Bäumen sehr stark Anwendung. Statt die detaillierte Form jedes einzelnen Blattes mittels vielen Polygonen darzustellen, reicht ein einfaches Polygon, auf das eine Blatttextur gelegt wird. Befindet sich nun dieselbe Textur als schwarzweiß-Version im Alpha Kanal, so werden alle Bereiche des Polygons, welche nicht zum Blatt gehören, ausgeblendet (Abb. 2.50). Damit ist es möglich, komplexe Geometrie mittels sehr wenigen Polygonen exakt abzubilden.

Abb. 2.49: Das selbe Material mit und ohne aktiviertem Displacement

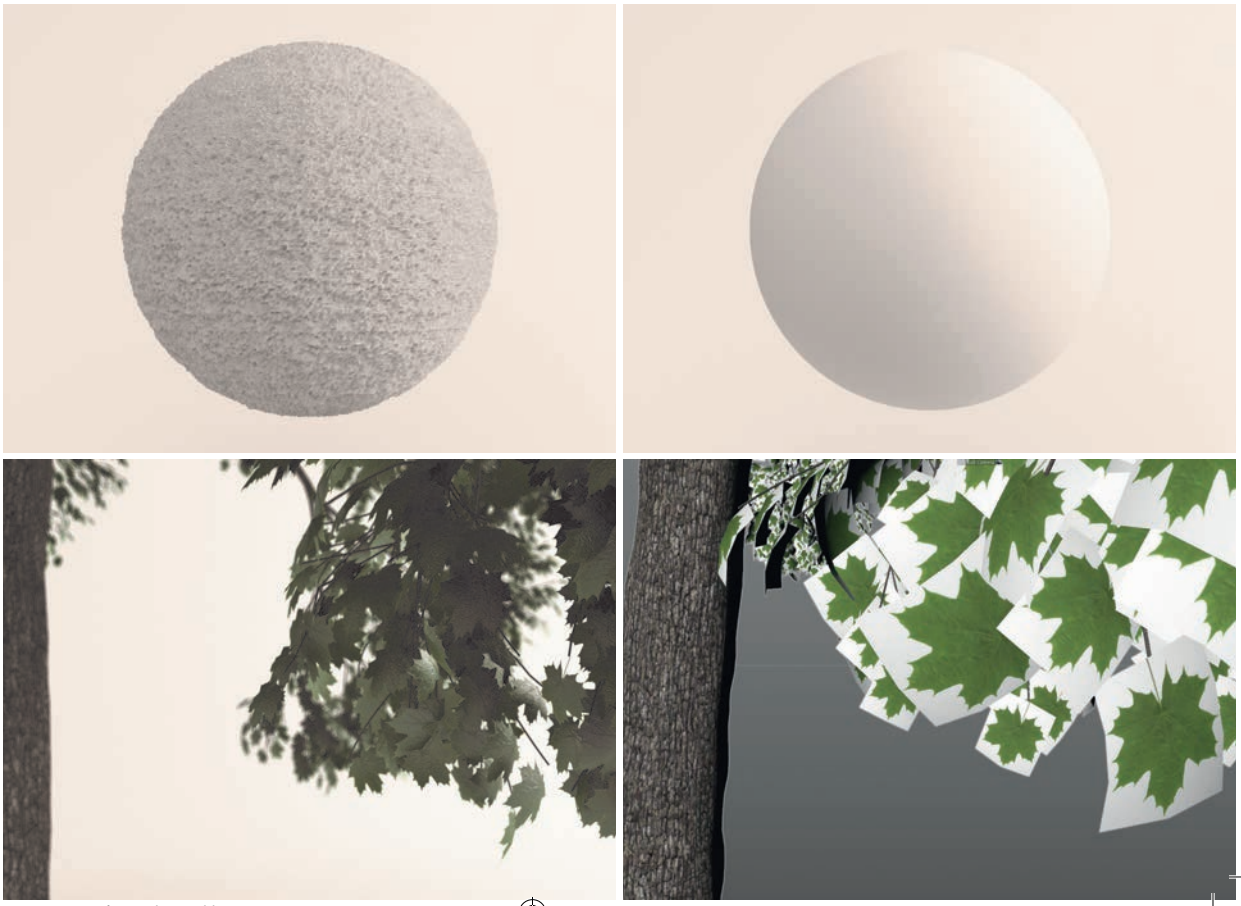


Abb. 2.50

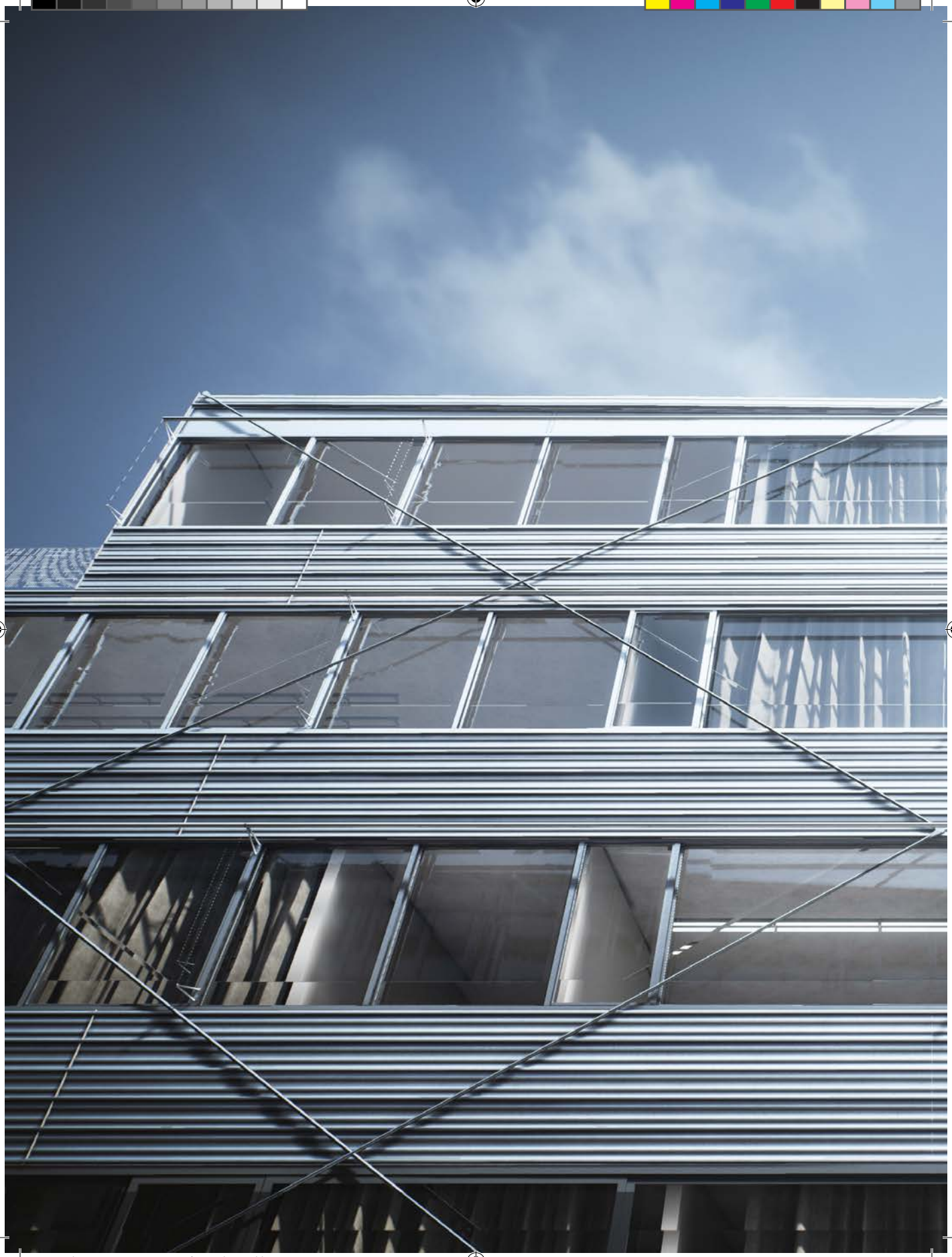


Abb. 0.6

2.4 - Postproduktion

Klassische Postproduktion

Als Postproduktion (auch Nachbearbeitung) bei Architekturvisualisierung werden sämtliche Arbeitsschritte genannt, die nachträglich am fertig berechneten Bild erfolgen. Nach dem Rendervorgang werden die Bilder (bei Animationen: die Bildfolge) in Drittanwendungen exportiert, um dort weiter bearbeitet zu werden. Meist handelt es sich dabei um jene Programme, die auch in der Film & Fotoindustrie verwendet werden (z.B. Adobe Photoshop oder After Effects). Dort erfolgen Korrekturen, Retuschen und Optimierungen in unterschiedlichem Ausmaß, welche die Qualität der Visualisierung deutlich steigern.

Geht es um den Einsatz und die Bedeutung von Postproduktion in der Architekturvisualisierung, gibt es grundsätzlich zwei Ansätze:

- So viel wie möglich in 3D darstellen – nur wenig nachbearbeiten
- Nur das Wichtigste in 3D darstellen – den Rest in der Nachbearbeitung erledigen

Beide Arbeitsweisen haben sowohl Vor- als auch Nachteile. Die unterschiedlichen Herangehensweisen sollen durch zwei Beispiele verdeutlicht werden (Abb. 2.51 + 2.52).

Abb. 2.51: Die wichtigsten Elemente wurden in 3D modelliert und visualisiert. Der Großteil der Arbeit erfolgte jedoch nach dem eigentlichen Renderprozess in einer Bilderbearbeitungssoftware

Abb. 2.52: Der Großteil der Arbeit erfolgte in diesem Beispiel im 3D Programm. Später wurden nur noch kleine Korrekturen durchgeführt und einzelne Elemente hinzugefügt.

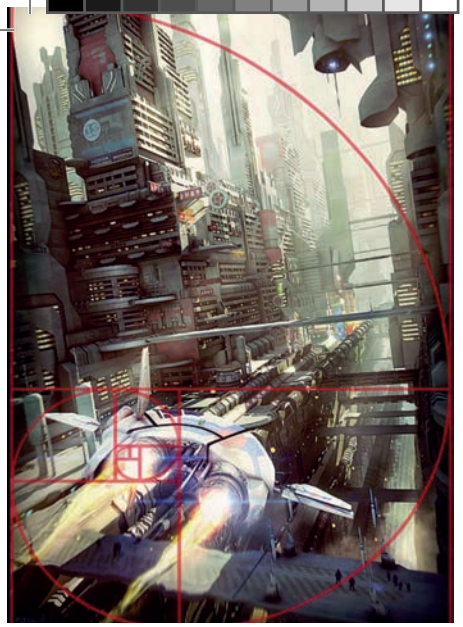


Abb. 2.53: Bildkomposition

Abb. 2.54: Bildkomposition

Abb. 2.55: Bildkomposition

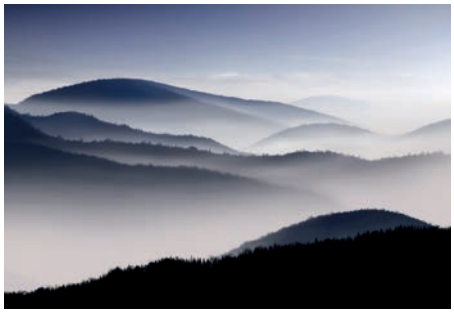


Abb. 2.56: Tiefenstaffelung

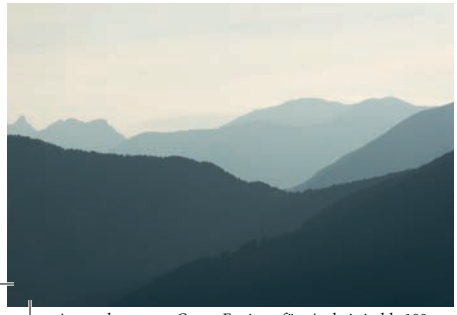


Abb. 2.58: Tiefenstaffelung

Um die neuen Möglichkeiten, welche sich durch Game Engines in der Nachbearbeitung eröffnen, zu verstehen, ist es wichtig, einen groben Einblick in die gängigsten Schritte der Postproduktion zu erhalten. Natürlich bestehen bei jedem Projekt individuelle Anforderungen an den Bildaufbau, den Inhalt, die Stimmung und viele andere Punkte. Trotzdem gibt es einige allgemein anerkannte Anhaltspunkte, die beachtet werden sollten, um eine qualitativ hochwertige Visualisierung produzieren zu können:

Kontrast, Helligkeit, Farbkorrekturen:

Die wohl bekannteste Form der Nachbearbeitung ist die Anpassung von Kontrast, Helligkeit oder Farben. In Bildbearbeitungsprogrammen steht eine große Anzahl an unterschiedlichen Werkzeugen und Methoden zur Verfügung, um diese grundlegenden Korrekturen durchzuführen.

Bildkomposition:

Hier geht es unter anderem um den Bildausschnitt, die Blickführung oder die Positionierung von Objekten innerhalb des Bildes. Es gibt eine Vielzahl anerkannter Regeln, die allesamt aus dem Film & Fotobereich stammen und bereits zahlreich erprobt wurden (Abb. 2.53 - 2.55).

Tiefenstaffelung:

Um ein spannendes Bild mit dem Eindruck von Tiefe zu erzeugen, sollte auf ein korrektes Kontrastverhältnis zwischen hellen und dunklen, aber auch nah und weit entfernten Stellen geachtet werden. Grundsätzlich verblassen Objekte mit steigender Distanz aufgrund von Nebel oder Unreinheiten in der Luft. Es wirkt, als ob sie heller und kontrastärmer werden würden. Im Gegensatz dazu erscheinen Objekte im Vordergrund dunkler und kontrastreicher (Abb.2.56 + 2.58).

Tiefenschärfe:

Je nach Brennweite und Blendeneinstellung des Objektivs erscheinen Bildbereiche, welche außerhalb des Fokus liegen, mehr oder weniger verschwommen. Dieser Effekt wird in der Fotografie oft verwendet und bietet – richtig angewandt - auch für Architekturvisualisierung eine große Qualitätssteigerung (Abb.2.59).

Vignette:

Viele Effekte, die in der Postproduktion bei Visualisierungen erstellt werden, stammen aus der Fotografie. Um ein möglichst fotorealistisches Ergebnis zu erzielen, wird versucht, Eigenschaften von Fotos, welche bedingt durch Kameraobjektive entstehen, nachzuahmen. Die Vignettierung – also die Abdunkelung des Bildes zu den Rändern hin – ist ein solcher Effekt (Abb.2.60) . Oft werden auch andere Besonderheiten, wie beispielsweise die chromatische Aberration, welche in der Fotografie eigentlich als „Fehler" bezeichnet wird, eingebaut, um das menschliche Auge noch stärker zu täuschen

Die beschriebenen Punkte bilden nur einen kleinen Ausschnitt aus den zahlreichen Möglichkeiten, die sich einem Visualisierer in der Postproduktion bieten. Sie sollen lediglich zur Veranschaulichung der Vorteile der Postproduktion mit Game Engines dienen. Anders als bei der klassischen Visualisierung mit bekannten Render Engines, bei der zunächst das Bild gerendert, exportiert und im Anschluss nachbearbeitet wird, beherrschen Game Engines die direkte, interne Postproduktion. Sehr viele wichtige Posteffekte, angefangen bei Helligkeit, Kontrast und Sättigung über Vignette, Tiefenschärfe und Farbkorrektur bis hin zu Bildrauschen oder Objektivverzerrung, können in Echtzeit implementiert werden. Mittels einer LUT Funktion („look-up-table") können sogar einzelne Bilder gespeichert, in einer separaten Bildbearbeitungsanwendung angepasst und wieder importier werden. Sämtliche Bearbeitungsschritte, die an diesem Bild vorgenommen wurden, werden dann automatisiert auf die gesamte Szene angewandt.

Dieses System bietet enorme Vorteile, da viele Effekte bereits frühzeitig sichtbar sind und angepasst werden können und nicht erst nachträglich hinzugefügt werden müssen. Außerdem lässt sich die Wirkung und Stimmung eines Bildes sehr stark und einfach beeinflussen, ohne die Szene an sich zu verändern. Dadurch können viele Szenarios erprobt werden und Entscheidungen für das Endprodukt schnell getroffen werden (Abb.2.62 - 2.64) [vgl. URL Composition].

Abb. 2.59: Tiefenschärfe

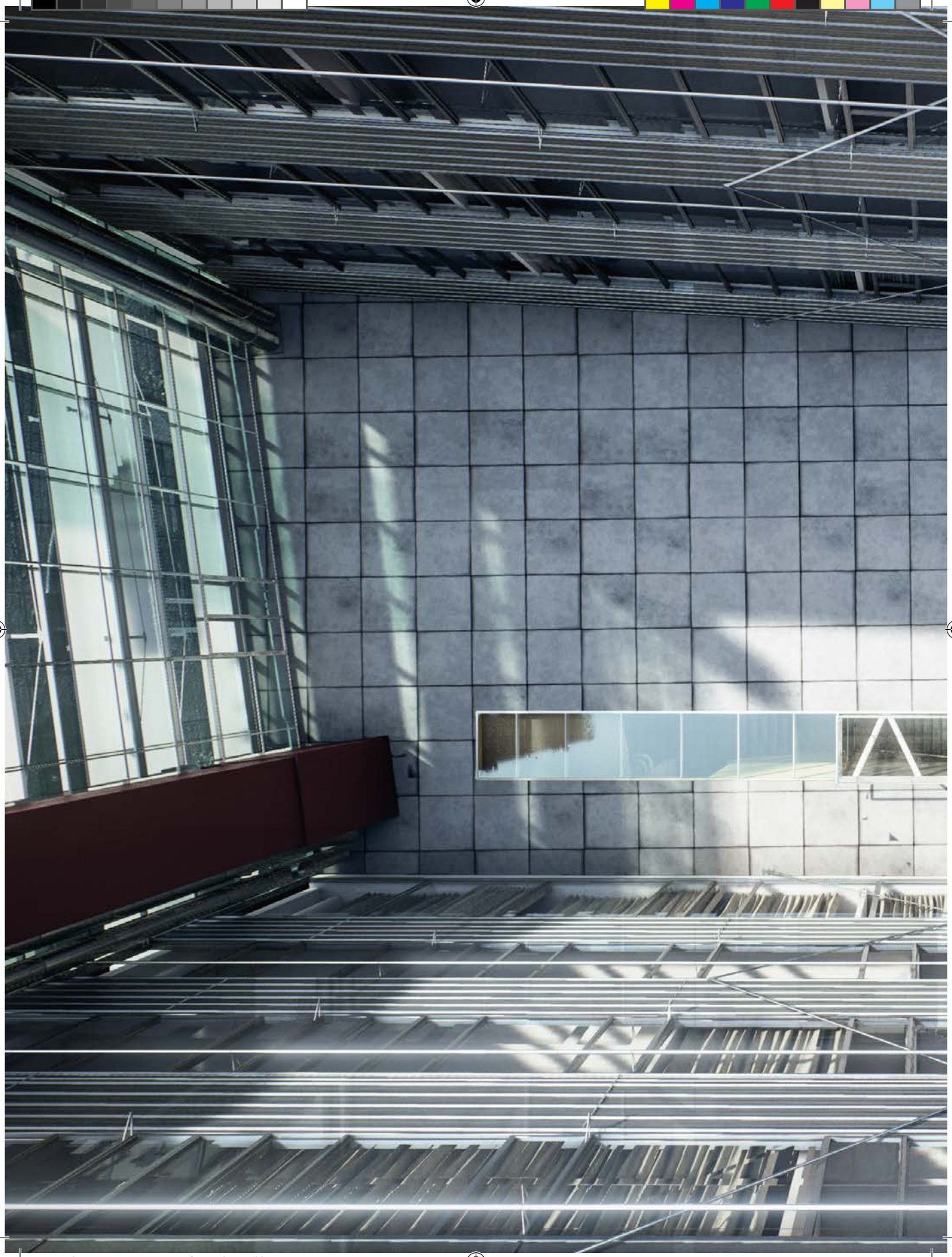
Abb. 2.60: Vignette

Abb. 2.62: Post-Processing

Abb. 2.63: Post-Processing

Abb. 2.64: Post-Processing

Abb. 2.61: Mit dem Post Process Volume können Effekte engineintern eingestellt werden.



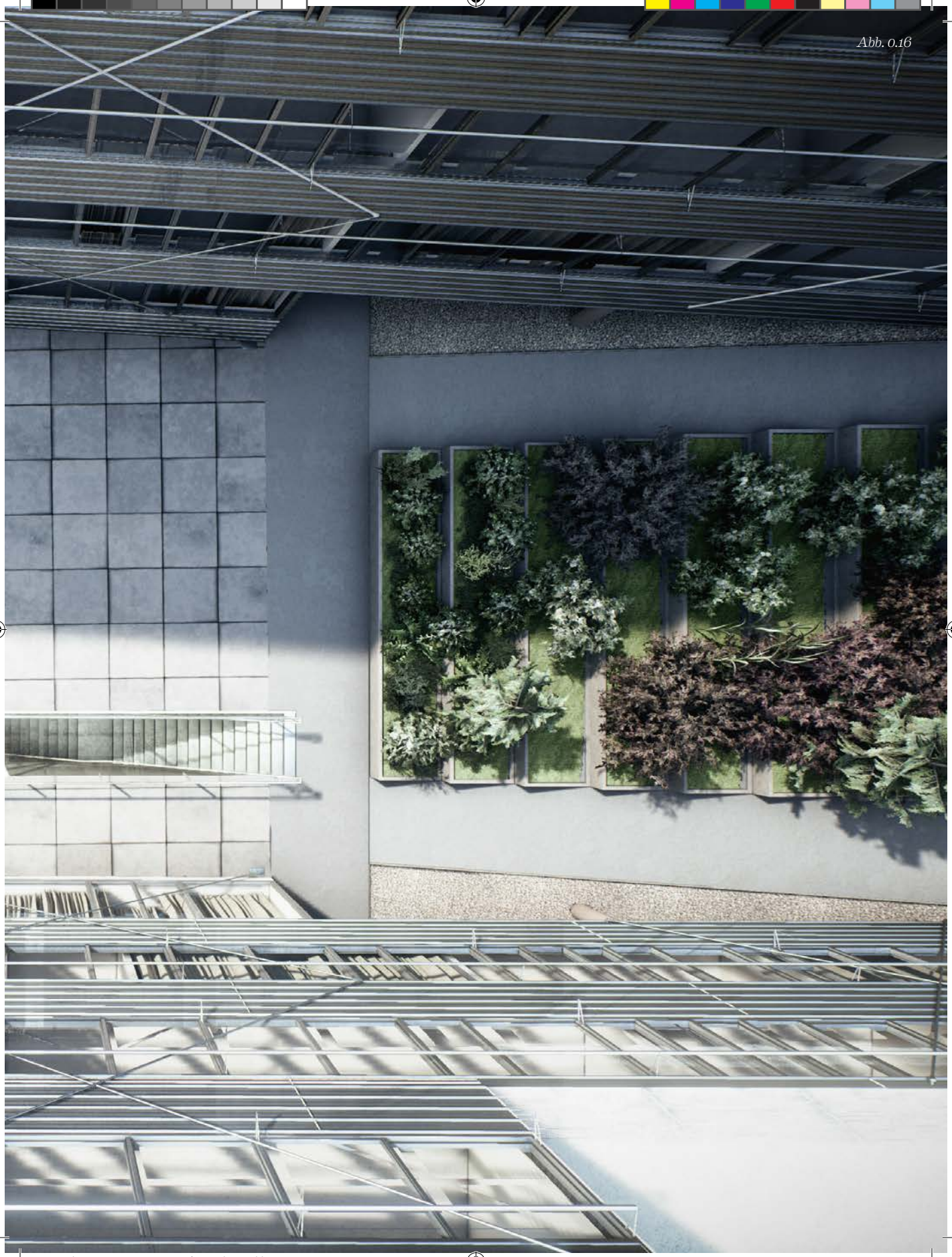


Abb. 0.16

Abb. 0.11

3 - Projektteil

3.1 - Der Architekt Helmut Richter

Helmut Richter war ein österreichischer Architekt und Hochschullehrer. Neben diversen Ausstellungsarchitekturen und den Restaurants Kiang zählen das Wohnhaus in der Brunner Straße und die Schule am Kinkplatz zu seinen Hauptwerken.

Geboren 1941 in Graz, studierte er Architektur an der TU Graz und Informationstheorie sowie System- und Netzwerktheorie an der University of California in Los Angeles. In Kalifornien war er auch als Forschungsassistent tätig, bevor er 1971 als Professor an der École nationale supérieure des beaux-arts de Paris lehrte. 1977 folgte die Gründung des eigenen Wiener Büros gemeinsam mit Heidulf Gerngroß. Die ersten Wohnbauten wie das Haus Königseder in Oberösterreich und die Glasfassade der Wohnhausanlage an der Brunner Straße sorgten für Aufsehen. Ab 1986 war Richter zusätzlich als Lektor an der Hochschule für angewandte Kunst und als Gastprofessor an der Gesamthochschule Kassel (bis 1987) tätig. Von 1991 bis 2007 war er Universitätsprofessor an der Technischen Universität Wien an der Abteilung für Hochbau 2.

Ende der 1960er Jahre entwarf Helmut Richter mehrere Wettbewerbsbeiträge und Prototypen für Möbel. 1967 ging er beim „Wittmann-Möbel-Wettbewerb" mit seinem Entwurf für ein mechanisches Bett, das durch einfache Manipulation in eine Bank verwandelt werden kann, als Sieger hervor. Sein Prototyp wurde später bei einer Ausstellung im MAK präsentiert. Es folgten weitere Entwürfe (u.a. mechanischer Sessel, Möbel für „Die Erste Österreichische Sparcasse Bank") und - eines der ersten großen Projekte – das Haus Königseder in Baumgartenberg. Richter entwarf es gemeinsam mit seinem damaligen Partner Heidulf Gerngross 1977 bis 1980 für den Arzt Jörg Königseder. Das Architektenteam entwickelte einen fremdartig wirkenden Baukörper aus Paraschalen und kleinwelligem Blech, welcher an das bestehende Haus angekoppelt wurde. 2003 wurde der Bau im Zuge einer Dachsanierung über dem Ordinations- und Wohnbereich aufgestockt. „Als Bauherr wusste ich sowohl beim ersten, als auch beim zweiten Umbau, dass ich mich intensiv mit Richters Architektur auseinandersetzen muss. Seine Objekte sind keine „Ohrwürmer", die sofort schmeicheln. Ich spürte während seiner Planungsarbeit sein angestrengtes, beharrliches Bemühen, sein faustisches Streben nach Vollkommenheit" berichtet Jörg Königseder über die Zusammenarbeit mit Richter. Auch das 1982 fertig gestellte Haus Plattner in Sollenau ist ein früher Bau, der als „Systemkritik am Thema Landhaus" gilt [vgl. URL Helmut Richter 1, sowie MA19, „Projekte und Konzepte, Heft 3, Ganztageshauptschule Kinkplatz Wien 14", 1995]

1982 folgten das Bad Sares, ein unkonventionelles Bad in einer Stadtwohnung im dritten Wiener Gemeindebezirk, das gänzlich ohne Fliesen auskommt [vgl Schartner/Fasch, „Ein Buch für Helmut Richter", 2007, 116 f], und das Restaurant Kiang I in der Rotgasse [vgl. URL Helmut Richter 3]. Im „Buch für Helmut Richter" schreibt Dietmar Steiner:

„Man kann sich heute kaum mehr vorstellen, welche Aufregung das erste Kiang-Restaurant im Wien des Jahres 1985 erzeugte. [...] Mit diesem Lokal wurde ein architektonischer Kontrapunkt zur inzwischen schon klassischen Wiener „kleinen Architektur" gesetzt. Alles ist hier härter, internationaler. Die klaren Farben, die radikalen Materialien. Antizipiert ist hier eine metropolische Alltäglichkeit, die sich den konstruktiven und ästhetischen Möglichkeiten heutiger Zeit vergewissert." (Dietmar Steiner, 2007)

Abb. 3.1: Haus Königseder

Dietmar Steiner übernahm zu jener Zeit die beratende Funktion für ein durch Erhard Busek und Jörg Mauthe initiiertes Programm, wodurch Architekten auch für den sozialen Wohnbau engagiert werden sollten. Durch seine Beteiligung wurde Helmut Richter beauftragt, das Ergebnis war die Wohnanlage Brunner Straße. Als Schallschutz gegen die stark befahrene Durchzugsstraße wurde eine Verglasung auf verzinkten Stahlrahmen verwendet. Beton in Skelettbauweise kam für die Tragkonstruktion zum Einsatz. Die Erschließung erfolgt über Laubengänge, welche durch die rahmenlose Glaswand vom Lärm der Straße abgeschirmt sind. Gleichzeitig ließ der Architekt zu den Wohnungen reichlich Abstand, sodass der Zugang über kurze Querbrücken erfolgt. Nach weiteren Wohnbauten und Ausstellungen folgte schließlich zwischen 1992 und 94 die Fertigstellung seines Hauptwerkes, der Schule am Kinkplatz 21 in Penzing [vgl. MA19, „Projekte und Konzepte, Heft 3, Ganztageshauptschule Kinkplatz Wien 14“, 1995]

„Ich wollte eine Schule machen, bei der nicht gleich das Unangenehme, das bei Schulen immer so auffällt, sich bemerkbar macht.“ (Helmut Richter, 1995)

Ähnlich wie beim Wohnbau in der Brunnerstraße, wird auch hier eine lange, transparente Erschließungszone mit klar davon abgesetzten Trakten angewandt. Größtenteils kommen Stahlkonstruktionen mit einer Glashaut zum Einsatz. Während die Architekturkritik einen Meilenstein der zeitgenössischen Österreichischen Architektur sieht, war das Projekt aufgrund der Neuartigkeit der Bautechnologien auch immer wieder negativer Kritik ausgesetzt. Wie seine beiden Mitarbeiter Gerd Erhartt und Jakob Dunkl beschreiben, begegnete er Bemerkungen über Ästhetik oder die Reinigung verschmutzter transparenter Flächen stets mit entwaffnenden Argumenten wie *„Ästhetik ist eine Frage der Gewohnheit“* oder *„Schmutziges Glas ist durchsichtiger als Beton“* [vgl. Schartner/Fasch, „Ein Buch für Helmut Richter“, 2007].

Zwischen 1991 und 2007 war Helmut Richter, neben seiner Tätigkeit als Architekt, Universitätsprofessor am Institut für Hochbau 2 an der TU Wien. In dieser Zeit betreute er mehr als 500 Diplomarbeiten [vgl. Schartner/Fasch, „Ein Buch für Helmut

Abb. 3.2: Bad Sares

Richter“, 2007, s. 5].

„Das Ergebnis, die überragende Qualität seiner Lehre, prägt eine Wiener Architektengeneration und inspirierte erfolgreiche „Unternehmensgründungen“ [...] Helmut Richter wurde so zu einem Qualitätssiegel weit über die Grenzen Österreichs hinaus. Absolventen, die bei ihm Diplom gemacht haben, können sich überall erfolgreich bewerben.“ (Johannes Baar-Baarenfels, 2007)

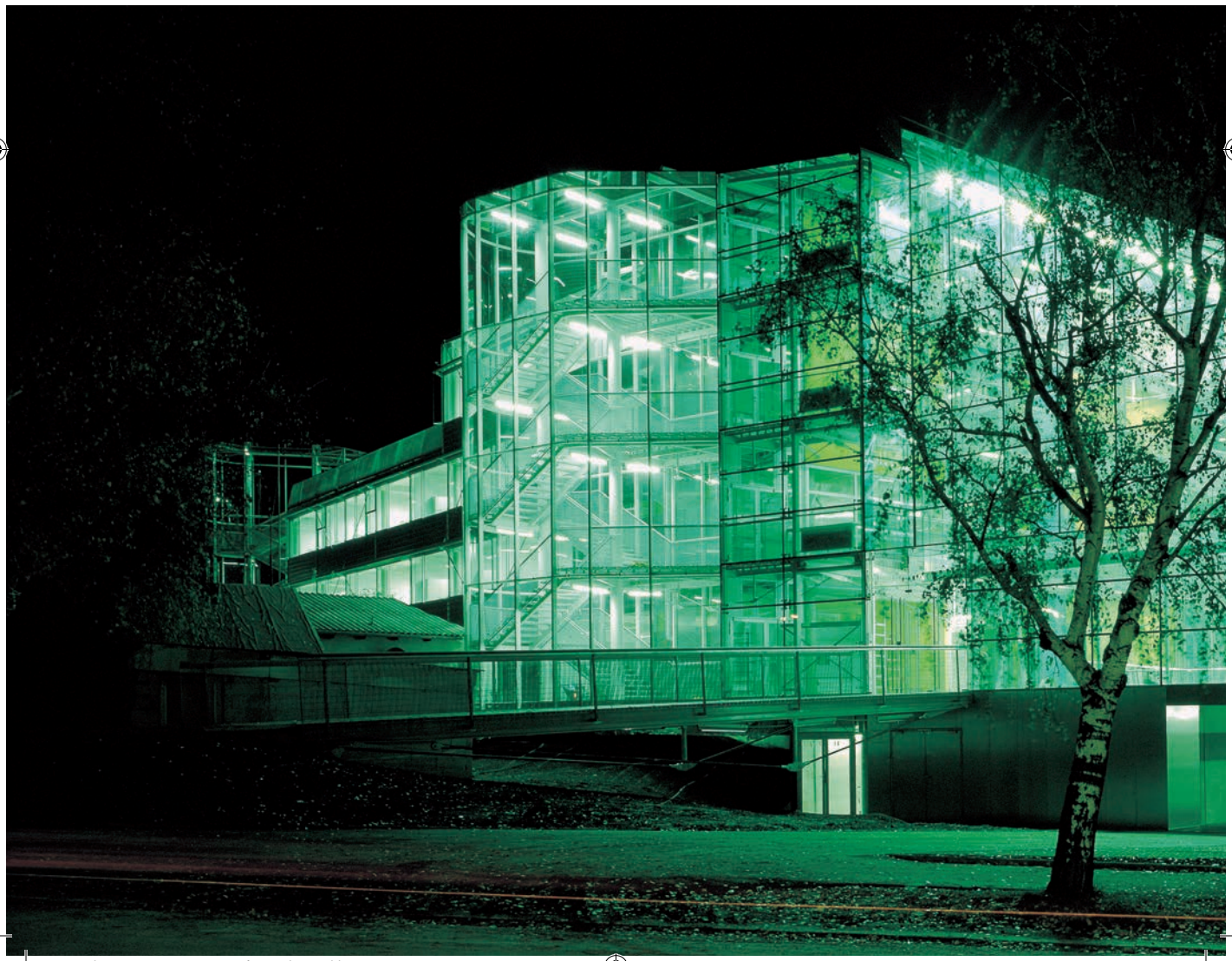
Helmut Richter verstarb, infolge seiner langjährigen Erkrankung, am 15.Juni 2014 in Wien.

o.: Abb. 3.3: Wohnhaus Brunnerstr.
u.: Abb. 3.4: Wohnhaus Brunnerstr.

o.: Abb. 3.5: Bad Sares
u.: Abb. 3.6: Wohnhaus Brunnerstr.

o.: Abb. 3.7
u.: Abb. 3.8

o.: Abb. 3.9: Restaurant Kiang
u.: Abb. 3.10: Restaurant Kiang



3.2 - Die Schule am Kinkplatz

„Das [...] Bauwerk manifestiert in vorbildlicher Weise, wie sich der bildungs- und gesellschaftspolitische Auftrag an die Stadtverwaltung mit weittragenden stadtplanerischen Absichten verbinden lässt. Das Schulbauprogramm 2000 mobilisiert über sieben Milliarden Schilling für Neu- und Umbauten an den Pflichtschulen. Der Wiener Schulbau steht damit wieder im Mittelpunkt internationalen Interesse und kann stellvertretend für das offensive Gesamtbild der gegenwärtigen Wiener Stadtentwicklung gelten." (Dr. Hannes Swoboda, 1995)

100 neue Pflichtschulen wurden im Rahmen des Schulbauprogrammes 2000 in Wien errichtet. Im Zuge dessen wurde 1994 auch die Schule am Kinkplatz von Helmut Richter fertiggestellt. Das Bauwerk macht auf den ersten Blick klar, dass Wiener Tradition gebrochen wird – durch den hohen Transparenzgrad und die Auflösung gewohnter Raumgrenzen erfährt es - auf diesem Gebiet bisher unerreichte - Großzügigkeit und einen starken räumlichen Reiz [vgl. MA19, „Projekte und Konzepte, Heft 3, Ganztageshauptschule Kinkplatz Wien 14", 1995, s.3].

Abb. 3.11

Eingebettet zwischen den Glashäusern der Erwerbsgärtner im Süden und bestehenden Wohnbauten im Norden wird Richters These spürbar, wonach der Bauplatz nicht als primärer Begründungshorizont für die Formgebung eines Baukörpers taugt. Die beiden südlich gelegenen, keilförmigen Glaskörper, welche die Eingangs- und Turnhalle sowie die Erschließungszone umhüllen, prägen das Erscheinungsbild maßgeblich und sind schon von weitem her sichtbar. Drei zweihüftige Klassentrakte erstrecken sich fingerartig gegen Norden, dazwischen liegen die Pausenhöfe.

Ähnlich wie beim Wohnbau in der Brunnerstraße zeigt sich die Erschließungszone, die gemeinsam mit Aula, Turnsaal und dazwischen liegendem Pausenhof den Südteil bildet, als langgestreckter, offener Gang, über den die Bedienung der drei nach Norden abgesetzten Trakte erfolgt. Diese werden über kurze Querbrücken erschlossen. Während der südliche Teil rechtwinkelig gegliedert ist, brechen die im Norden anschließenden Blöcke mit ihren nicht parallelen Fassaden aus diesem System heraus und verleihen den zwischengeschalteten Innenhöfen dadurch mehr Tiefenwirkung. Über den ersten Hof gelangen die Nutzer in den im Untergeschoss gelegenen Gymnastiksaal.

Während Turnsäle und Aula von allen Nutzern des Gebäudes gemeinsam genutzt werden, sind die drei Trakte zwei unterschiedlichen Schulen zugewiesen. Die

Abb. 3.12

Obergeschosse sind zweihüftig organisiert und beherbergen jeweils die getrennten Unterrichtsräume der beiden Schulen (Neue Mittelschule mit Schwerpunkt Informatik und Ganztagsschule Kinkplatz), wobei der mittlere Block wiederum beiden Schulen zugeordnete Sonderunterrichtsräume enthält. Erschlossen werden diese Bereiche jeweils über kurze Stege, den Abschluss bilden große verglaste Flächen mit anschließenden, außenliegenden Fluchtstiegenhäusern.

In den Schnitten werden der – durch das Gebäude fließende – Hangverlauf und die beiden separaten Einheiten des Projektes besonders deutlich sichtbar. Beachtenswert ist in diesem Zusammenhang auch die Unterscheidung in zwei unterschiedliche statische Systeme im Bereich der beiden Hallen. Während das Tragwerk der Aula mit einer weiten Auskragung gelöst ist, kommt über dem Turnsaal ein schräg abgestützter Rahmen zum Einsatz. Mit dem Ziel einer größtmöglichen Entmaterialisierung wurde für die Befestigung der Fassadenelemente eine punktgestützte Verglasung gewählt.

