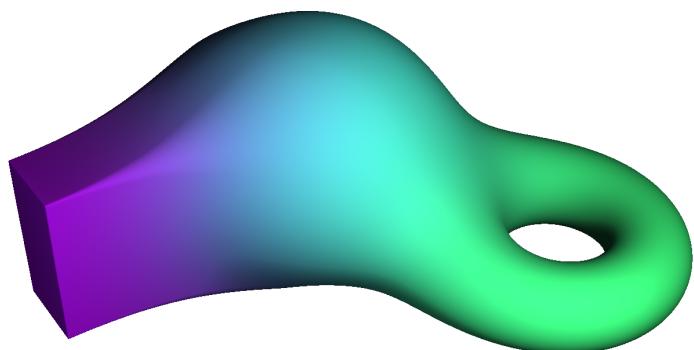


Raymarching

Studienarbeit Mathematik

Moritz Kronberger



Hochschule Augsburg
Interaktive Medien Semester 4
SoSe 2021

Inhaltsverzeichnis

1 Einleitung	2
2 Begriff der Bildsynthese	2
3 Verdeckungsberechnung	2
3.1 Grundlagen des Raymarchings	4
3.2 Signed Distance Functions (SDFs)	8
3.2.1 Einfache SDFs	8
3.2.2 SDF Quader	9
3.2.3 Transformation von SDFs	11
4 Shading	12
4.1 Pseudo-Depth-Mapping	12
4.2 Normalenberechnung	13
4.3 Blinn-Phong-Modell	14
5 Globale Lichtverteilung	16
5.1 Einfache Schatten	17
5.2 Weiche Schatten	18
6 Alternativen zur Minimumsfunktion	19
6.1 Mengenoperationen	19
6.2 Smooth Minimum	22
7 Einsatzbereiche des Raymarchings	25
8 Ausblick	25
9 Links der Anwendungen	27
10 Quellen	27

1 Einleitung

Diese Arbeit soll sich einer möglichen Herangehensweise an die Aufgabe der Bildsynthese, dem Raymarching, oft auch als Spheretracing bezeichnet, widmen. Der Fokus liegt hierbei auf der Darstellung implizit über Signed Distance Functions (SDFs) definierter Geometrie. Dabei sollen neben der Raymarching-Technik selbst auch allgemeinere Bildsynthesetechniken Erwähnung finden.

Grundlage der Ausarbeitung ist eine, in der Spiele-Engine Unity entwickelte, Anwendung. Bei dieser wird die Unity-Umgebung hauptsächlich als grafisches Interface genutzt sowie von einigen der eingebauten Funktionen, wie dem Kameramodell, Gebrauch gemacht. Ein weiterer Grund für die Wahl von Unity ist die Unterstützung von *Compute Shadern*, programmiert in HLSL (High Level Shader Language), was eine schnelle, parallele Verarbeitung des Raymarchings auf der Grafikkarte ermöglicht.

Die Erläuterung der Raymarching-Technik wird zudem durch eine interaktive Visualisierung mithilfe von p5.js unterstützt.

Zur Anfertigung ergänzender Grafiken kamen das Funktionsplotting-Programm Desmos, sowie Adobe Illustrator zum Einsatz.

2 Begriff der Bildsynthese

Zu Beginn soll der Begriff der Bildsynthese, auch als Rendering bezeichnet, etwas genauer definiert und unterteilt werden.

Es sei also zunächst eine einfache Szene gegeben, die bestimmte Grundobjekte (auch geometrische Primitive genannt) sowie Lichter, mit bestimmten Eigenschaften enthält. Durch den Vorgang der Bildsynthese soll nun festgelegt werden, wie diese für einen bestimmten Betrachter, in der Regel eine Kamera, bestehend aus Augpunkt und Projektionsebene, dargestellt werden sollen.

Dieser Prozess kann in drei Hauptaufgaben eingeteilt werden:

Verdeckungsberechnung: Bei der Verdeckungsberechnung (auch Sichtbarkeitsentscheid genannt) soll ermittelt werden, welche Primitive an welcher Position und in welchem Umfang für den Betrachter sichtbar sein sollen.

Shading: Das Shading simuliert im Anschluss die Materialeigenschaften der angezeigten Primitive, beispielsweise deren Farbigkeit oder Oberflächenbeschaffenheit, unter Berücksichtigung der Lichter in der Szene.

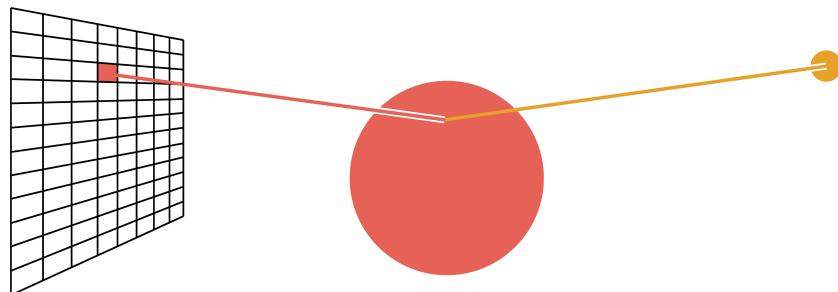
Globale Lichtverteilung: Zuletzt kann die globale Lichtverteilung innerhalb der Szene berechnet werden. Hierzu zählt beispielsweise die indirekte Beleuchtung durch Reflexion des Lichts durch die Primitive oder der Schattenwurf der Primitive untereinander.

3 Verdeckungsberechnung

Der Themenschwerpunkt dieser Arbeit, das Raymarching, ist eine Technik zur Berechnung des Sichtbarkeitsentscheids. Einige Eigenheiten des Raymarchings haben allerdings auch Auswirkungen auf das Shading und die globale Lichtverteilung.

Zur Umsetzung der Verdeckungsberechnung gibt es unzählige verschiedene Techniken. Um eine Intuition für die grundlegende Herangehensweise beim Raymarching zu erhalten lohnt es sich zunächst (sehr stark vereinfacht) eine "reale Bildsynthese", das Aufnehmen eines digitalen Fotos zu betrachten:

Hierbei soll die Szene aus einem Primitiv (der roten Kugel), einer Punktlichtquelle und dem Sensor (Pixelraster) einer Kamera bestehen:



Die Lichtquelle emittiert nun Photonen, welche sich bis zu einer Kollision geradlinig als Strahl durch den Raum bewegen. Bei einer Kollision mit einem Primitiv wird nun je nach dessen Oberflächenbeschaffenheit ein Teil der Energie des Photons absorbiert, transmittiert und reflektiert. Trifft das reflektierte Photon nun schlussendlich auf einen Pixel des Kamerasensors, so kann die verbleibende Energie des Photons in einen Farbwert "übersetzt" werden, der dem Farbwert an der letzten Kollision entspricht. Geschieht dies für jeden Pixel des Sensors entsteht ein Abbild der Szene. Ein möglicher Ansatz zur Bildsynthese wäre also diesen physikalischen Ablauf modellhaft nachzubilden.

Hierzu wird der Vorgang des Aussendens von Strahlen übernommen, jedoch direkt eine enorme Ineffizienz des obigen Modells behoben. Man kann sich vorstellen, dass die Lichtquelle eine unfassbar große Anzahl an Lichtstrahlen in alle unterschiedlichen Richtungen emittiert, von welchen allerdings nur ein äußerst geringer Bruchteil tatsächlich auf die Pixel des Sensors trifft. Idealerweise sollte natürlich nur Rechenleistung für Strahlen verwendet werden, die für die Berechnung des Bildes relevant sind, also auf den Sensor treffen.

Dies lässt sich sicherstellen, indem anstatt von der Lichtquelle, von jedem Pixel des Sensors ausgehend, ein Strahl in die Szene geschickt wird.

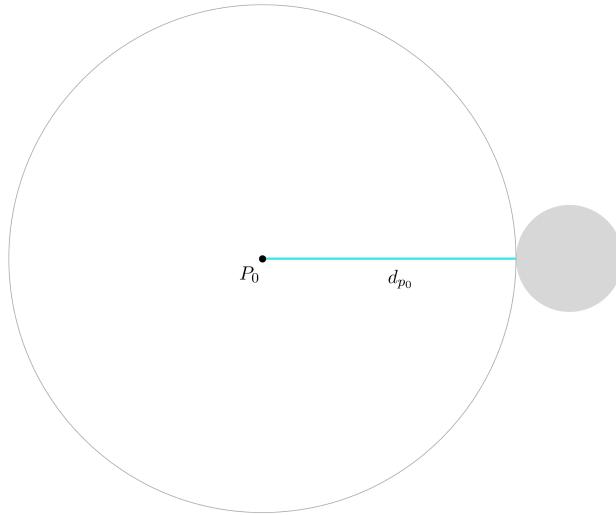
Interessant ist, dass diese Umkehr selbst bei einem physikalisch perfekten Renderer möglich wäre, da das Helmholtz Reziprozitätsprinzip vereinfacht gesprochen besagt, dass es für die "Endenergie" des Lichtstrahls unerheblich ist auf welcher Seite sich Sensor und Lichtquelle befinden.