Durch die transparente Fassade ist das schlanke, bis an die Grenzen des Standes der Technik ausgereizte Tragwerk erkennbar. Zusätzlich verstärken die sichtbar belassenen Aussteifungselemente die technoide Wirkung des Gebäudes. Der Transparenzgrad der Glashaut wird maßgeblich durch das vorherrschende Umgebungslicht beeinflusst. Während tagsüber eher der Eindruck einer grünlich schimmernden, spie-

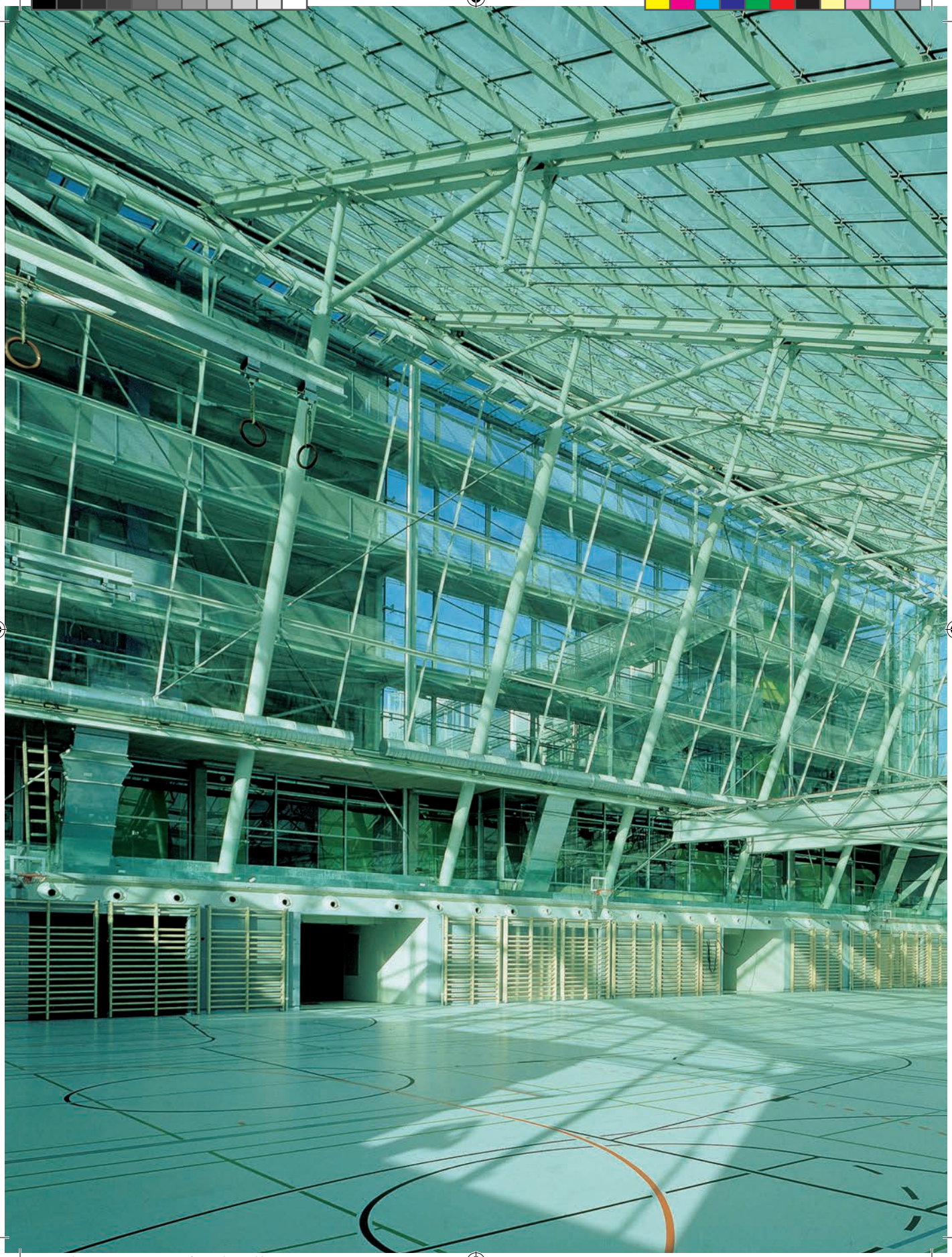
gelnden Oberfläche entsteht, sind abends Einblicke ins Innere der Schule möglich. Das Erscheinungsbild befindet sich damit ständig im Wandel.

Die zwischen den Trakten eingebetteten Halbhöfe stellen besonders prägende Elemente dar. Sie sind sowohl quer miteinander als auch längs mit der Erschließungszone verbunden und nach Süden zur Wohnbebauung hin offen. Gegen Norden entsteht je nach Lichtsituation ebenfalls ein offener Eindruck, da Licht durch die verglaste Erschließungszone dringen kann und somit starke Transparenz entsteht. Der erste Hof wird auf der Nordseite zwischen Aula und Turnsaal fortgesetzt und gliedert dort die beiden Glaspulte klar in zwei Bereiche.

Die Planung der Schule wurde von Dipl.-Ing. Erich Panzhauser von der Fakultät für Architektur und Raumplanung der TU Wien bauphysikalisch begleitet. Aufgrund des hohen Verglasungsgrades und der damit zu erwartenden sommerlichen Überhitzung, wurden eine Vielzahl wichtiger Einflussfaktoren und deren Zusammenspiel in die bauphysikalischen Überlegungen mit einbezogen. Geeignete Maßnahmen zur Erhaltung eines angenehmen Innenraumklimas wurden mit Hilfe eines Programmes zur Simulation des thermischen Verhaltens untersucht. Die Berechnungen führten zunächst zur Wahl eines Sonnenschutzglases mit einem u-Wert von 1,8 W/m2K. In Kombination mit innenliegenden hochreflektierenden Screens konnte der zu erwartende Wärmeeintrag stark reduziert werden. Neben der natürlichen Belüftbarkeit wurde eine zusätzliche, mechanische Stützbelüftung mit auf bis zu 18° vorgekühlter Luft vorgesehen [vgl. MA19, „Projekte und Konzepte, Heft 3, Ganztageshauptschule Kinkplatz Wien 14“, 1995, s. 4 ff].

Abb. 3.13

Abb. 3.14



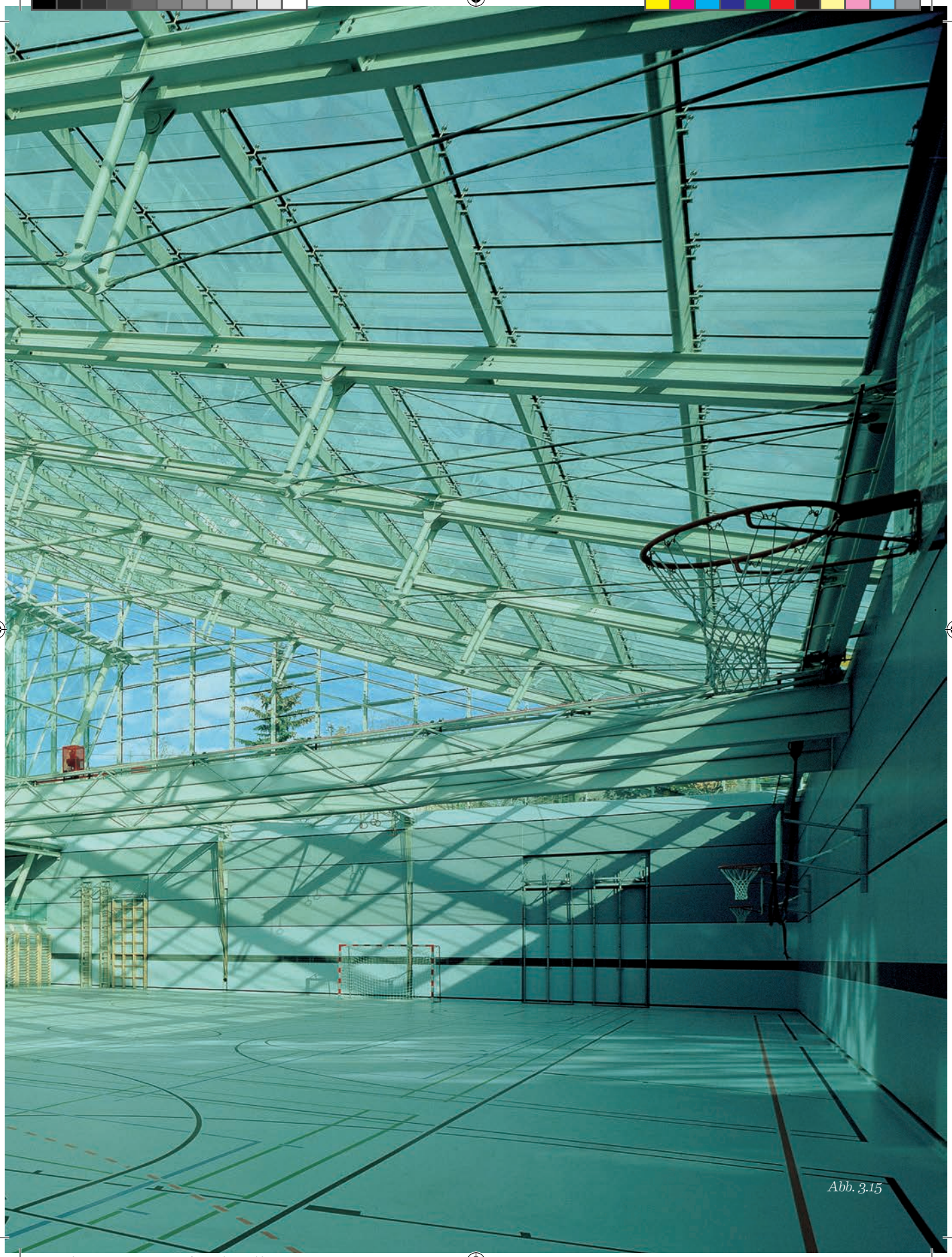
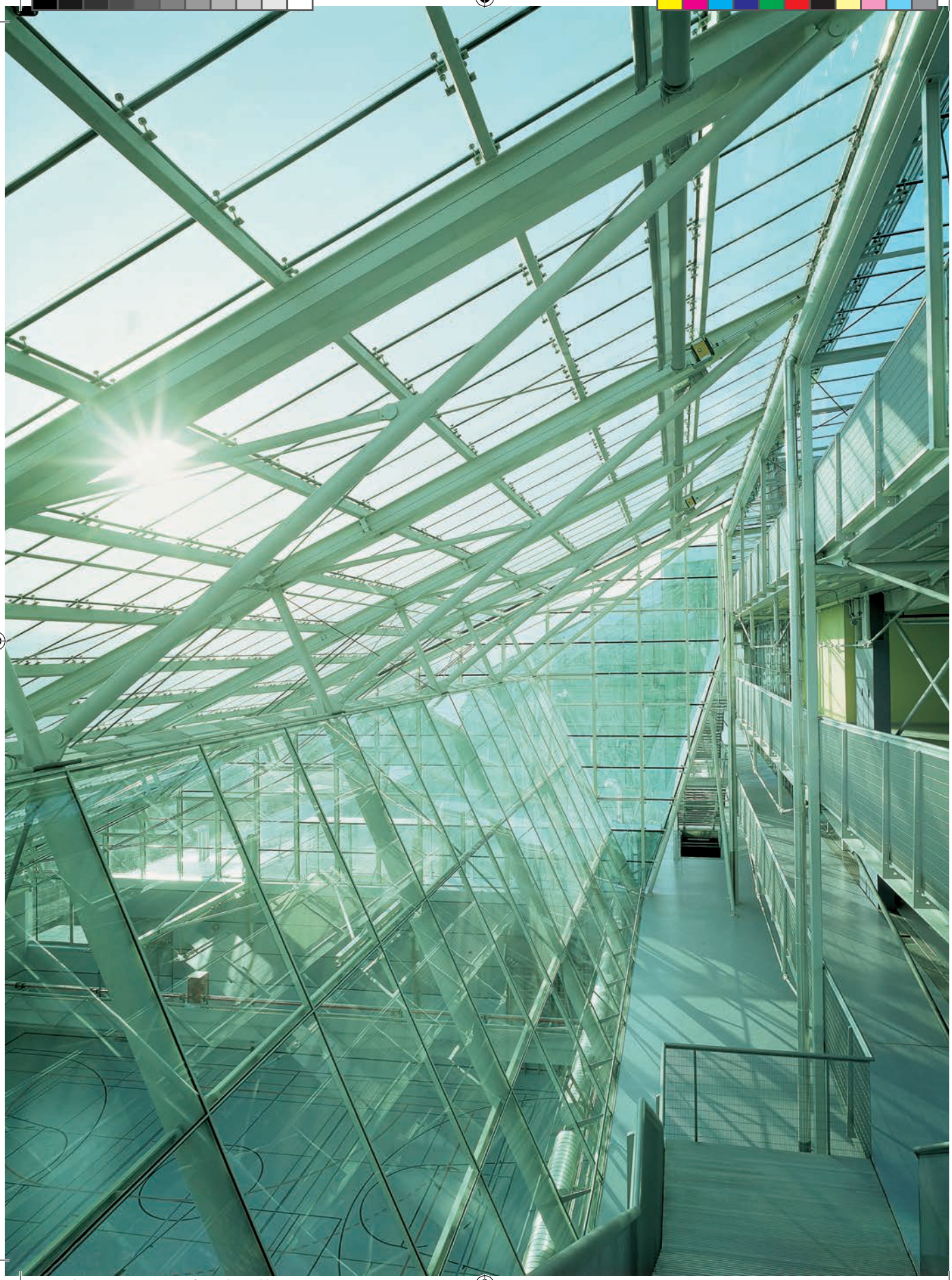


Abb. 3.15



Trotz der vorangegangenen bauphysikalischen Überlegungen war die Schule in der Vergangenheit unglücklicherweise immer wieder negativer Kritik hinsichtlich ihrer Nutzbarkeit ausgesetzt. Spätere Nachbesserungen brachten nicht den erwarteten Effekt, oder führten selber zu neuen Problemen. Hauptsächliche Kritikpunkte waren unzumutbare sommerliche Temperaturen, Wassereintritt und Schimmelbildung, sowie schadhafte Bauteile. Seit einigen Jahren gibt es Überlegungen zur Sanierung der Schule. Zum gegenwärtigen Zeitpunkt steht nicht fest, ob die Schule am Kinkplatz generalsaniert oder tatsächlich vom Abbruch bedroht ist. Fest steht, dass die Nutzer vorraussichtlich im Juni 2017 - für einen Zeitraum von fünf Jahren - in ein Ausweichquartier umziehen sollen. Dieses wird in Form einer wieder verwendbaren Containerschule im 14. Bezirk (Braillegasse/Torricelligasse) aufgestellt und später teilweise vom Österreichischen Bundesheer weiterbenutzt [vgl. URL Helmut Richter 3&4].

„[...] Helmut Richters kompromisslose Haltung muss beim Bauen und ganz besonders in Wien, der Kalkputzstadt, zu Konflikten führen und sie waren zahlreich. Aber er war nicht allein, auffallend die Parallelen: Roland Rainer mit dem Stadthallenbad, es war ebenso wie Helmut Richters Schule am Kinkplatz vom Abbruch oder Entstellung bedroht, oder Ernst Hiesmayr mit dem Juridikum - allesamt international ausgezeichnete Bauten, Stahl - Glas Konstruktionen, transparent konstruiert, bis ins Detail, technische Infrastrukturen offen gelegt, innovativ. Sie sind Schlüsselwerke der österreichischen Nachkriegsmoderne. [...] Am weiteren Schicksal der Schule am Kinkplatz werden wir erkennen, ob die zahlreichen Nachrufe von PolitikerInnen zum Tod von Helmut Richter ernst zu nehmen sind. Es ist zu hoffen.“
(Gerhard Steixner, 2014)

Abb. 3.16

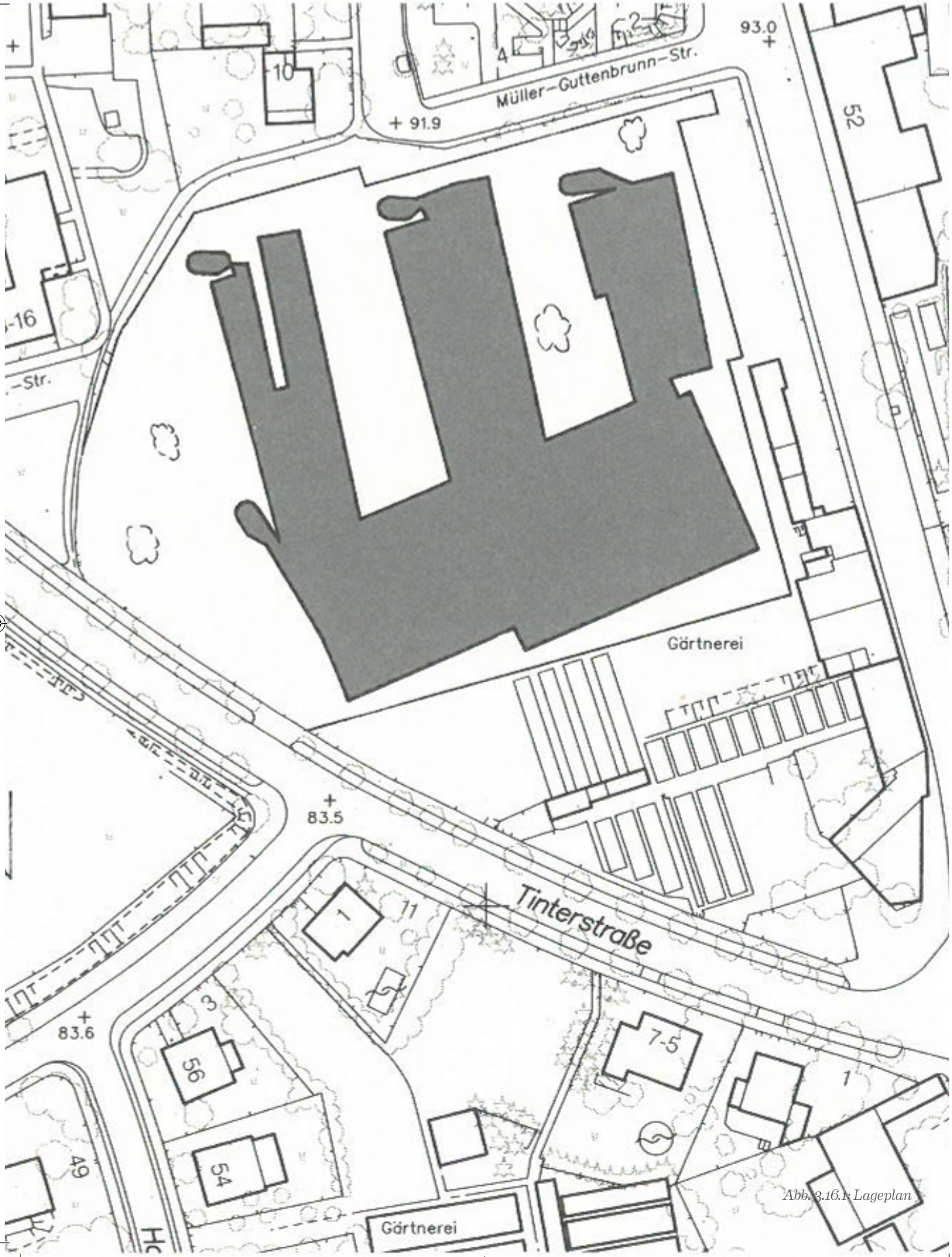


Abb. 3.16.1: Lageplan

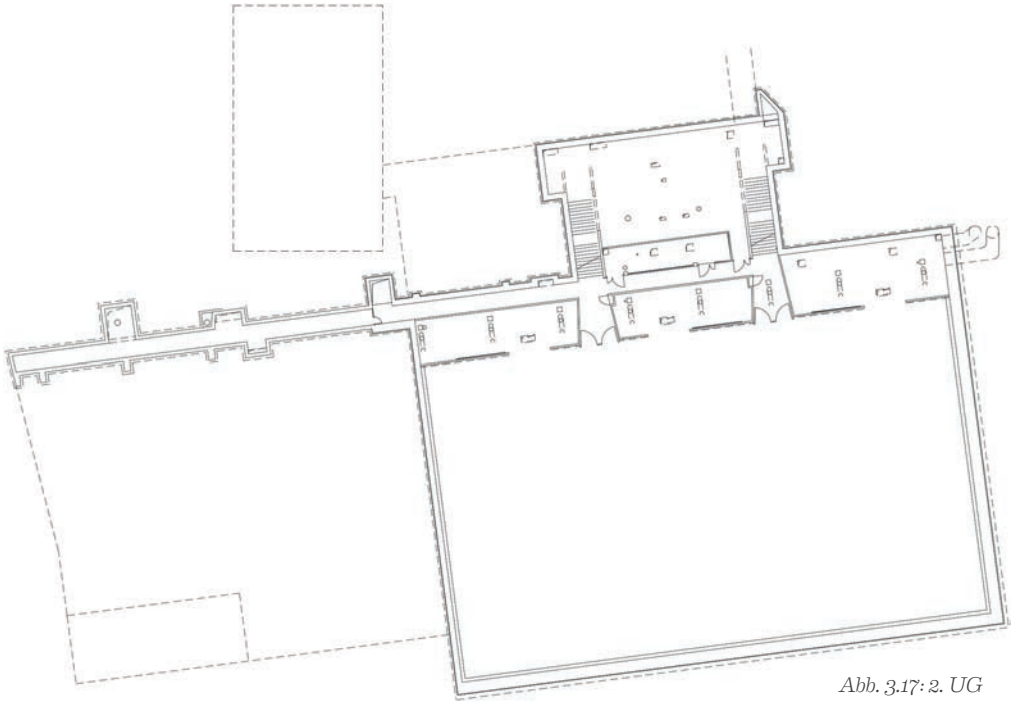


Abb. 3.17: 2. UG

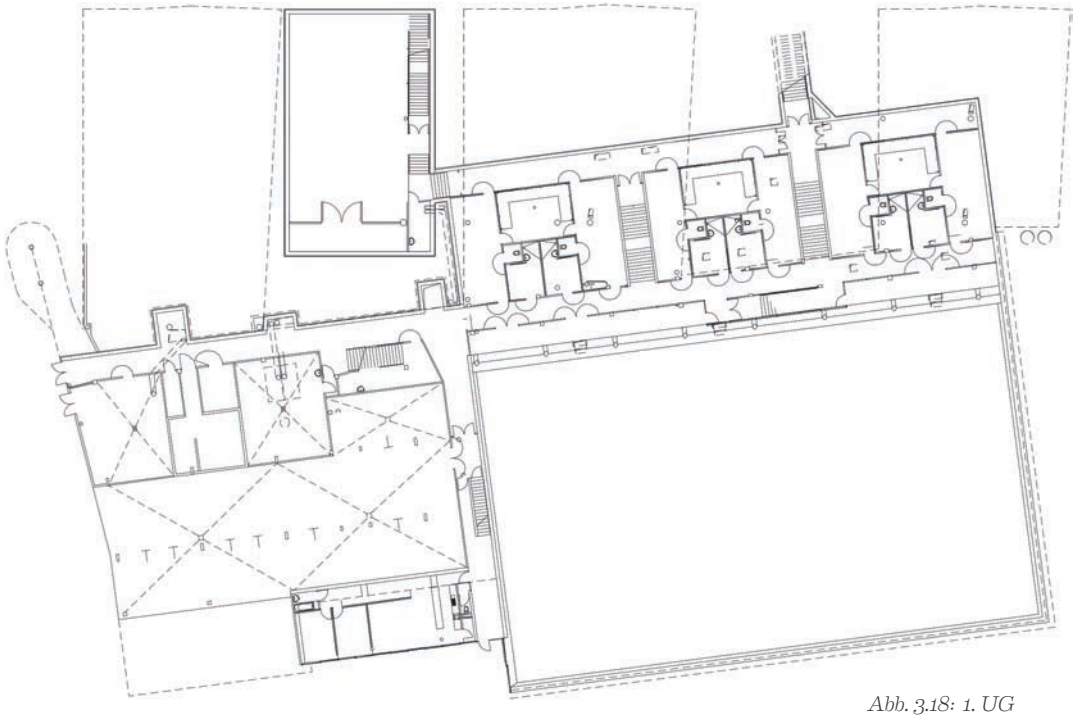


Abb. 3.18: 1. UG

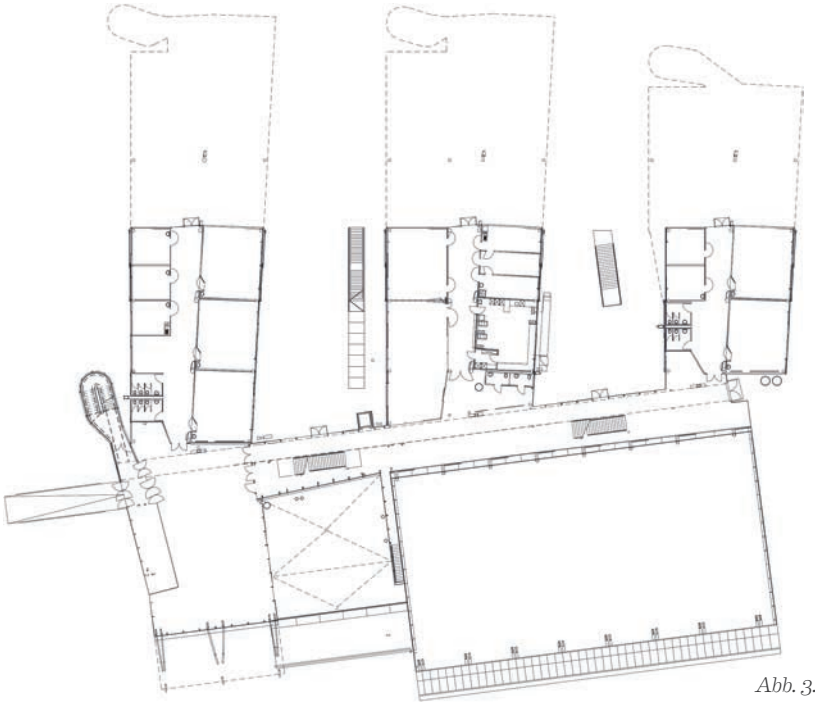


Abb. 3.19: EG

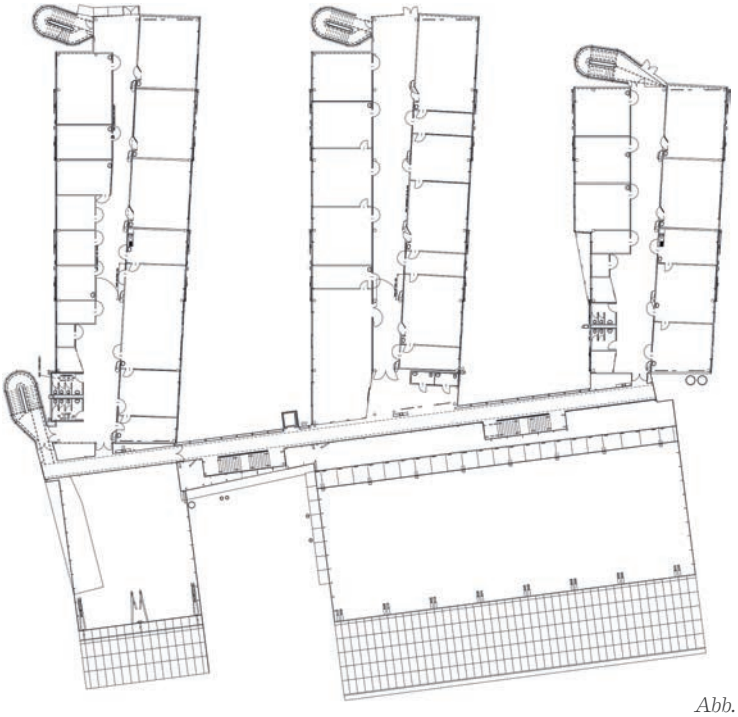


Abb. 3.20: 1. OG

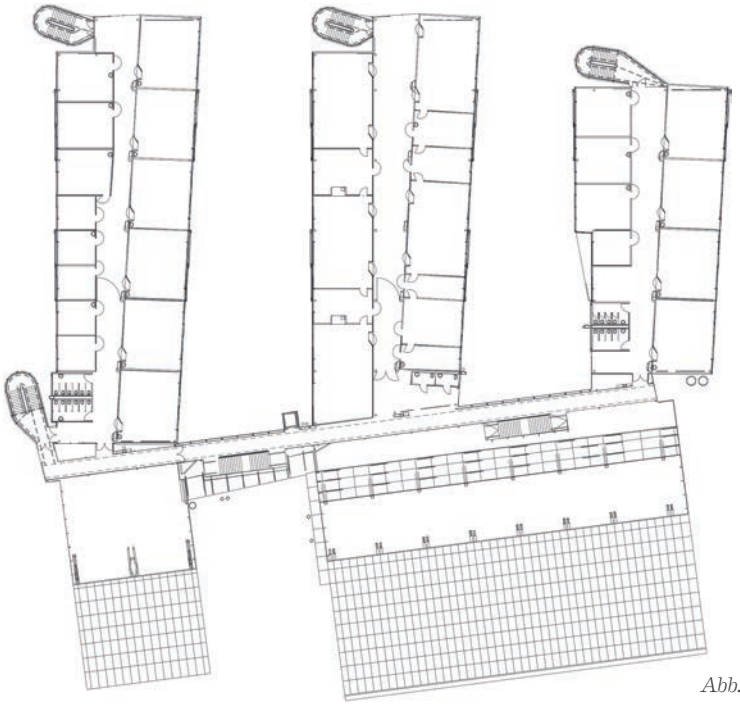


Abb. 3.21: 2. OG

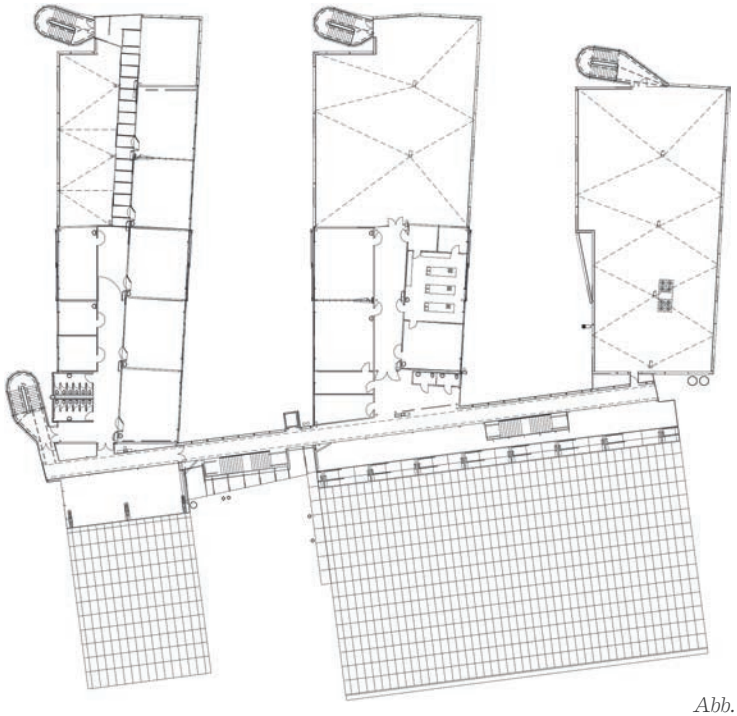


Abb. 3.22: 3. OG

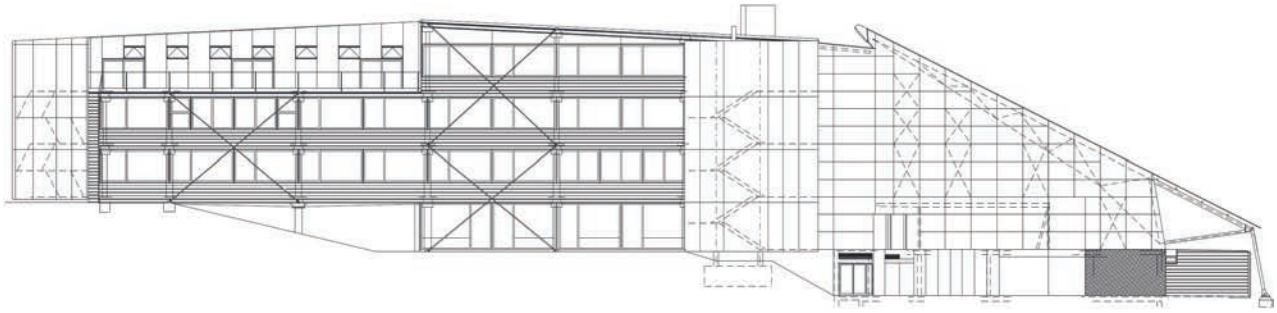


Abb. 3.23: Ansicht West

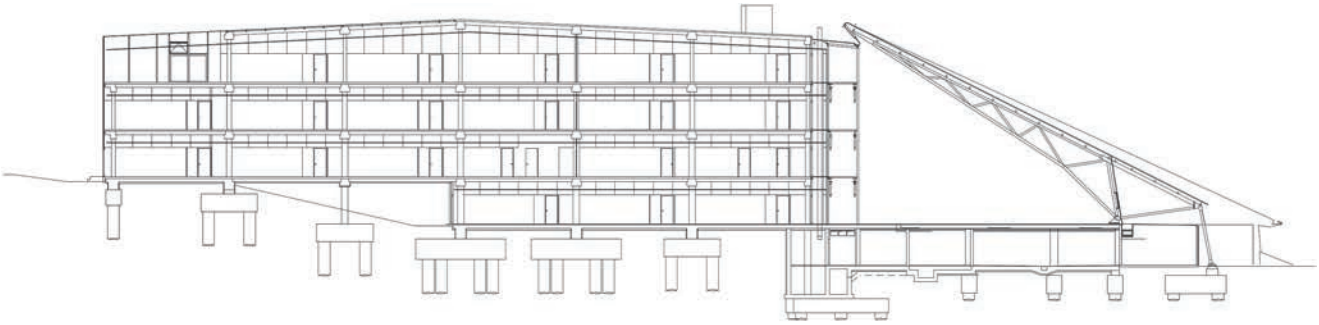


Abb. 3.24: Schnitt durch Eingangshalle

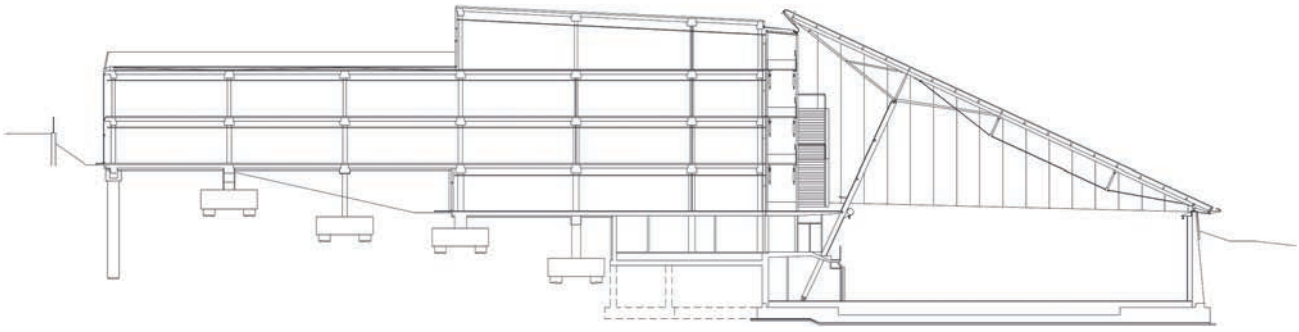


Abb. 3.25: Schnitt durch Turnsaal

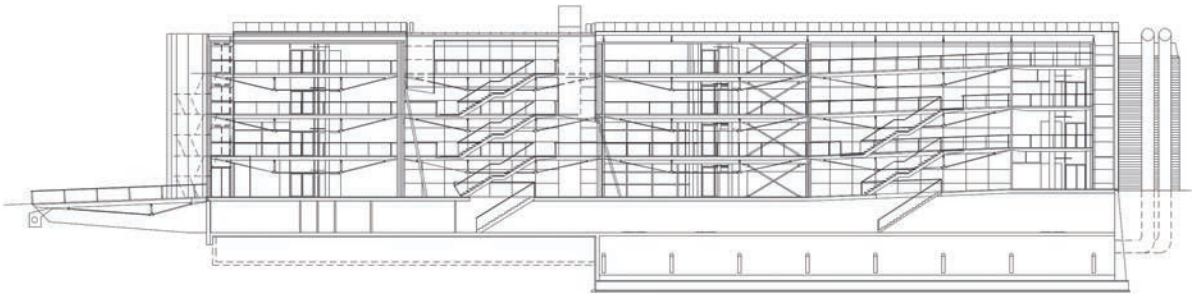


Abb. 3.26: Schnitt durch Erschließungszone

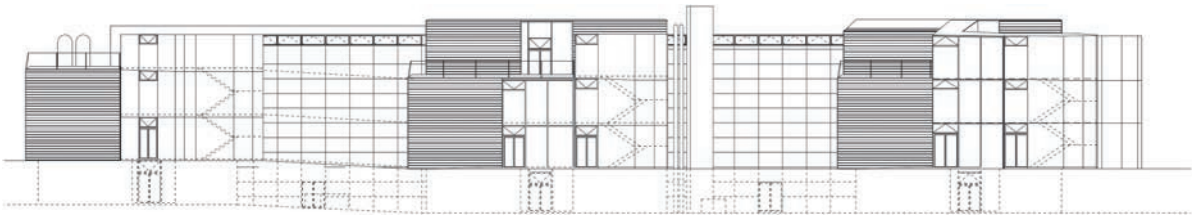


Abb. 3.27: Schnitt durch Klassentrakte

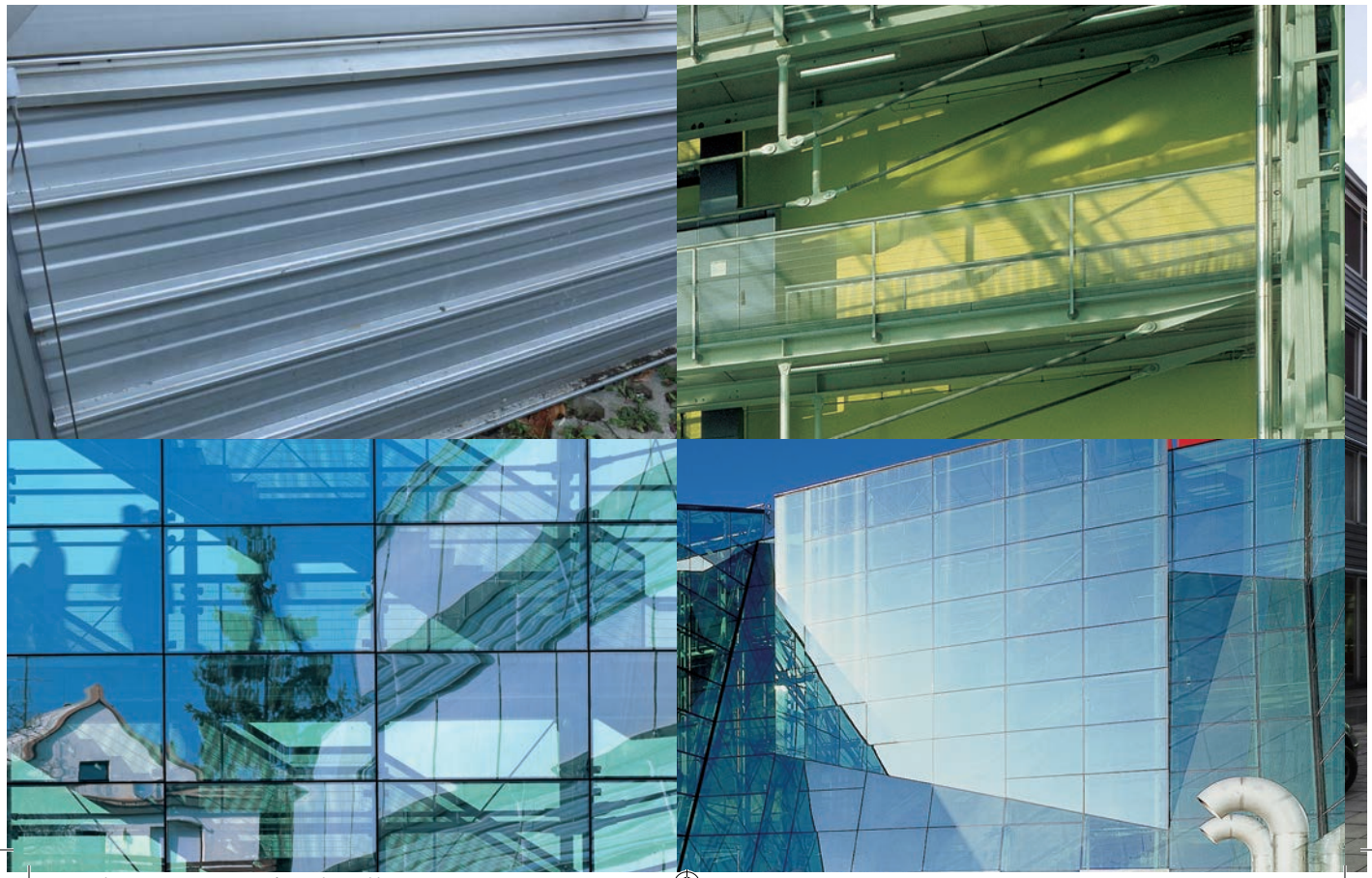
Abb. 3.28

Die Schule am Kinkplatz von Helmut Richter wurde für die Visualisierung gewählt, um einen Vergleich mit der Realität zu ermöglichen. Mit den unterschiedlichen Lichtsituationen und den für Game Engines herausfordernden Materialien, bietet sich das Bauwerk gut an, die Leistungsfähigkeit aktueller Software zu erforschen.

Das Erscheinungsbild der Schule wird maßgeblich durch die verwendeten Stahl- und Glaselemente geprägt. Die Dreifachisoliergläser mit einer partiell reflektierenden Schicht auf der äußersten Scheibe erscheinen je nach Lichteinfall in einem lichten Blau bis Grün. Während die südseitig gelegenen Hallen über eine minimierte Stahlkonstruktion mit Unterspannungen verfügen, sind die Klassentrakte als Betonskelett mit Fertigteildecken ausgebildet. Für die Fassaden, welche die Pausenhöfe bilden, kommen industrielle Stahlelemente zum Einsatz. Der hohe Glasanteil schafft größtmögliche Transparenz und damit eine helle und freundliche Atmosphäre. Diese soll durch die verwendete gelbe Wandfarbe in den Gängen weiter verstärkt werden. Je nach Tageszeit ist die Farbe durch die grünlich schimmernden Glasflächen auch von außen sichtbar [vgl. MA19, „Projekte und Konzepte, Heft 3, Ganztageshauptschule Kinkplatz Wien 14“, 1995 sowie URL Helmut Richter 5].

Um eine möglichst realitätsgetreue virtuelle Abbildung der Schule schaffen zu können, war es notwendig, die tatsächlich verwendeten Materialien detailliert zu rekonstruieren. Für diese Aufgabe wurden verschiedene Werkzeuge und Programme verwendet, welche im folgenden Kapitel näher erklärt werden. Die Glaselemente, die Stahlelemente der Hoffassaden und die gelben Gangflächen wurden repräsentativ gewählt, um die zur Verfügung stehenden Möglichkeiten für die virtuelle Darstellung von Materialien zu testen.

oben li: *Abb. 3.29: Stahlelemente der Fassade*
oben re.: *Abb. 3.30: Gelbe Wandfarbe*
unten li.: *Abb. 3.31: Glaselemente*
unten re.: *Abb. 3.32: Glaselemente*



In Kombination mit den verwendeten Materialien entstehen je nach Jahres-, Tageszeit und Wetter sehr unterschiedliche Lichtsituationen. Abhängig vom Sonnenstand und Bewölkung können stark unterschiedliche Stimmungen beobachtet werden. Besonders die zwischen den Klassentrakten gelegenen Pausenhöfe werden durch die natürliche Belichtung stark beeinflusst. Sie sind entweder direktem Sonnenlicht ausgesetzt, stark verschattet und lediglich diffus belichtet oder erscheinen - bedingt durch den Sonnenlichteinfall durch die südlich davon gelegene verglaste Erschließungszone - in einem schimmernd grünen Licht. Diese Lichtsituationen sind auch im darunter gelegenen Turnsaal spürbar, wobei dieser notwendigerweise über eine zusätzliche künstliche Beleuchtung mittels Leuchtstoffröhren verfügt.

Während der Pausenhof zwischen den Klassentrakten nach Süden durch die verglaste Erschließungszone begrenzt wird, bleibt er gegen Norden zur Straße hin offen. Aufgrund des natürlichen Hangverlaufs entstehen mehrere Treppen, welche als begrünte Flächen ausgebildet sind. Es kommt eine Kombination aus kleinwüchsigen Gräsern und Büschen mit mittelgroßen Nadelhölzern zur Anwendung. Auf der Gegenüberliegenden Straßenseite befinden sich ebenfalls (in Privatgärten gelegene) Nadelhölzer sowie Laubbäume.

Neben einer naturgetreuen Abbildung der verwendeten Materialien, war die Repräsentation der vorkommenden Lichtstimmungen sowie der Vegetation ein ausschlaggebendes Kriterium für eine qualitative virtuelle Nachbildung des Projektes. Genau wie für die Materialerstellung, war es auch hier notwendig, bestimmte Werkzeuge und Anwendungen in den Workflow einzubeziehen, um das entsprechende Ergebnis zu erzielen.

vorherige Seite:
links: *Abb. 3.33: Sonnenlicht fällt durch Er schließungszone in Innenhof*
rechts o.: *Abb. 3.34: Nur diffus belichteter Innenhof*
rechts u.: *Abb. 3.35: Mit direktem Sonnenlicht belichteter Innenhof*

Vor Ort vorkommende Vegetation
li.o.: Abb. 3.36: *Büsche im Innenhof*
li.u.: Abb 3.37: *Birken in der Umgebung*
re.o.: Abb 3.38: *Nadelhölzer in den gegen über liegendene Privatgärten*
re.u.: Abb 3.39: *kleinwüchsige Nadelhölzer im Innenhof*




Um das Potential von Game Engines zur Erstellung fotorealistischer Renderings für Architekturprojekte zu überprüfen, wurde bei diesem Projekt die Unreal Engine 4 von Epic ausgewählt. Seit der Veröffentlichung der Version 4 im Jahr 2015 können mit dieser Render Engine fotorealistische Ergebnisse in Echtzeit erzielt werden, die sich qualitativ auf Augenhöhe mit Visualisierungen von bewährten Renderern befinden. Über das Onlineforum können Benutzer Feedback an die Entwickler bei Epic geben. Durch diesen direkten Kontakt, erscheinen regelmäßig Updates, welche die Leistungsfähigkeit der Unreal Engine für Architekturvisualisierungen rasch vorantreiben.

Im folgenden Kapitel wird die Entstehung des Projekts näher beschrieben. Es werden die Modellierung, der Export aus der Modelliersoftware beziehungsweise der Import in die Unreal Engine, die Belichtung, Texturierung sowie der Umgang mit der Vegetation und die Postproduktion erläutert.

3.3 - Modell

Um für jede Phase eines Projektes das bestmögliche Werkzeug zur Verfügung zu haben, werden sogenannte Production-Pipelines verwendet. Diese beschreiben die einzelnen Aufgaben und die Position einer Software innerhalb des Projektverlaufs. Zum Beispiel gibt es Software, die mehr Möglichkeiten zum Erstellen eines 3D Modells auf Grundlage von Plänen bietet, andere wiederum sind auf dem Gebiet der Freiformmodellierung leistungsfähiger. Um die Geometrie von einem Programm ins nächste übernehmen zu können, ist ein Export beziehungsweise Import notwendig. Dafür stehen oft Werkzeuge zur Verfügung, um das 3D Modell möglichst optimiert und verlustfrei weiterbearbeiten zu können. Im Folgenden werden die - für dieses Projekt angewandte - Production-Pipeline und ihre einzelnen Phasen näher erläutert.

Der erste Schritt um ein passendes 3D Modell für die Visualisierung kreieren zu können, bestand in der Sammlung von Plänen der Schule. Diese wurden von Architektin Mag. Silja Tillner (Tillner & Willinger ZT GmbH) zur Verfügung gestellt. Nachdem bis dato keine 3D Daten von dem Projekt vorhanden waren, wurde das Modell von Grund neu aufgebaut. Die Pläne und Fotografien, welche vor Ort aufgenommen wurden, stellten die Basis dafür dar. Obwohl nur einige Bereiche der Schule stellvertretend im Detail dargestellt werden, wurde ein grobes Modell der gesamten Schule erstellt. Das sorgt für realistische Lichtsituationen und bietet Hintergrundmaterial für die erstellten Visualisierungen. Dieses erste Modell wurde mit Rhinoceros 3D (in der Folge „Rhino“ genannt) erstellt.

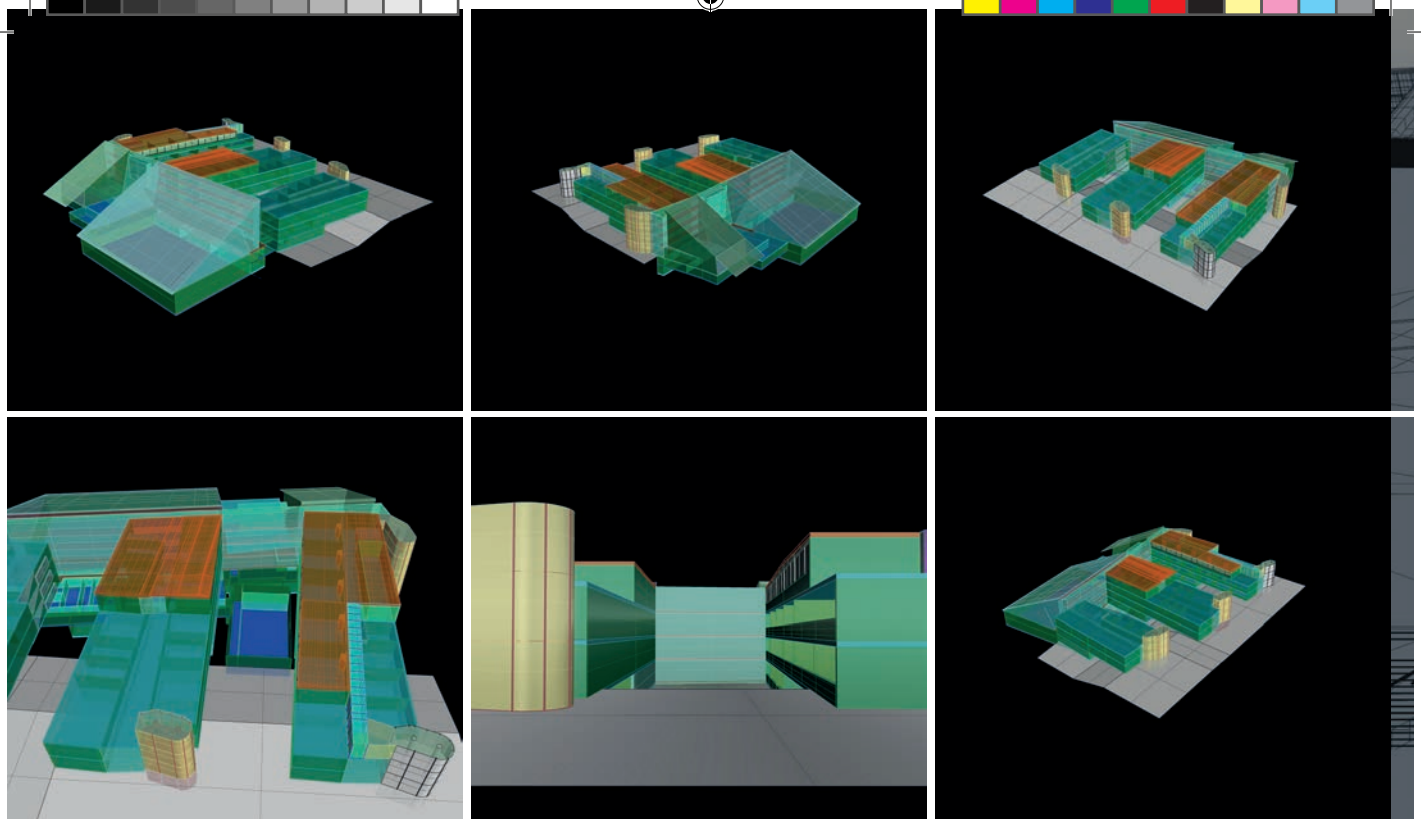


Abb. 4.1: Rhino Modell

Modellierung mit Rhino

Für die erste Modellierphase wurde Rhino verwendet. Mit dieser Anwendung war es möglich in angemessener Zeit den gesamten Baukörper in groben Umrissen aufzubauen (Abb. 4.1). Es handelt sich dabei um eine 3D Modelliersoftware des Herstellers Robert McNeel & Associates. Im Vergleich zu polygonbasierender Software basiert Rhino auf dem NURBS Modell. Der Hauptfokus liegt dabei auf der Erstellung mathematisch korrekter Repräsentationen von Kurven und Freiformflächen.

Mittels der Fang-Funktionen (Snapping) war ein schnelles und präzises Arbeiten möglich. Die Pläne und Fotos wurden als Referenzen in den Hintergrund geladen, direkt vor Ort genommene Maße dienten zusätzlich zur Orientierung. Um die Übersichtlichkeit zu wahren, wurden für alle Geschosse eigene Ebenen erstellt und die einzelnen Bauteile (Außenwände, Innenwände, Decken, Boden, Türen, Fenster,...) auf Unterebenen gelegt. Dies ermöglicht einerseits das rasche Ausblenden ganzer Stockwerke beziehungsweise einzelner Bauteilgruppen, andererseits kann diese Hierarchie auch exportiert und in anderen Programmen weiterverwendet werden.

Export aus Rhino nach Cinema 4D

Die Detaillierung der drei Hauptbereiche erfolgte mittels Cinema 4D. Zunächst war es notwendig das in Rhino erstellte Grundmodell nach Cinema 4D zu exportieren.

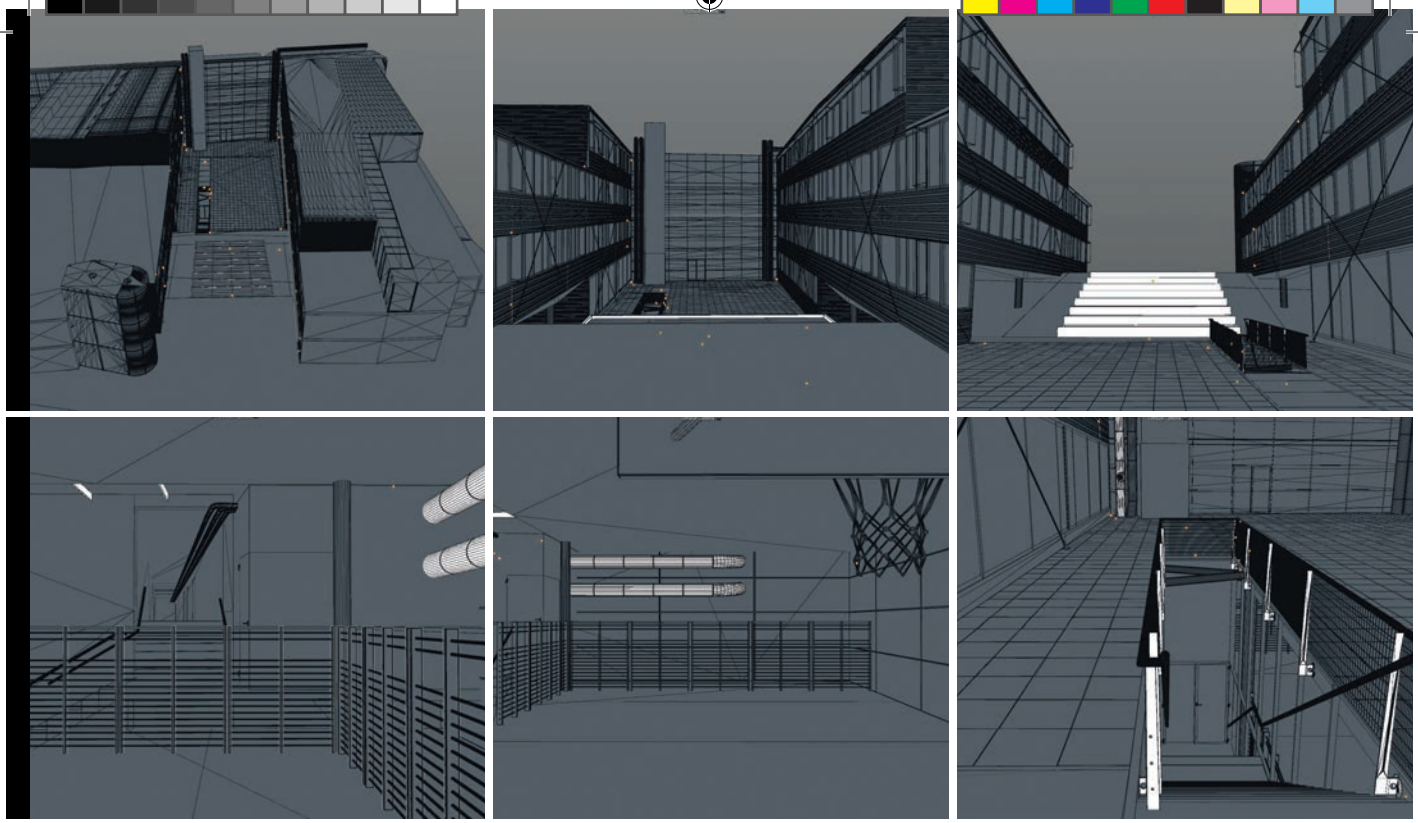
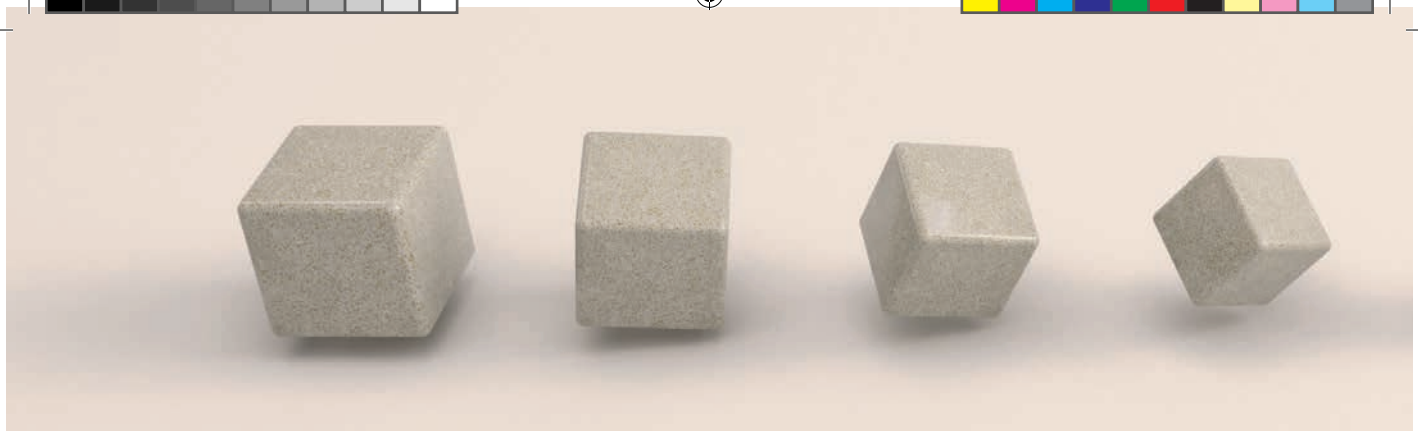


Abb. 4.2: Cinema 4D Modell

Dafür wurde das Plug-in Rhino IO verwendet. Dieses Import/Export Plug-In ermöglicht es, Rhino Dateien (.3dm) direkt in Cinema 4D zu öffnen. Sämtliche Objekte und Ebenenstrukturen, die zuvor in Rhino erstellt wurden, werden übernommen und können weiter bearbeitet werden. Das sorgt für gute Übersichtlichkeit und eine saubere Cinema 4D Datei. Außerdem ist es möglich das 3D Modell in Rhino später zu überarbeiten und in Cinema 4D zu aktualisieren. Sämtliche Einstellungen, welche zuvor schon in Cinema 4D getätigt wurden (z.B. Materialzuweisungen) werden auf die aktualisierte Geometrie übernommen.

Detaillierung in Cinema 4D

Nachdem die grundlegende Geometrie aus Rhino vorhanden war, wurden in Cinema 4D zusätzliche Details erstellt (Abb. 4.2). Dabei handelt es sich um verschiedene Objekte im Raum, welche nicht zur Gebäudearchitektur direkt zählen. Dazu gehören hauptsächlich die Geräte in den beiden Turnsälen (Turngeräte, Sprossenwände, Basketballkörbe...) und die Einrichtungsgegenstände im Klassenzimmer (Tische, Stühle, Kasten...). Außerdem wurden Elemente wie Tür- und Fensterbeschläge, Lüftungen und Tragwerksdetails ergänzt. Sämtliche 3D Modelle wurden neu erstellt, es wurde keine bereits vorhandene Geometrie verwendet. Dabei wurde besonderes Augenmerk auf die Topologie der Modelle gelegt, um eine saubere Geometrie zu gewährleisten. Das ist gerade bei der Visualisierung mit der Unreal Engine ausschlaggebend, um später in kurzer Zeit korrekte UV-Maps für die Belichtung erstellen zu können.



Boxmodeling-Workflow

Hauptsächlich wurde ein Boxmodeling-Workflow verwendet, das heißt die Objekte wurden - von einem Grundkörper (Würfel, Kugel, Zylinder...) ausgehend – schrittweise mittels verschiedener Befehle erstellt. Die am häufigsten verwendeten Befehle bei dieser Arbeitsweise waren:

- die Skalierung, Rotation und das Verschieben
- die Extrusion und innere Extrusion
- das Beveln
- das Hinzufügen von Punkten und Kanten mittels des Messer Werkzeugs
- das Erstellen neuer Polygone zum Schließen von Öffnungen

Außerdem wurden verstärkt Selektionswerkzeuge wie Ring- und Loopselektionen für die schnelle Auswahl mehrerer zusammenhängender Polygone, Punkte oder Kanten eingesetzt. Für die korrekte Funktion dieser Selektionsarten ist ebenfalls eine saubere Geometrie ausschlaggebend.

MoGraph

Cinema 4D bietet mit MoGraph (Motion Graphics) ein leistungsfähiges, integriertes Paket zum Kreieren verschiedenster parametrischer Effekte. Ursprünglich für den Bereich des Motion Design programmiert, lassen sich viele Funktionen auch für Visualisierungen in der Architektur verwenden. Beim Modellieren von 3D Objekten kommt es häufig vor, dass ein- und dasselbe Element öfter verwendet wird, um Zeit und Ressourcen zu sparen. Es kann sich dabei um einzelne Teile innerhalb eines Modells handeln (die Sprossen einer Sprossenwand) oder auch um ganze Objekte (einzelne, zu einer großen Wand aneinandergereihte Sprossenwände). Für solche Aufgaben bietet das Kloner Objekt in MoGraph eine leistungsstarke Lösung. Beim 3D Modell der Schule von Helmut Richter wurde es beispielsweise für die Sprossenwände, die Lüftungsöffnungen und die Modelle der Turngeräte verwendet. Im Speziellen bei der Tragwerksmodellierung war dieses Werkzeug besonders hilfreich, da die sich wiederholenden Träger sehr schnell, präzise und gleichmäßig platziert werden konnten. Dem Nutzer stehen mehrere unterschiedliche Modi zur Verfügung (Linear, Radial, Raster...), mit

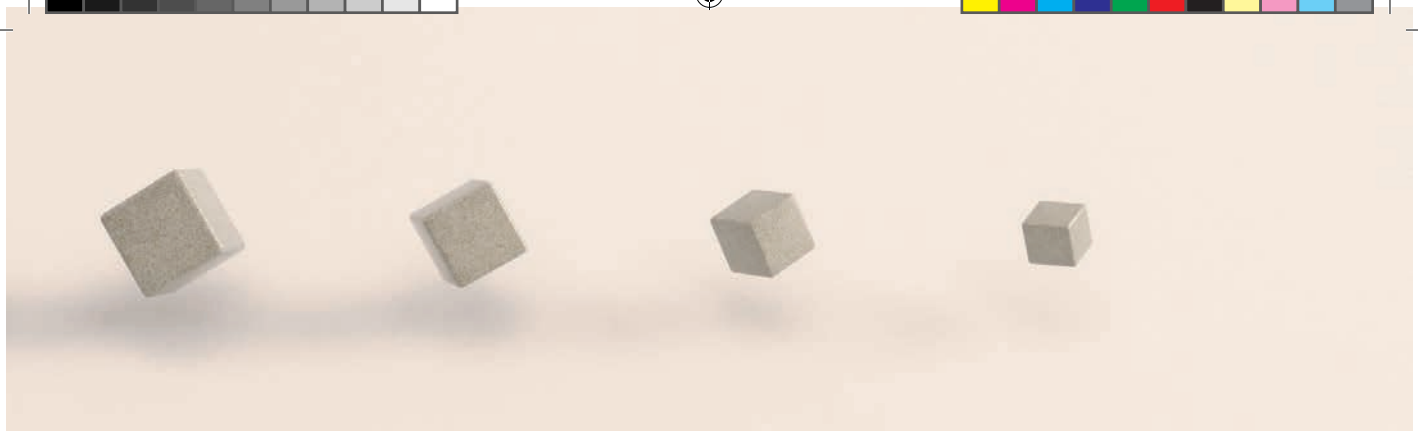


Abb. 4.3: Mittels MoGraph erstellte und transformierte Klone eines Objektes.

welchen verschiedenste Anordnungen realisierbar sind. Außerdem ist es möglich die Klone entlang eines Objektes (z.B. eines Splines) zu verteilen. Durch die parametrische Funktionsweise von MoGraph wird die Anordnung in Echtzeit aktualisiert, wenn das Objekt verändert wird (Abb.4,3). Neben der guten Steuerbarkeit der Klone ist vor allem die Instanzfunktion eine wichtige und ressourcenschonende Funktion.

Renderinstanzen

Renderinstanzen werden bei Visualisierungen verwendet, um Ressourcen zu sparen und Renderzeiten zu verkürzen. Es handelt sich dabei im Prinzip um Kopien von Objekten, mit dem Unterschied, dass die Geometrie nicht kopiert, sondern nur referenziert wird. Erstellt man eine Renderinstanz eines Objektes, erscheint diese im Editorfenster wie ein Duplikat des Originals. Tatsächlich bestimmt die Instanz nur die Position, Skalierung und Ausrichtung im Raum, die Geometrie selbst wird vom instanzierten Objekt übernommen. Aus diesem Grund ist es auch nicht möglich Polygone, Punkte oder Kanten einer Renderinstanz zu verändern. Wird das Originalmesh verändert, werden alle Instanzen automatisch aktualisiert.

Der große Vorteil dieses Systems liegt darin, dass die Geometrie eines oft verwendeten Objektes nur einmal geladen und berechnet werden muss. Alle anderen Renderinstanzen sind – was die Ressourcen angeht – praktisch frei verfügbar. Das ermöglicht überhaupt erst die Berechnung bestimmter Szenen (z.B. großer Grasflächen), da ein und dasselbe Objekt oft wiederverwendet wird. Um eine gleichmäßige, unnatürliche Verteilung zu vermeiden, werden die Renderinstanzen gedreht und skaliert. Dies kann auch zufallsgesteuert über diverse Shader (meist Noiseshader) passieren.

Cinema 4D nach 3DsMax

Das in Cinema 4D fertig gestellte Modell wurde mittels des FBX (.fbx) Formats in 3DsMax importiert. 3DsMax wurde hauptsächlich verwendet, um die Geometrie zu optimieren und diese für den Export in die Unreal Engine vorzubereiten. Aufgrund der starken Verbreitung dieser Software ist eine Vielzahl an Skripten vorhanden, die eine Automation der benötigten Schritte zulassen. Das erleichtert und beschleunigt den Workflow in dieser Phase immens, da ansonsten alle Modellteile händisch bearbeitet werden müssten. Mit Hilfe der Skripten wurden die sogenannten „Smoothing Groups" erstellt (Skript: STR Tools), die benötigten UV Koordinaten angelegt (Skript: Steamroller) und das Modell im – für die Unreal Engine lesbaren - FBX Format exportiert (Skript: UE4FBX).

Smoothing Groups mit STR Tools

Smoothing groups funktionieren in 3Ds Max ähnlich wie das zuvor beschriebene Phong-Tag in Cinema 4D. Es handelt sich dabei um eine Technik, welche durch Polygone angenäherte Rundungen vollkommen glatt erscheinen lässt. Dies geschieht über die Belichtung und verbraucht – im Vergleich zu modellierter Geometrie – nahezu keine Ressourcen. Die zu glättenden Kanten müssen vom Nutzer definiert werden. Dies kann entweder manuell oder durch eine für das ganze Objekt gültige Winkelbeschränkung geschehen. So lässt sich definieren, dass sämtliche Kanten bis zu einem bestimmten Winkel geglättet werden. Alle Kanten mit größeren Winkeln verbleiben unverändert. Dadurch wird verhindert, dass der Effekt an unerwünschten Stellen (z.B. Wandecken) auftritt. Mit dem Skript STR Tools kann dieser Vorgang für eine große Zahl von Objekten gleichzeitig und automatisiert durchgeführt werden. Die Smoothing-Groups werden in den Eigenschaften von jedem Objekt gespeichert und können später von der Unreal Engine korrekt interpretiert werden.

UV Koordinaten mit Steam Roller

Wie erwähnt, bilden korrekte UV Koordinaten eine wichtige Grundlage für die spätere Belichtung in Game Engines. Mit der Unreal Engine besteht zwar die Möglichkeit diese während des Imports automatisiert anlegen zu lassen, allerdings birgt dies hohes Fehlerpotenzial und die Ergebnisse sind meist nicht zufrieden stellend. Aus diesem Grund sollten UV Koordinaten immer zuvor in einer 3D Anwendung angelegt werden. Dabei ist wichtig zu bedenken, dass für Visualisierungen in Game Engines zwei unabhängige Sets an Koordinaten für jedes Objekt angelegt werden müssen. Diese werden auf unterschiedlichen Kanälen gespeichert, wobei jene auf Kanal 1 später für die Materialbelegung und jene auf Kanal 2 für die Vorberechnung von Licht verwendet werden. Grundsätzlich können UV Koordinaten händisch ausgelegt werden, was aber nur bei sehr komplexen (vor allem organischen) Körpern notwendig ist. Im Bereich der Architekturvisualisierung ist es meist ausreichend die automatisierten Algorithmen zu verwenden. Diese finden sich zwar von Haus aus in 3DsMax, allerdings wäre es auch in diesem Fall notwendig, sämtliche Objekte einzeln anzuwählen und die Koordinaten auf den beiden UV Kanälen zu erstellen. Deshalb kommt auch hier ein Skript (Steam-Rol-

ler) zum Einsatz. Es ermöglicht die UV Koordinaten auf einem bestimmten Kanal, für sämtliche Objekte in einer Szene, in einem einzigen Schritt zu erstellen (Abb. 4.4 + 4.5). Diese werden, genauso wie die Informationen zu Smoothing-Groups, ebenfalls innerhalb jedes Objektes gespeichert und beim späteren Import in die Unreal Engine übernommen.

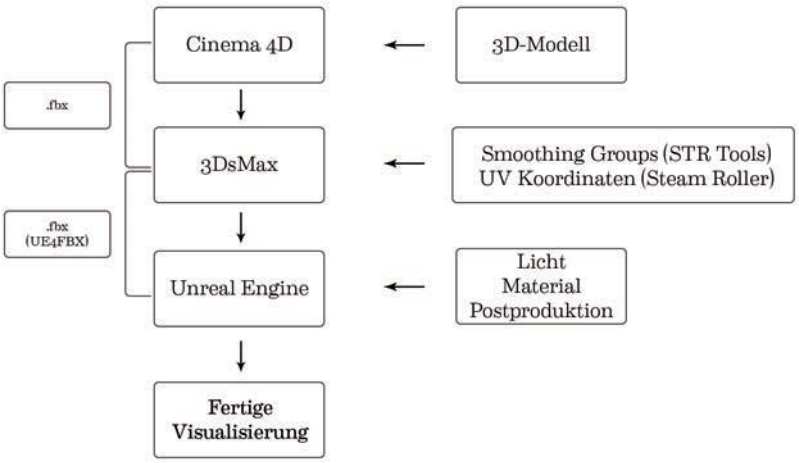
Export mit UE4FBX

Um Geometrie aus 3D Anwendungen in Unreal Engine importieren zu können, ist es notwendig das FBX (Filmbox) Dateiformat zu verwenden. Es handelt sich dabei um ein, von Kaydara entwickeltes und seit 2006 in Besitz von Autodesk befindliches, Dateiformat, welches zur Interoperabilität zwischen verschiedenen 3D Anwendungen geschaffen wurde. Beim Exportieren sollte auf einige wichtige Einstellung geachtet werden. Es muss sichergestellt werden, dass die zuvor erstellten smoothing groups gespeichert werden und alle Polygone trianguliert werden. Wurde zuvor auf die richtige Modelltopologie geachtet, nimmt dieser Vorgang nur wenige Sekunden in Anspruch. Außerdem sollte darauf geachtet werden, dass die Einheiten auf „Automatisch" gestellt sind und das FBX 2014 Dateiformat verwendet wird. Dadurch behält man beim Import in die Unreal Engine die korrekte Skalierung bei. Das verwendete Skript UE4FBX bietet all diese Optionen und ermöglicht den direkten Export aller selektierten Elemente.

Import in Unreal Engine 4

Nach dem erfolgreichen Export aus 3Ds Max wurden die 3D Objekte in die Unreal Engine 4 importiert. Ähnlich wie beim Export sollten auch beim Import in die Unreal Engine einige wichtige Dinge beachtet werden, um das erwartete Ergebnis zu erzielen. Alle gewünschten Objekte können per simplem Drag & Drop im Mesh Editor der Engine platziert werden. In erster Linie sollte die Option zur automatisierten Erstellung von UV Koordinaten deaktiviert werden, da ansonsten die zuvor erstellten Koordinaten überschrieben werden. Außerdem sollten die Objekte als Einzelgeometrie platziert und nicht zu einem großen Mesh kombiniert werden, um die weitere Bearbeitung zu vereinfachen.

Abb. 4.9.2: Ablaufdiagramm für den Export-Workflow



o.: Abb. 4.4: UV Maps im Editor
u.: Abb. 4.5: UV Maps am Objekt

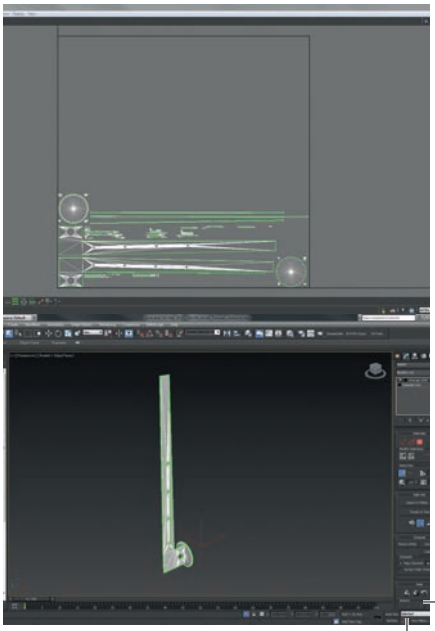


Abb. 4.9.1: Unreal Engine 4 Benutzeroberfläche

Epic Unreal Engine 4

Die Unreal Engine ist eine 1998 von Epic Games veröffentlichte Spiel-Engine. Seit ihrer Veröffentlichung wurde sie auf diversen Betriebssystemen verwendet und auf zahlreiche Konsolen portiert. Während die ersten drei Versionen vorrangig für die Entwicklung von Computerspielen verwendet wurden, bietet die Version 4, welche im März 2014 vorgestellt wurde, außerdem viel Potenzial im Bereich der Architekturvisualisierung (Abb. 4.6 - 4.9). Das Framework basiert auf der Grafik-Engine, der Skriptsprache UnrealScript und weiteren Hilfsprogrammen (z.B. Leveleditor). Die Engine bietet damit ein eigenständiges Paket, mit dem alle für Visualisierungen essentielle Schritte angefangen bei der Belichtung über Materialität bis hin zu Animationen und Kamerafahrten ermöglicht werden.

Die Benutzeroberfläche der Unreal Engine

Die Nutzeroberfläche der Engine kann je nach Bedarf angepasst werden. Grundsätzlich gestaltet sich der Aufbau Modular. Der wichtigste Teil ist das Editorfenster, das die Navigation in der Szene und die Selektion von Objekten beziehungsweise deren Bearbeitung erlaubt. Je nach Bedarf können zusätzliche Fenster wie der Materialeditor, Mesheditor oder der Inhalts-Browser geöffnet werden.