Wenn nun das Aussenden von Strahlen in die Szene zunächst zur Verdeckungsberechnung genutzt werden soll, erscheint es intuitiv jeden Strahl auf einen Schnittpunkt mit einem Primitiv in der Szene zu überprüfen. Dies setzt jedoch voraus, dass für jedes Objekt der Szene ein konkreter Schnittpunkt mit einer (Halb-)Geraden berechnet werden kann, was einige Limitierungen mit sich bringt. Klassischerweise wird dieses Problem, beispielsweise beim Raytracing, gelöst, indem komplexe Objekte aus, zwischen Punkten aufgespannten, Drei- oder Mehrecken zusammengesetzt werden, deren Schnittpunkte einfach berechnet werden können. Es ergeben sich die allgemein bekannten Polygon-Meshes.

Raymarching ist allerdings vor allem deshalb interessant, weil hier das Problem der Schnittpunktberechnung vollständig eliminiert wird, sodass Objekte auch implizit, also rein durch mathematische Formeln, definiert werden können.

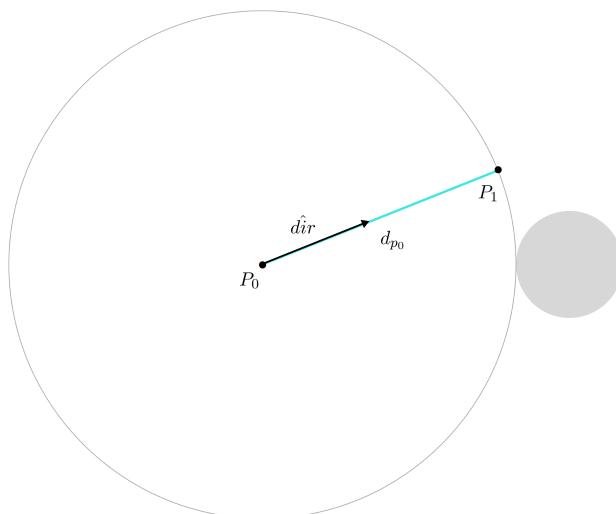
3.1 Grundlagen des Raymarchings

Es sei eine Szene, in der sich ein Augpunkt P_0 und verschiedene Primitive an beliebigen Stellen im Raum befinden.

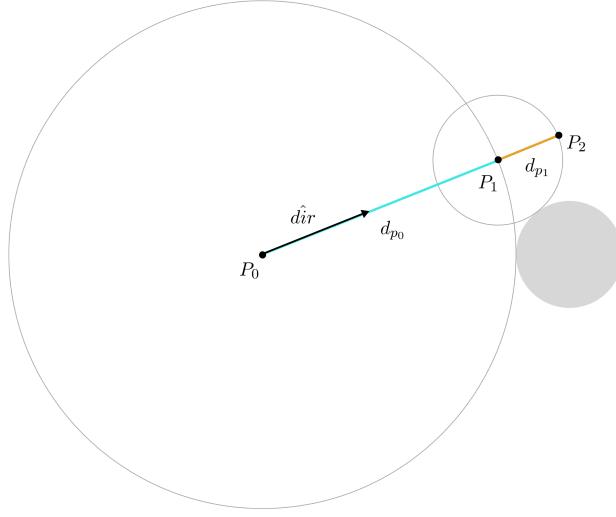


Es wird nun angenommen, dass eine Funktion \minDst existiert, welche den minimalen Abstand d_{p_0} von P_0 zu allen Primitiven der Szene angibt. Es können nun also von P_0 beliebige Strahlen der Länge d_{p_0} in alle Richtungen ausgesandt werden, ohne Gefahr zu laufen, dass die Strahlen ein Primitiv schneiden.

Es liegt nun ein normierter Richtungsvektor \hat{dir} vor, welcher die Richtung eines Strahls angibt. Dieser Richtungsvektor kann nun mit dem Skalar d_{p_0} multipliziert und zu P_0 addiert werden um den Ortsvektor eines neuen Punktes P_1 zu erhalten, welcher somit um d_{p_0} von P_0 entfernt am Ende des Strahls liegt.

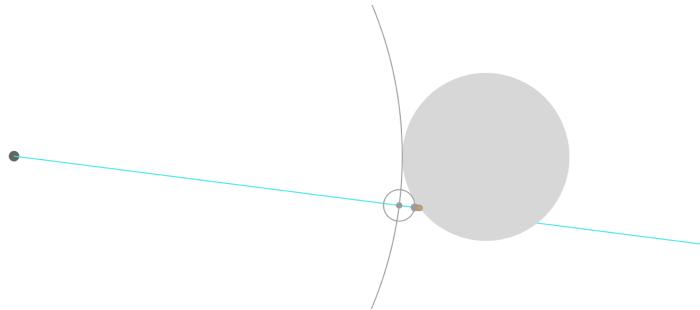


Für P_1 kann nun erneut mit \minDst der minimale Abstand d_{p_1} zu den Primitiven der Szene berechnet werden. Dieser Ablauf entspricht später einem Schritt des Algorithmus'. Skalarmultipliziert diesen erneut mit \hat{dir} und addiert das Ergebnis wieder zu P_1 ergibt sich der Ortsvektor eines ein Punktes P_2 mit dem Abstand $d_{p_0} + d_{p_1}$ zu P_0 in Richtung \hat{dir} .



Man unterscheidet an dieser Stelle zwischen einem Globalen Abstand d_g , also dem Abstand des letzten Punktes P_n von P_0 und einem lokalen Abstand d_{p_n} , also dem Ergebnis von \minDst für den jeweiligen Punkt.

Erhält man für d_{p_n} den Wert Null liegt der Punkt P_n auf der Oberfläche eines Primitivs, es wurde also an diesem Punkt eine Kollision mit einem der Primitive festgestellt.



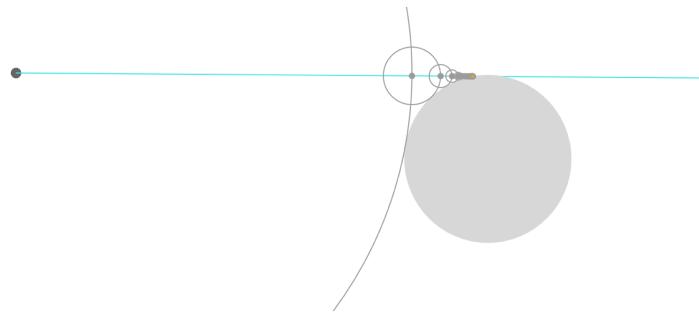
Wird für jeden Pixel der Kamera ein entsprechender Strahl, definiert durch einen normierten Richtungsvektor mit Fuß im Augpunkt und Spitze im entsprechenden Mittelpunkt des Pixels, ausgesandt, kann so die Verdeckungsberechnung für das gesamte Bild erfolgen. Kollidiert ein Strahl mit einem der Primitive, erhält man also auf dem Weg des Strahls einen lokalen Abstand von 0, kann der dem Strahl zugehörige Pixel dementsprechend markiert werden. Für den Fall, dass ein Strahl mit keinem Primitiv kollidieren sollte, muss zudem eine maximale, globale Distanz d_{max} oder maximale Schrittzahl s_{max} definiert werden, ab welcher die Berechnung abgebrochen wird.

Für die tatsächliche Implementierung eines solchen Algorithmus, sollte noch eine weitere Veränderung vorgenommen werden:

Man kann sich außerdem vorstellen, dass sich der Strahl beispielsweise einer Kugel in einem derart ungünstigen Winkel annähert, dass der minimale Abstand zwar immer kleiner wird, jedoch eine enorme Schrittzahl benötigt wird, bevor man tatsächlich einen Abstand von Null erhält.

Eine absolut genaue Berechnung des Kollisionspunktes ist in der Realität jedoch unnötig, nachdem dieser später durch einen Pixel dargestellt wird, welcher selbst eine gewisse Größe hat. Es reicht daher, ab einem gewissen Minimalabstand eine Kollision anzunehmen. Dieser wird konventionell durch die Konstante ϵ angegeben.

Im folgenden Beispiel nähert sich der Abstand zwar immer weiter Null an, erreicht diesen jedoch nie, bzw. in der Praxis erst beim Auslaufen der Gleitkommastellen des Systems. Mit einem ϵ -Wert von 0.01 wird hingegen schon nach unter 40 Schritten eine Kollision festgestellt.



Der Algorithmus läuft also folgendermaßen ab:

Für jeden Pixel:

Normierten Richtungsvektor \hat{dir} vom Augpunkt P zum Pixel-Mittelpunkt aufstellen

Globale Distanz d_g und Schrittzahl s mit 0 initialisieren

Solange $d_g < d_{max}$ und $s < s_{max}$:

Lokalen Abstand $dist$ von P mit $minDst$ berechnen

Wenn $dist < \epsilon$:

Kollision gefunden

Neuen Strahlen-Startpunkt P berechnen mit $P = P + dist \cdot \hat{dir}$

$dist$ zu d_g addieren

s erhöhen

Keine Kollision gefunden

In der Anwendung wird dieser Algorithmus mit folgendem Code umgesetzt:

```
[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID){

    // track global distance and steps marched
    float globalDistance = 0;
    int steps = 0;

    // create "uv-coordinates" with centered origin
    float2 uv = id.xy / Resolution.xy *2 -1;

    // create Ray from Camera to uv-coordinate
    Ray ray = CreateCameraRay(uv);

    Result[id.xy] = float4(backgroundColor, 1);

    while(globalDistance <= maxDistance && steps < maxSteps){

        steps++;

        float4 evalSceneForPoint = minDst(ray.origin);

        float dist = evalSceneForPoint.w;

        // check if collision occurred
        if(dist < epsilon){

            // set color at point
            globalDistance += dist;

            float3 color = evalSceneForPoint.xyz;

            Result[id.xy] = float4(color, 1);
            break;
        }

        // update new point position
        ray.origin += ray.direction * dist;

        // update current distance
        globalDistance += dist;
    }
}
```

3.2 Signed Distance Functions (SDFs)

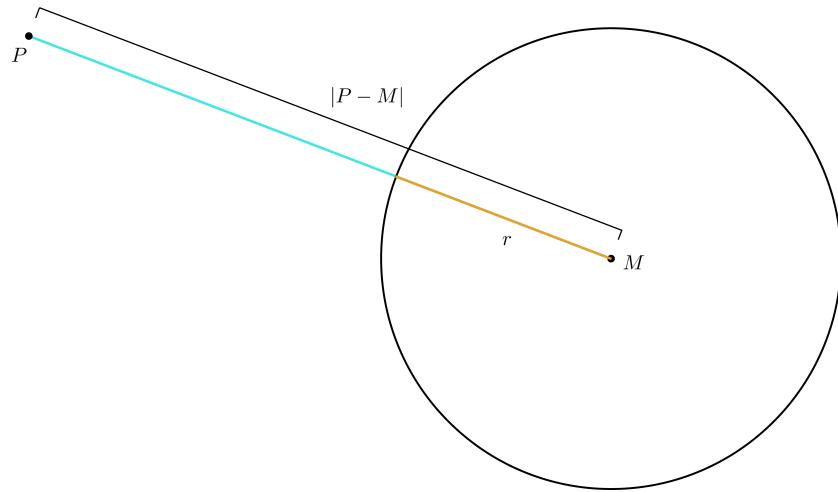
Bisher wurde die Funktion \minDst , welche die minimale Distanz von einem Punkt zu einem oder mehreren Primitiven angibt, einfach als gegeben angenommen. Tatsächlich kommen beim Raymarching dafür in der Regel sogenannte *Signed Distance Functions* (SDFs) zum Einsatz. Neben der minimalen Distanz eines Punktes zu einem Objekt geben SDFs dabei zusätzlich über das Vorzeichen der Distanz an, ob der Punkt außerhalb (positives Vorzeichen) oder innerhalb (negatives Vorzeichen) des Objekts liegt.

3.2.1 Einfache SDFs

Ein sehr anschauliches und einfaches Beispiel hierfür ist folgende SDF einer Kugel mit Radius r und Mittelpunkt M für einen Punkt P :

$$\text{sdf}_{\text{Kugel}} = |P - M| - r$$

Es wird also lediglich von der Länge von \overrightarrow{MP} der Radius r subtrahiert. Das Vorzeichen dieser Funktion ist somit für Abstände $|P - M| < r$, also Punkte innerhalb der Kugel, negativ, für $|P - M| > r$, also Punkte außerhalb der Kugel, positiv.



Noch simpler ist die SDF der sogenannten *Groundplane*, also einer Ebene, welche von der x- und z-Achse aufgespannt wird:

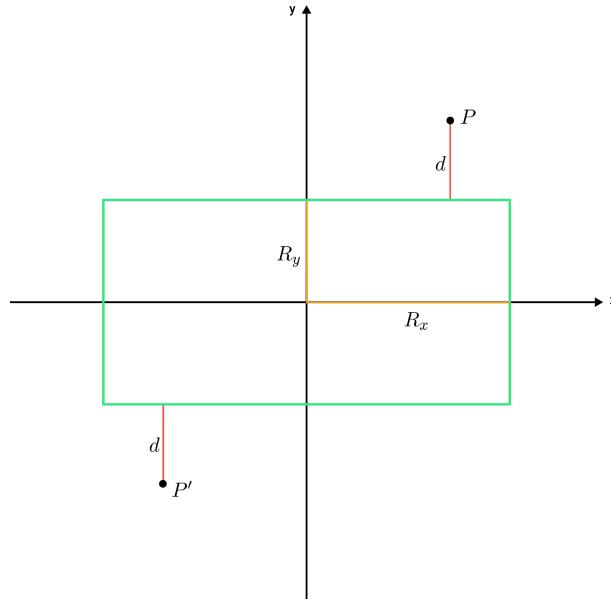
$$\text{sdf}_{\text{gPlane}} = P_y$$

Der Abstand des Punktes zur Groundplane entspricht dabei immer seiner y-Koordinate, für Punkte unterhalb der x-z-Ebene wird das Vorzeichen richtigerweise negativ.

3.2.2 SDF Quader

Etwas interessanter ist bereits die Herleitung einer SDF für Quader.

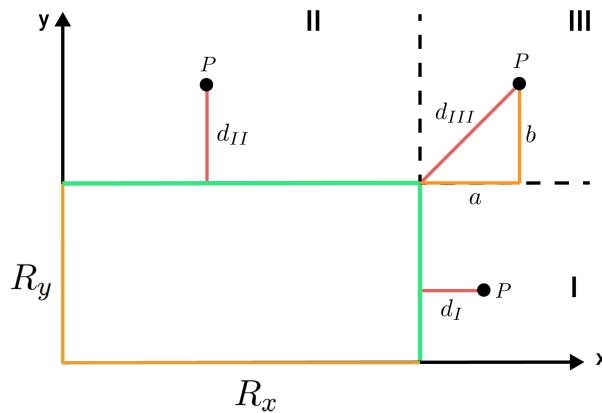
Zunächst sei ein Rechteck R , welches mit seinem Mittelpunkt im Ursprung zentriert und achsenparallel ist und dessen halbe Breite mit R_x und halbe Höhe mit R_y bezeichnet werden:



Für ein solches Rechteck kann zur Abstandsberechnung seine Symmetrie ausgenutzt werden.

Wie in der Grafik beispielhaft angedeutet, existiert für jeden Punkt P' , mit Ortsvektor $\vec{r}_p = \begin{pmatrix} p_x \\ p_y \end{pmatrix}$ und einem Abstand d zum Rechteck R ein Punkt P im ersten Quadranten mit äquivalenten Abstand.

Die Herleitung der SDF kann also anhand des ersten Quadranten erfolgen. Dieser lässt sich in die drei folgenden Zonen einteilen:



In Zone I berechnet sich der Abstand d_I von P zum Rechteck mit:

$$d_I = p_x - R_x$$

In Zone II gilt für d_{II} :

$$d_{II} = p_y - R_y$$

In Zone III handelt es sich beim Abstand d_{III} um die Distanz zwischen dem Punkt P und der Ecke des Rechtecks. Dieser lässt sich wie folgt mithilfe des Satz des Pythagoras berechnen:

$$d_{III} = \sqrt{a^2 + b^2}$$

Hierbei gilt $a = p_x - R_x$ und $b = p_y - R_y$, es folgt also für d_{III} :

$$d_{III} = \sqrt{(p_x - R_x)^2 + (p_y - R_y)^2}$$

Diese Fallunterscheidung für die drei verschiedenen Zonen ist allerdings für eine performante Implementierung der SDF hinderlich. Besser wäre es eine einzige Formel zu finden, welche für alle Zonen gleichermaßen gilt.