Abb. 4.6: Unreal Engine 1

Abb. 4.7: Unreal Engine 2

Abb. 4.8: Unreal Engine 3

Abb. 4.9: Unreal Engine 4

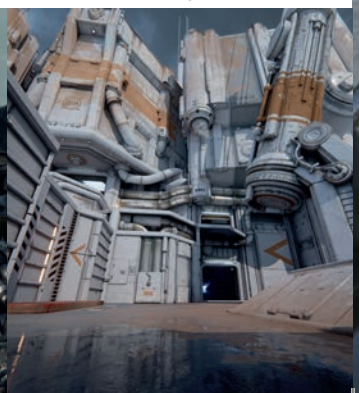
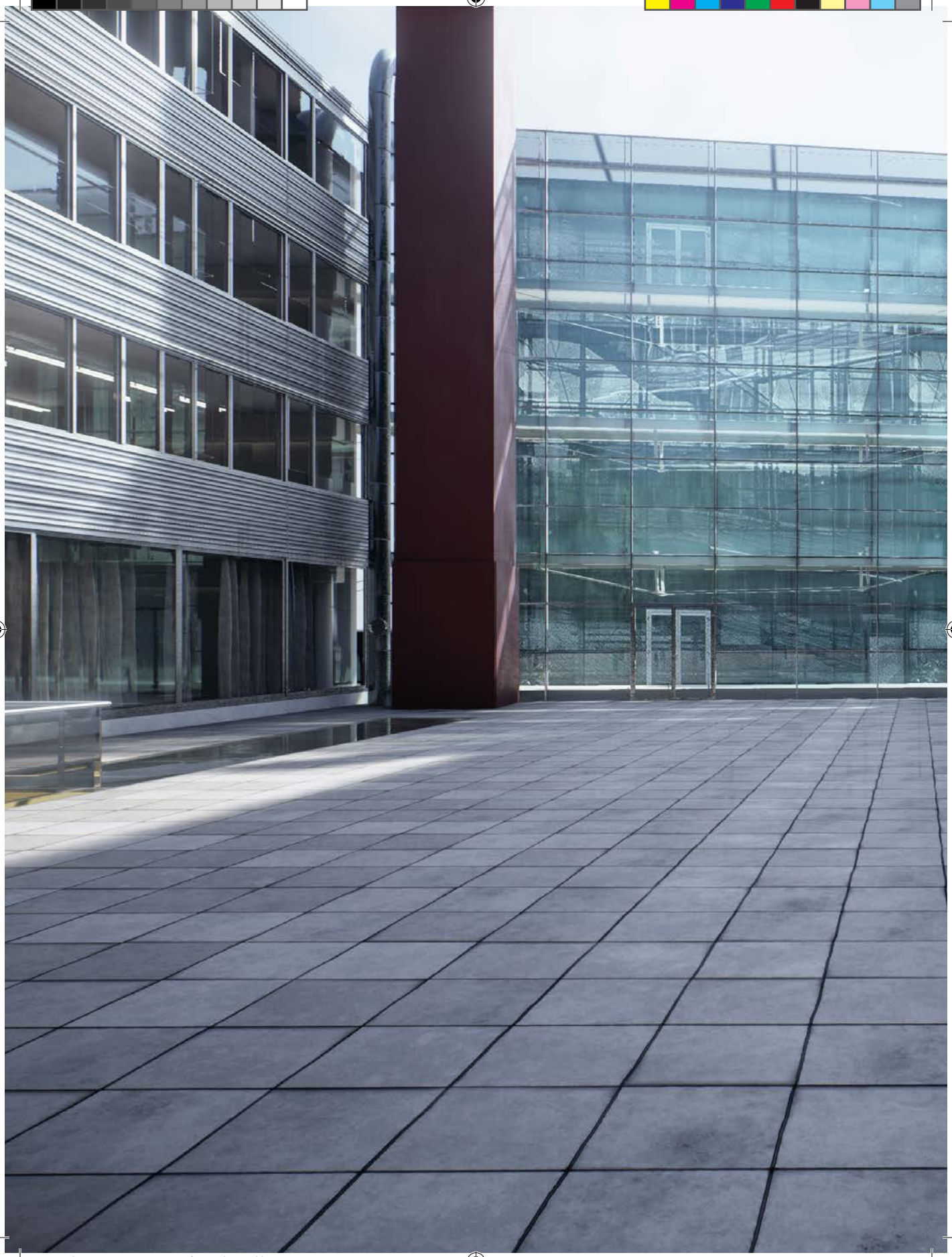


Abb. 0.18



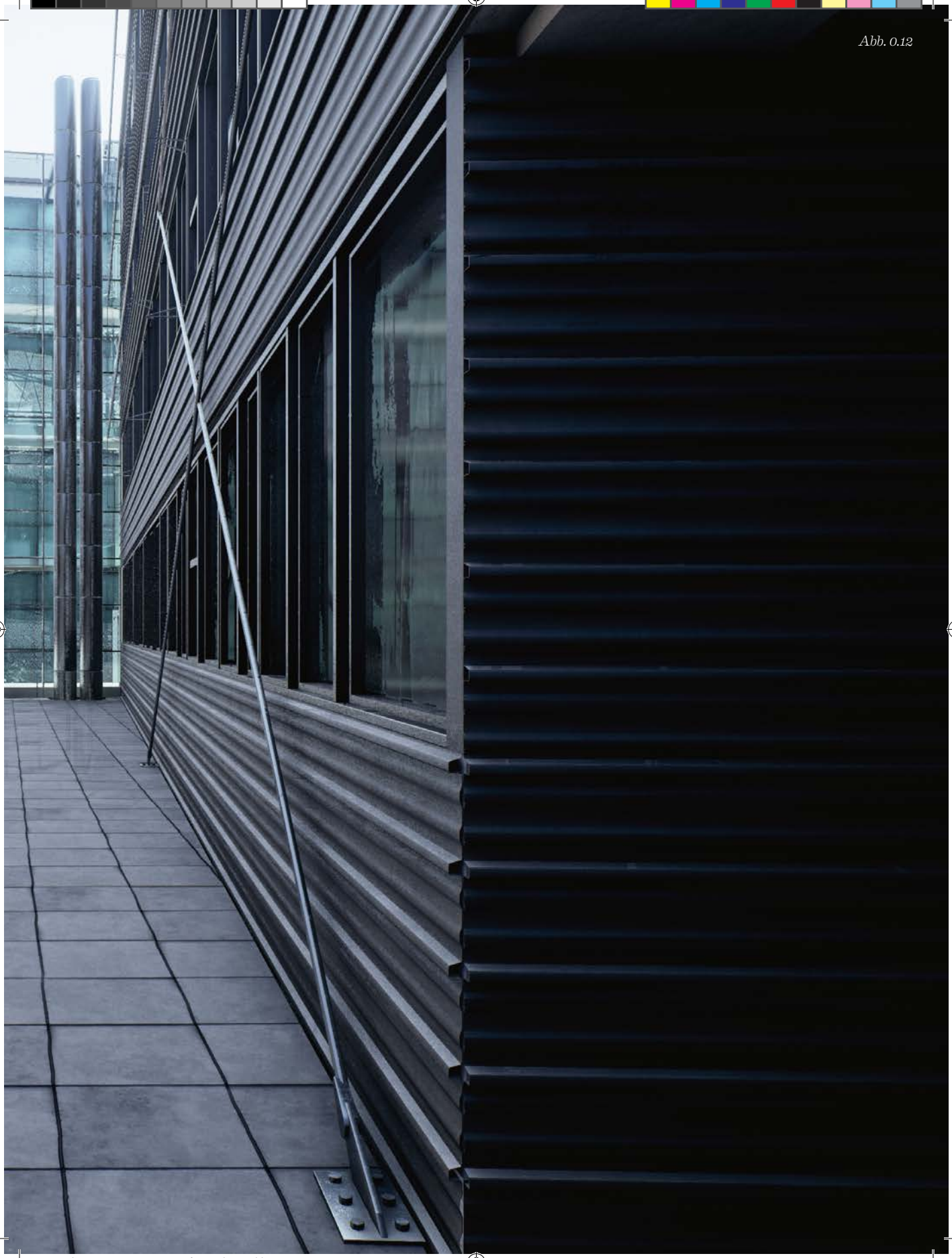


Abb. 0.12

3.5 - Licht

Für die Belichtung des Beispielprojektes kam eine Kombination aus natürlicher und künstlicher Belichtung zum Einsatz. Wie zuvor beschrieben, wurden Lichtquellen grundsätzlich so gesetzt, wie sie auch in der echten Welt vorkommen würden, um ein möglichst fotorealistisches Ergebnis zu erzielen. Eines der wichtigsten Kriterien in diesem Zusammenhang war die korrekte und saubere Berechnung der globalen Illumination:

Lightmass Global Illumination

Als Lightmass wird das Global Illumination System der Unreal Engine bezeichnet. Es ist dafür verantwortlich sogenannte „Lightmaps" mit komplexen Lichtinteraktionen wie zum Beispiel Flächenschatten und indirekter Belichtung zu erstellen. Diese werden dafür verwendet die GI darzustellen und somit ein realistisches Ergebnis überhaupt erst zu ermöglichen. Bei den möglichen Eigenschaften muss zwischen statischen und stationären Lichtquellen unterschieden werden, da nicht alle Effekte, die für statische Lichtquellen zur Verfügung stehen, auch bei stationären verfügbar sind und umgekehrt. Als GI Algorithmus kommt Photon Mapping in Kombination mit Final Gather zum Einsatz.

Die Schritte bei der GI Berechnung in Lightmass:

1. Photonen werden von der Lichtquelle aus in die Szenen geschickt (direkte Lichtstrahlen).
2. Von diesen direkten Strahlen werden jene, welche auf Geometrie treffen, einmal als indirektes Licht reflektiert (indirekte Lichtstrahlen).
3. Jene direkten Strahlen, die dazu führen, dass die daraus resultierenden ersten indirekten Strahlen wieder auf Geometrie stoßen, werden gespeichert. Durch diesen Vorgang werden besonders helle Lichtflecken, welche durch direkten Lichteinfall entstehen und deren Reflektion andere Geometrie im Raum indirekt beleuchtet, identifiziert (z.B. ein heller Fleck am Boden, der durch Sonneneinfall durch ein kleines Fenster entsteht).
4. Jene direkten Strahlen, welche zwar auf Geometrie treffen, deren erster indirekter Strahl danach aber keine weitere Geometrie trifft, (weil er beispielsweise durch eine Öffnung im Raum ins unendliche verschwindet), werden hingegen nicht benötigt.
5. Viele weitere Photonen werden entlang der gespeicherten Strahlen von der Lichtquelle ausgesandt. Diese werden je nach eingestellter Anzahl („bounces“) öfter im Raum abgestoßen. Mit steigender Anzahl der „bounces“ sinkt deren Auswirkung auf die Belichtung stark, somit sind mehr als zwei in der Regel nicht sinnvoll.
6. Nun ist die Photonenkarte fertig und die Lichtberechnung für jeden einzelnen Texel beginnt. Ein Texel lässt sich mit einem Pixel bei einem Rasterbild vergleichen. Allerdings handelt es sich hierbei um den Teil einer Textur (UV Koordinaten) statt eines Rasterbildes.
7. Zunächst erfolgt die Berechnung der direkten Schatten. Um große, qualitativ hochwertige Halbschatten zu berechnen, ist eine höhere Anzahl an Samples notwendig. Hier kommt die im Absatz Photon Mapping beschriebene Optimierung zum Einsatz, bei welcher zwischen komplett verschatteten Punkten (U) und im Halbschatten liegenden Punkten (P) unterschieden wird.
8. Angrenzende Photonen um jeden Texel werden gesammelt. Dadurch werden besonders helle Stellen, von denen indirektes Licht abgestrahlt wird, bestimmt.
9. Der Final Gather Prozess beginnt. Strahlen werden aus der Hemisphäre jedes Texels in die Szene geschickt. Wird Geometrie getroffen, so wird das Licht der Photonenkarte an dieser Stelle gesammelt. Wird keine Geometrie getroffen und der Strahl endet im unendlichen bzw. im Himmel, so wird die Umgebungsfarbe oder Himmelsfarbe und Intensität aufgenommen.
10. Weitere verfeinerte Final Gather Schritte folgen. Die Hemisphäre wird unterteilt und weitere Strahlen ausgesandt. In Richtungen, aus welchen starkes indirektes Licht kommt (heller Fleck am Boden), wird stärker unterteilt (importance sampling oder importance driven final gathering). Auch in jenen Bereichen, wo große Unterschiede zwischen zwei benachbarten Strahlen herrschen, erfolgt eine stärkere Unterteilung um Bildrauschen zu vermeiden (adaptive sampling) [vgl. URL Lightmass 1].

Wichtige Effekte, die bei der GI Berechnung mittels Lightmass möglich sind:

Diffuse Interreflektionen / Indirekte Beleuchtung

Die Simulation von indirektem Licht ist bei weitem die wichtigste Funktion des Lightmass Algorithmus. Sie trägt maßgeblich zur Realität der Visualisierung bei.

Auf Abb. 4.29 ist eine Szene in der Unreal Engine sichtbar. Sie wird mit nur einer gerichteten Lichtquelle belichtet und es ist lediglich das direkte Licht sichtbar. Wie zu erkennen ist, sind alle Bereiche, die nicht direkt angestrahlt werden, komplett schwarz.

Auf Abb. 4.30 wird lediglich der erste „Bounce" – also die erste Reflektion von indirektem Licht – dargestellt. Die Intensität der indirekten Beleuchtung ist abhängig von der Stärke der Lichtquelle und deren Abstand zum jeweiligen Objekt. Hinter dem Sessel ist ein Effekt erkennbar, der als indirekter Schatten bezeichnet wird. Dieser Schatten entsteht durch indirekten Lichteinfall und wäre ohne GI nicht möglich.

Beim nächsten „Bounce" (Abb. 4.31) ist das Licht viel schwächer geworden und gleichmäßiger verteilt.

Kombiniert man alle „Bounces" so erhält man das finale mit GI belichtete Bild (Abb 4.32).

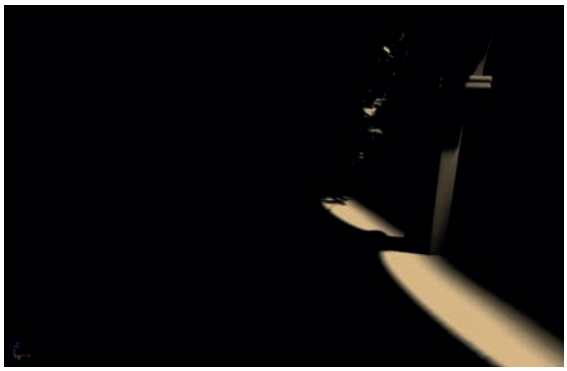


Abb. 4.29:

Abb. 4.30

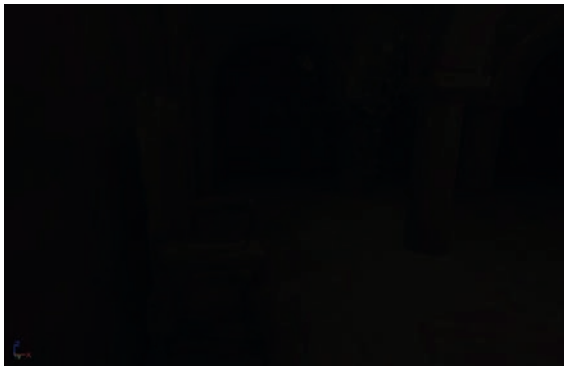


Abb. 4.31

Abb. 4.32

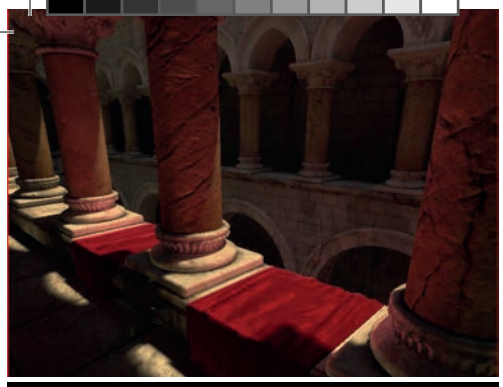


Abb. 4.33

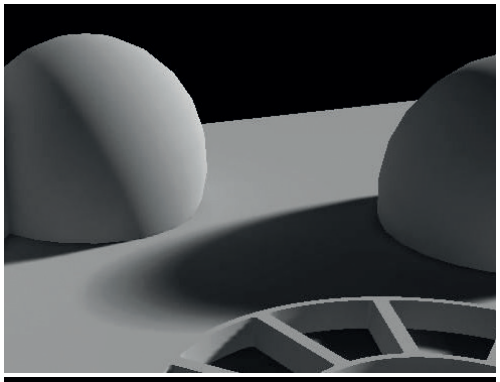


Abb. 4.34

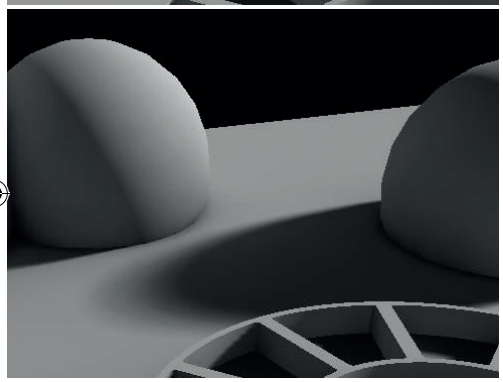


Abb. 4.35

Abb. 4.36

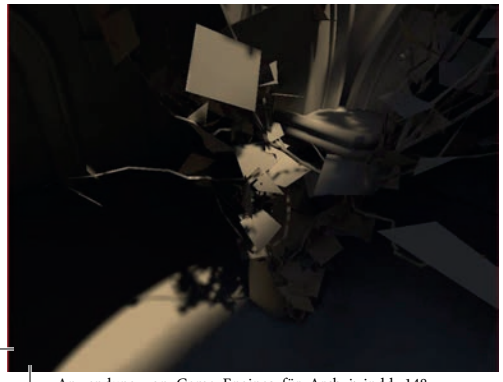


Abb. 4.37

Color Bleeding

In Abb. 4.33 ist sichtbar, dass die indirekten Strahlen, welche bei der GI Berechnung von dem roten Teppich reflektiert wurden, dessen Farbe aufgenommen haben und nun als rotes indirektes Licht die umgebende Geomet-rie beleuchten. Dieser Effekt wird im Englischen oft als Color Bleeding bezeichnet.

Ambient Occlusion (AO) / Umgebungsverdeckung

Bei der Umgebungsverdeckung handelt es sich um eine shading Methode, die in der Computergrafik verwendet wird um noch realistischere Ergebnisse zu erzielen. Sie basiert auf der Tatsache, dass in Ecken und Kanten meist geringere Beleuchtungsstärken vorliegen und somit der Eindruck einer Verschattung entsteht. Bei Entfall dieses Effektes erscheint es oft als würden Körper über dem Boden „schweben“

AO kann sowohl direkt beim Rendervorgang berechnet werden, als auch in der Post Production mittels eines separaten „AO-Pass“ hinzugefügt werden. Lightmass unterstützt die Berechnung von AO sowohl für direktes als auch indirektes Licht.

Abb. 4.34 zeigt eine Berechnung ohne AO. Abb. 4.35 ist mit aktivierter AO berechnet. Wo Objekte aufeinandertreffen, wird ein Schatten berechnet. Diese Simulation ist zwar physikalisch nicht korrekt, täuscht dem menschlichen Auge aber trotzdem eine realistische Darstellung vor.

Maskierte Schatten

Blätter von Pflanzen werden aufgrund der Polygonanzahl nicht modelliert, sondern vereinfacht als einzelnes Polygon erstellt. Die richtige Form erhalten sie erst durch einen Alphakanal im Shader. Damit auch der Schatten dieser Blätter korrekt und nicht als einfaches Polygon dargestellt wird, berücksichtigt Lightmass bei der Schattenberechnung den Alphakanal im Material und beschneidet auch die Schatten korrekt (Abb. 4.36 + 4.37).

Flächenlichter und Flächenschatten

In der Realität haben alle Lichtquellen eine bestimmte Fläche. Auch einzelne Glühbirnen, welche in der Computergrafik oft als (unendlich kleine) Punktlichter dargestellt werden, haben in Wirklichkeit eine bestimmte physikalische Ausdehnung. Die Übergänge zwischen vollkommen verschatteten und direkt belichteten Flächen werden als Halbschatten oder Penumbra bezeichnet. Die Schärfe dieser Bereiche wird maßgeblich durch die Größe (Fläche) der Lichtquelle bestimmt. Das ist einer der Gründe, warum in Fotostudios sogenannte Softboxen mit großen Flächen verwendet werden. Einerseits wird dadurch eine gleichmäßige Lichtverteilung erreicht, andererseits werden harte Schattenkanten vermieden.

Neben der Fläche der Lichtquelle wird die Form der Penumbra außerdem durch den Abstand des Schattenwerfers zum verschatteten Objekt selbst bestimmt. Je weiter der Schattenwerfer entfernt ist, desto unschärfer und weicher erscheint der Übergang.

Dieser Effekt wird auch in der Unreal Engine simuliert und trägt damit stark zum Realismus bei. In dem Beispiel ist zu erkennen, dass der Schatten, der von der Säule auf den Boden geworfen wird, mit steigendem Abstand an Unschärfe zunimmt (Abb. 4.38 + 4.39) [vgl. URL Lightmass 2].

Im Vergleich zu der Darstellung in Abb. 4.38 variiert die Schärfe des Schattens in Abb. 4.39 je nach Abstand zum schattenwerfenden Objekt

Abb. 4.38

Abb. 4.39

Lichteinstellung Kinkplatz

Nachdem das 3D Modell erfolgreich in der Unreal Engine importiert und platziert wurde, folgte zunächst ein simples Lichtsetup, um eine grundlegende Belichtung zu erhalten.

Dieses besteht aus einem Himmelslicht, das mittels eines HDRIs erstellt wurde, und einer unendlichen Lichtquelle zur Simulation von Sonnenlicht. Zum Erlangen des HDRI wurde auf eine käufliche erworbene Bibliothek zurückgegriffen. Das Himmelslicht gleicht in seiner Funktionsweise der von bereits beschriebenen Domelights und ist als unendlich große Halbkugel vorstellbar, auf dessen Innenseite ein HDRI projiziert wird. Über dieses erfolgt die Belichtung (Abb. 4.40 - 4.42). Die Intensität und Farbe des Lichts, welches von dieser Halbkugel in die Szene geschickt wird, ist abhängig vom genutzten HDRI – wodurch eine sehr natürliche Belichtung erzielt wird. Das HDRI wurde gleichzeitig als Hintergrundbild für die Szene verwendet. Um die Schatten und den Kontrast zwischen hellen und dunklen Stellen besser kontrollieren zu können, kommt außerdem eine unendliche Lichtquelle zum Einsatz. Über diese kann die Stärke des Sonnenlichts und der daraus resultierende Schattenwurf einfach und präzise kontrolliert werden.

Um diese grundlegende, natürliche Belichtung später in Echtzeit anpassen zu können, wurden dynamische Lichtquellen gewählt. Dadurch ist es möglich sowohl Intensität als auch Farbe und Richtung des Sonnenlichts bei Bedarf jederzeit anzupassen (Abb. 4.43 - 4.45).

Abb. 4.40 - 4.42: *die Auswirkugen, die unterschiedliche HDRIs auf die Szene haben.*
Abb. 4.43 - 4.45: *dynamische Änderung des Sonnenlichts. Eine Neuberechnung ist nicht notwendig.*
Abb. 4.45.1 (ganz unten): *für die Belichtung verwendetes HDRI (Peter Guthrie, Sun Blue Sky 1433)*

o.: Abb. 4.40 *o.: Abb. 4.41* *o.: Abb. 4.42*
u.: Abb. 4.43 *u.: Abb. 4.44* *u.: Abb. 4.45*

Portallichter

Nachdem das grundlegende Lichtsetup erstellt war, wurde zu Testzwecken eine Vorberechnung des Lichts erstellt. In dieser Phase wurden die Vorteile von Testbildern mit einheitlichem weißem Material deutlich, da Berechnungsfehler bei Licht und Schatten gut sichtbar waren.

Oft ist es möglich, viele dieser Fehler mit höheren Einstellungen bei der Berechnungsqualität zu beseitigen. Doch besonders die problematischen Bereiche im Untergeschoss waren auf eine zu geringe Strahlendichte und dadurch schlecht berechnete globale Illumination zurückzuführen. Solche Probleme entstehen meist bei der Visualisierung von Innenräumen, welche lediglich durch verhältnismäßig kleine Öffnungen mit natürlichem Licht belichtet werden sollen. Bedingt durch den geringen Lichtanteil in solchen Szenarios ist es schwierig eine saubere und zufriedenstellende Ausleuchtung zu erhalten. Abhilfe schaffen hier die seit der Version 4.11 eingeführten Portallichter. Grundsätzlich sind diese nichts Neues, sie werden bei anderen Engines schon seit längerem für eben jene beschriebenen Situationen eingesetzt. Im Prinzip bewirken sie nichts anderes, als die Konzentration der von der Lichtquelle ausgesandten Lichtstrahlen auf gewisse, durch den Benutzer definierte Bereiche. Sie werden in Öffnungen wie Fenster oder Türen eingesetzt, um Licht auf diese Bereiche zu fokussieren und damit eine höhere Qualität der Berechnung durch den GI Algorithmus (bei Unreal Lightmass genannt) zu ermöglichen. Wichtig ist hierbei, dass diese Portallichter selbst kein Licht aussenden – sie steuern lediglich das von anderen Lichtquellen ausgehende Licht in bestimmte Bereiche.

Durch den Einsatz von Portallichtern wurde eine stark verbesserte GI Qualität erreicht. Außerdem konnte der Lightmass Algorithmus ressourcenschonender arbeiten, was sich positiv auf die Vorberechnungszeit auswirkte.

Lightmass Importance Volume

Eine weitere Optimierung der Berechnungszeit konnte durch den Einsatz von so genannten Lightmass Importance Volumes erzielt werden. Wie bereits im vorigen Kapitel beschrieben, ist mit diesen eine Unterscheidung zwischen wichtigen und unwichtigen Bereichen einer Szene möglich. Die GI Berechnung wird auf die essentiellen Teile des 3D Modells konzentriert, was – wie das nachfolgende Beispiel verdeutlicht – die Renderzeit optimiert.

Durch Verwendung von Portallichtern und eine Vorberechnung mit höherer Qualität konnte eine angemessene und zufriedenstellende Lichtqualität mit vertretbarer Berechnungszeit durch rein natürliche Belichtung erreicht werden. Durch schlechte GI entstehende Flecken wurden minimiert und Schatten detaillierter dargestellt. Dank des Einsatzes von Portallichtern und Lightmass Importance Volumes konnten Ressourcen effizient genutzt und die Renderzeit optimiert werden. Mit diesem Ergebnis war vorerst eine Weiterarbeit möglich, zusätzliche Verbesserungen konnten zu einem späteren Zeitpunkt – nachdem die Texturierung erfolgt war – stattfinden [vgl .URL Lightmass 3].

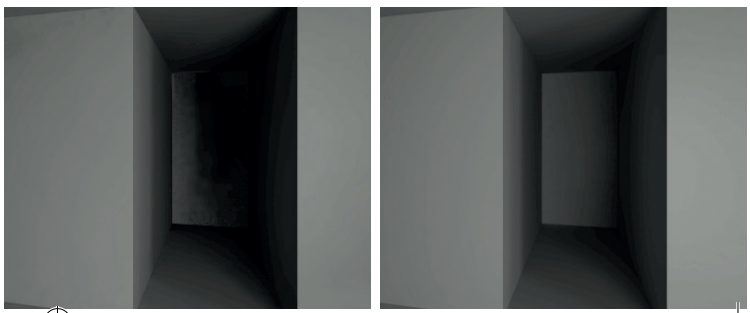


Abb. 4.46

Das linke Bild wurde ohne die Anwendung von Portallichtern berechnet. Die unsaubere Schattenberechnung ist deutlich erkennbar. Im rechten Bild kamen Portallichter zum Einsatz, das Resultat ist eine saubere globale Illumination

Kunstlicht

In Ergänzung zur natürlichen Belichtung, wurden im Untergeschoss der Schule – wie es auch im echten Vorbild der Fall ist – Kunstlichter eingesetzt. Im Hinblick auf eine möglichst ressourcenschonende Arbeitsweise wurde bei diesen Lichtern auf eine Einstellbarkeit in Echtzeit verzichtet.

Abb. 4.47: Turnsaal im UG mit deaktivierter künstlicher Belichtung.

Abb. 4.48: Turnsaal im UG mit aktivierter künstlicher Belichtung.

Virtuelle Repräsentation der tatsächlich vorkommenden Lichtsituationen

Mit dem beschriebenen Lichtsetup ist es möglich, die tatsächlich vorherrschenden unterschiedlichen Lichtsituationen präzise zu simulieren. Beispielhaft sind hier vier stark von der Belichtung abhängige Stimmungen dargestellt. Dank der Möglichkeiten von dynamischen Lichtern in der Unreal Engine kann schnell und ohne zusätzlichen Renderzeiten zwischen diesen gewechselt werden.

o: Abb. 4.49.1: Hof mit direktem Sonnenlicht
u: Abb. 4.49.4: Turnsaal mit direktem Sonnenlicht

o: Abb. 4.49.2: Hof mit Sonnenlicht durch Erschließungszone

o: Abb. 4.49.3: Hof ohne Sonnenlicht
u: Abb. 4.49.5: Turnsaal ohne Sonnenlicht

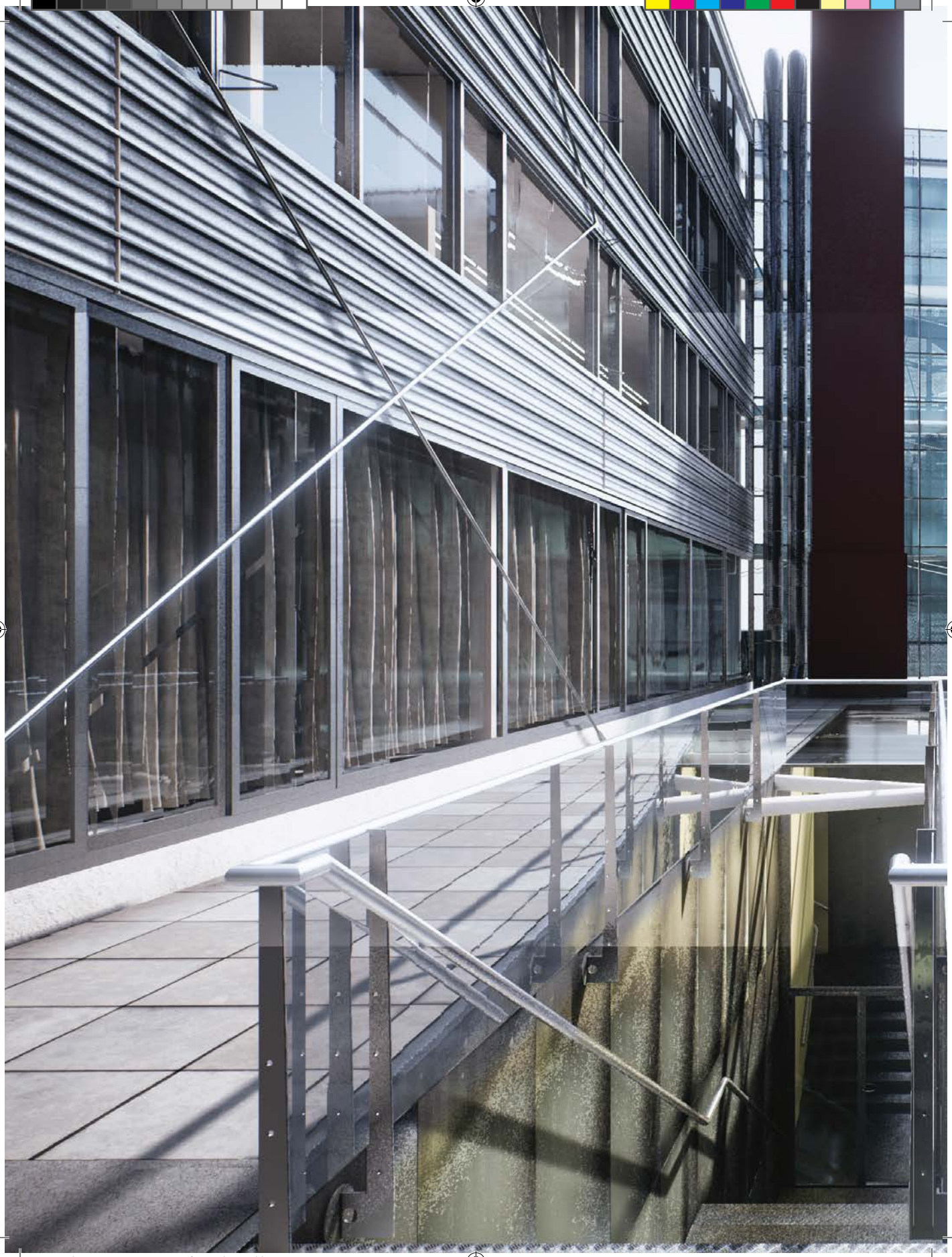
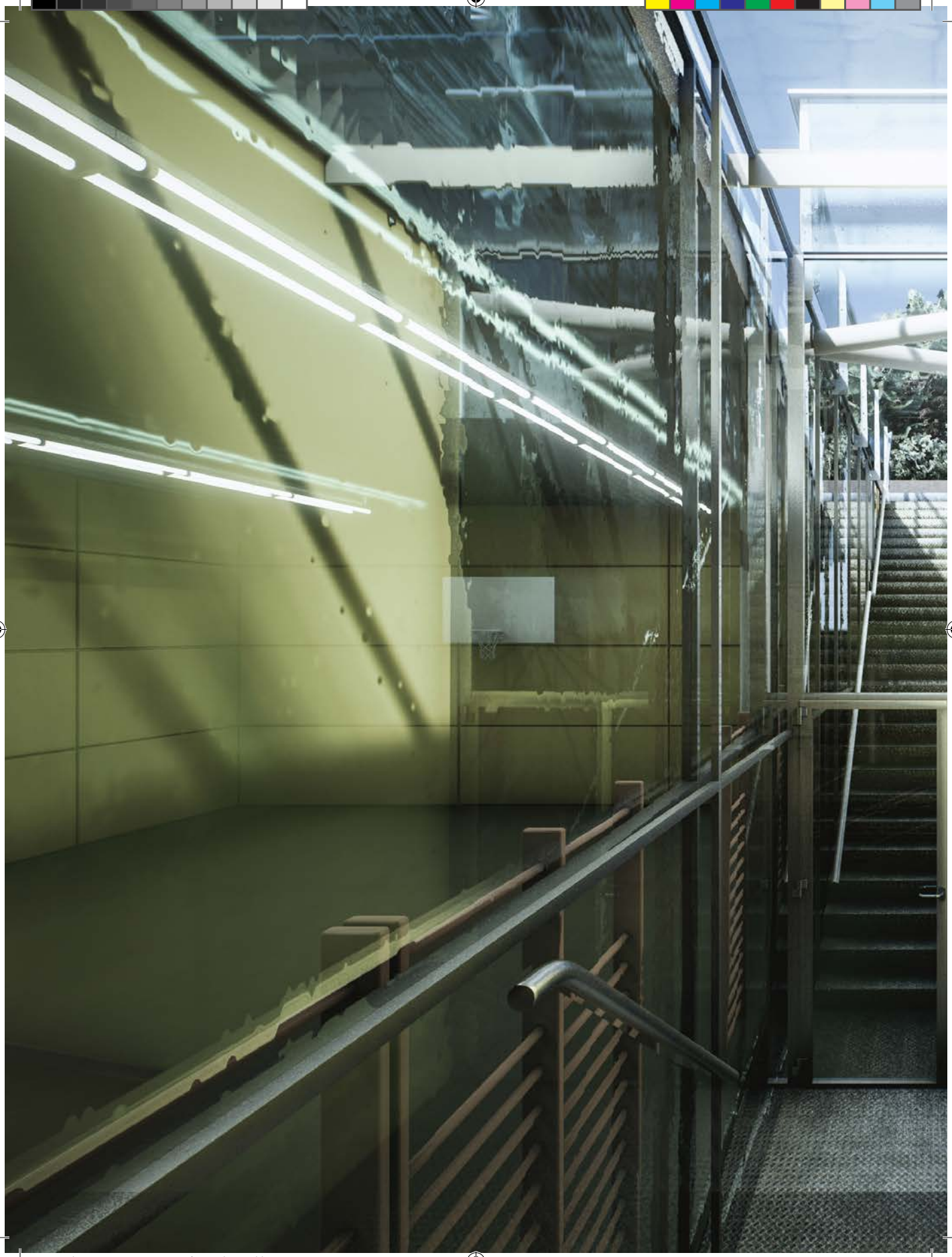