Betrachtet man die Formel für d_{III} und die restlichen obigen Formeln, stellt man fest, dass die a -Komponente aus d_{III} der Formel für d_I und die b -Komponente der Formel für d_{II} entspricht. Würde a nun für Punkte in Zone II zu Null und b für Punkte in Zone I zu Null evaluiert, könnte d_{III} gleichermaßen für alle Zonen gelten. Mit Blick auf die Grafik ergibt sich in beiden dieser Fälle für die jeweiligen Komponenten ein negativer Wert. Mithilfe einer \max -Funktion, welche den größeren zweier Werte zurückgibt, lässt sich so eine, im ersten Quadranten allgemeingültige, Formel aufstellen:

$$d_{\text{Rechteck}} = \sqrt{(\max(p.x - R_x, 0))^2 + (\max(p.y - R_y, 0))^2}$$

Das Rechteck kann ebenfalls in Vektorschreibweise angegeben werden:

$$\vec{R} = \begin{pmatrix} R_x \\ R_y \end{pmatrix}$$

Die Komponenten a und b lassen sich nun allgemeingültig für alle Quadranten als Vektor \vec{q} wie folgt berechnen:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \text{abs}(p_x) \\ \text{abs}(p_y) \end{pmatrix} - \begin{pmatrix} R_x \\ R_y \end{pmatrix} = \vec{q}$$

Durch das Verwenden der absoluten Werte der Komponenten von \vec{r}_p wird jeder beliebige Punkt auf einen Punkt mit äquivalentem Abstand im ersten Quadranten übertragen. Die Länge von \vec{q} entspricht dabei der Formel von d_{III} :

$$|\vec{q}| = \sqrt{(|p_x| - R_x)^2 + (|p_y| - R_y)^2}$$

In Vektorschreibweise gilt folgende Formel also für beliebige, im Ursprung zentrierte und achsenparallele Rechtecke:

$$d_{\text{Rechteck}} = |\max(\text{abs}(\vec{r}_p) - \vec{R}, 0)| = |\max(\vec{q}, 0)|$$

Die \max -Funktion wird im Folgenden immer komponentenweise angewandt.

Allerdings handelt es sich hierbei noch nicht um eine vollständige SDF, diese sollte für Punkte innerhalb des Quaders negative Werte zurückgeben.

Die einzelnen Komponenten von \vec{q} werden für Punkte innerhalb des Quaders zwar negativ, jedoch wurden diese Werte ja explizit durch $\max(\vec{q}, 0)$ eliminiert.

Daher seien nun eine Funktion \maxcomp , welche für jeden Vektor dessen maximale Komponente, und eine Funktion \min , welche den geringeren zweier Werte zurückgibt. Mithilfe dieser Funktionen lässt sich nun eine tatsächliche SDF für ein Rechteck definieren:

$$\text{sdf}_{\text{Box}} = |\max(\vec{q}, 0)| + \min(\maxcomp(\vec{q}), 0)$$

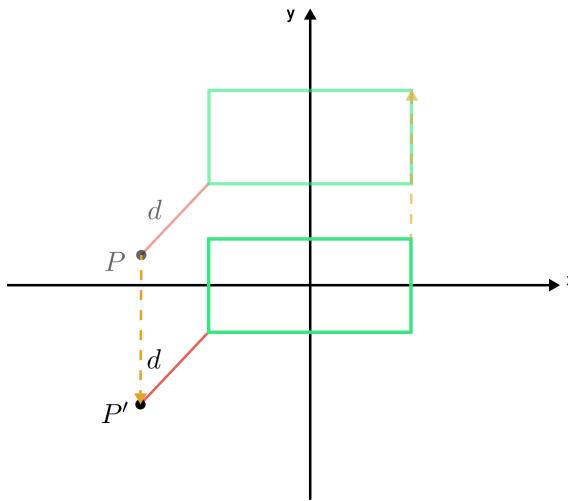
Diese gibt für einen Punkt P außerhalb des Quaders dessen minimalen Abstand zum Primitiv zurück, da $\min(\maxcomp(\vec{q}), 0)$ in diesem Fall zu Null evaluiert. Umgekehrt wird die größte Vektorkomponente von \vec{r}_p zurückgegeben, sollte der Punkt innerhalb des Rechtecks liegen. Das Vorzeichen dieser Komponente ist dabei immer negativ.

Nachdem die Vektorschreibweise dieser Formel keine Einschränkungen bei den Dimensionen vorgibt, ist diese SDF nicht nur in zwei oder drei, sondern $n \in N$ Dimensionen gültig. Eine schöne Visualisierung einer so erzeugten, vierdimensionalen Box findet sich in [diesem Video](#).

3.2.3 Transformation von SDFs

Auch wenn die SDF eines Quaders unabhängig von der Dimension ist, ist sie dafür umso eingeschränkter was die Position des Quaders anbelangt. Wer zudem SDFs für weitere Primitive aus beispielsweise [Ingo Quilez' Blog](#) in seine Anwendung einbauen möchte, stellt fest, dass diese in der Regel keine Parameter für die Position des Primitivs enthalten und nur im Ursprung gültig sind.

Was also tun, wenn die Primitive an anderen Positionen im Raum dargestellt werden sollen? Es sei daran erinnert, dass beim Raymarching lediglich der minimale Abstand von P zur Szene relevant ist. Für *Euklidische Transformationen*, also Transformationen, welche bei gleicher Anwendung auf zwei Punkte deren Abstand unverändert lassen, wie beispielsweise *Translationen* oder *Rotationen*, kann also anstatt des Abstands vom Punkt P zum transformierten Primitiv der Abstand zwischen dem Primitiv im Ursprung und eines umgekehrt transformierten Punktes P' berechnet werden:



In der Anwendung wird hierzu die aus der Vorlesung bekannte Rotationsmatrix nach der z-y'-x'-Konvention transponiert um die inverse Rotation zu erhalten. Auch die Translationen werden mit einem negativen Vorzeichen umgekehrt. Hier ist zu beachten, dass aufgrund der umgekehrten Reihenfolge, für eine nachgelagerte Translation des Primitivs, der Punkt bereits vor der Rotation translatiert werden muss:

```
float3 rotateAndTranslate(float3 v, float3 rot, float3 trans){
    rot = radians(rot);
    float a = rot.x;
    float b = rot.y;
    float g = rot.z;

    float4x4 rotation = {cos(b)*cos(g),  cos(a)*sin(g)+sin(a)*sin(b)*cos(g),  sin(a)*sin(g)-cos(a)*sin(b)*cos(g),  0,
                          -cos(b)*sin(g),  cos(a)*cos(g)-sin(a)*sin(b)*sin(g),  sin(a)*cos(g)+cos(a)*sin(b)*sin(g),  0,
                          sin(b),          -sin(a)*cos(b),                      cos(a)*cos(b),                      0,
                          0,                0,                            0,                            1 };

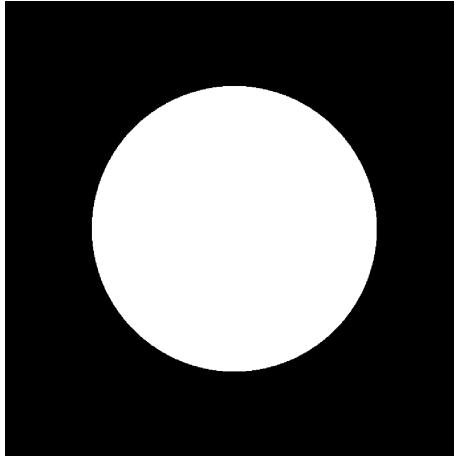
    float4x4 translation = {1, 0, 0, -trans.x,
                           0, 1, 0, -trans.y,
                           0, 0, 1, -trans.z,
                           0, 0, 0, 1};

    float4 v4 = float4(v, 1);
    float4 v_new = mul(rotation,mul(translation,v4));
    return float3 (v_new.x/v_new.w, v_new.y/v_new.w, v_new.z/v_new.w);
}
```

Ein interessantes Detail mit Bezug zur Vorlesung ist, dass Unity unter der Haube Quaternionen für Drehungen benutzt. Will man also die Winkel aus dem Unity-GUI in obiger Transformationsmatrix benutzen, müssen diese zuerst in Euler-Winkel zurückkonvertiert werden.

4 Shading

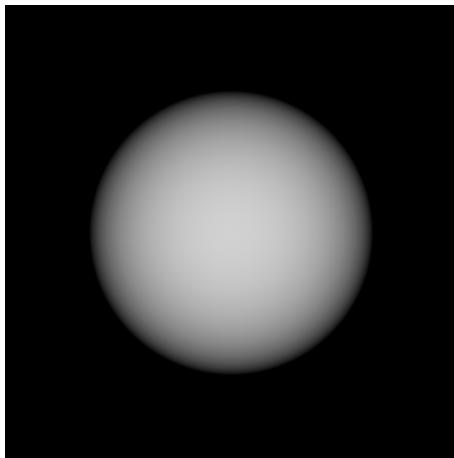
Mit obigem Algorithmus, den SDFs und deren Transformationen lassen sich nun bereits einige Primitive an beliebigen Positionen im Raum darstellen. Die entstehenden Bilder wirken allerdings noch sehr zweidimensional:



Das liegt daran, dass bisher lediglich der erste Schritt der Bildsynthese durchgeführt wurde, die Verdeckungsberechnung. Neben der reinen Existenzfrage eines Primitivs an einem bestimmten Pixel, ist selbstverständlich auch das Aussehen dieses Pixels, in Form einer Kombination der jeweiligen Materialeigenschaften mit der Lichtsituation, interessant. Die für die Ermittlung der lokalen Beleuchtung nötigen Berechnungen werden als *Shading* bezeichnet.