Abb. 0.2



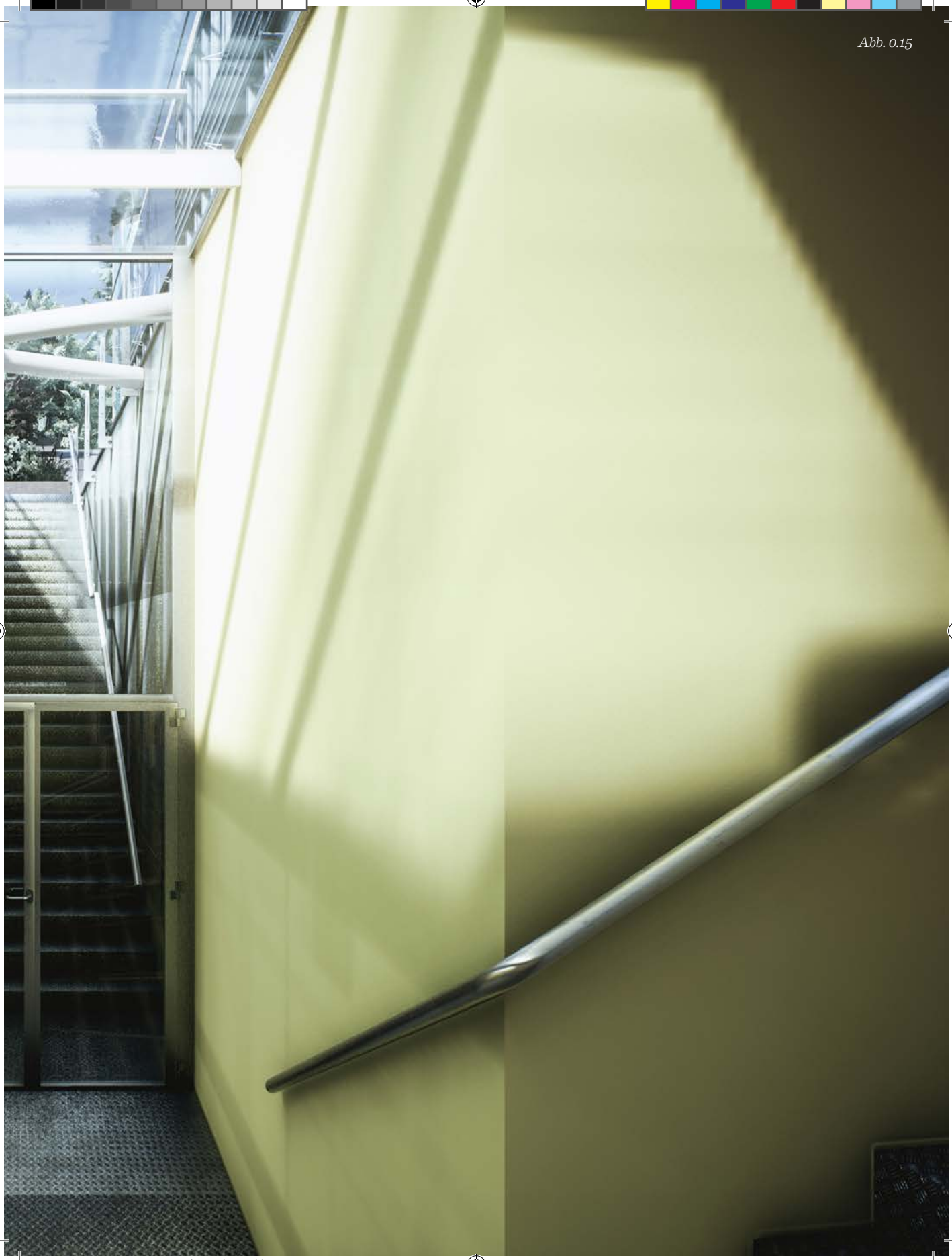
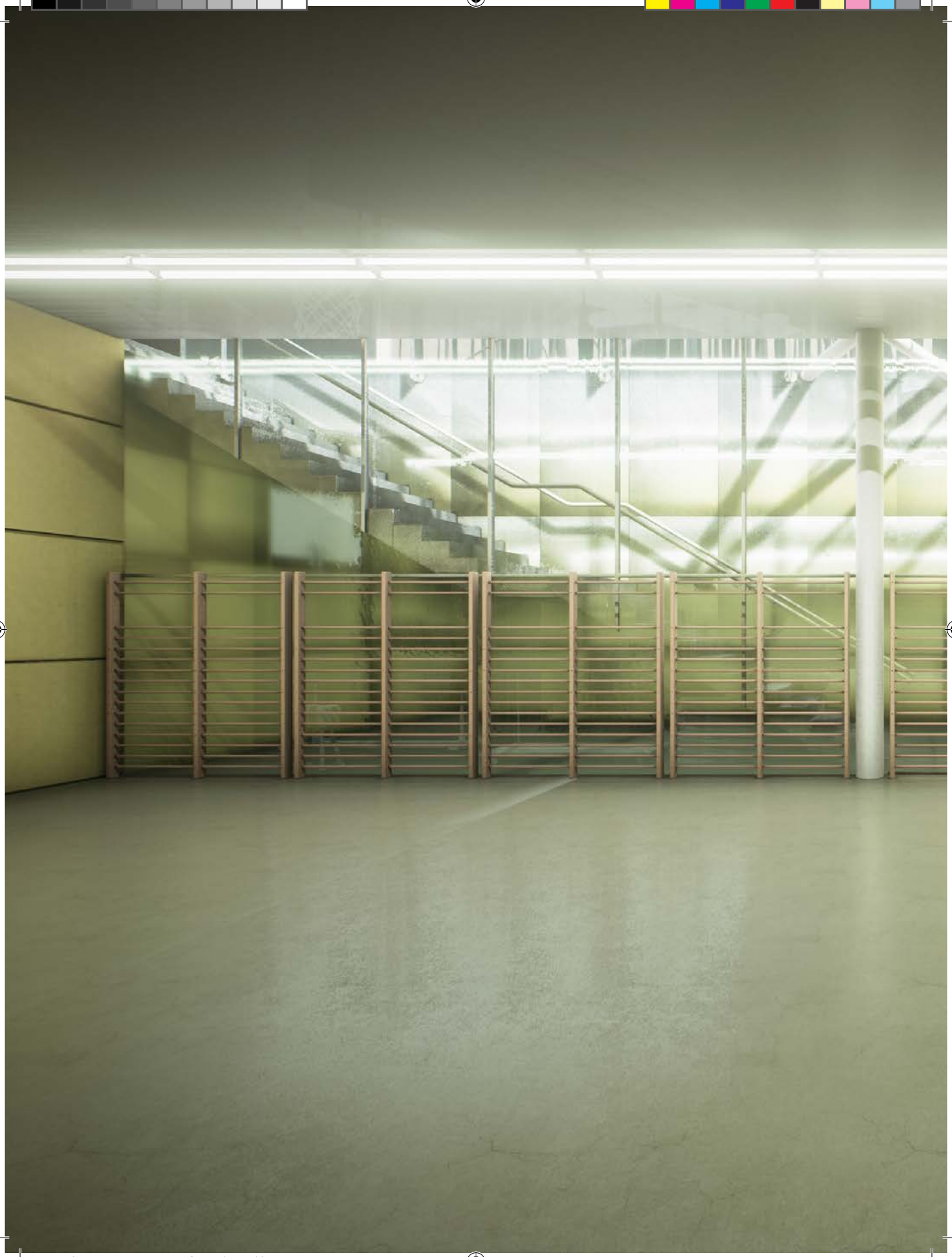


Abb. 0.15



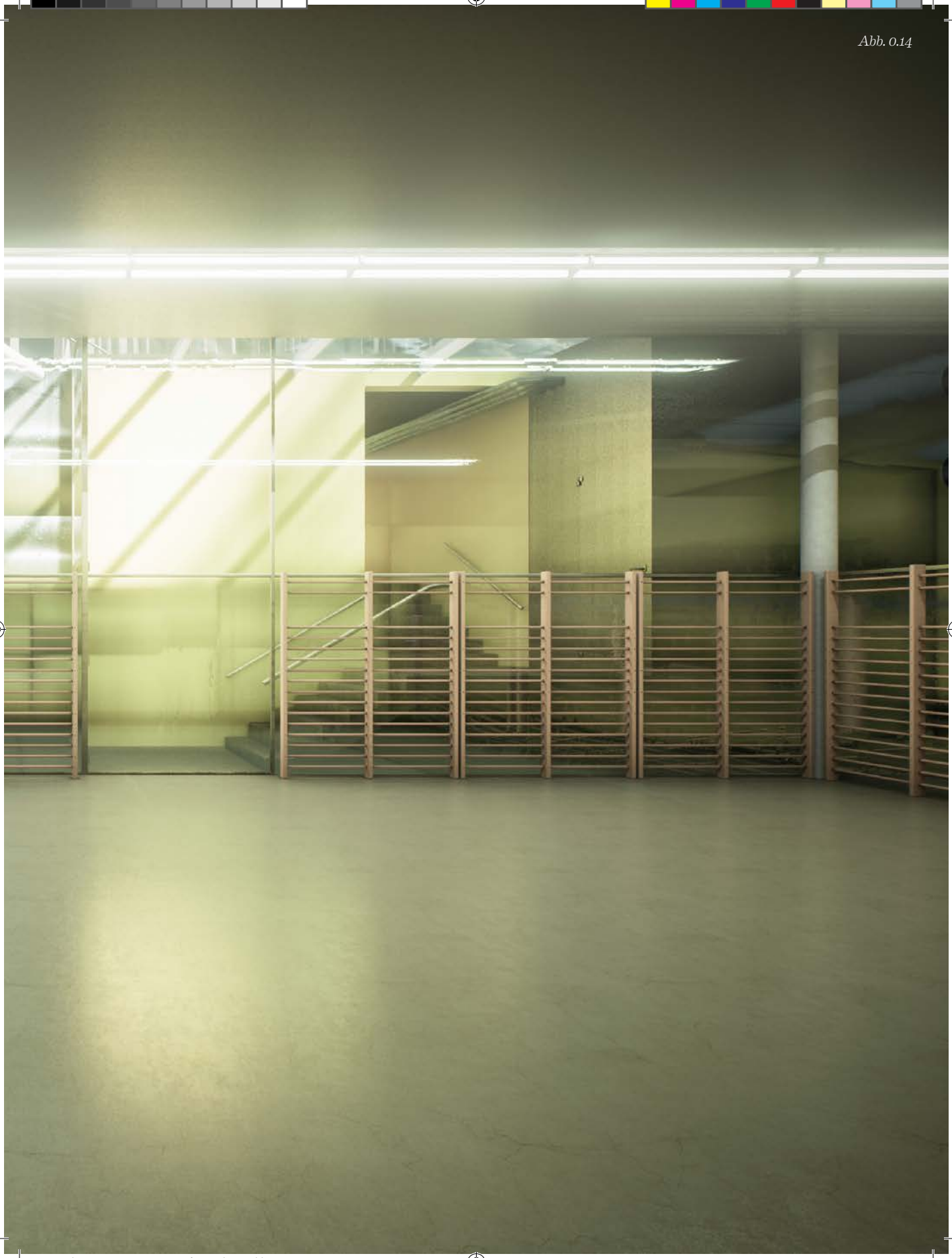


Abb. 0.14

Abb. 4.481: Unreal Material Editor

Abb. 4.49: Unreal Material Editor

Abb. 4.50: Unreal Material Editor

Abb. 4.51: Unreal Material Editor

Abb. 4.52: Instanz-Editor

3.6 - Material

Für die Erstellung der notwendigen Materialien wurden sowohl der engineinterne Materialeditor, als auch Anwendungen von Drittanbietern (Substance Designer) verwendet. Der Materialeditor der Unreal Engine ist parametrisch aufgebaut. Dadurch sind spätere Änderungen an Materialien leicht möglich, da Elemente einfach hinzugefügt oder gelöscht, beziehungsweise mit den unterschiedlichen integrierten Shadern angepasst werden können. Die Idee besteht darin, verschiedene Bilder, Effekte und Einstellungen miteinander zu verbinden, um bestimmte Texturen zu generieren. Diese werden dann, genau wie bei traditionellen Materialeditoren bei anderen Engines, mit dem jeweiligen Kanal verbunden. Die Gesamtheit aller Kanäle ergibt das fertige Material.

Die Unreal Engine beinhaltet innerhalb des Materialeditors ein Vorschaufenster. Hier können Materialien an einem abstrakten Objekt kontrolliert werden. Die Aktualisierung erfolgt in Echtzeit, was eine sehr schnelle und intuitive Arbeitsweise ermöglicht (Abb. 4.48.1 - 4.51). Außerdem werden Änderungen sofort bei sämtlichen Objekten im Ansichtsfenster aktualisiert sobald das Material gespeichert wird.

Für die Visualisierung der Schule am Kinkplatz wurde der Unreal Material Editor beispielsweise für die Putz- und Glasoberflächen verwendet. Nun tauchen diese beiden Materialien öfter in leicht abgewandelter Form im Projekt auf. Statt – wie üblich - eine Kopie des vorhandenen Materials zu erstellen, die Änderungen vorzunehmen und das Material erneut zu verwenden, bietet die Unreal Engine für solche Fälle eine elegantere und ressourcenschonendere Lösung an:

Materialinstanzen in der Unreal Engine

Das Instanz-System der Unreal Engine erlaubt viele unterschiedliche Materialien aus einem einzigen sogenannten Mastermaterial zu erstellen. Dazu wird zunächst eine Instanz des jeweiligen Materials erstellt. Es handelt sich dabei im Prinzip um eine exakte Kopie – mit dem Unterschied, dass erstmal keine Parameter geändert werden können und grundsätzlich alle Änderungen, die am Mastermaterial vorgenommen werden, auch durch die Instanz übernommen werden. Im nächsten Schritt werden jene Einstellungen definiert, welche in den Instanzen angepasst werden sollen. Es kann sich dabei um jegliche Materialeigenschaften, wie zum Beispiel Farbe, Spiegelung oder Oberflächenunebenheiten, handeln. Selbst Bilder, die zum Erstellen von Texturen verwenden werden, können für einen späteren Tausch freigegeben werden. Welche und wie viele Eigenschaften in den Instanzen angepasst werden können, bleibt alleine dem Nutzer überlassen.

Durch einen Klick auf die Materialinstanz öffnet sich der Instanz-Editor (Abb. 4.52). Dort können alle – zuvor im Mastermaterial freigegebenen – Parameter adjustiert werden. Im Unterschied zu „gewöhnlichen“ Materialien, werden Änderungen an Instanzen in Echtzeit im Ansichtsfenster sichtbar. Das erleichtert Einstellungen immens, da die Auswirkungen sofort in der finalen Umgebung betrachtet werden können.

Durch die konsequente Anwendung von Materialinstanzen kann sowohl Zeit als auch Leistung eingespart werden, da diese nur einen Bruchteil des Ressourcenbedarfs von Standardmaterialien haben. Außerdem kann die Materialbibliothek auf diese Weise schlank und übersichtlich gehalten werden.

Substance Designer

Eine Möglichkeit die Materialerstellung für Projekte in der Unreal Engine weiter zu optimieren, bietet die Anwendung von Software, welche speziell für diesen Bereich entwickelt wurde. Bei der vorliegenden Visualisierung wurde der Substance Designer von Allegorithmic verwendet. Es handelt sich dabei um ein Programm, das die parametrische Erstellung von Texturen erlaubt. Grundsätzlich ähnelt die Funktionsweise stark jener des engineinternen Materialeditors, jedoch bieten sich weitaus mehr Möglichkeiten. Außerdem können im Substance Designer Texturen unabhängig von der Engine erstellt werden, was große Vorteile bei der Anwendung von mehreren Renderern birgt.

Meist sind es wenige, vergleichsweise simple Eingriffe, die den Unterschied zwischen einem qualitativ hochwertigen und minderwertigen Material ausmachen. Ausschlaggebend ist nicht die Komplexität des Materials sondern vielmehr die Qualität der einzelnen verwendeten Texturen. Besonders wichtig ist in diesem Zusammenhang die so genannte „Height-Map". Sie ist maßgeblich für das Erscheinungsbild eines Materials verantwortlich und mittels wenigen Schritten können aus ihr alle anderen Texturen generiert werden. Es bietet sich also an, diese zunächst zu erstellen und in der Folge auf ihr aufzubauen. Eines der Materialien, bei denen diese Arbeitsweise angewandt wurde, ist beispielsweise die für den Weg verwendete Asphaltoberfläche. Mehrere verschiedene, parametrisch erstellte Texturen wurden überblendet, um eine entsprechende „Height-Map" zu kreieren (Abb. 4.56). Diese wurde im nächsten Schritt mittels eines Farbverlaufs eingefärbt, um eine passende Textur für den Diffus-Kanal zu erhalten. Außerdem wurde sie für die Erstellung einer Maske verwendet, die zur Steuerung der Verfärbungen dient. Damit spätere Anpassungen in der Spiele-Engine möglich sind, wurden bestimmte Parameter freigegeben. Diese werden beim Export gespeichert und sind für spätere Einstellungen verfügbar. Die so erstellten Texturen wurden kombiniert, um alle benötigten Kanäle für das fertige Material zu erhalten (Abb. 4.57), das mittels des

Abb. 4.56: Height-Map Graph

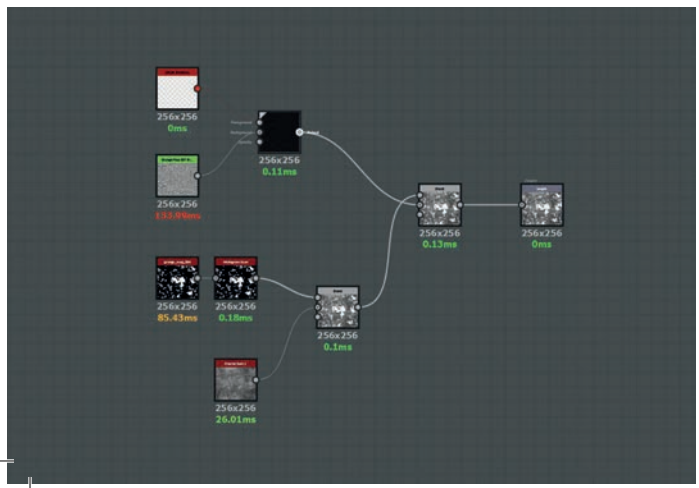
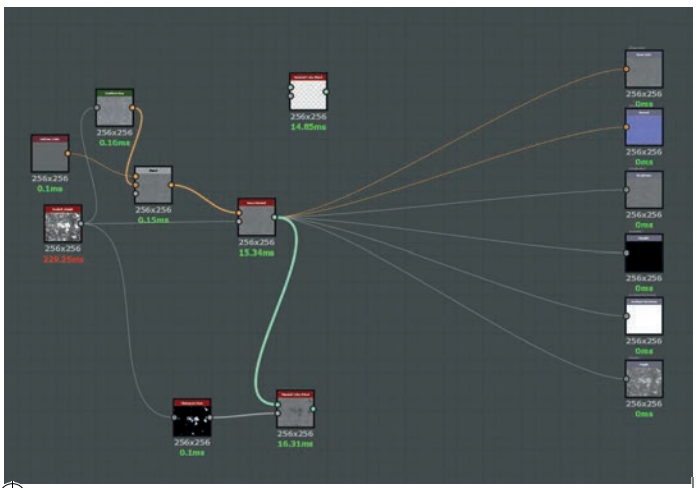


Abb. 4.57: Graph für das fertige Material



.sbsar Formats in der Unreal Engine importiert wurde (Abb. 4.56 - 4.55).

Innerhalb der Unreal Engine kann das Material bei Bedarf weiter angepasst werden. Im vorliegenden Beispiel, wurden einige simple Ergänzungen vorgenommen, welche das Einstellen der Ausrichtung und Kachelung des Materials erlauben. Außerdem wurden Instanzen verwendet, um Variationen des Materials zu erhalten, ohne dieses selbst zu verändern. Änderungen im Substance Designer bzw. eine Aktualisierung der .sbsar Datei, wurden auf diese Art und Weise automatisch für alle Instanzen übernommen.

Es handelt sich beim beschriebenen Beispiel um kein besonders komplexes Material. Es dient lediglich zur Veranschaulichung der Möglichkeiten, die sich durch zusätzliche Software bei der Erstellung von Materialien bieten. Trotzdem war es mit wenigen einfachen Schritten möglich, ein ansprechendes Material zu erstellen, welches – verglichen mit „Standardmaterialien" – wesentlich lebendiger und realistischer wirkt.

Im Arbeitsalltag kommt es häufig vor, dass ähnliche Materialien immer wieder zum Einsatz kommen. Um Zeit und Kosten zu sparen, bietet sich das Anlegen einer Materialbibliothek an. Auch bei diesem Projekt wurde eine Bibliothek verwendet um die Übersicht zu bewahren. Es handelt sich dabei um eine simple Ordnerstruktur, in der die jeweiligen .sbsar Dateien abgelegt wurden. Zu Vorschauzwecken wurde der Substance Player verwendet. Mit dieser Anwendung konnten Materialien und deren Einstellungen schnell erprobt werden. Diese Vorgangsweise bietet sich besonders bei Szenarios an, wo es keine Möglichkeit der Echtzeitvorschau innerhalb der verwendeten Engine gibt (bei Renderern, die nicht in Echtzeit agieren).

Abb. 4.53: Vorschau im Substance Player

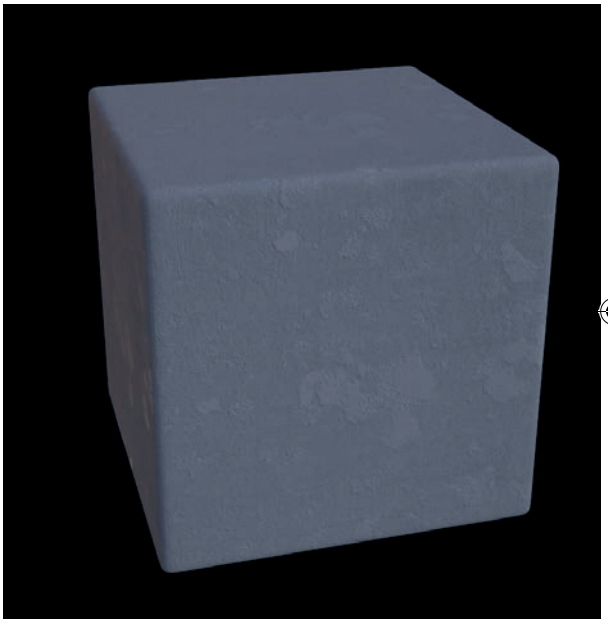


Abb. 4.54: Vorschau im Substance Player

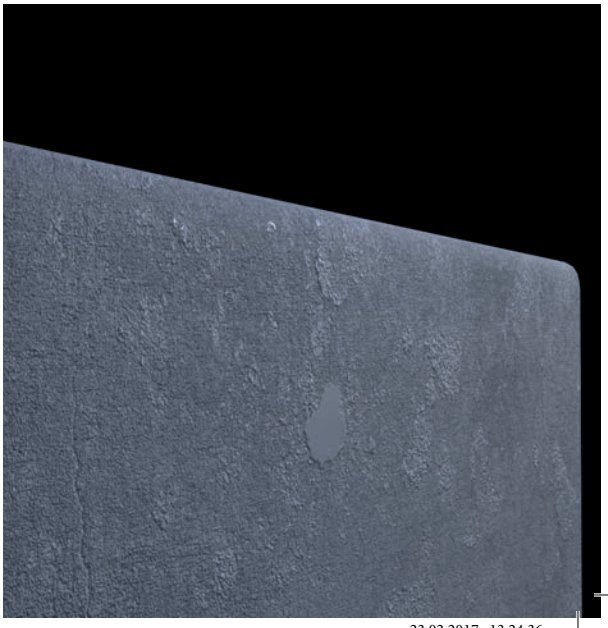
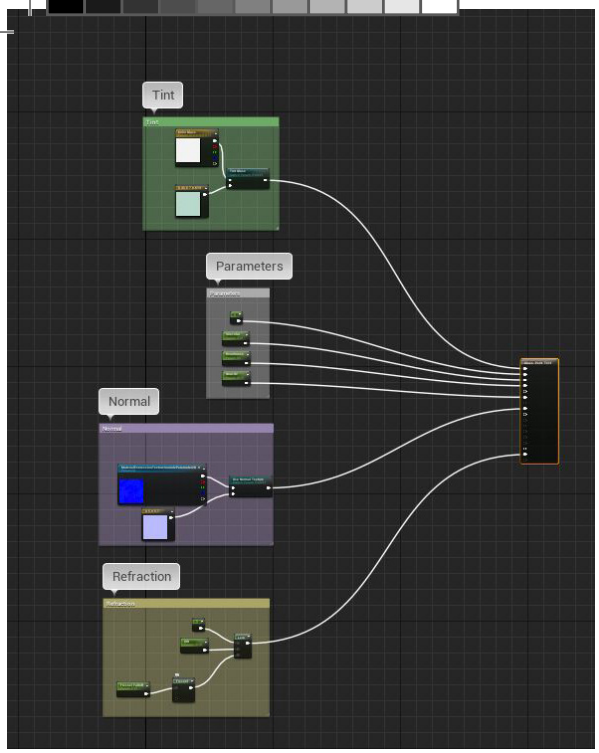
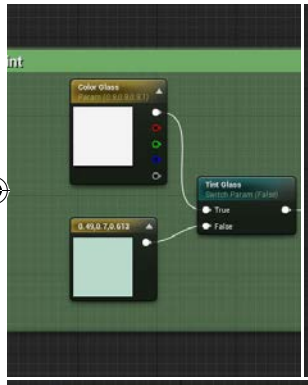
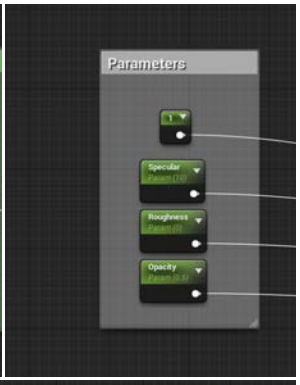
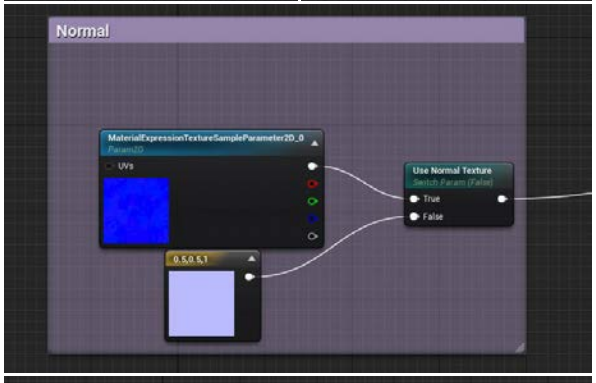


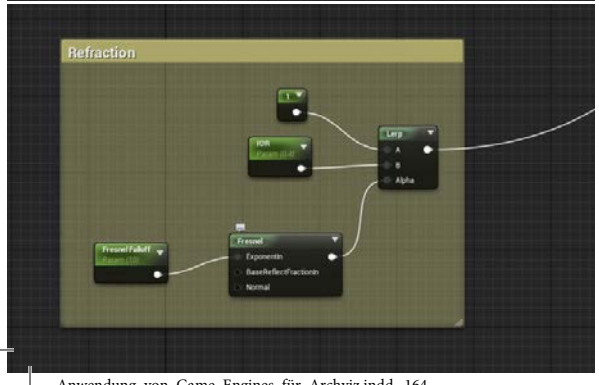
Abb. 4.55: Vorschau im Substance Player











Virtuelle Repräsentation der in der Realität verwendeten Materialien

Um eine möglichst realitätsgetreue virtuelle Nachbildung der Schule schaffen zu können, war eine präzise Nachbildung aller sichtbarer Materialien notwendig. Grundsätzlich wurden ähnliche Techniken wie bei der beschriebenen Asphaltoberfläche angewandt, wenn auch einige Materialien direkt im internen Materialeditor der Unreal Engine anstatt mit Substance Designer erstellt wurden. Drei für das Projekt wichtige Materialien sind im Folgenden im Detail beschrieben.

Glaselemente der Fassade

Base Color: Hier wird die Tönung des Glases definiert. Für ein realistisches Ergebnis ist auch bei grundsätzlich farblosem Glas eine leichte grün-bläuliche Tönung wichtig. Über eine Wahr/Falsch Funktion kann die Tönung später ein- und ausgeschalten werden.
Normal: Hier werden Unebenheiten an der Oberfläche des Glases mittels einer Textur definiert. In der Realität weist nahezu jedes Glas irgendeine Art von Unebenheiten auf. Die Simulation dieser ist für ein realitätsgetreues Ergebnis wichtig. Über eine Wahr/Falsch Funktion kann der Effekt später ein- und ausgeschalten werden.
Refraction: Über diesen Kanal wird die Lichtbrechung und damit die Art und Weise gesteuert, wie Objekte, die sich hinter dem Glas befinden, erscheinen. Es kommt ein Funktion zur Simulation des Fresnel Effektes zum Einsatz. Diese lässt Spiegelungen je nach Blickwinkel unterschiedlich stark erscheinen. Über den IOR wird der Index of Refraction (Brechungsindex) angegeben. Er definiert, wie stark Licht, das durch das Glas fällt, gebrochen wird.
Parameters: Hier werden die Werte für das Glanzlicht (Specular), die Schärfe der Spiegelung (Roughness) und die Transparenz (Opacity) des Glases definiert. Sie werden in diesem Fall über Werte zwischen 0 und 1 gesteuert.

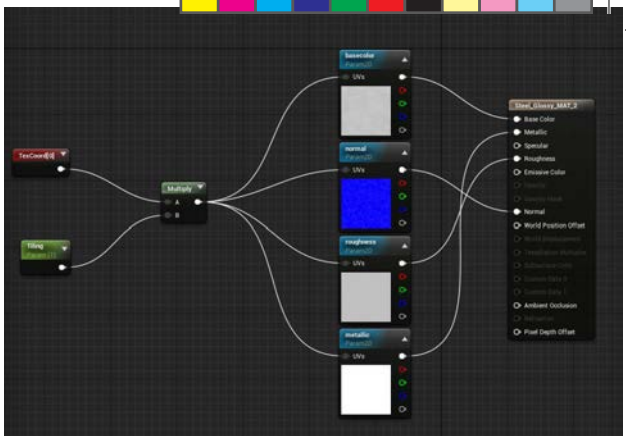
v.o.n.u.:
Abb. 4.58.1: Gesamtmaterial
Abb. 4.58.2:Base Color
Abb. 4.58.3: Parameters
Abb. 4.58.4: Normal
Abb. 4.58.5: Refraction

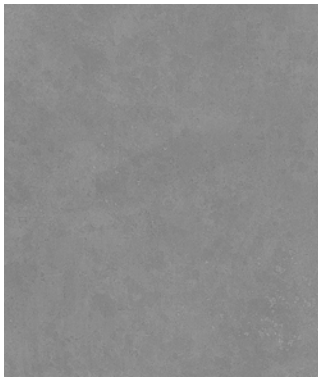
Fassadenelemente aus Stahl

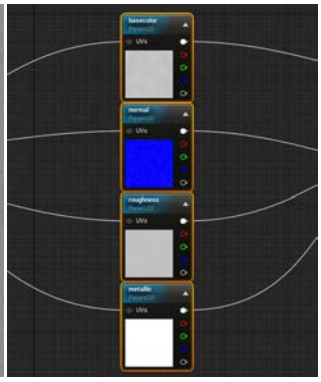
Die Texturen für dieses Material wurden mittels Substance Designer erstellt. Genau wie beim zuvor beschriebenen Asphaltmaterial wurden mehrere Noise Shader miteinander multipliziert, um eine Textur mit leichten Unregelmässigkeiten zu erhalten. Dies sorgt für ein natürliches Erscheinungsbild. Die Texturen wurden in Unreal importiert und im internen Editor zu einem Material zusammengefügt.
Mit den Kanälen Base Color, Normal, Roughness und Metallic erfolgte die Definition des Materials. Die Schärfe der Spiegelung (Roughness) ließe sich grundsätzlich auf über einen Wert zwischen 0 und 1 für das gesamte Material steuern. Die hier verwendete Steuerung über eine Textur mit unterschiedlichen Grauwerten sorgt aber für Variationen innerhalb des Materials und damit für ein wesentlich realistischeres Ergebnis.

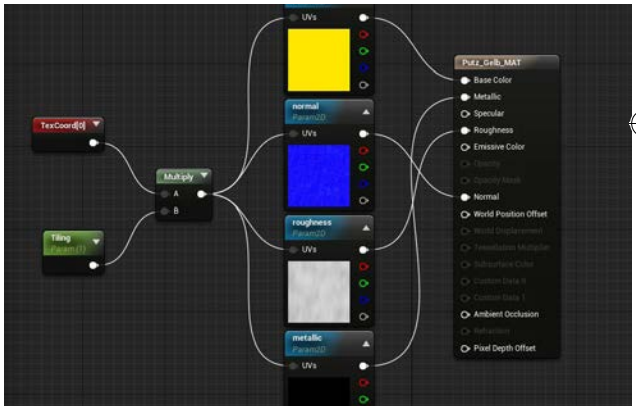
Gelbe Wände am Gang

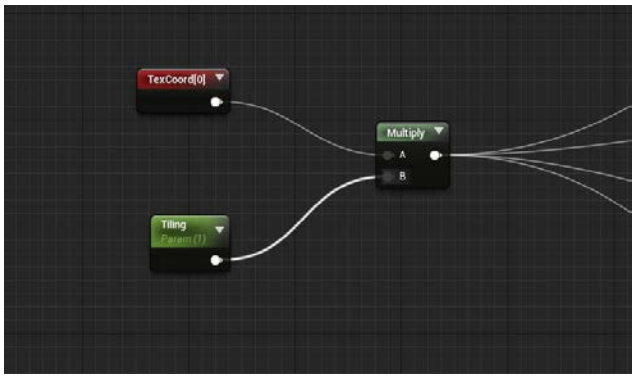
Für das gelbe Putzmaterial am Gang, wurden wiederum im Substance Designer Noise Shader kombiniert, um die entsprechenden Texturen zu erhalten. Für den Normal Kanal wurde eine kleinteilige Textur mit vielen kleinen Unebenheiten erstellt, um die in der Realität vorkommenden Poren im Material zu simulieren. Bei der Roughness sorgt eine Textur mit unterschiedlichen Graustufen für Variationen innerhalb des Materials. Für die Farbtextur war in diesem Fall eine einfache, eintönige Farbe ausreichend. Eine Besonderheit in diesem Zusammenhang stellt die Steuerung der Kachelung dar. Sie wurde implementiert, um später steuern zu können, wie oft die Textur auf einer Fläche wiederholt wird und damit die Feinheit des Materials beeinflussen zu können.

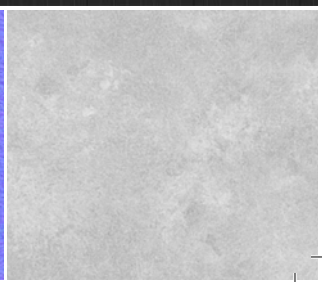












v.o.n.u.:
Abb. 4.58.6: Stahlelemente Gesamtmaterial
Abb. 4.58.7:Stahlelemente Roughness Textur
Abb. 4.58.8: Stahlelemente Texturen in den einzelnen Kanälen
Abb. 4.58.9: Gelbe Wände Gesamtmaterial
Abb. 4.58.10: Gelbe Wände Kachelung
Abb. 4.58.11: Gelbe Wände Normal
Abb. 4.58.12: Gelbe Wände Roughness

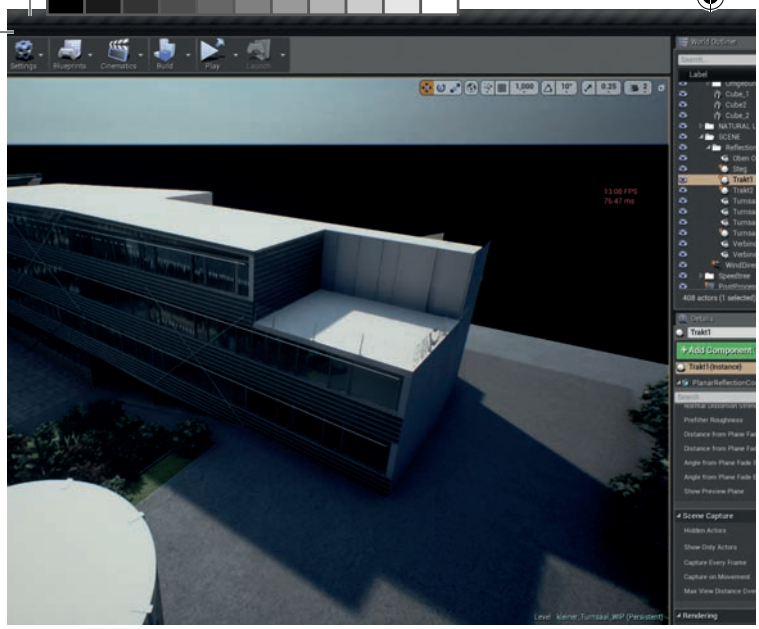


Abb. 4.58: Reflection Environment in der Unreal Engine

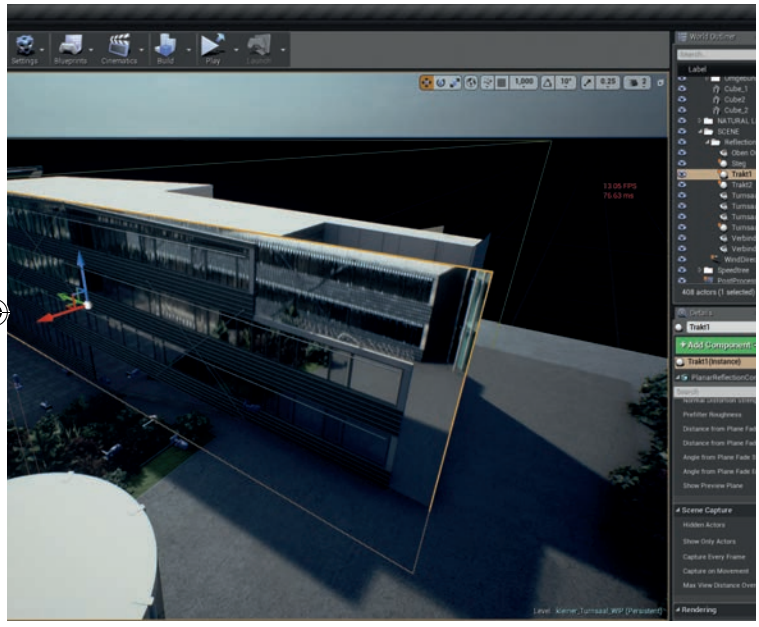


Abb. 4.59: Reflection Environment in der Unreal Engine

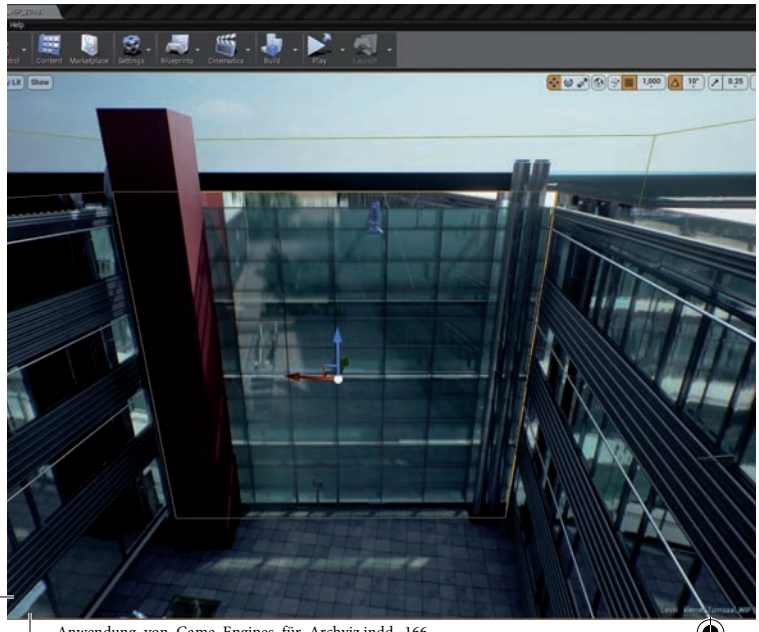


Abb. 4.59.1: Reflection Environment in der Unreal Engine

Spiegelungen in der Unreal Engine

Um realistische Spiegelungen bei Visualisierungen in der Unreal Engine zu erhalten, ist die Anwendung der Spiegelungsumgebung („Reflection Environment“) notwendig. Sie wird über so genannte „Reflection Captures“ gesteuert. Dabei handelt es sich um Objekte, welche die Szene in ihrer Umgebung auf eine einfache Form (Kugel, Würfel oder Ebene) projizieren und diese als Spiegelung auf Materialien wiedergeben (Abb. 4.58 - 4.59.1). Sie können vom Nutzer beliebig platziert und skaliert werden, um die Reflektionen anzupassen. Dabei ist wichtig zu beachten, dass kleinere Captures größere überschreiben. Dadurch besteht die Möglichkeit, einige große für die allgemeine Spiegelung in einer Szene und zusätzlich kleinere zur Detailierung in bestimmten Bereichen zu verwenden.

Bei der Visualisierung der Schule am Kinkplatz wurden für die allgemeinen Reflektionen im Hof und im Turnsaal kugelförmige Reflection Captures verwendet. Um realistische Ergebnisse zu erzielen, wurden zusätzlich ebenenförmige Captures im Bereich der Glasflächen verwendet. Da die Aktualisierung auch hier in Echtzeit erfolgt, können Auswirkungen gut abgeschätzt werden.