4.1 Pseudo-Depth-Mapping

Raymarching ermöglicht extrem einfach die Tiefe des angezeigten Primitivs im Bild darzustellen, da ja für jeden Pixel der Abstand d_g zwischen dem Punkt auf der Primitivoberfläche und dem Augpunkt bekannt ist. Dieser lässt sich beispielsweise zwischen 0 und dem Maximalabstand d_{max} als Graustufenwert interpolieren:



Der "Pseudo"-Zusatz stammt daher, dass hier der Abstand des Punktes zur Kamera benutzt wird. Für einen tatsächlichen Z-Buffer wird allerdings die z-Koordinate des Punktes im Raum genutzt. Diese ließe sich diese allerdings enorm simpel berechnen.

4.2 Normalenberechnung

Für etwas weiterführende Shading-Techniken wird in der Regel für den zu shadenden Punkt auf dem Primitiv dessen Normale, also ein Vektor, welcher mit dem Fuß im entsprechenden Punkt und senkrecht zur Primitivoberfläche steht, benötigt. In der Regel liegt dieser Vektor in normierter Form vor.

Bei SDFs kann dieser Normalenvektor mit dem Gradienten der SDF berechnet werden.

Der Gradient einer Funktion weist bekanntlich immer in die Richtung der größten Wertänderung. Da SDFs den Abstand eines Punktes zu einem Primitiv angeben, welcher für Punkte im Inneren des Körpers negativ und für Punkte nahe der Oberfläche immer kleiner wird, je näher diese an die Oberfläche rücken, kann man sich vorstellen, dass die maximale Abstandsvergrößerung senkrecht von der Oberfläche weg erfolgt. Der Gradient der SDF f im Punkt p entspricht somit der Normalen von p :

$$\nabla f(p) = \left(\frac{df(p)}{dx}, \frac{df(p)}{dy}, \frac{df(p)}{dz} \right) = \vec{n}_p$$

Da die exakte Berechnung des Gradienten je nach SDF nicht besonders trivial ist, wird meist vom *Vorwärtsdifferenzquotienten* Gebrauch gemacht:

$$\frac{df(p)}{dx} = \frac{f(p + (h, 0, 0)) - f(p)}{h}$$

Als Differenz h zwischen p und dem Sample-Punkt dient die bekannte Konstante ϵ , da h möglichst klein gewählt werden sollte, um den Fehler der Annäherung möglichst gering zu halten, ohne jedoch in Rundungsfehler aufgrund der Gleitkommaverarbeitung zu laufen. Die Division durch die Differenz ϵ kann entfallen, da diese gleichermaßen für alle partiellen Differenzquotienten erfolgt und bei dem normierten Normalenvektor lediglich die Richtung, nicht der tatsächliche Wert des Gradienten ausschlaggebend ist.

Der Normalenvektor \hat{n}_p berechnet sich also mit:

$$\begin{aligned} \vec{n}_p &= \begin{pmatrix} f(p + (\epsilon, 0, 0)) - f(p) \\ f(p + (0, \epsilon, 0)) - f(p) \\ f(p + (0, 0, \epsilon)) - f(p) \end{pmatrix} \\ \hat{n}_p &= \frac{\vec{n}_p}{|\vec{n}_p|} \end{aligned}$$

Etwas genauere Annäherungen an den Gradienten liefert der *zentrale Differenzquotient*, mit dem sich \vec{n}_p wie folgt berechnet:

$$\vec{n}_p = \begin{pmatrix} f(p + (\epsilon, 0, 0)) - f(p - (\epsilon, 0, 0)) \\ f(p + (0, \epsilon, 0)) - f(p - (0, \epsilon, 0)) \\ f(p + (0, 0, \epsilon)) - f(p - (0, 0, \epsilon)) \end{pmatrix}$$

Dieser hat gegenüber dem Vorwärtsdifferenzquotienten allerdings einen gewissen Performance-Nachteil, da der Wert der SDF für p bereits bekannt ist, die SDF hier also einmal mehr evaluiert werden muss.

4.3 Blinn-Phong-Modell

Um nun bei der Darstellung der Primitive die Lichtsituation der Szene und die jeweiligen Materialeigenschaften zu berücksichtigen wurde in der Anwendung auf das Blinn-Phong-Modell zurückgegriffen. Dieses basiert zwar nicht auf physikalischen Prinzipien, erzielt aber mit recht einfachen Mitteln sehr ansehnliche Ergebnisse.

Nachdem das Blinn-Phong-Modell nicht exklusiv für Raymarching verwendet wird, sondern im Gegenteil eine der beliebtesten Shading Techniken ist, soll an dieser Stelle nur eine recht oberflächliche Betrachtung erfolgen:

Zunächst muss die Szene um mindestens ein Punktlicht erweitert werden. Dieses wird durch seine Position im Raum P_{Licht} sowie eine Angabe der Lichtintensität I_{in} definiert.

Im Phong-Modell setzt sich das Shading immer aus einer ambienten, einer diffusen und einer spiegelnden Komponente zusammen (I_{amb} , I_{diff} , I_{spec}).

Diese Komponenten werden zur finalen Farbe I_{out} aufaddiert. Dabei werden die diffusen und spiegelnden Anteile zuerst für alle Lichter der Szene addiert, bevor abschließend der globale ambiente Anteil hinzugefügt wird.

$$I_{\text{out}} = I_{\text{amb}} + \sum_i I_{\text{diff}} + I_{\text{spec}}$$

Die am einfachsten zu berechnende Komponente ist I_{amb} . Diese kompensiert für das Fehlen von globaler Beleuchtung, also das Einbeziehen des von Objekten der Szene reflektierten Lichts. Stattdessen wird einfach ein gewisses Licht-“Offset” hinzugefügt, welches daher auch keine Rücksicht auf die Position der Lichtquellen nimmt:

$$I_{\text{amb}} = I_a k_{\text{ambient}}$$

Bei I_a handelt es sich um die Intensität des Umgebungslichts, k_{ambient} bildet die Materialkonstante des Primitivs für ambiente Reflexionen. Die Intensität I_a wird global für alle Primitive gleich modelliert.

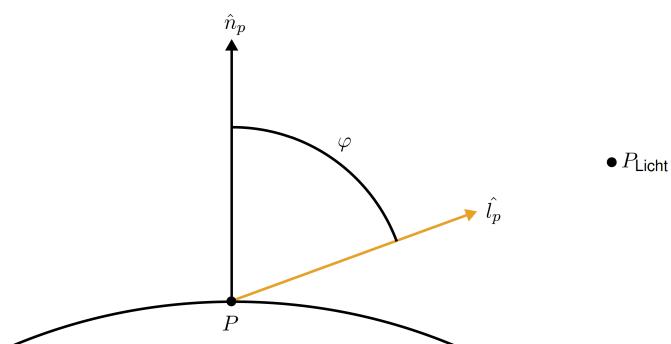
Die diffuse Komponente berechnet sich durch:

$$I_{\text{diff}} = I_{\text{in}} k_{\text{diff}} \cos\varphi$$

mit

$$\cos\varphi = \hat{l}_p \cdot \hat{n}_p$$

wobei \hat{l}_p dem normierten Richtungsvektor vom Punkt P zur Position des Punktlichts P_{Licht} entspricht. Bei k_{diff} handelt es sich erneut um eine diffuse Materialkonstante, welche auch in der Anwendung für jedes Primitiv individuell gewählt wird.



Die Stärke der diffusen Komponente ist also von der Intensität des Punktlichts, der diffusen Materialkonstante und dem Winkel φ zwischen der Normalen des Punktes und des einfallenden

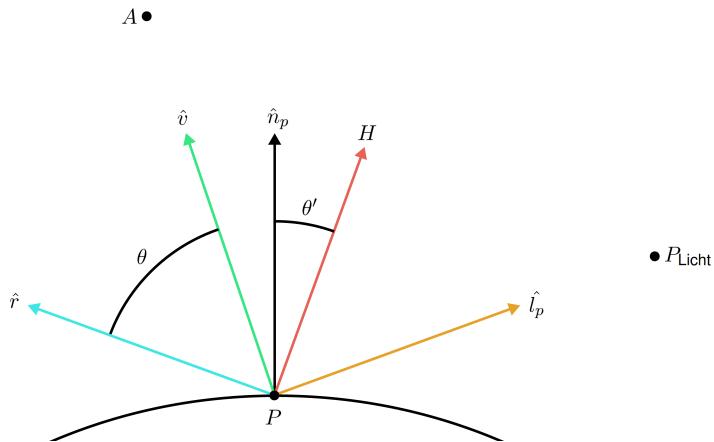
Lichtstrahls abhängig. Dabei fällt die diffuse Komponente umso größer aus je geringer φ ist, je geradliniger das Licht also auf die Oberfläche trifft.