Vegetation mit Speed Tree

Durch die Implementierung von Vegetation kann die Qualität einer Visualisierung stark steigen. Bei Anwendung von Game Engines bietet sich zudem die Möglichkeit, Bäume und Gras einfach und mit geringem Ressourcenverbrauch zu animieren, was dem Realismus weiter zugutekommt. Bei diesem Projekt wurde zum Erstellen der Vegetation die Software Speed Tree von IDV verwendet. Diese ist stark verbreitet und findet sonst in der Film- und Spielebranche Anwendung. Sie ermöglicht die prozedurale Erstellung von Bäumen, Büschen und Gräsern. Außerdem bietet sie die Möglichkeit, die für Game Engines benötigten UV Maps direkt zu erstellen und zu optimieren.

Die einzelnen Komponenten eines Baumes werden durch so genannte „Nodes" dargestellt (Abb. 4.60). Diese bieten zahlreiche unterschiedliche Einstellmöglichkeiten, mit denen sich das Aussehen des jeweiligen Modells individuell anpassen lässt. Werkzeuge, die die Elemente per Zufallsprinzip ändern, sorgen für einen realistischen Eindruck.

Wie auch beim Substance Designer stehen für Speed Tree bei den meisten Engines notwendige Plug-Ins zur Verfügung. Diese steuern den Import/Export. Auch die für die Belichtung in Game Engines notwendigen UV Koordinaten lassen sich direkt in der Software erstellen (Abb. 4.61). Über ein Farbsystem wird deren Auflösung und Qualität dargestellt, was eine einfache Optimierung ermöglicht.

Nach der erfolgreichen Erstellung in Speed Tree wurden die benötigten Bäume per Drag&Drop in die Unreal Engine importiert. Das Material wird automatisch zugewiesen, es erfolgten lediglich kleine Anpassungen im Materialeditor. Um ein realistischeres Ergebnis zu erzielen, wurde ein zweiseitiger Shader verwendet, der die Simulation der leichten Transparenz von Blättern erlaubt. Um die Vegetation in der Szene zu positionieren, wurde das Unreal Paint Tool verwendet (Abb. 4.63 - 4.65). Dieses ermöglicht die schnelle Verteilung von Instanzen. Ähnlich wie bei Materialien weisen diese einen geringen Ressourcenverbrauch auf. Dadurch wird es möglich, die sehr hohe Zahl an Objekten, die bei Vegetation für ein naturgetreues Ergebnis notwendig ist, mit aktueller Hardware gut zu bewältigen. Außerdem können mittels des Paint Tools große Flächen sehr schnell und ohne großen manuellen Aufwand bespielt werden.

Der realitätsnahen Bewegung von Vegetation kommt bei Kamerafahrten und Walkthroughs eine entscheidende Rolle zu. Ohne diese wirken Animationen unnatürlich, starr und leblos. Mit der engineinternen Windquelle bietet sie eine effiziente und einfache Lösung für die schnelle Erstellung von Windeffekten an. Über verschiedene Parameter können die gewünschten Effekte eingestellt werden.

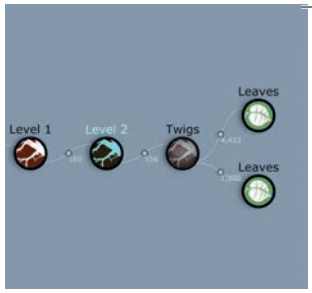


Abb. 4.60: Nodes in Speed Tree


Abb. 4.61: UV Maps in Speed Tree

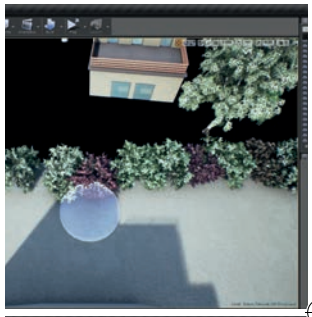


Abb. 4.62: Paint Tool

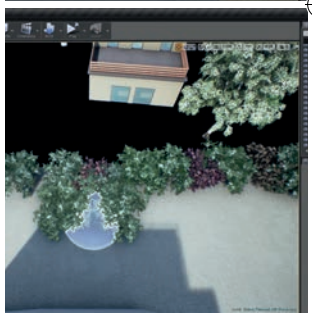


Abb. 4.63: Paint Tool

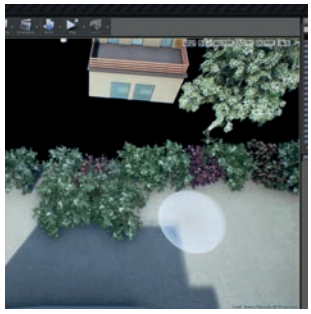


Abb. 4.64: Paint Tool

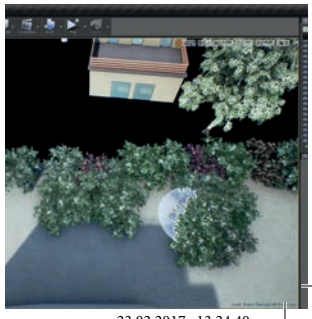
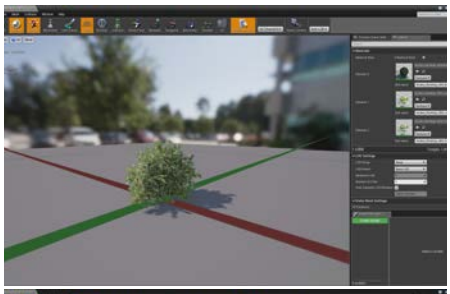


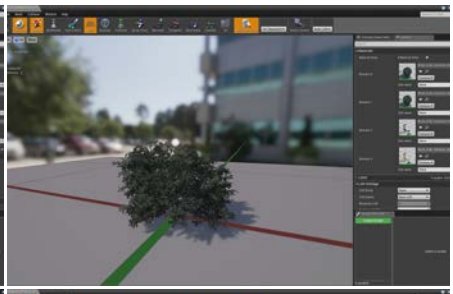
Abb. 4.65: Paint Tool

Virtuelle Repräsentation der tatsächlich vorkommenden Vegetation mittels Speed Tree

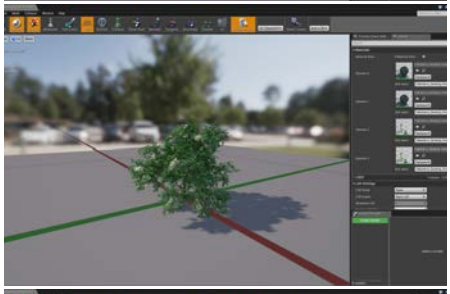
Um die tatsächlich vor Ort vorkommende Vegetation in der Echtzeitvisualisierung zu simulieren, wurde eine Kombination aus mehreren 3D Modellen verwendet. Die Standardmodelle wurden in Speed Tree angepasst um den benötigten Anforderungen zu entsprechen. Die Auflösung der Lightmap für die Lichtberechnung wurde angehoben und es wurde lediglich der höchste Detaillevel (Level of Detail) verwendet, um eine möglichst realitätsgetreue Abbildung zu schaffen.

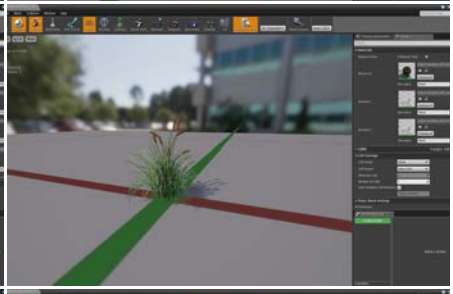
li: Abb. 4.66.1: Busch Var. 1
re: Abb. 4.66.2: Busch Var. 2



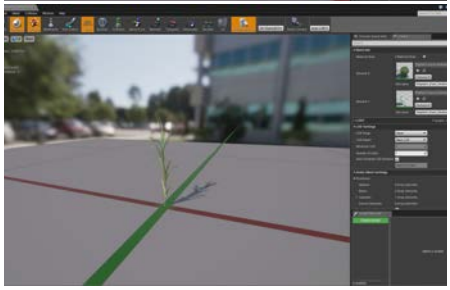


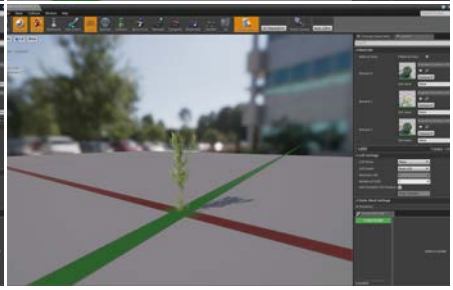
li: Abb. 4.66.3: Busch Var. 3
re: Abb. 4.66.4: Gras Var. 1



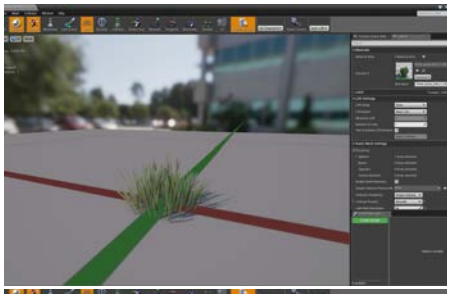


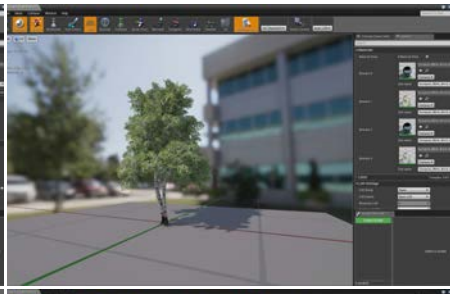
li: Abb. 4.66.5: Gras Var. 2
re: Abb. 4.66.6: Gras Var. 3



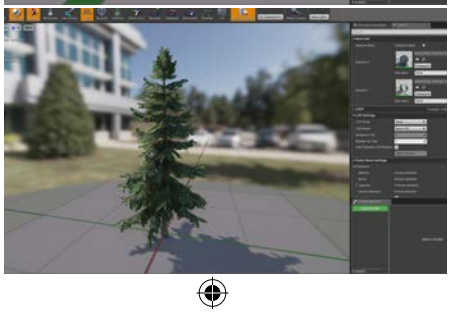


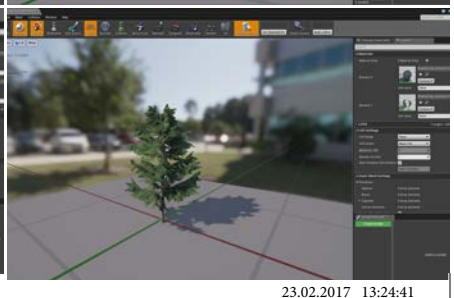
li: Abb. 4.66.7: Gras Var. 4
re: Abb. 4.66.8: Laubbaum

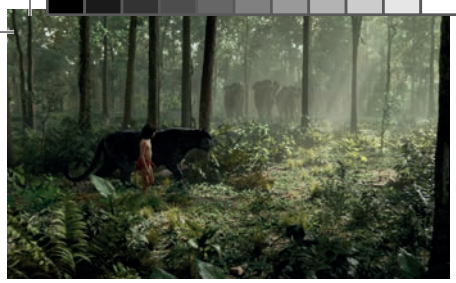
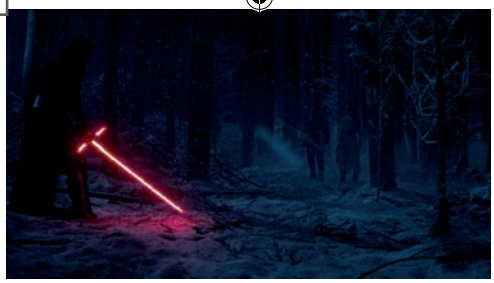




li: Abb. 4.66.9: Nadelholz
re: Abb. 4.66.10: Nadelholz jung







links: Abb. 4.67 - 4.72
Einige Filmszenen, bei denen Bäume, die mittels Speed Tree erstellt wurden, zum Einsatz kommen.

3.7 - Ergebnisse

Abb. 0.10.1

Abb. 0.10

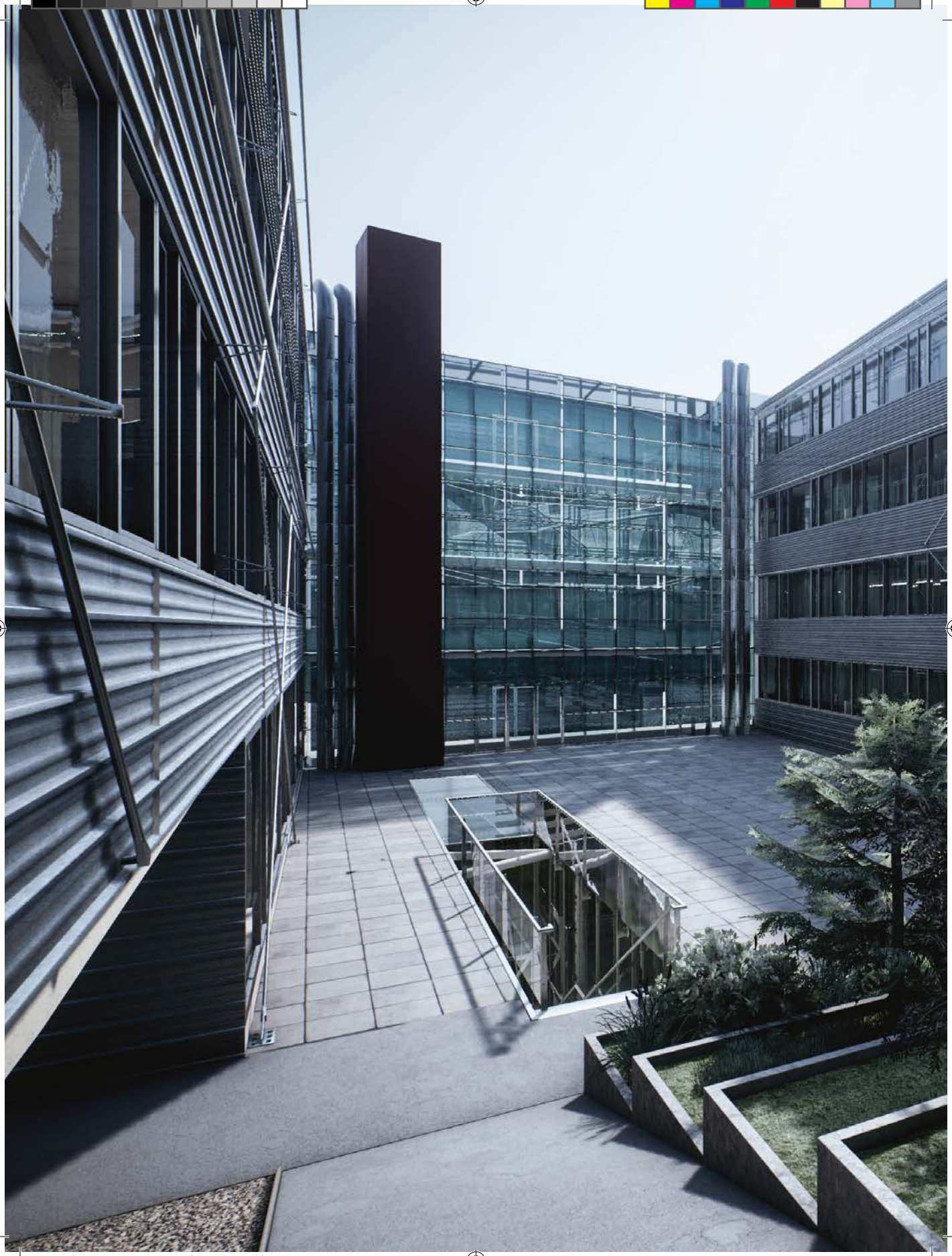
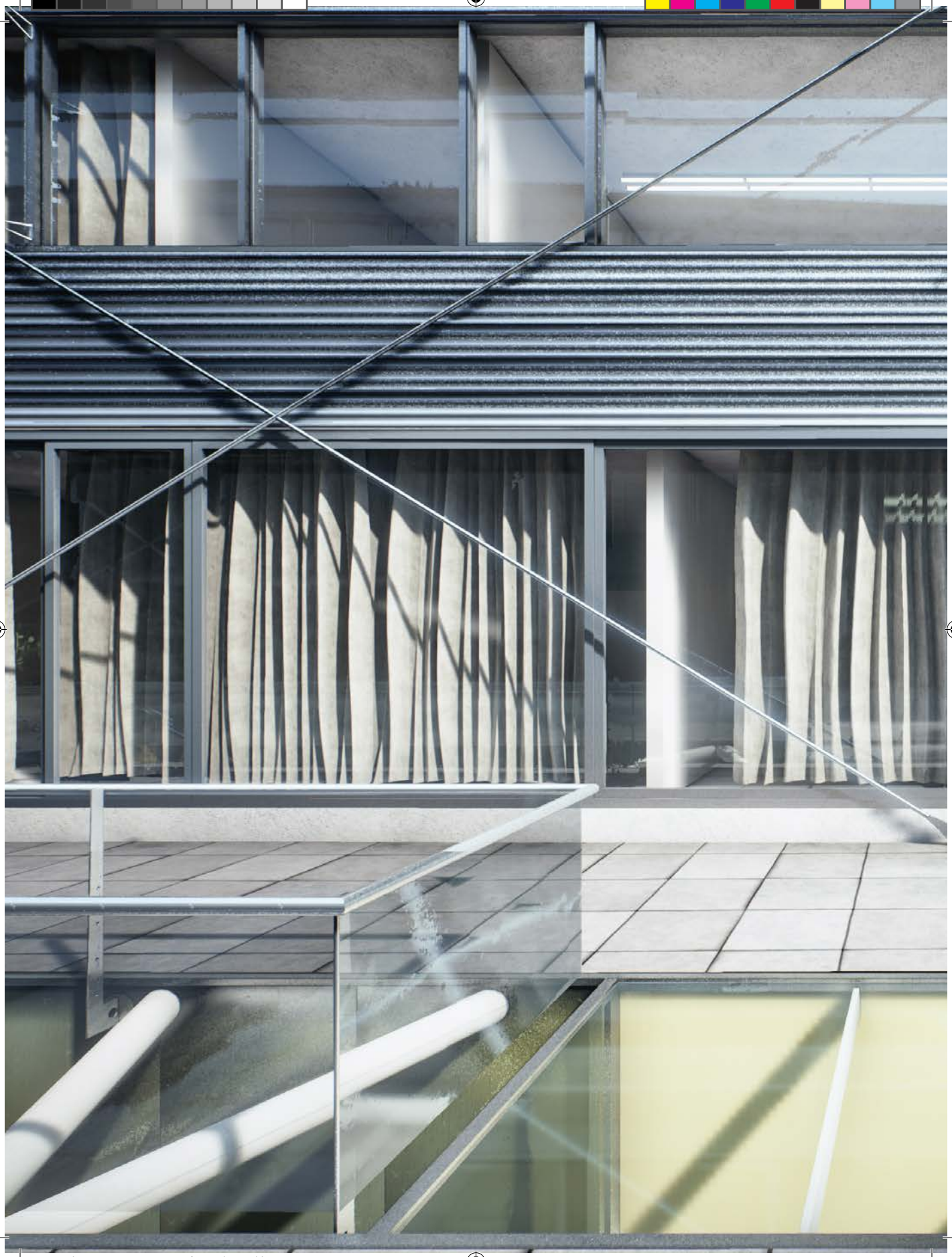


Abb. 0.3



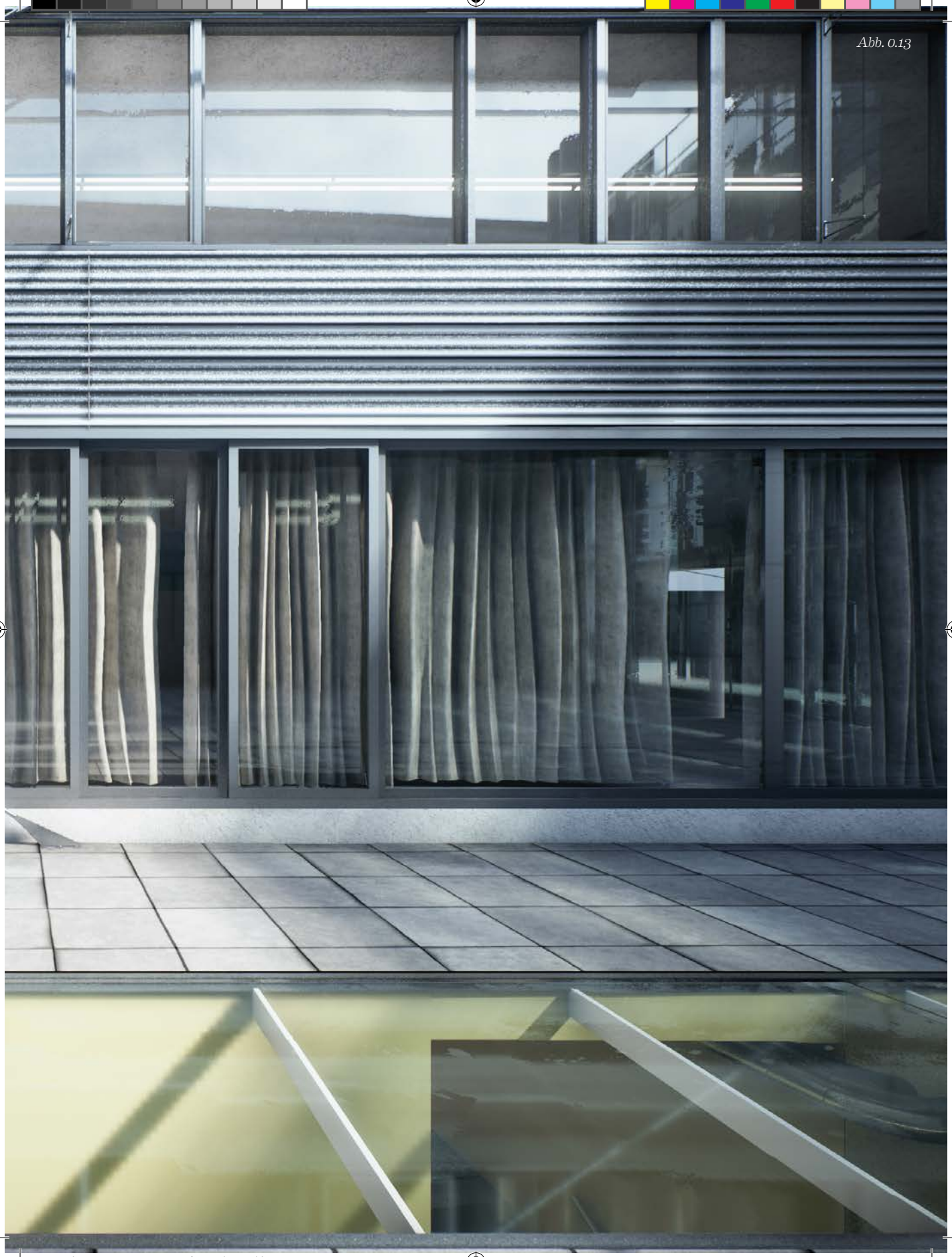
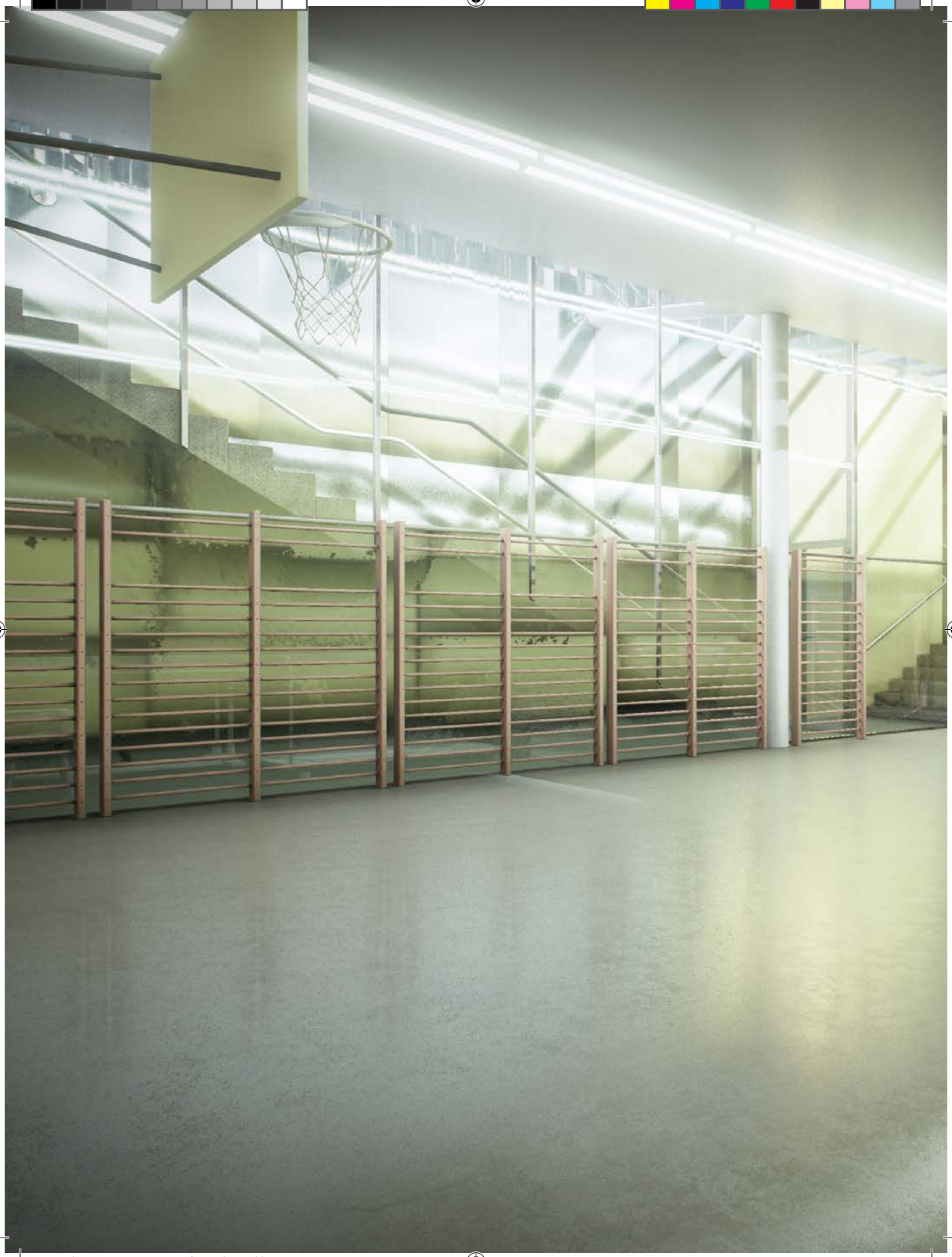


Abb. 0.13



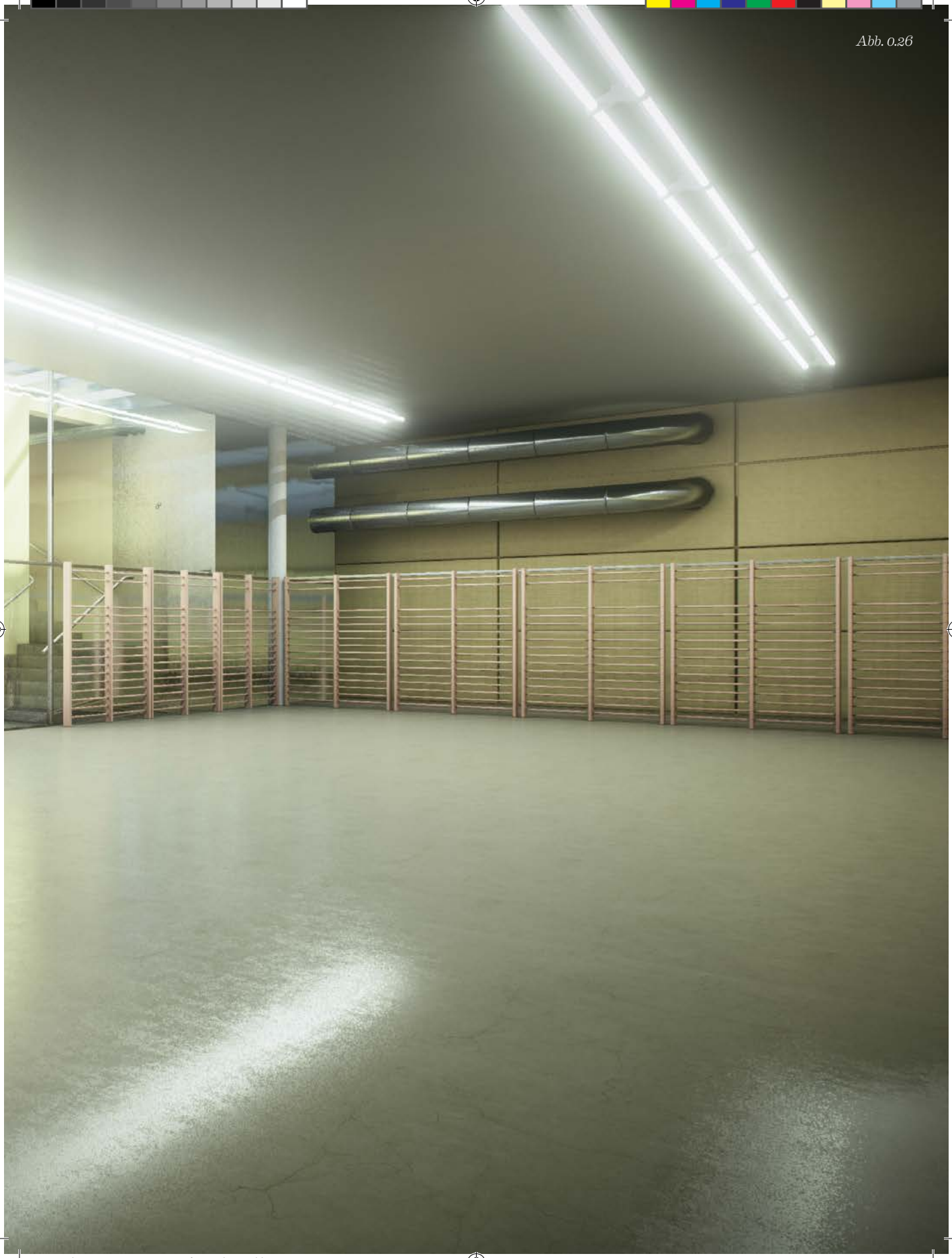
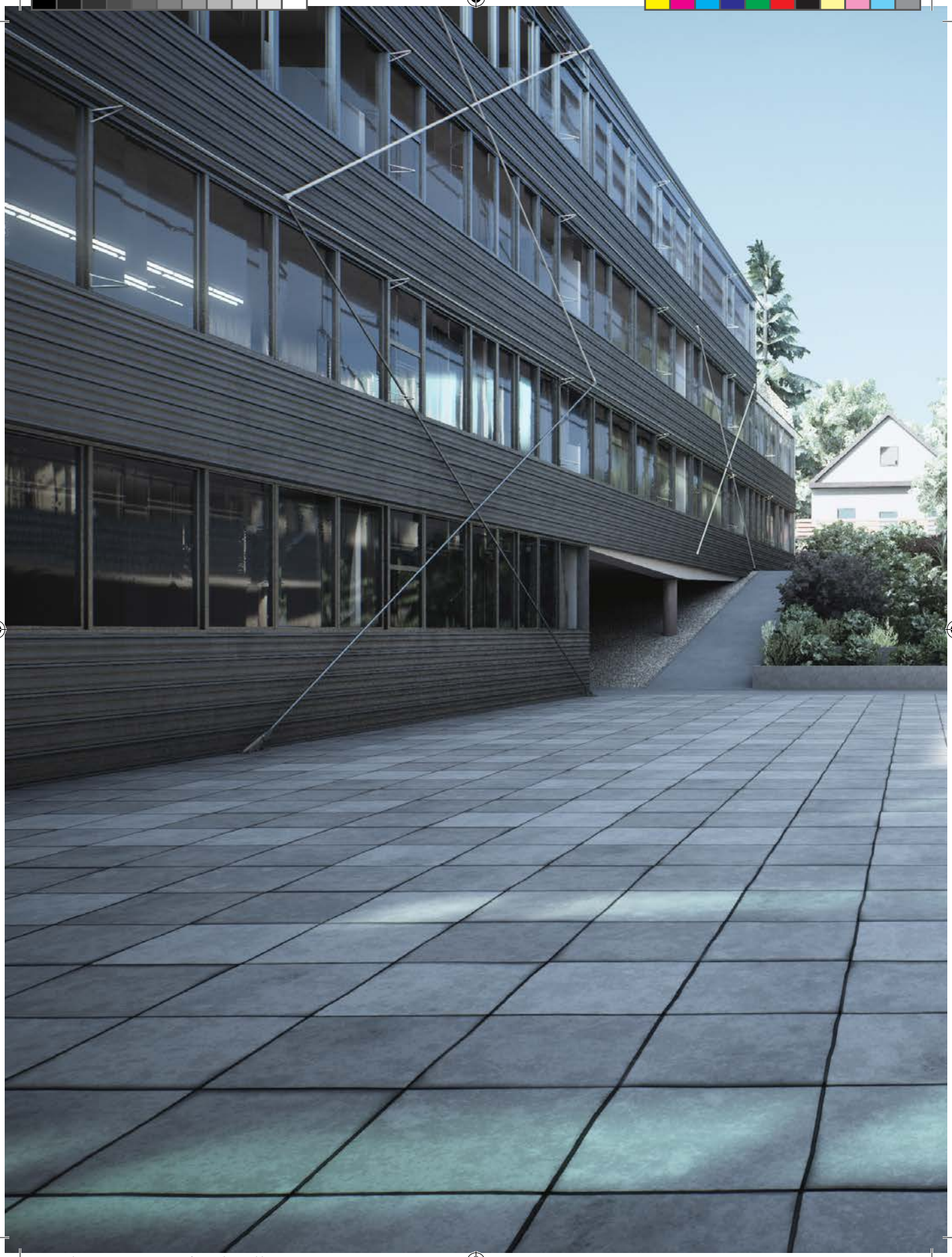


Abb. 0.26



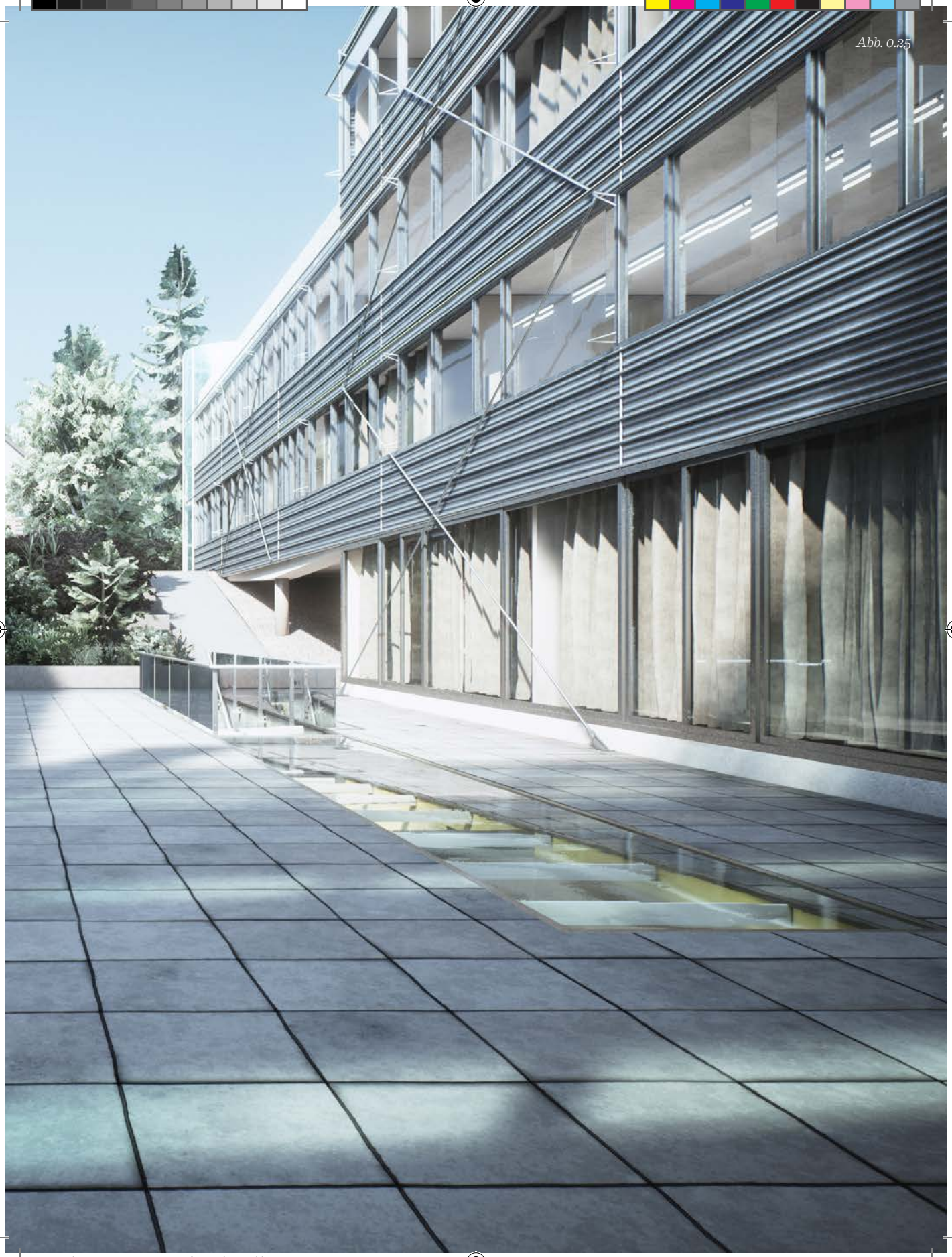
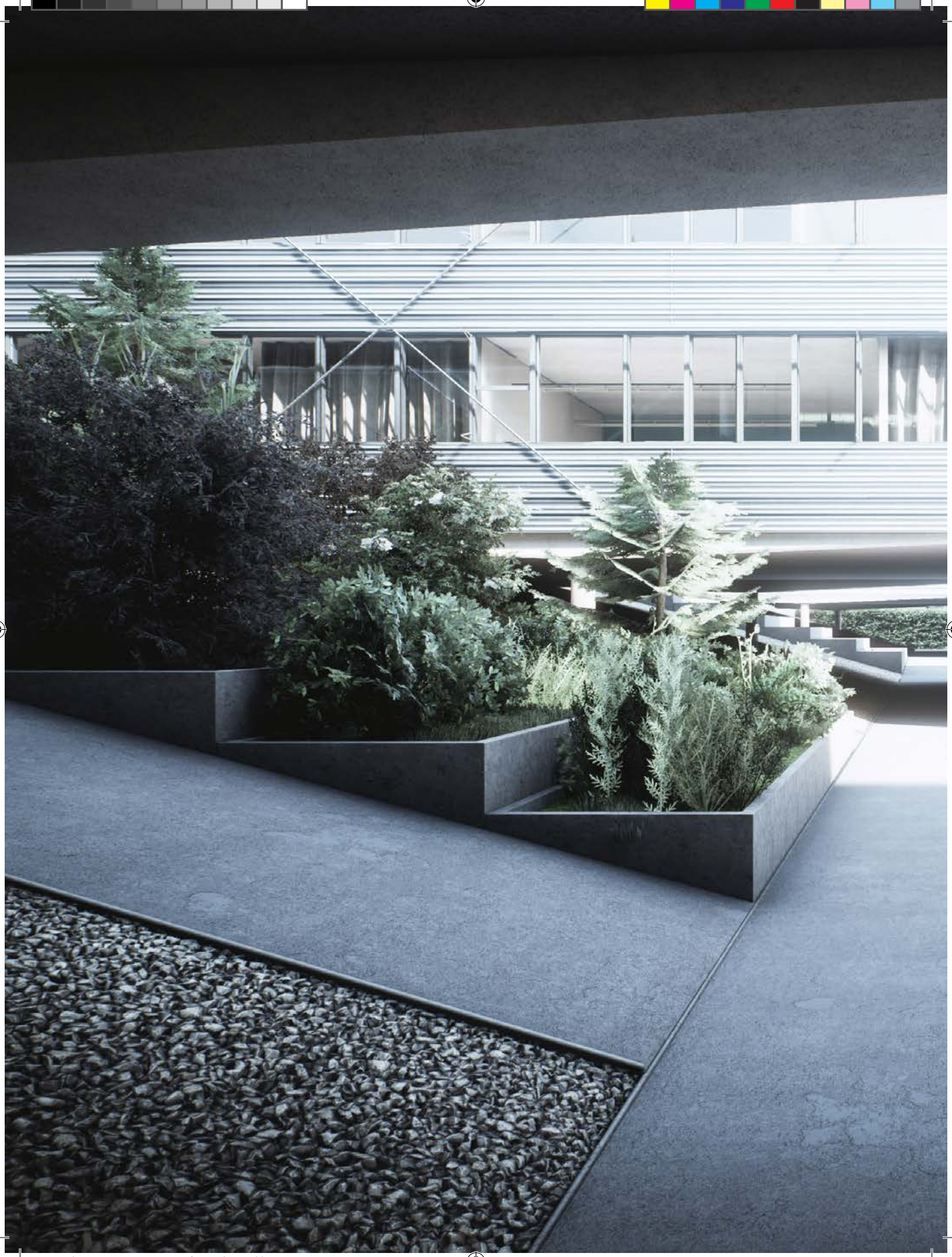