Bei der spiegelnden Komponente spielt hingegen auch der Standpunkt des Betrachters (also der Augpunkt der Kamera) eine Rolle. Sie berechnet sich mit:

$$I_{\text{spec}} = I_{\text{in}} k_{\text{spec}} (\cos\theta)^h$$

$$\cos\theta = \hat{r} \cdot \hat{v}$$

Der Winkel θ entspricht dabei dem Winkel zwischen dem normierten Richtungsvektor \hat{r} der Reflexion des Lichtstrahls und dem normierten Richtungsvektor \hat{v} vom Augpunkt der Kamera zum Punkt P . Der Cosinus dieses Winkels wird dabei natürlich wieder maximal, je kleiner dieser Winkel ist, die Reflexion wird also umso heller, je direkter man in die Reflexionsrichtung blickt.

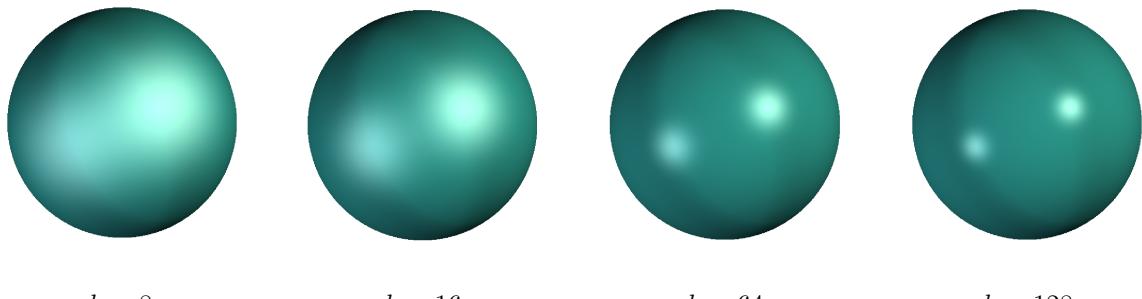


Die Berechnung dieses Winkels ist in der Praxis recht rechenintensiv, weshalb eine Weiterentwicklung, das Blinn-Phong-Modell, anstatt θ den Winkel θ' zwischen der Normalen \hat{n}_p und der Winkelhalbierenden H (auch als *Halfway Vektor* bezeichnet) zwischen \hat{v} und \hat{l}_p , verwendet:

$$H = \frac{\hat{l} + \hat{v}}{|\hat{l} + \hat{v}|}$$

$$\theta' = H \cdot \hat{n}_p$$

Der sogenannte *Phong Exponent* h (auch als *Specular Hardness* bezeichnet) sorgt durch die wiederholte Multiplikation des Cosinus Werts (welcher in der Regel < 1 ist) mit sich selbst dafür, dass die Reflexion für größere Exponenten immer kleiner und schärfer ausfällt.



$h = 8$

$h = 16$

$h = 64$

$h = 128$

Der Winkel θ' ist dabei annähernd halb so groß wie θ , weshalb für ein ähnliches Ergebnis beim Blinn-Phong-Modell ein viermal so großer Wert für h gewählt werden muss.

Die Werte der Skalarprodukte zur Berechnung von $\cos\varphi$ und $\cos\theta'$ sollten zudem auf den Bereich $[0, 1]$ eingeschränkt werden, da negative Werte die jeweilige Komponente negativ werden lassen würden und somit "Licht absorbiert" würde.

Die Lichtintensitäten I_a und I_{in} sind in der Anwendung als RGB-Farbvektoren, welche mit einem Intensitätsfaktor skalarmultipliziert werden, gegeben. Sie können also als Lichtfarbe aufgefasst werden. Es darf dabei das eigentliche RGB-Maximum von 1, bzw. 255 überschritten werden, was für realistisch dunkle Schatten sorgt. Zur Vereinfachung ist in der Anwendung die Lichtfarbe des ambienten Lichts als weiß angenommen.

Die Materialkonstanten werden in der Anwendung mit entsprechenden Farbvektoren skalarmultipliziert. Zur Vereinfachung wird eine einheitliche "Objektfarbe" sowohl für die ambiente, als auch die diffuse Komponente verwendet. Für die spiegelnde Konstante wird angenommen, dass das einfallende Licht ohne Farbveränderung reflektiert wird. Je nach Anwendung kann es sinnvoll sein k_{diff} und k_{spec} komplementär zu halten.

Zudem kann das Abfallen der Lichtstärke mit zunehmender Entfernung modelliert werden, indem (lose vom Inverse Square Law inspiriert) die Lichtintensität durch das Quadrat der Länge des Vektors \vec{v} (dem Abstand zwischen P und P_{Licht}) dividiert wird:

```
float3 diffAndSpec(float3 p, float3 pNormal, float3 pColor, float3 lightPos, float distanceToLight, float brightness, float diffuse,
float specular, float power){

    float kd = diffuse;
    float ks = specular;

    // falloff light intensity
    float3 lIn = brightness / pow(distanceToLight,2);
    // vector from Point to Lightsource (normalized)
    float3 l = lightPos - p;
    l = normalize(l);
    // vector from Point to Camera (normalized)
    float3 v = mul(CameraCoord_to_WorldCoord, float4(0,0,0,1)).xyz - p;
    v = normalize(v);

    // Halfway Vektor
    float3 h =normalize(l+v);

    // Blinn-Phong model for diff and spec
    float3 lightDiffuse = lIn * kd * pColor * max(dot(l,pNormal),0);
    float3 lightSpecular = lIn * ks * (pow(max(dot(h, pNormal),0), power));

    return lightDiffuse + lightSpecular;
}

float3 phongLight(float3 p, float3 pNormal, float3 pColor, float diffuse, float specular, float power){
    float3 pointlight = 0;
    // add diff and spec of all lights
    for(int i = 0; i<lightCount; i++){
        Light light = lights[i];
        float distanceToLight = distance(p, light.position);
        float3 diffSpec = diffAndSpec(p, pNormal, pColor, light.position, distanceToLight, light.brightness, diffuse, specular, power);
        pointLight += diffSpec;
    }
    pointLight += pColor * ambientLightIntensity;
    return pointLight;
}
```

5 Globale Lichtverteilung

Das Blinn-Phong-Modell erzielt bereits gute Beleuchtungsergebnisse bei den einzelnen Primitiven, jedoch stellt man fest, dass beispielsweise eine Kugel nach wie vor keinen Schatten auf eine Groundplane wirft. Das liegt daran, dass das Shading lediglich die Auswirkungen der Lichter auf die Primitive, nicht aber die der Primitive untereinander behandelt. Letzteres geschieht bei der Berechnung der globalen Lichtverteilung. Hierzu zählt der bereits angesprochene Schattenwurf, aber auch beispielsweise die Reflexionen der anderen Objekte in glänzenden Objekten (der Grund warum Primitive mit $k_{diff} = 0$ und $k_{spec} = 1$ bis auf die Reflexionen schwarz erscheinen).

In dieser Arbeit soll beispielhaft der Schattenwurf behandelt werden.

5.1 Einfache Schatten

Schattenwurf mithilfe von Raymarching zu erzeugen ist grundsätzlich extrem simpel. Für jeden Punkt P , an dem ein Primitiv erkannt wurde, kann bestimmt werden, ob ein, von der jeweiligen Punktlichtquelle von P_{Licht} nach P ausgesendeter, Strahl P erreicht oder ob es zuvor zu einer Kollision mit einem Primitiv kommt, der Punkt also im Schatten liegt.

Hierzu wird der gleiche Raymarching-Algorithmus verwendet wie schon zum Sichtbarkeitsentscheid. Einzig der Richtungsvektor muss in diesem Fall umgekehrt von der Lichtquelle hin zum Punkt aufgestellt werden, da der minimale Abstand bei P natürlich sofort kleiner als ϵ wäre. Als maximale Distanz kann die Länge des Vektors $\overrightarrow{PP_{\text{Licht}}}$ verwendet werden. Wird diese Distanz erreicht, wird der Punkt von der Lichtquelle beleuchtet, falls es zuvor zu einer Kollision kommt, liegt der Punkt im Schatten. Es ist außerdem wichtig nicht die gesamte Distanz zwischen Punkt und Lichtquelle als Maßstab zu nehmen, sondern diesen um einen gewissen Puffer zu reduzieren. Der Strahl nähert sich dem Primitiv nun in einem anderen Winkel, weshalb es durch den ϵ -Wert etwas früher oder später als gewünscht zu einer Kollision kommen kann, was zu sichtbaren Artefakten führt:

```
float basicShadow( float3 p, float3 pNormal, float3 lightPos){
    // create new Ray from Light to Point
    float3 direction = p - lightPos;
    direction = normalize(direction);
    float distanceToLight = distance(p, lightPos);

    Ray shadowRay = CreateRay(lightPos, direction);

    float globalDistance = 0;
    int maxSteps = 50;
    int steps = 0;

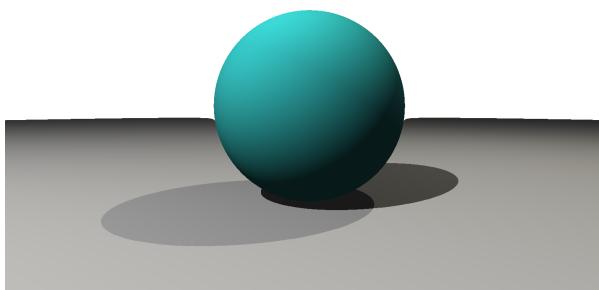
    while(globalDistance + epsilon*50 < distanceToLight && steps < maxSteps){

        float4 evalSceneForPoint = minDst(shadowRay.origin)[0];

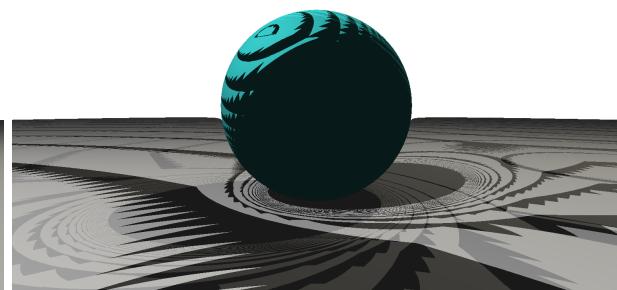
        float dist = evalSceneForPoint.w;
        steps++;

        // check if collision occurred
        if(dist < epsilon){
            return 0;
        }

        // update new point position
        shadowRay.origin += shadowRay.direction * dist;
        // update current distance
        globalDistance += dist;
    }
    return 1;
}
```



mit Puffer



ohne Puffer

5.2 Weiche Schatten

Auch wenn dieser Algorithmus in der Lage ist mit einfachen Mitteln genaue Schatten zu erzeugen, wirken diese dennoch unnatürlich scharf. In der Realität handelt es sich bei einer Lichtquelle nie um einen einzelnen Punkt, sondern um ein Objekt mit einer gewissen Größe. Die Lichtquelle wird also oft nicht aus jedem Winkel vollständig von einem Objekt verdeckt, weshalb um den tatsächlichen Kernschatten herum ein weicherer Halbschatten auftritt. Dieser kann beim Rendern aber auch für eine Punktlichtquelle erzeugt werden, was zwar nicht der Realität entspricht, jedoch dennoch optisch überzeugend wirkt.

Anstatt nun binär entweder einen Schatten für Strahlen mit Kollision oder keinen Schatten für Strahlen ohne Kollision zurückzugeben, wird zusätzlich auch für Punkte, an denen es nur beinahe zu einer Kollision kommt, ein Schatten erzeugt. Dieser soll umso dunkler werden, je näher dabei der Abstand zum Kollisionsobjekt war (Auslaufen des Schattens) und je näher dies am zu schattierenden Punkt geschieht. Letzteres imitiert das Verhalten eines tatsächlichen Halbschattens.

Betrachtet man den Schatten-Algorithmus erkennt man, dass beide dieser Distanzen jederzeit als $dist$ und ...Differenz zwischen d_g und Abstand zwischen Punkt und Lichtquelle verfügbar sind. Das Hinzufügen eines extrem simplen Terms erzeugt somit mit kaum gesteigertem Rechenaufwand weiche Schatten:

```
float softShadow(float3 p, float3 pNormal, float3 lightPos, float k){
    // create new Ray from Light to Point
    float3 direction = p - lightPos;
    direction = normalize(direction);
    float distanceToLight = distance(p, lightPos);

    Ray shadowRay = CreateRay(lightPos, direction);

    float globalDistance = 0;
    float shadow = 1;
    int maxSteps = 50;
    int steps = 0;

    while(globalDistance + epsilon*5 < distanceToLight && steps < maxSteps){

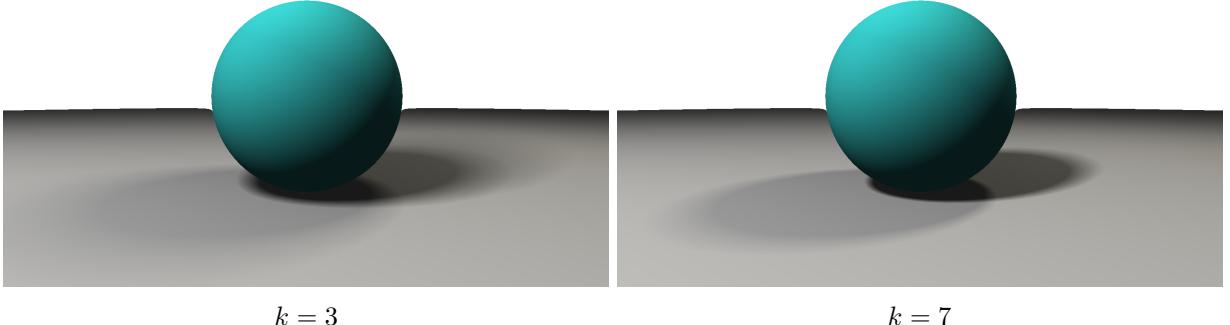
        float4 evalSceneForPoint = minDst(shadowRay.origin)[0];

        float dist = evalSceneForPoint.w;
        steps++;

        // get distance between point-to-shade and current ray-end
        float dstObjP = distanceToLight - globalDistance;

        // get shadow intensity
        shadow = min(shadow, k * dist/dstObjP);

        // update new point position
        shadowRay.origin += shadowRay.direction * dist;
        // update current distance
        globalDistance += dist;
    }
    return shadow;
}
```



$k = 3$

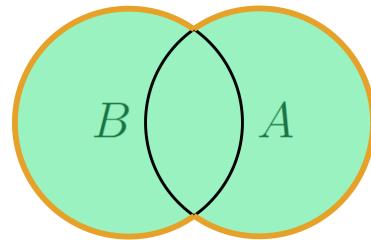
$k = 7$

Der Faktor k bestimmt dabei die Schärfe des Schattens, wobei ein höherer Wert für k mit einer schärferen Grenze des Schattens korrespondiert.

6 Alternativen zur Minimumsfunktion

6.1 Mengenoperationen

Befinden sich in der Szene mehrere Primitive, werden bei der Berechnung der Umgebungsverdeckung für einen Punkt P deren SDFs eine nach der anderen evaluiert und die erhaltene Distanz mithilfe der Minimumsfunktion \min mit der Distanz des Vorgängerprimitivs verglichen, um so die minimale Distanz aller Primitive zum Punkt P zu erhalten. Diese minimale Distanz d_{\min} wird anschließend mit der Konstante ϵ verglichen um für den Fall $d_{\min} < \epsilon$ festzustellen, dass am Punkt P ein Primitiv existiert. Überlappen sich mehrere Primitive, wird dadurch immer das näher an P , also "außen" gelegene, angezeigt.



Betrachtet man diesen Vorgang als Mengenoperation entspricht obiger Fall der *Vereinigung* von zwei Mengen, bzw. SDFs. Es wären somit aber auch andere Mengenoperationen wie *Durchschnitt* oder *Differenz* denkbar.

Relativ intuitiv vorstellbar ist die Umsetzung einer Durchschnittsoperation. Dabei soll, angenommen es existieren ein Primitiv A mit SDF_A und ein Primitiv B mit SDF_B , lediglich die Schnittmenge von A und B angezeigt werden. Die jeweiligen SDFs werden wie gewohnt für einen Punkt P evaluiert und geben dabei eine Distanz $dist_A$, bzw. $dist_B$ zurück. Anstatt der \min -Funktion, existiert nun eine andere Funktion f , welche von den Distanzen $dist_A$, und $dist_B$ die jeweilige Distanz d_f zurückgibt, welche nur für den Fall, dass P sowohl von A als auch B weniger als ϵ entfernt ist, eine Kollision angeben soll. Die Distanz d_f soll also nur für den Fall $dist_A < \epsilon$ und $dist_B < \epsilon$ kleiner als ϵ werden. Bei f handelt es sich also um die Maximumsfunktion \max .