Abb. 0.25



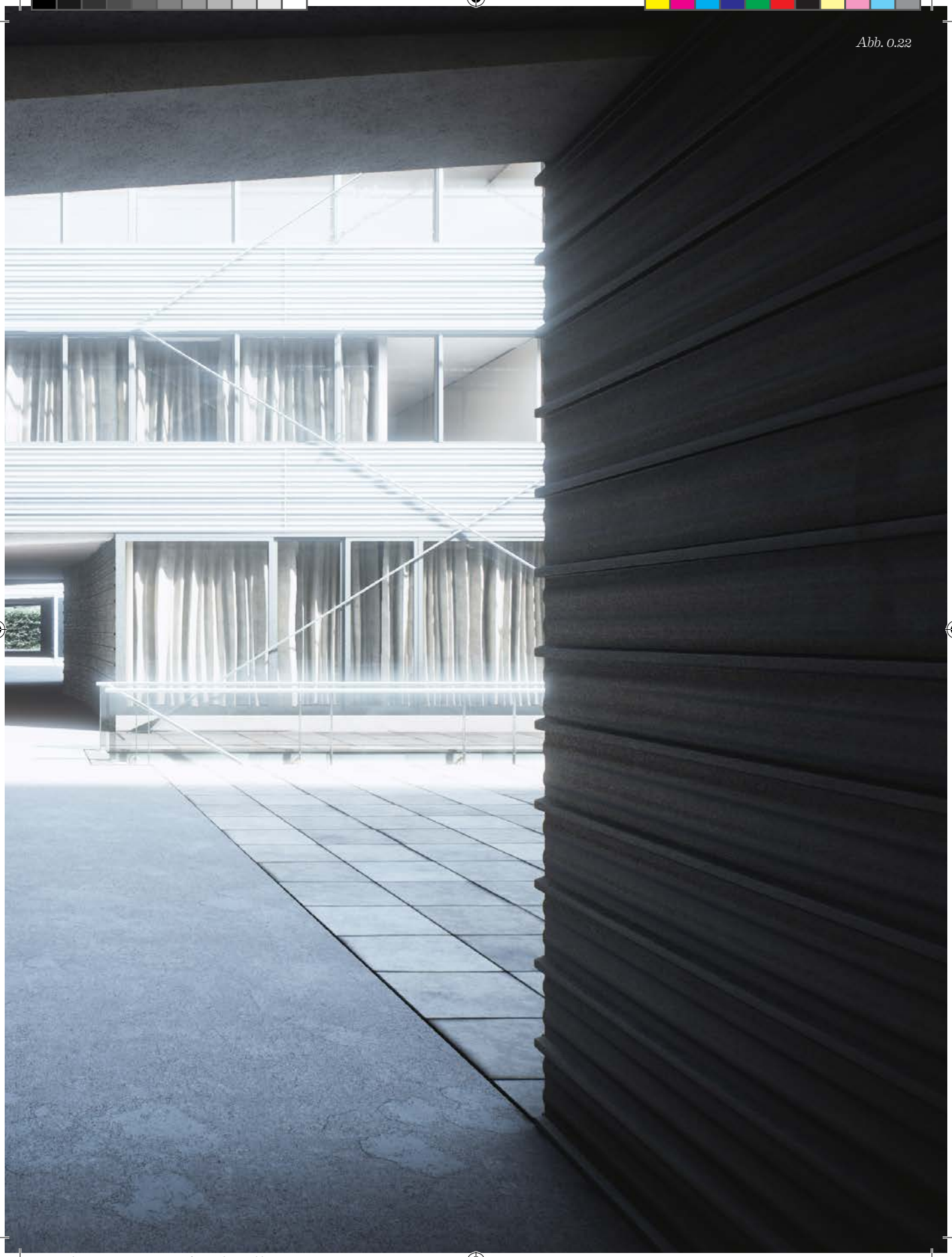


Abb. 0.22

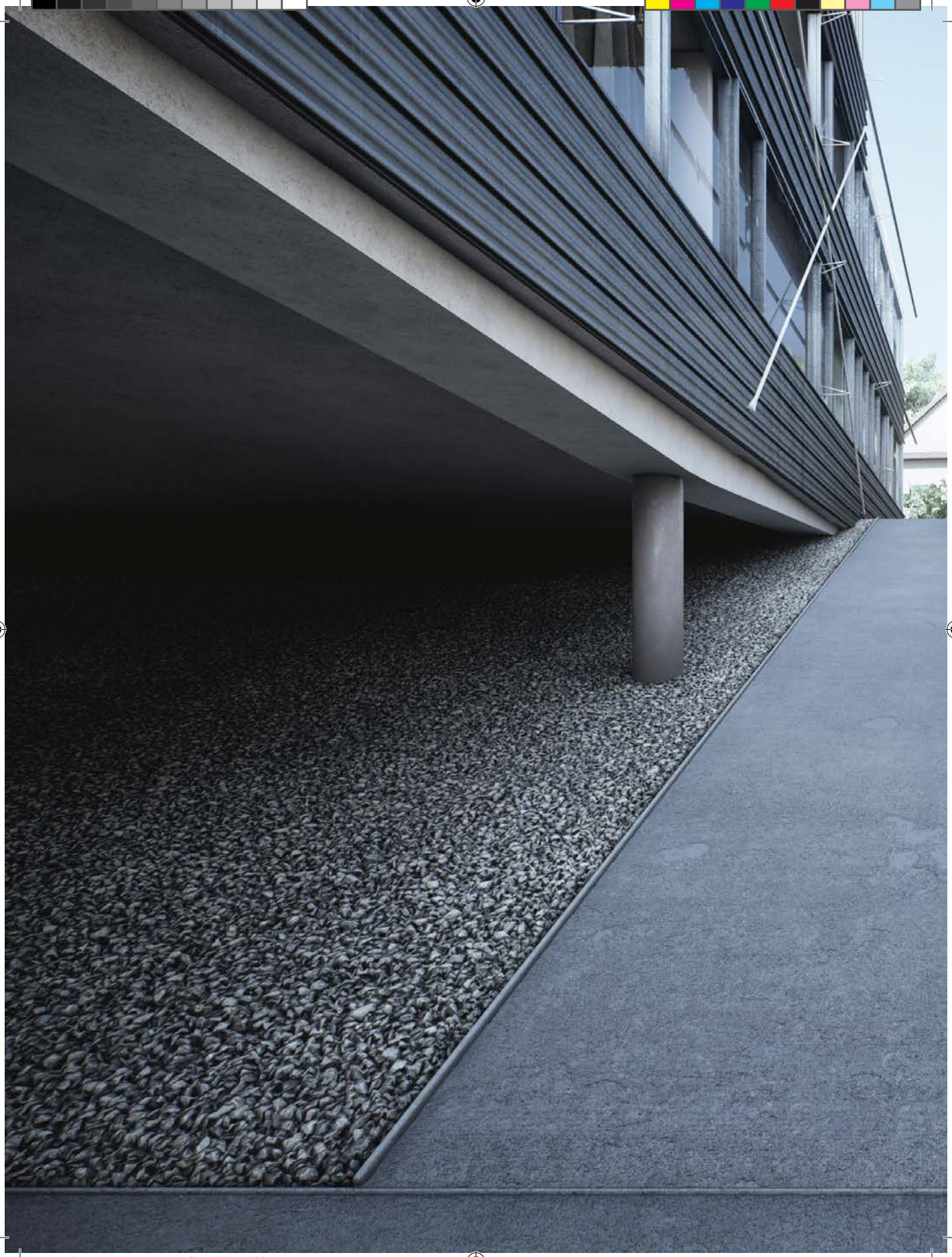


Abb. 6.20

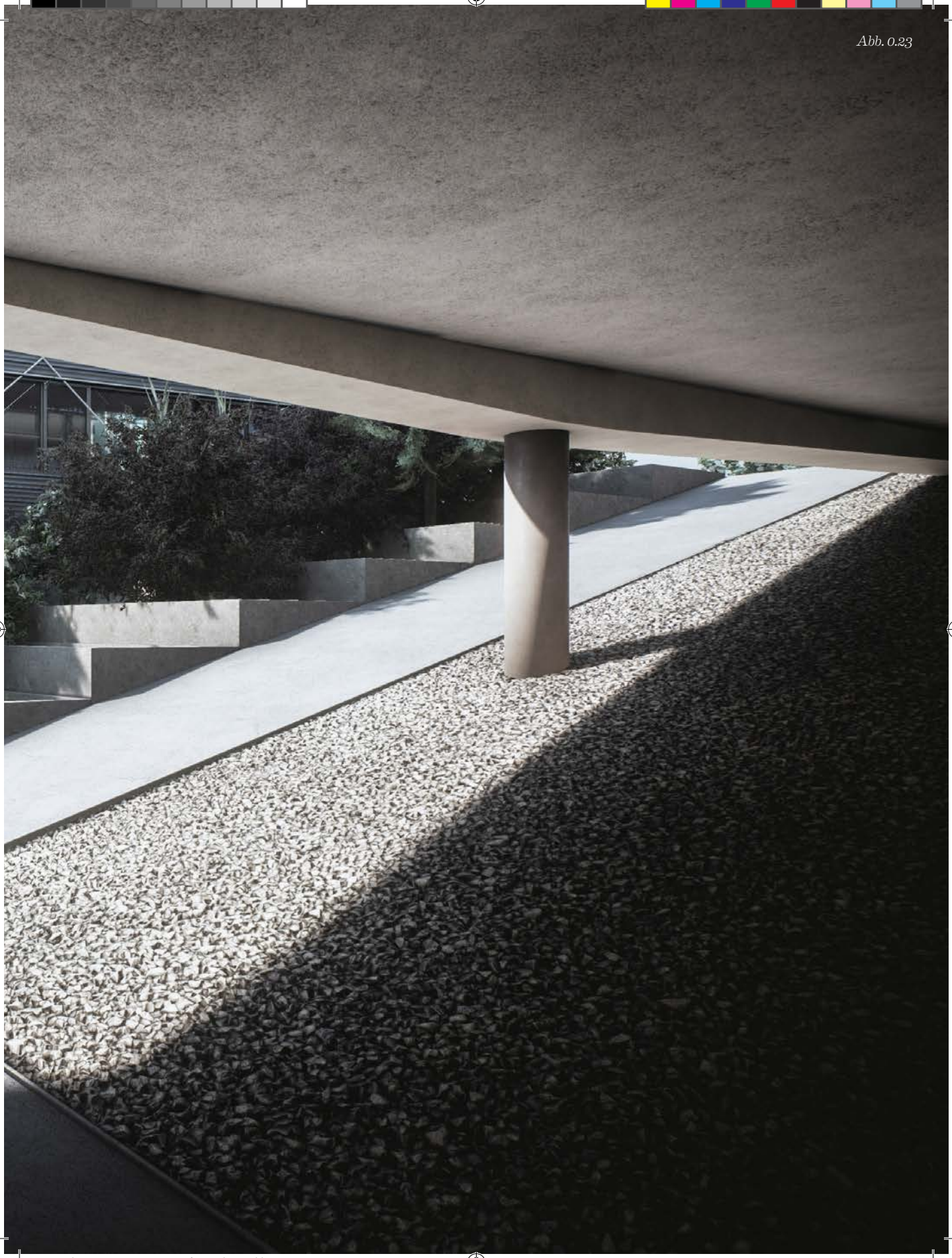


Abb. 0.23

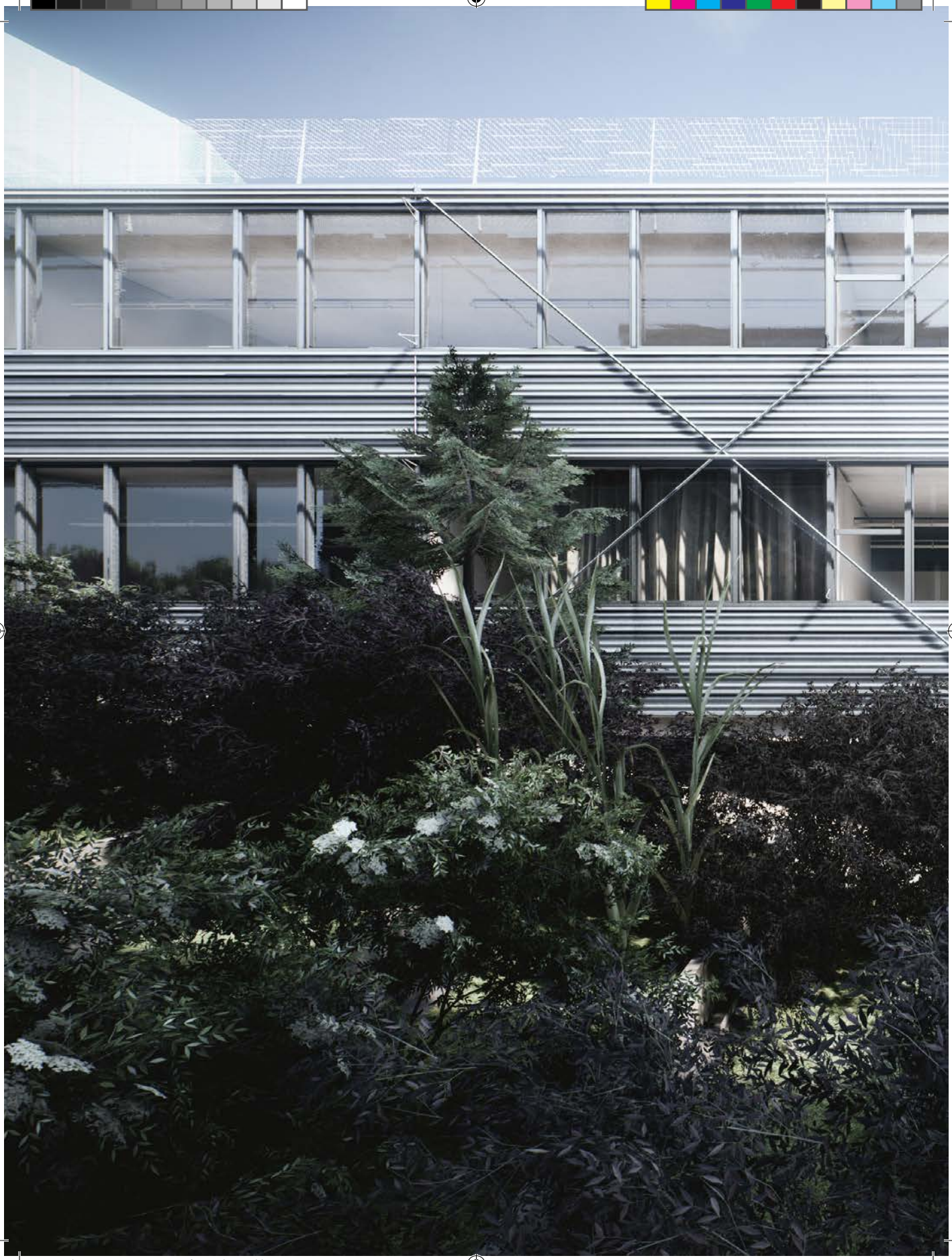
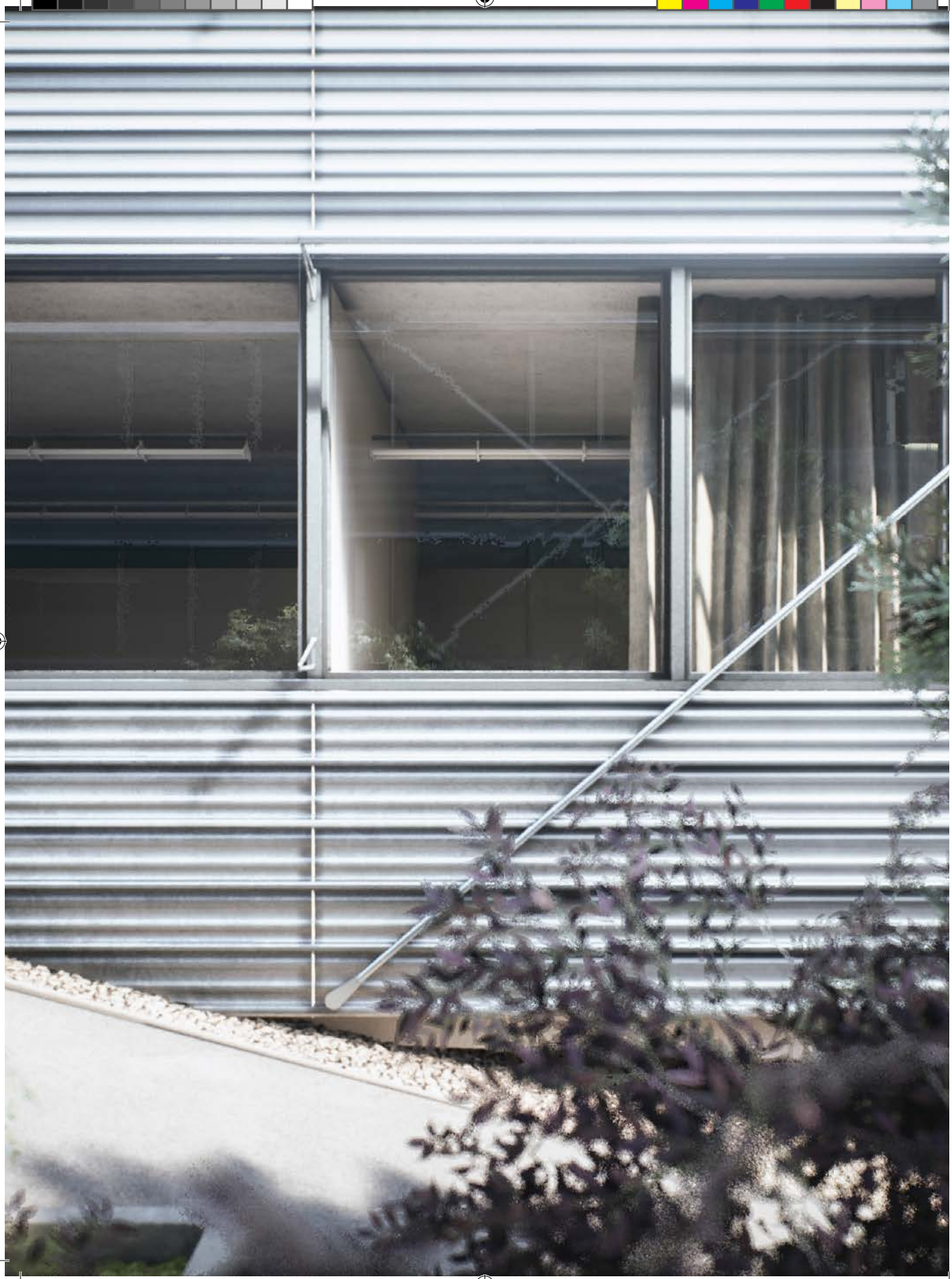


Abb. 9.2:

Abb. 0.24



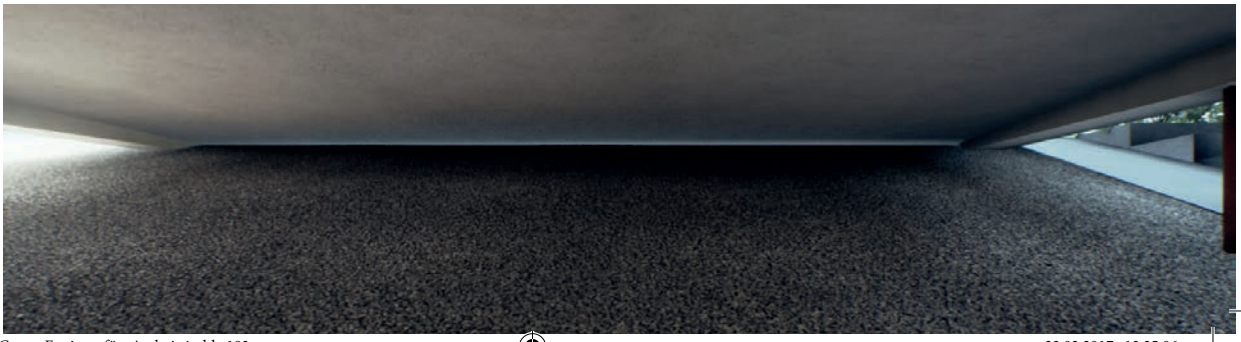
3.8 - Problematische Bereiche

Wie anfangs bereits erwähnt, reicht die Qualität von Game Engines – den Fotorealismus betreffend – bislang noch nicht ganz an jene von bekannten Engines heran, wenn auch die Unterschiede in bestimmten Fällen mittlerweile verschwindend gering sind. Die deutlichsten Unterschiede sind bei der Licht & Schattenberechnung wahrnehmbar. Betrachtet man das fertige Gesamtprojekt, so können Fehler in diesen Bereichen oft durch qualitativ hochwertige Modelle und Materialien, oder geschickt gesteuerte Kamerafahrten überblendet werden. Im Sinne einer objektiven Analyse wollen wir die problematischen Bereiche hier isoliert betrachten:

Light Leaking

„Leaking" bezeichnet die unerwünschte Übertragung von Licht auf Geometrie in bestimmten Bereichen des Modells. Es tritt häufig an Schnittstellen zwischen zwei Objekten auf, wenn sich auf einer Seite eine starke Lichtquelle (z.B. Sonnenlicht) befindet. Ursachen für dieses Phänomen sind meist zu niedrige Einstellung der Lightmass GI Berechnung. Der Algorithmus hat für die anspruchsvollen Bereiche zu wenig Samples zur Verfügung und weiß deshalb nicht, wo genau die Grenze zwischen belichteter und nicht-belichteter Geometrie verläuft. Das Ergebnis sind verwaschene Schatten und Lichtflecken an Stellen, die eigentlich im Dunkeln liegen sollten. Oft schafft eine Erhöhung der Berechnungsqualität in solchen Fällen Abhilfe. Außerdem sollte darauf geachtet werden, dass sich die Geometrie an der betroffenen Stelle auch tatsächlich (bzw. stark genug) überschneidet. Tritt „Leaking" beispielsweise an Wandecken innerhalb eines Raumes auf, kann eine Erhöhung der Wandstärken das Problem beseitigen.

Abb. 4.73: Am Schnittpunkt der zwei Elemente ist „Leaking" erkennbar



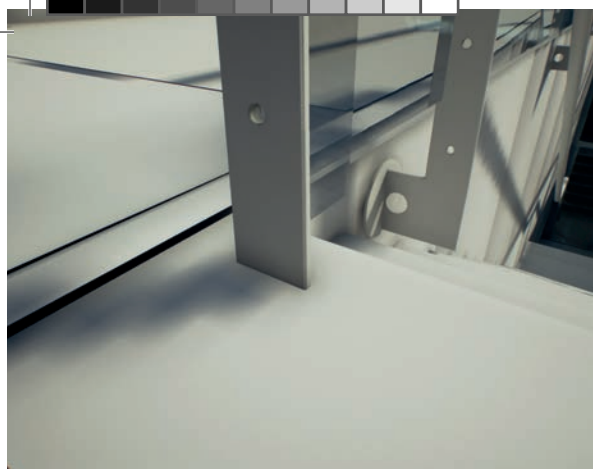


Abb. 4.74: schlecht berechnete Ambient Occlusion

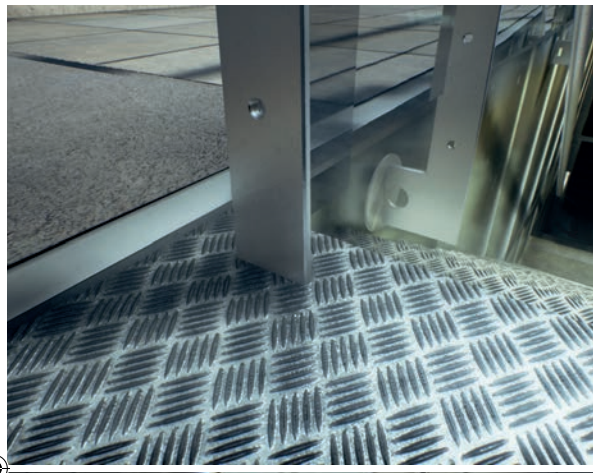


Abb. 4.75

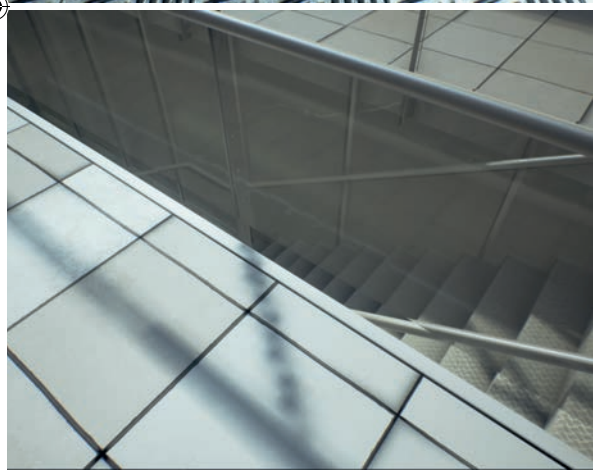


Abb. 4.76: geringe Schattenauflösung

Abb. 4.77: geringe Schattenauflösung

Kontaktschatten, Umgebungsverdeckung, Ambient Occlusion

Unter Ambient Occlusion wird die Berechnung von Kontaktschatten zwischen zwei sich berührenden Objekten verstanden. Ohne diesen Effekt erscheinen dreidimensionale Körper oft falsch belichtet, oder sie „schweben" im Raum. Abhängig von der Lichtsituation und Geometrie kommt es bei der Lichtberechnung in der Unreal Engine fallweise vor, dass die Ambient Occlusion zu schwach oder schlecht berechnet wird (Abb. 4.74). Dieser „Fehler" fällt vor allem in frühen Phasen verstärkt auf, wenn noch einheitliche weiße, statt komplexer Materialien, verwendet werden. Er gehört jedenfalls zu jenen, die gut durch die Anwendung qualitativ hochwertiger Shader kaschiert werden können (Abb. 4.75).

Schattenauflösung

Die Licht & Schattenberechnung ist in der Unreal Engine maßgeblich von der Auflösung der UV-Maps abhängig. Niedrige Auflösungen sorgen einerseits für schnelle Renderzeiten, erzeugen andererseits aber minderwertige Ergebnisse. Dies wird insbesondere bei scharfen Schattenkanten sichtbar – sie erscheinen abgetreppt und „verpixelt", wenn die UV-Map zu klein gewählt wird (Abb. 4.76 + 4.77). Stark betroffen sind in dieser Hinsicht Bereiche mit besonders detailreichen Schatten. Hier schaffen hohe Lightmap Auflösungen Abhilfe, jedoch gilt es immer zwischen Qualität und Ressourcenverbrauch abzuwägen. Besonders hohe Auflösungen sollten nur auf großen Flächen und in Bereichen gewählt werden, welche später auch deutlich sichtbar sind.

Spiegelungen

Während in anderen Renderern die Spiegelungen automatisch anhand des Umgebungslichts und der Materialeigenschaften berechnet werden, muss in der Unreal Engine die Einstellung mittels Reflection Captures manuell erfolgen. Das bedeutet für den Visualisierer zusätzlichen Zeitaufwand – vor allem auch deshalb, weil auf diese Art erstellte Reflektionen immer nur eine Annäherung an die Realität darstellen und dieser nie vollkommen entsprechen. Dadurch ist es manchmal notwendig mehrere Anordnungen zu testen, um ein zufriedenstellendes und naturgetreues Ergebnis zu erhalten. Dies wird, dank der Möglichkeit planare Reflektionen zu erstellen (seit Engineversion 4.11), stark erleichtert. Zuvor war es lediglich möglich, die Spiegelungen mittels einer Kugel- oder Würfelform anzunähern – was gerade bei ebenen Flächen, wie Fensterverglasungen, oft zu unnatürlichen Resultaten führte. Die Spiegelungen in der Unreal Engine funktionieren gut und es sind sehr realitätsgetreue Ergebnisse möglich, jedoch muss der zusätzliche Zeitaufwand für die Einstellung beim Erstellen von Visualisierungen bedacht werden.

Abb. 0.28

4 - Zusammenfassung & Ausblick

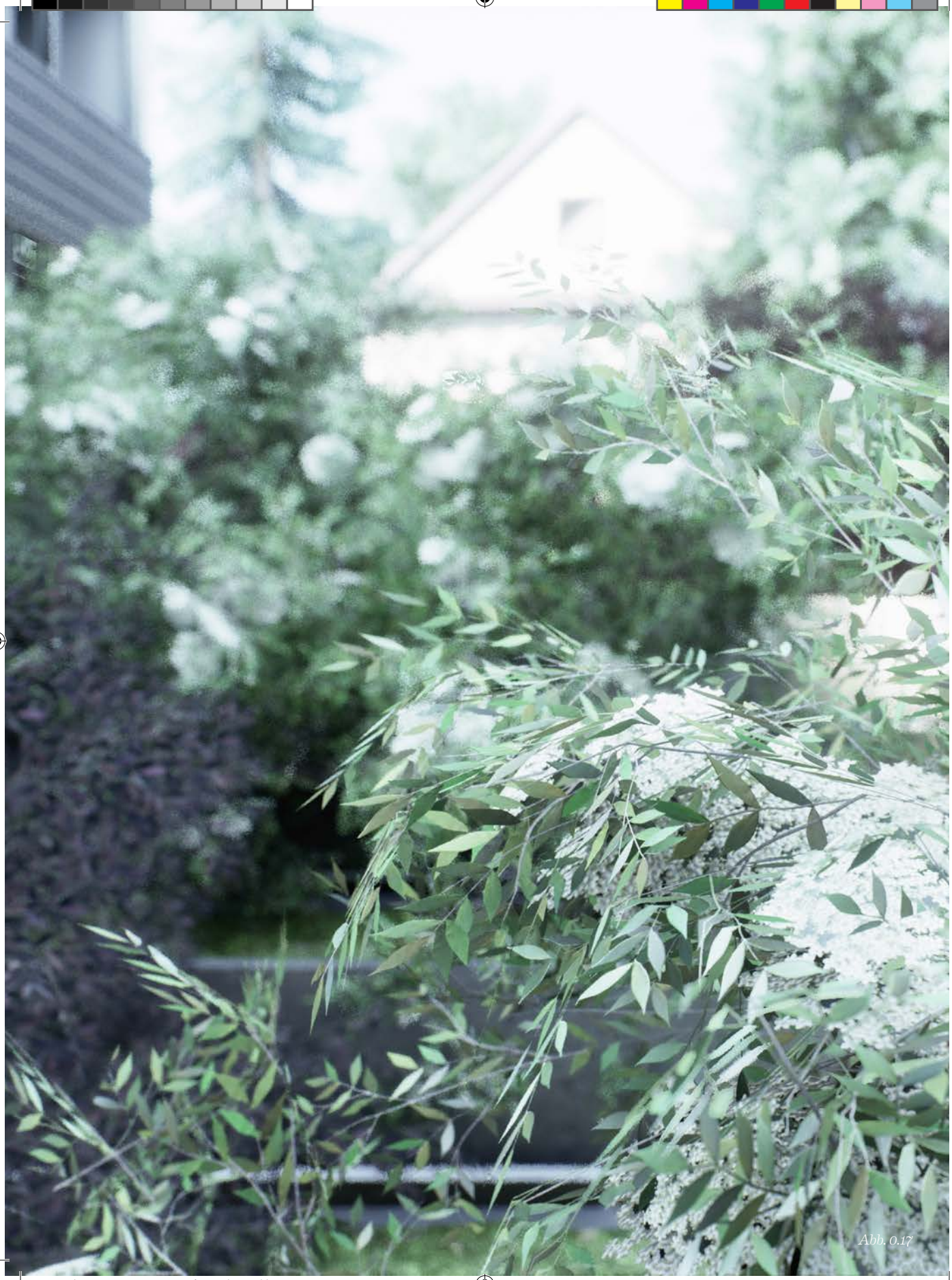


Abb. 0.17

4.1 Zusammenfassung

Die Schule von Helmut Richter wurde gewählt, um die Leistungsfähigkeit von Game Engines für Architekturvisualisierungen an einem konkreten Objekt testen zu können. Die Darstellung eines bereits gebauten Projektes erlaubt den Vergleich zwischen Fotos des realen Gebäudes und Visualisierungen des virtuellen Modells. Dieser belegt, dass aktuelle Game Engines in der Lage sind fotorealistische Ergebnisse zu erzielen und diese in Echtzeit darzustellen. Sowohl die Simulation der tatsächlichen Lichtverhältnisse als auch der vorkommenden Materialien und Vegetation lassen sich gut lösen. Mit entsprechender Hardware ist eine flüssige Echtzeitdarstellung von komplexen Szenen, wie sie bei der Visualisierung von Architektur vorkommen, unproblematisch.

Die Erfahrung aus der Projektarbeit zeigt aber auch, dass verglichen mit klassischen Rendermethoden ein gewisser Mehraufwand im Workflow entsteht, der im Hinblick auf das gewünschte Ergebnis entsprechend bedacht werden muss. Gleichwohl bestätigt der Versuch, dass die Anwendung von Game Engines für Architekturvisualisierungen bislang unerreichte Möglichkeiten und großes Potential für diesen Projektbereich birgt. Wurden Visualisierungen bislang hauptsächlich nach erfolgter Planung zu Präsentationszwecken verwendet, ist es mit den nun zur Verfügung stehenden Werkzeugen effektiver möglich, die Vorteile von computergenerierten Bildern auch in anderen Phasen zu nutzen. Die Darstellung in Echtzeit ermöglicht eine weitaus schnellere und intuitivere Arbeitsweise, als sie mit klassischen Render Engines möglich wäre.

Game Engines sind insbesondere dann gut geeignet. wenn der Fokus auf einer schnellen, anpassungsfähigen Projektvisualisierung liegt. Die unterschiedlichen Arten der Interaktion lassen einen äußerst flexiblen Umgang mit der Szene zu, wenn es darum geht Lichtstimmungen, Material- oder Modelländerungen zu testen. Varianten können somit um ein Vielfaches schneller ausprobiert und überprüft werden, was Game Engines insbesondere für die Nutzung in frühen Projektphasen interessant macht. Durch die Möglichkeit das gesamte Projekt in Echtzeit zu erkunden, kann im Vergleich zu normalen Visualisierungen eine deutlich gesteigerte Raumwahrnehmung erzielt werden. Raumfolgen und die Bewegung innerhalb eines Projektes – sowohl in der Horizontalen als auch Vertikalen – können überzeugend simuliert werden. Dieser Effekt kann durch die Nutzung von Virtual Reality Technologien weiter gesteigert werden.

Hinsichtlich der visuell erreichbaren Qualität sind Game Engines sehr leistungsfähig und bewegen sich etwa auf dem Niveau etablierter Renderer. Auch wenn in einigen Bereichen nicht immer vollkommener Fotorealismus erzielt werden kann, wirken die Ergebnisse aufgrund des überzeugenden Gesamtbildes durchaus „fotorealistisch“.

Bedingt durch die vergleichsweise junge Technologie und rasch voranschreitende Entwicklung von Game Engines, erscheint der Zugang noch relativ schwierig. Dies ist dem noch geringen Maß an Informationen und Hilfestellungen geschuldet, welche derzeit in Form von Literatur oder Onlinetutorials zur Verfügung stehen. Funktionen müssen von Nutzern manchmal mittels Trial & Error Prinzip erprobt werden, um das gewünschte Ergebnis zu erreichen. Oft gestaltet es sich auch schwierig die ursprüngliche Herkunft dieser Software in der visuellen Ästhetik der fertigen Projekte zu verdecken und hoher Arbeitsaufwand ist notwendig, um mit Game Engines erstellte Visualisierungen nicht wie Computerspiele wirken zu lassen.

Der Workflow ist aufgrund der Tatsache, dass Game Engines erst seit kurzem für Architekturvisualisierungen verwendet werden, nicht für diese ausgelegt und somit komplizierter als bei klassischen Renderern. Ein Rendern innerhalb der 3D Applikation, wie es bei anderen Render Engines der Fall ist, ist derzeit noch nicht möglich. Zusätzliche Schritte sind notwendig, bevor aus einem 3D Modell eine fertige Visualisierung erstellt werden kann. Diese können zwar mit unterschiedlichen Skripten vereinfacht und beschleunigt werden, bedeuten aber trotzdem einen erhöhten Zeitaufwand und bieten mehr Raum für Fehler. Aufgrund der Tatsache, dass Game Engines eine eigenständige Anwendung darstellen, müssen Materialzuweisungen beziehungsweise Einstellungen nach dem Import neu durchgeführt werden. Der Umstand, dass durch die Visualisierung in Echtzeit viel mehr vom gesamten Projekt sichtbar ist, als es bei Standbildern oder vorher aufgenommenen Animationen der Fall wäre, geht ebenfalls mit einem erhöhten Aufwand einher, da alle relevanten Bereiche modelliert, texturiert und belichtet werden müssen.

Aufgrund der genannten Punkte hinsichtlich des komplexeren Workflows erscheint der erforderliche Aufwand bei der Architekturvisualisierung mittels Game Engines zunächst höher als bei der Verwendung von klassischen Render Engines. Durch die Vorteile von Game Engines wird dieser aber im späteren Arbeitsverlauf um ein Vielfaches wieder wettgemacht. Voraussetzung dafür ist jedoch ein entsprechender Bedarf bei der jeweiligen Aufgabe. Für die Erstellung einiger zuvor definierter Bilder bei einem mehr oder weniger fertig entwickelten Projekt wäre die Anwendung von Game Engines derzeit aus ökologischen Gesichtspunkten noch ungeeignet. Für solche Anforderungen sind viele etablierte Renderer vorhanden, deren erprobte Workflows eine problemlose und schnelle Arbeitsweise erlauben. Außerdem würde solch ein Einsatz das Potential von Game Engines vernachlässigen.

Um die Möglichkeiten der Software auszuschöpfen sind insbesondere Projekte geeignet, die sich in frühen Entwicklungsphasen befinden und bei denen viele, weitreichende Änderungen in rascher Folge stattfinden. Auch solche Aufgaben, die nach einem hohen Maß an Interaktivität oder nach bewegten Bildern (zum Beispiel für Präsentationen) verlangen, können mit Game Engines gut gelöst werden. Ab einer gewissen Menge an zu produzierenden Visualisierungen kann es wirtschaftlich rentabel sein, Game Engines auch in Projekte miteinzubeziehen, die im ersten Moment besser für klassische Engines geeignet erscheinen. Viele Einzelbilder aus verschiedenen Perspektiven von ein und demselben 3D Modell können in kurzer Zeit erstellt werden, um den Anforderungen gemäß Menge und Ressourcenverbrauch gerecht zu werden.

Trotz der zuvor genannten Herausforderungen bei der Arbeit mit Game Engines sind diese für gewisse Aufgaben der Architekturvisualisierung von Vorteil. Die Eignung ist jedoch stark vom Projekt abhängig und muss im Einzelfall abgeschätzt werden.

4.2 Ausblick

Verglichen mit etablierten Renderern bieten Game Engines große Vorteile und neue Möglichkeiten auf dem Gebiet der Architekturvisualisierung. Hinsichtlich der visuellen Darstellung bieten sie einen ähnlich hohen Qualitätsstandard, doch ist besonders beim Workflow noch Verbesserungspotential vorhanden.

Die Optimierung der Schnittstellen zwischen 3D Applikationen und Game Engines und die Entwicklung eines allgemein anerkannten und erprobten Exportworkflows würde die Arbeit stark vereinfachen und beschleunigen. Bei der Verwendung von einheitlichen Materialbibliotheken (zum Beispiel Substance Designer) sowohl innerhalb der 3D Modelliersoftware als auch der verwendeten Game Engine, könnte die Schnittstelle die Materialübernahme steuern, um dem Nutzer die erneute Materialzuweisung zu ersparen. Auch die Arbeit innerhalb der Game Engines bedarf weiteren Anpassungen, um das aus anderen Engines gewohnte Niveau zu erreichen. Problematische Bereiche stellen derzeit vor allem die Licht- & Schattenberechnung dar. Fallweise werden physikalisch nicht korrekte Ergebnisse produziert, welche vom Visualisierer schwer vorhersehbar sind und die Qualität des finalen Produktes beeinflussen können. Auch die Einstellung der Spiegelungen erfordert einen vergleichsweise hohen Zeitaufwand – eine automatische Berechnung wäre hier von Vorteil.

Es muss jedoch bedacht werden, dass Game Engines erst seit kurzer Zeit zur Erstellung von Architekturvisualisierungen angewandt werden und einer ständigen Entwicklung in diesem Bereich unterliegen. Sie werden auch verstärkt in der Immobilien- und Automobilbranche verwendet. Die Vorteile einer Echtzeitdarstellung lassen sich natürlich auch hier gut ausnutzen – geht es zum Beispiel um die Konfiguration von Kraftfahrzeugen oder die Ausbauvarianten von Innenräumen. Unabhängig vom Anwendungsgebiet lässt sich das erlebte Realitätsgefühl durch die Anwendung von zusätzlichen Technologien weiter erhöhen. Hierzu zählen insbesondere Virtual und Augmented Reality.

Virtual Reality („virtuelle Realität“)

Die Anwendung von Virtual Reality Headsets erlaubt das Empfinden von computergenerierten Szenen zu steigern. Dabei handelt es sich um Brillen, die dem Nutzer das Gefühl vermitteln, sich mitten in der Szene zu befinden, statt diese am Monitor zu betrachten. Mittels eingebauter Sensoren werden Kopfbewegungen in Kamerabewegung in der virtuellen Welt umgesetzt. Zusätzlich ermöglichen zwei händisch bediente Controller weitere Interaktionen innerhalb der virtuellen Realität.

Abb. 5.7: Anwendung eines Virtual Reality (VR) Headset zur Modellierung eines Levels.

Diese Technologie findet zurzeit hauptsächlich Anwendung in der Spielindustrie (Abb. 5.7). In Kombination mit Architekturvisualisierungen sind aber Anwendungsformen im Bereich der Architektur durchaus denkbar. Geht es um die Erkundung oder die Präsentation von Entwürfen, stellen Virtual Reality Headsets einen hohen Mehrwert dar. Das Erstellen von 3D Modellen kann mit ihnen ebenso erleichtert werden, wie die Überprüfung von Raumsituationen oder Lichtstimmungen. Wenngleich diese aus dem Gebiet des Level-Designs bei Spielen stammen, wurden erste Anwendungsbeispiele in dieser Hinsicht bereits getestet und vorgestellt. Eine Adaption solcher Techniken für andere Industrien erscheint durchaus plausibel.

Augmented Reality („erweiterte Realität")

Im Allgemeinen wird unter diesem Begriff die computergestützte Erweiterung der Realitätswahrnehmung verstanden. Es kann sich dabei um Zusatzinformationen, welche auf einem Display (z.B. Smartphone) über die tatsächliche reale Umgebung gelegt werden, handeln (Abb. 5.1).

Abb. 5.1

Abb. 5.2

Abb. 5.3

Abb. 5.4

Abb. 5.5

Abb. 5.1 - 5.5: Augmented Reality

Umgekehrt gedacht bietet Augmented Reality in Kombination mit Game Engines die Möglichkeit virtuelle Szenen als Basis zu nutzen, um Objekte aus der realen Welt darin zu platzieren. Ähnlich bekannten Technologien aus der Filmindustrie (Bluebox, Greenbox) können somit Menschen in beliebigen Umgebungen fotorealistisch dargestellt werden (Abb. 5.2). Im Unterschied zu bisherigen Techniken kann die virtuelle Szene dank der Möglichkeiten von Game Engines jedoch in Echtzeit verändert werden (Abb. 5.3). Gleichzeitig reagieren sowohl die umgebende Szene, als auch das reale Objekt auf Änderungen der Lichtsituation, Schatten oder Spiegelungen (Abb. 5.4 + 5.5) [vgl. URL Virtual Reality].

Eine Sonderform der Augmented Reality bietet Microsofts Hololens. Grundsätzlich handelt es sich dabei um eine Virtual Reality Brille in Kombination mit Zusatzinhalten, wie sie für gewöhnlich bei Augmented Reality verwendet werden. Diese werden mittels des in der Brille eingebauten Displays über die tatsächliche Welt „gelegt" und versorgen den Nutzer so mit Informationen. Die Bedienung erfolgt per Gesten- oder Sprachsteuerung.

Weitere Anwendungsgebiete von Game Engines

Abgesehen von „klassischen" Visualisierungen für Entwurfs- und Präsentationszwecke sind weitere Aufgabengebiete denkbar. Geht es zum Beispiel um Simulation von Fluchtwegen oder körperlich bedingten Einschränkungen im Alltag, können Game Engines ein hilfreiches Werkzeug darstellen. Die Rauchentwicklung innerhalb eines Gebäudes und die damit einhergehende Sichteinschränkung könnten simuliert werden, um die Anordnung von Fluchtwegmarkierungen innerhalb eines Gebäudes zu testen. Eine geringere Körpergröße (wie etwa bei Kindern) oder Einschränkungen durch bestehende Sehschwäche könnten ebenso dargestellt werden, um Probleme bei der Orientierung (beispielsweise in einem Krankenhaus) früh behandeln zu können.

Auch die Bauforschung und Bauaufnahme bieten Anwendungsgebiete für Echtzeitvisualisierungen. Denkbar ist ein Einsatz etwa zur Hilfestellung bei der Erforschung historischer Bausubstanz oder bei der Zugänglichmachung von Orten, die normalerweise nicht für die Öffentlichkeit begehbar sind. Durch die Möglichkeiten welche sich bei der Simulierung bieten, können verschiedene zeitliche Bauphasen oder auch der altersbedingte Verfall verdeutlicht werden. Moderne Engines bieten darüber hinaus Systeme zur Darstellung von künstlicher Intelligenz, die zudem mittels der engineinternen Programmiersprachen um eigene Funktionen erweitert werden können. Mit ihnen lässt sich menschliches Verhalten simulieren und für Analysen nutzen (zum Beispiel von Bewegungsströmen oder von menschlichem Verhalten innerhalb eines Gebäudes). Zusätzlich können solche Anwendungen mit akustischen Informationen versehen werden und mit all ihren Funktionen zu eigenen (Architektur-)Applikationen entwickelt werden.

Hinsichtlich der genannten Punkte wird deutlich, dass Game Engines sich in ihrer Anwendung für Architektur noch am Anfang der Entwicklung befinden. Betrachtet man jedoch die aktuellen Möglichkeiten, dann wird deutlich, dass sie zukünftig einen wichtigen Bestandteil im Aufgabengebiet der Architekturvisualisierung bilden werden.

Literatur

Ein Buch für Helmut Richter, Technische Universität Wien, Fakultät für Architektur- und Raumplanung, Schartner/Fasch, ISBN: 978-3-9501497-7-7

MA19 Stadtplanung Wien, Projekte und Konzepte, Ganztageshauptschule Kinkplatz Wien 14. ISBN: 3-901210-57-1

Chramosta, „Helmut Richter, Bauten und Projekte“, 1999, ISBN-10: 0792362314

Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1

derStandard, 21.06.2014, Architektur

Galileo Design: Cinema 4D, Das umfassende Handbuch, Andreas Asanger, ISBN-10: 383621797X

Jacqueline Esen: Digitale Fotografie, Grundlagen und Fotopraxis, ISBN-10: 38421001183

Francesco Legrnezi, Vray The Complete Guide, second edition, 2010

Möller/Haines/Hoffman, „Real-time Rendering“, 2008, ISBN-10: 1568814240

Chaosgroup, „Guide to GPU v. 1.0“, 2016

Shiratuddin/Thabet, „Virtual Office Walkthrough Using a 3D Game Engine“, 2002

Paar/Herwig, „Game Engines: Tools for Landscape Visualization and Planning?“, 2004

Indraprastha/Shinozaki, „The Investigation on using Unity3D Game Engine in Urban Design Study“, 2009

Rontsinsky, „Echtzeit-Rendering-Techniken für die Planung und Visualisierung von Architektur-Projekten, 2012].

Weblinks

URL Game Engines 1

http://www.theverge.com/2015/3/4/8150057/unreal-engine-4-epic-games-tim-sweeney-gdc-2015

URL Computer Graphics 1:

https://en.wikipedia.org/wiki/Computer_graphics#cite_note-1

URL Computer Graphics 2:

Michael Friendly (2008). „Milestones in the history of thematic cartography, statistical graphics, and data visualization".

URL Computer Graphics 3:

http://www.graphics.cornell.edu/online/tutorial/

URL Game Engine 2:

https://en.wikipedia.org/wiki/Game_engine

URL Game Engine 3:

http://www.kinephanos.ca/2014/game-engines-and-game-history/

URL Game Engine 4:

https://www.theguardian.com/technology/gamesblog/2009/dec/14/games-gameculture

URL Game Engine 5:

factions.pidbaq.com/whats-a-good-game-engine/

URL Echtzeit Engine:

http://www.software3d.de/blog/unterschied-externe-render_vray.html

URL Vito Miliano:

http://old.hirevito.com/oldportfolio/unrealty/vsmm99/

URL Ben Prince:

https://www.youtube.com/watch?v=K_YM-vnl8J8

URL Falling Water:

http://www.breeed.com/

URL Bryan Mock:

https://www.youtube.com/watch?v=KTLQxNxO_n8

URL Dereau Benoit:

http://www.benoitdereau.com/

URL Rafreis:

https://80.lv/articles/ue4arch-interview-about-realiscit-environments-in-ue4/

URL Vineyard Challenge:

https://www.unrealengine.com/blog/winners-of-the-vineyard-challenge-revealed

URL 3D Modelling:

https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704

URL GI 1:

http://home.lagoa.com/2014/04/ray-tracing-vs-path-tracing-in-plain-english/

URL GI 2:

http://www.scratchapixel.com/index.php

URL GI 3:

http://www.evermotion.org/tutorials/show/8056/3ds-max-what-is-final-gather

URL Unreal Engine 4.14:

https://www.unrealengine.com/blog/unreal-engine-4-14-released

URL Composition:

https://en.wikipedia.org/wiki/3D_modeling

URL Helmut Richter 1:

https://de.wikipedia.org/wiki/Helmut_Richter_(Architekt)

URL Helmut Richter 2:

http://www.nextroom.at/actor.php?id=4369

URL Helmut Richter 3:

https://www.architektur-aktuell.at/projekte/helmut-richter-die-glasschule-am-kink-platz-revisited

URL Helmut Richter 4:

http://www.meinbezirk.at/penzing/lokales/mittelschule-kinkplatz-ungluecksgebaeude-wird-endlich-generalsaniert-oder-abgerissen-d1756518.html

URL Helmut Richter 5:

https://www.architektur-aktuell.at/projekte/helmut-richter-die-glasschule-am-kink-platz-revisited

URL Lightmass 1:

https://forums.unrealengine.com/showthread.php?88952-Lets-make-LightURL 5:

mass-EPIC-%28and-understandable%29

URL Lightmass 2:

https://docs.unrealengine.com/latest/INT/

URL Lightmass 3:

https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/Basics/

URL Virtual Reality:

https://www.youtube.com/watch?v=pEeN_4h1DYQ

Weitere Quellen

http://www.archdaily.com/607849/unreal-visualizations-3-pros-and-3-cons-of-rendering-with-a-video-game-engine/

http://blog.digitaltutors.com/understanding-global-illumination/

https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/

http://www.thepolygoners.com/tutorials/GIIntro/GIIntro.htm

http://www.scratchapixel.com/index.php

http://www.evermotion.org/tutorials/show/8056/3ds-max-what-is-final-gather-

Möller, Haines, Hoffman: Real-time Rendering 2008

https://en.wikipedia.org/wiki/Real-time_rendering

https://de.wikipedia.org/wiki/3D-Echtzeit

https://en.wikipedia.org/wiki/Polygonal_modeling

https://de.wikipedia.org/wiki/Non-Uniform_Rational_B-Spline

https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704

https://en.wikipedia.org/wiki/Rhinoceros_3D

http://www.vol.at/schulbauprogramm-2000-abgeschlossen/vienna-migrate-45343

http://www.nextroom.at/building.php?id=2381

https://de.wikipedia.org/wiki/UV-Koordinaten

https://www.archtoolbox.com/materials-systems/electrical/readlightdistributiondiagram.html

0 - Projektbilder

Abb. 0.1 Verfasser der Arbeit, 2017
Abb. 0.2 Verfasser der Arbeit, 2017
Abb. 0.3 Verfasser der Arbeit, 2017
Abb. 0.4 Verfasser der Arbeit, 2017
Abb. 0.5 Verfasser der Arbeit, 2017
Abb. 0.6 Verfasser der Arbeit, 2017
Abb. 0.7 Verfasser der Arbeit, 2017
Abb. 0.8 Verfasser der Arbeit, 2017
Abb. 0.9 Verfasser der Arbeit, 2017
Abb. 0.10 Verfasser der Arbeit, 2017
Abb. 0.11 Verfasser der Arbeit, 2017
Abb. 0.12 Verfasser der Arbeit, 2017
Abb. 0.13 Verfasser der Arbeit, 2017
Abb. 0.14 Verfasser der Arbeit, 2017
Abb. 0.15 Verfasser der Arbeit, 2017
Abb. 0.16 Verfasser der Arbeit, 2017
Abb. 0.17 Verfasser der Arbeit, 2017
Abb. 0.18 Verfasser der Arbeit, 2017
Abb. 0.19 Verfasser der Arbeit, 2017
Abb. 0.20 Verfasser der Arbeit, 2017
Abb. 0.21 Verfasser der Arbeit, 2017
Abb. 0.22 Verfasser der Arbeit, 2017
Abb. 0.23 Verfasser der Arbeit, 2017
Abb. 0.24 Verfasser der Arbeit, 2017
Abb. 0.24.1 Verfasser der Arbeit, 2017
Abb. 0.25 Verfasser der Arbeit, 2017
Abb. 0.26 Verfasser der Arbeit, 2017
Abb. 0.28 Verfasser der Arbeit, 2017
Abb. 0.29 Verfasser der Arbeit, 2017
Abb. 0.10.1 Verfasser der Arbeit, 2017

1 - Prolog

Abb. 1.0 UE4Arch, https://www.youtube.com/watch?v=9h0hVZ3WIs4&t=13s
Abb. 1.1 UE4Arch, https://www.youtube.com/watch?v=Gah8sHA1r_8&t=62s
Abb. 1.2 UE4Arch, https://www.youtube.com/watch?v=Gah8sHA1r_8&t=62s
Abb. 1.3 koooolalala, https://www.youtube.com/watch?v=slc--V2pi5c
Abb. 1.4 Pixelflakes, https://d1b0j0gf6r3vez.cloudfront.net/app/uploads/2016/11/15151335/16036_202_Turntable-Platform_PF-copy.jpg
Abb. 1.5 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 1.6 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 1.7 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 1.8 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 1.9 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 1.10 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 1.11 Verfasser der Arbeit, 2017
Abb. 1.12 Verfasser der Arbeit, 2014
Abb. 1.12.1 Pedro Fernandes, Arqui9, http://www.3dartistonline.com/users/4109/thm1024/13694032 71_arqui9.jpg
Abb. 1.13 Verfasser der Arbeit, 2014
Abb. 1.14 http://2012books.lardbucket.org/books/mass-communication-

Abb. 1.15 https://musingsofamariominion.wordpress.com/tag/spacewar/
Abb. 1.16 https://en.wikipedia.org/wiki/IBM_2250
Abb. 1.17 Verfasser der Arbeit, 2017
Abb. 1.18 https://de.wikipedia.org/wiki/Macintosh
1.18.1 https://de.wikipedia.org/wiki/Amiga
Abb. 1.19 https://en.wikipedia.org/wiki/Money_for_Nothing_(song)
Abb. 1.20 http://www.playbuzz.com/ashleighburns10/do-you-remember-toy-story-1
Abb. 1.21 http://www.gamehackstudios.com/quake-free-download/
Abb. 1.22 http://media.moddb.com/images/downloads/1/75/74481/TOUT-FacingWorlds.jpg
Abb. 1.23 http://uncannyvalleymag.blogspot.co.at/2012/12/pimp-my-fic-2-final-fantasy-spirits.html
Abb. 1.24 Verfasser der Arbeit, 2017
Abb. 1.25 http://www.rpg-palace.com/maxy/beaumonde%20cathedral.png
https://en.wikipedia.org/wiki/Wargame_Construction_Set
https://en.wikipedia.org/wiki/Pinball_Construction_Set
Abb. 1.26 http://www.gamona.de/games/tom-clancy-s-the-division,
xbox-one-version-nicht-in-full-hd-mit-framedrops-bis-unter:news.html
Abb. 1.27 https://home.otoy.com/octanerender-3-and-roadmap-update/
Abb. 1.28 chaosgroup.com, Guide to GPU, v-ray.com, labs.chaosgroup.com
Abb. 1.29 chaosgroup.com, Guide to GPU, v-ray.com, labs.chaosgroup.com
Abb. 1.30 https://www.youtube.com/watch?v=-P28LKWTzrI
Abb. 1.32 chaosgroup.com, Guide to GPU, v-ray.com, labs.chaosgroup.com
Abb. 1.33 chaosgroup.com, Guide to GPU, v-ray.com, labs.chaosgroup.com
Abb. 1.35 Verfasser der Arbeit, 2017
Abb. 1.36 Verfasser der Arbeit, 2017
Abb. 1.37 Verfasser der Arbeit, 2017
Abb. 1.38 Verfasser der Arbeit, 2017
Abb. 1.39 Verfasser der Arbeit, 2017
Abb. 1.40 Verfasser der Arbeit, 2017
Abb. 1.41 Verfasser der Arbeit, 2017
Abb. 1.42 Verfasser der Arbeit, 2017
Abb. 1.43 koooolalala, https://www.youtube.com/watch?v=gC0PdspqNYk
Abb. 1.44 koooolalala, https://www.youtube.com/watch?v=DFPOsnC_ETQ
Abb. 1.45 Евгений Евдокимов, https://www.youtube.com/watch?v=XuoFq1ZXVsY&t=1s
Abb. 1.46 UE4Arch, https://www.youtube.com/watch?v=yxdp0icxcC0
Abb. 1.47 http://old.hirevito.com/oldportfolio/unrealty/vsmm99/
Abb. 1.48 http://researchrepository.murdoch.edu.au/id/eprint/7372/1/Virtual_office_walkthrough.pdf
Abb. 1.49 http://researchrepository.murdoch.edu.au/id/eprint/7372/1/Virtual_office_walkthrough.pdf
Abb. 1.50 http://researchrepository.murdoch.edu.au/id/eprint/7372/1/Virtual_office_walkthrough.pdf
Abb. 1.51 Herwig/Paar "Game Engines: Tools for Landscape Visualizastion and Planning?", 2004
Abb. 1.52 Indraprastha/Shinozaki, „The Investigation on using Unity3D Game Engine in Urban Design Study“, 2009
Abb. 1.53 Indraprastha/Shinozaki, „The Investigation on using Unity3D Game Engine in Urban Design Study“, 2009
Abb. 1.54 Indraprastha/Shinozaki, „The Investigation on using Unity3D Game Engine in Urban Design Study“, 2009
Abb. 1.55 Indraprastha/Shinozaki, „The Investigation on using Unity3D Game Engine in Urban Design Study“, 2009
Abb. 1.56 Rontsinsky, „Echtzeit-Rendering-Techniken für die
Planung und Visualisierung von Architektur-Projekten, 2012
Abb. 1.57 Rontsinsky, „Echtzeit-Rendering-Techniken für die
Planung und Visualisierung von Architektur-Projekten, 2012
Abb. 1.58 Rontsinsky, „Echtzeit-Rendering-Techniken für die
Planung und Visualisierung von Architektur-Projekten, 2012
Abb. 1.59 Rontsinsky, „Echtzeit-Rendering-Techniken für die
Planung und Visualisierung von Architektur-Projekten, 2012
Abb. 1.60 Rontsinsky, „Echtzeit-Rendering-Techniken für die
Planung und Visualisierung von Architektur-Projekten, 2012

Abb. 1.61 Rontsinsky, „Echtzeit-Rendering-Techniken für die Planung und Visualisierung von Architektur-Projekten, 2012
Abb. 1.62 Rontsinsky, „Echtzeit-Rendering-Techniken für die Planung und Visualisierung von Architektur-Projekten, 2012
Abb. 1.63 Rontsinsky, „Echtzeit-Rendering-Techniken für die Planung und Visualisierung von Architektur-Projekten, 2012
Abb. 1.64 URL 10
Abb. 1.65 URL 10
Abb. 1.66.1 URL 11
Abb. 1.66 URL 12
Abb. 1.67 URL 12
Abb. 1.68 URL 13
Abb. 1.69 URL 13
Abb. 1.70 URL 13
Abb. 1.71 URL 14
Abb. 1.72 URL 14
Abb. 1.73 URL 14
Abb. 1.74 URL 14
Abb. 1.75 URL 15

2 - Workflow

Abb. 2.1 Verfasser der Arbeit, 2017
Abb. 2.2 Verfasser der Arbeit, 2017
Abb. 2.3 http://forums.newtek.com/showthread.php?131319-Another-Box-Model-Head-Steps-Thread
Abb. 2.4 Verfasser der Arbeit, 2017
Abb. 2.5 https://www.pinterest.com/source/grasshopper3d.com
Abb. 2.6 http://www.rhino3dportugal.com/site/wp-content/uploads/2012/10/Background-Grasshopper.jpg
Abb. 2.7 https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704
Abb. 2.8 Verfasser der Arbeit, 2017
Abb. 2.9 Verfasser der Arbeit, 2017
Abb. 2.10 Verfasser der Arbeit, 2017
Abb. 2.11 Verfasser der Arbeit, 2017
Abb. 2.12 Verfasser der Arbeit, 2017
Abb. 2.13 Verfasser der Arbeit, 2017
Abb. 2.14 Verfasser der Arbeit, 2017
Abb. 2.15 Verfasser der Arbeit, 2017
Abb. 2.16 Verfasser der Arbeit, 2017
Abb. 2.17 Verfasser der Arbeit, 2017
Abb. 2.18 https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704
Abb. 2.19 https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704
Abb. 2.20 https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704
Abb. 2.21 https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704
Abb. 4.9.2 Verfasser der Arbeit, 2017
Abb. 4.9.3 Verfasser der Arbeit, 2017

Abb. 4.10 http://dudka.cz/rrv/files/screenshot/room4-step079-snapshot000.png (RRV - Radiosity Renderer and Visualizer, dudka.cz/rrv), CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=4457401
Abb. 4.11 http://www.lafortune.eu/publications/dissertation/images.html
Abb. 4.12 http://www.lafortune.eu/publications/dissertation/images.html
Abb. 4.13 https://de.wikipedia.org/wiki/Raytracing
Abb. 4.14 https://de.wikipedia.org/wiki/Raytracing
Abb. 4.15 https://de.wikipedia.org/wiki/Raytracing
Abb. 4.16 https://de.wikipedia.org/wiki/Raytracing
Abb. 4.17 https://de.wikipedia.org/wiki/Raytracing
Abb. 4.23 Verfasser der Arbeit, 2017
Abb. 4.24 Verfasser der Arbeit, 2017
Abb. 4.25 Vray, The Complete Guide, Francesco Legrenzi
Abb. 4.26 Vray, The Complete Guide, Francesco Legrenzi
Abb. 4.28 http://www.evermotion.org/tutorials/show/8056/3ds-max-what-is-final-gather-
Abb. 2.22 Peter Guthrie, HDRI Sky, 1433 Sun Blue Sky
Abb. 2.23 Verfasser der Arbeit, 2017
Abb. 2.24 Verfasser der Arbeit, 2017
Abb. 2.25 Verfasser der Arbeit, 2017
Abb. 2.26 Verfasser der Arbeit, 2017
Abb. 2.27 Verfasser der Arbeit, 2017
Abb. 2.28 Verfasser der Arbeit, 2017
Abb. 2.29 Verfasser der Arbeit, 2017
Abb. 2.30 Verfasser der Arbeit, 2017
Abb. 2.30.1 https://www.archtoolbox.com/materials-systems/electrical/readlightdistributiondiagram.html
Abb. 2.31 Verfasser der Arbeit, 2017
Abb. 2.32 Verfasser der Arbeit, 2017
Abb. 2.33 Verfasser der Arbeit, 2017
Abb. 2.33.1 Von Zephyris at en.wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7202834
Abb. 2.34 https://www.unrealengine.com/blog/unreal-engine-4-14-released
Abb. 2.35 Verfasser der Arbeit, 2017
Abb. 2.36 Verfasser der Arbeit, 2017
Abb. 2.37 Verfasser der Arbeit, 2017
Abb. 2.38 Verfasser der Arbeit, 2017
Abb. 2.39 Verfasser der Arbeit, 2017
Abb. 2.40 Verfasser der Arbeit, 2017
Abb. 2.41 Verfasser der Arbeit, 2017
Abb. 2.42 Verfasser der Arbeit, 2017
Abb. 2.43 Verfasser der Arbeit, 2017
Abb. 2.44 Verfasser der Arbeit, 2017
Abb. 2.45 Verfasser der Arbeit, 2017
Abb. 2.46 Verfasser der Arbeit, 2017
Abb. 2.47 koooolalala, https://www.youtube.com/watch?v=_nLGoqqDc0w
Abb. 2.48 koooolalala, https://www.youtube.com/watch?v=_nLGoqqDc0w
Abb. 2.49 Verfasser der Arbeit, 2017
Abb. 2.50 Verfasser der Arbeit, 2017
Abb. 2.51 arqui9, https://scontent.xx.fbcdn.net/v/t1.0-0/s526x395/15219612_1240508706022552_5345395136209773357_n.jpg?oh=1b503fb7ce2398be79358dd0567c83a4&oe=58DE073C
Abb. 2.52 BEHF Architekten, 2016
Abb. 2.53 https://www.youtube.com/watch?v=O8i7OKbWmRM&list=PLlh8Uv1hC6mBkJ3pIzEdC8tTuvFhkp2fU&index=1&t=2s%20-%3e%20tamas%20medve,%20colin%20levy

Abb. 2.54 https://www.youtube.com/watch?v=O8i7OKbWmRM&list
=PLlh8Uv1hC6mBkJ3pIzEdC8tTuvFhkp2fU&index=1&t=2s%20-%3e%20tamas%20medve,%20colin%20levy

Abb. 2.55 https://www.youtube.com/watch?v=O8i7OKbWmRM&list
=PLlh8Uv1hC6mBkJ3pIzEdC8tTuvFhkp2fU&index=1&t=2s%20-%3e%20tamas%20medve,%20colin%20levy

Abb. 2.56 https://interfacelift.com/wallpaper/downloads/date/any/
Abb. 2.58 https://interfacelift.com/wallpaper/downloads/date/any/
Abb. 2.59 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 2.60 Alex Roman, The Third & The Seventh // From Bits to the Lens, ISBN: 10 84-616-5240-1
Abb. 2.61 Verfasser der Arbeit, 2017
Abb. 2.62 Verfasser der Arbeit, 2017
Abb. 2.63 Verfasser der Arbeit, 2017
Abb. 2.64 Verfasser der Arbeit, 2017

3 - Projektteil

Abb. 3.1 Ein Buch für Helmut Richter, Technische Universität Wien,
Fakultät für Architektur- und Raumplanung, ISBN: 978-3-9501497-7-7
Abb. 3.2 Ein Buch für Helmut Richter, Technische Universität Wien,
Fakultät für Architektur- und Raumplanung, ISBN: 978-3-9501497-7-7
Abb. 3.3 http://www.nextroom.at/actor.php?id=4369
Abb. 3.4 http://www.nextroom.at/actor.php?id=4369
Abb. 3.5 Ein Buch für Helmut Richter, Technische Universität Wien,
Fakultät für Architektur- und Raumplanung, ISBN: 978-3-9501497-7-7
Abb. 3.6 http://www.nextroom.at/actor.php?id=4369
Abb. 3.7 http://www.nextroom.at/actor.php?id=4369
Abb. 3.8 http://www.nextroom.at/actor.php?id=4369
Abb. 3.9 http://www.nextroom.at/actor.php?id=4369
Abb. 3.10 http://www.nextroom.at/actor.php?id=4369
Abb. 3.11 Stadtplanung Wien, Projekte und Konzepte, Ganztageshauptschule
Kinknplatz Wien 14, Mischa Erben, ISBN: 3-901210-57-1
Abb. 3.12 Helmut Richter
Abb. 3.13 Stadtplanung Wien, Projekte und Konzepte, Ganztageshauptschule
Kinknplatz Wien 14, Croce+Wir, ISBN: 3-901210-57-1
Abb. 3.14 Helmut Richter
Abb. 3.15 Stadtplanung Wien, Projekte und Konzepte, Ganztageshauptschule
Kinknplatz Wien 14, Croce+Wir, ISBN: 3-901210-57-1
Abb. 3.16 Helmut Richter
Abb. 3.16.1 Helmut Richter
Abb. 3.17 Helmut Richter
Abb. 3.18 Helmut Richter
Abb. 3.19 Helmut Richter
Abb. 3.20 Helmut Richter
Abb. 3.21 Helmut Richter
Abb. 3.22 Helmut Richter
Abb. 3.23 Helmut Richter
Abb. 3.24 Helmut Richter
Abb. 3.25 Helmut Richter
Abb. 3.26 Helmut Richter
Abb. 3.27 Helmut Richter
Abb. 3.28 Helmut Richter

Abb. 3.29 Verfasser der Arbeit, 2016
Abb. 3.30 Helmut Richter
Abb. 3.31 Helmut Richter
Abb. 3.32 Helmut Richter
Abb. 3.33 Helmut Richter
Abb. 3.34 Helmut Richter
Abb. 3.35 Helmut Richter
Abb. 3.36 Verfasser der Arbeit, 2016
Abb. 3.37 Mischa Erben, 1995
Abb. 3.38 Verfasser der Arbeit, 2016
Abb. 3.39 Verfasser der Arbeit, 2016
Abb. 4.0 Verfasser der Arbeit, 2017
Abb. 4.1 Verfasser der Arbeit, 2017
Abb. 4.2 Verfasser der Arbeit, 2017
Abb. 4.3 Verfasser der Arbeit, 2017
Abb. 4.4 Verfasser der Arbeit, 2017
Abb. 4.5 Verfasser der Arbeit, 2017
Abb. 4.6 http://www.dadsoft.net/2013/12/unreal-gold-game.html
Abb. 4.7 http://www.oldpcgaming.net/unreal-tournament-2003-review/
Abb. 4.8 http://www.gamehackstudios.com/unreal-tournament-3-free-download/
Abb. 4.9 https://i.imgur.com/c4vBMUn.jpg
Abb. 4.9.1 Verfasser der Arbeit, 2017
Abb. 4.9.2 Verfasser der Arbeit, 2017
Abb. 4.29 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.30 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.31 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.32 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.33 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.34 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.35 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.36 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.37 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.38 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.39 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.40 Verfasser der Arbeit, 2017
Abb. 4.41 Verfasser der Arbeit, 2017
Abb. 4.42 Verfasser der Arbeit, 2017
Abb. 4.43 Verfasser der Arbeit, 2017
Abb. 4.44 Verfasser der Arbeit, 2017
Abb. 4.45 Verfasser der Arbeit, 2017
Abb. 4.45.1 Peter Guthrie
Abb. 4.46 https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/
Abb. 4.47 Verfasser der Arbeit, 2017
Abb. 4.48 Verfasser der Arbeit, 2017
Abb. 4.48.1 Verfasser der Arbeit, 2017
Abb. 4.49 Verfasser der Arbeit, 2017
Abb. 4.50 Verfasser der Arbeit, 2017
Abb. 4.51 Verfasser der Arbeit, 2017
Abb. 4.52 Verfasser der Arbeit, 2017

Abb. 4.53 Verfasser der Arbeit, 2017
Abb. 4.54 Verfasser der Arbeit, 2017
Abb. 4.55 Verfasser der Arbeit, 2017
Abb. 4.56 Verfasser der Arbeit, 2017
Abb. 4.57 Verfasser der Arbeit, 2017
Abb. 4.58 Verfasser der Arbeit, 2017
Abb. 4.58.1 Verfasser der Arbeit, 2017
Abb. 4.58.2 Verfasser der Arbeit, 2017
Abb. 4.58.3 Verfasser der Arbeit, 2017
Abb. 4.58.4 Verfasser der Arbeit, 2017
Abb. 4.58.5 Verfasser der Arbeit, 2017
Abb. 4.58.6 Verfasser der Arbeit, 2017
Abb. 4.58.7 Verfasser der Arbeit, 2017
Abb. 4.58.8 Verfasser der Arbeit, 2017
Abb. 4.58.9 Verfasser der Arbeit, 2017
Abb. 4.58.10 Verfasser der Arbeit, 2017
Abb. 4.58.11 Verfasser der Arbeit, 2017
Abb. 4.58.12 Verfasser der Arbeit, 2017
Abb. 4.59 Verfasser der Arbeit, 2017
Abb. 4.59.1 Verfasser der Arbeit, 2017
Abb. 4.60 Verfasser der Arbeit, 2017
Abb. 4.61 Verfasser der Arbeit, 2017
Abb. 4.62 Verfasser der Arbeit, 2017
Abb. 4.63 Verfasser der Arbeit, 2017
Abb. 4.64 Verfasser der Arbeit, 2017
Abb. 4.65 Verfasser der Arbeit, 2017
Abb. 4.66.1 Verfasser der Arbeit, 2017
Abb. 4.66.2 Verfasser der Arbeit, 2017
Abb. 4.66.3 Verfasser der Arbeit, 2017
Abb. 4.66.4 Verfasser der Arbeit, 2017
Abb. 4.66.5 Verfasser der Arbeit, 2017
Abb. 4.66.6 Verfasser der Arbeit, 2017
Abb. 4.66.7 Verfasser der Arbeit, 2017
Abb. 4.66.8 Verfasser der Arbeit, 2017
Abb. 4.66.9 Verfasser der Arbeit, 2017
Abb. 4.66.10 Verfasser der Arbeit, 2017
Abb. 4.67 http://www.speedtree.com/
Abb. 4.68 http://www.speedtree.com/
Abb. 4.69 http://www.speedtree.com/
Abb. 4.70 http://www.speedtree.com/
Abb. 4.71 http://www.speedtree.com/
Abb. 4.72 http://www.speedtree.com/
Abb. 4.73 Verfasser der Arbeit, 2017
Abb. 4.74 Verfasser der Arbeit, 2017
Abb. 4.75 Verfasser der Arbeit, 2017
Abb. 4.76 Verfasser der Arbeit, 2017
Abb. 4.77 Verfasser der Arbeit, 2017

Abb. 4.49.1 Verfasser der Arbeit, 2017
Abb. 4.49.2 Verfasser der Arbeit, 2017
Abb. 4.49.3 Verfasser der Arbeit, 2017
Abb. 4.49.4 Verfasser der Arbeit, 2017
Abb. 4.49.5 Verfasser der Arbeit, 2017

4 - Ausblick & Zusammenfassung

Abb. 5.1 http://bizarreality.co.za/ar
Abb. 5.2 Zero Density, https://www.youtube.com/watch?v=pEeN_4h1DYQ
Abb. 5.3 Zero Density, https://www.youtube.com/watch?v=pEeN_4h1DYQ
Abb. 5.4 Zero Density, https://www.youtube.com/watch?v=pEeN_4h1DYQ
Abb. 5.5 Zero Density, https://www.youtube.com/watch?v=pEeN_4h1DYQ
Abb. 5.7 http://www.pcgameshardware.de/Unreal-Engine-Software-239301/News/VR-Editor-HTC-Vive-1185376/