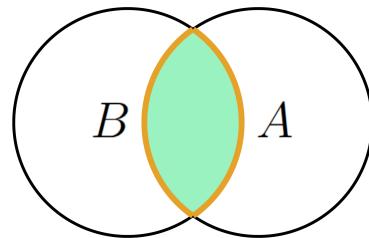
Es lassen sich dabei die folgenden Fälle betrachten:

$$\text{dist}_A > \epsilon \text{ und } \text{dist}_B > \epsilon \longrightarrow \max(\text{dist}_A, \text{dist}_B) > \epsilon$$

$$\text{dist}_A > \epsilon \text{ und } \text{dist}_B < \epsilon \longrightarrow \max(\text{dist}_A, \text{dist}_B) > \epsilon$$

$$\text{dist}_A < \epsilon \text{ und } \text{dist}_B > \epsilon \longrightarrow \max(\text{dist}_A, \text{dist}_B) > \epsilon$$

$$\text{dist}_A < \epsilon \text{ und } \text{dist}_B < \epsilon \longrightarrow \max(\text{dist}_A, \text{dist}_B) < \epsilon$$

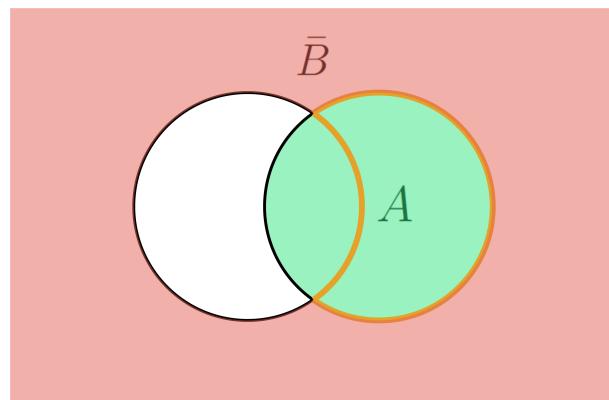


Mithilfe dieser Herleitung lässt sich auch die Bildung der Differenzoperation relativ leicht erklären. Es existieren erneut zwei Primitive A und B mit SDF_A und SDF_B , sowie deren dist_A und dist_B . Die SDFs evaluieren bekanntlich außerhalb des Primitivs zu einem positiven, innerhalb des Primitivs zu einem negativen Wert. Man kann sich also vorstellen, dass ein Primitiv negiert werden kann, indem der, durch die SDF ermittelten, Distanz ein negatives Vorzeichen vorangestellt wird.

Bildet man nun die Schnittmenge zwischen A und dem negierten B , existiert A also nur für Bereiche, die außerhalb von B liegen. Mit der Funktion

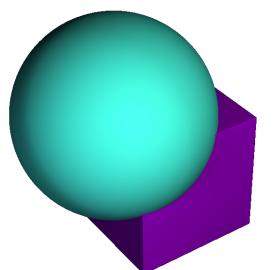
$$\max(\text{dist}_A, -\text{dist}_B)$$

wird also wie gewünscht B von A subtrahiert.



Nachdem in der Anwendung mehr als nur die Distanz verarbeitet werden muss, sind hier die Operationen als *if*-Anweisungen umformuliert:

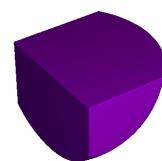
```
// Subtract
if(prim.combinationMode==1){
    if(minDistance < -primDistance){
        minDistance = -primDistance;
    }
}
// Intersect
else if(prim.combinationMode==2){
    if(minDistance < primDistance){
        minDistance = primDistance;
    }
}
// Union
else{
    if(minDistance > primDistance){
        minDistance = primDistance;
        pointColor = primColor;
    }
}
```



Vereinigung



Differenz

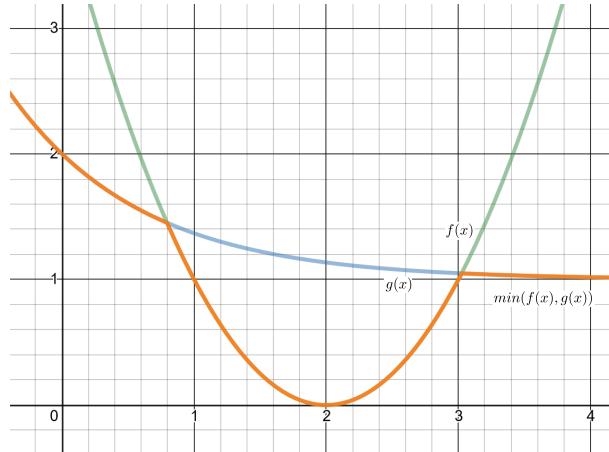


Durchschnitt

Denkt man an den bisherigen Raymarching Algorithmus zurück, war die Reihenfolge, in der die Distanzen von P zu den Primitiven miteinander verglichen wurden, nicht relevant. Dies ändert sich jedoch mit der Einführung der Mengenoperationen, da es selbstverständlich einen erheblichen Unterschied macht, ob nun beispielsweise B von A oder A von B subtrahiert wird. Um die Evaluationsreihenfolge der Primitive festlegen zu können existiert daher in der Anwendung ein "Evaluation Order"-Attribut für jedes Primitiv.

6.2 Smooth Minimum

Die Auswahl der kleinsten Distanz bei der Vereinigung mit der \min -Funktion hat einen "Nachteil". Der Übergang zwischen zwei sich überlappenden Primitiven ist immer geknickt.



Man kann sich dabei vorstellen dass zwei SDFs durch die Funktionen $f(x)$ und $g(x)$ repräsentiert werden. Die Knicke der Minimumsfunktion an den Schnittpunkten der Funktionsgraphen von f und g sind dabei gut erkennbar.

Anstatt der absoluten \min -Funktion soll nun eine geglättete Minimumsfunktion $smin$ festgelegt werden, welche, wenn die Werte von f und g nahe beieinander liegen, glatt zwischen diesen interpoliert und ansonsten regulär den minimalen der beiden Werte zurückgibt. Die Formulierung "nahe beieinander" wird mathematisch mit $f(x) - g(x) \in [-k, k]$ beschrieben.

Ein erster Ansatz für eine solche Funktion könnte beispielsweise folgendermaßen aussehen:

$$smin(F(x), g(x), k) = \min(f(x), g(x)) - w(f(x), g(x), k)$$

Bei $w(f(x), g(x), k)$ handelt es sich um eine Glättungsfunktion, deren Wert von der Minimumsfunktion subtrahiert wird. Für diese sollte also gelten:

$$\text{wenn für } x = a : f(x) - g(x) = k \rightarrow w(a) = 0$$

$$\text{wenn für } x = b : f(x) - g(x) = -k \rightarrow w(b) = 0$$

$$\text{wenn für } x = c : f(x) = g(x) \rightarrow w(c) = s$$

s entspricht dabei dem Maximalwert der Glättung, welcher am Schnittpunkt c der beiden Funktionen erreicht werden soll.

Mit k lässt sich das Intervall einstellen, in dem Glättung stattfinden soll. Sind die Beträge der Abstände zwischen $f(x)$ und $g(x)$ also $> k$, soll keine Glättung mehr erfolgen, sondern das absolute Minimum der Abstände zurückgegeben werden. Die Funktion w wird dabei gebildet mit:

$$w(f(x), g(x), k) = s h(f(x), g(x), k)^n$$

Die Wertemenge von h bewegt sich dabei im Intervall $[0, 1]$. Für $h(f(x), g(x), k)$ gilt:

$$h(f(x), g(x), k) = \begin{cases} 1 + \frac{f(x)-g(x)}{k}, & \text{wenn } x > c \\ 1 - \frac{f(x)-g(x)}{k}, & \text{wenn } x < c \end{cases}$$

Für den Fall $x = c$ wird h somit zu 1, es wird also s vollständig von der \min -Funktion subtrahiert. Für die Fälle $x = a$ oder $x = b$ erhält man für h wie erwartet 0. Für die Bereiche $|f(x) - g(x)| > k$ werden die Werte von h negativ, es wird also tatsächlich $\min(h, 0)$ in w verwendet.

Man kann die $smin$ -Funktion so auch zweiteilig, also mit einer Kurve rechts und einer Kurve links von c , betrachten. Nachdem die Funktion für einen glatten Übergang von f und g sorgen soll, sollte der Anschluss dieser beiden Teile im Punkt c also mindestens C^1 -stetig sein.

Dies ist der Fall, wenn die erste Ableitung des linken Teils im Punkt c der ersten Ableitung des rechten Teils im Punkt c entspricht, es soll also gelten:

$$f'(c) + w'_l(c) = g'(c) + w'_r(c)$$

$$f'(c) + \frac{ns(f'(c) - g'(c))(\frac{f(c)-g(c)+k}{k})^{n-1}}{k} = g'(c) - \frac{ns(f'(c) - g'(c))(\frac{-f(c)+g(c)+k}{k})^{n-1}}{k}$$

Nachdem bekanntlich $f(c) = g(c)$ gilt, kann zu

$$f'(c) + \frac{ns(f'(c) - g'(c))}{k} = g'(c) - \frac{ns(f'(c) - g'(c))}{k}$$

vereinfacht werden. Formt man die Gleichung zu

$$f'(c) - g'(c) + \frac{ns(f'(c) - g'(c))}{k} = -\frac{ns(f'(c) - g'(c))}{k}$$

um, ersetzt der Übersichtlichkeit halber $f'(c) - g'(c)$ mit u

$$u + \frac{uns}{k} = -\frac{uns}{k}$$

und formt weiter um zu

$$u = -u \frac{2ns}{k}$$

lässt sich erkennen, dass die Gleichung nur gilt, wenn

$$-\frac{2ns}{k} = 1$$

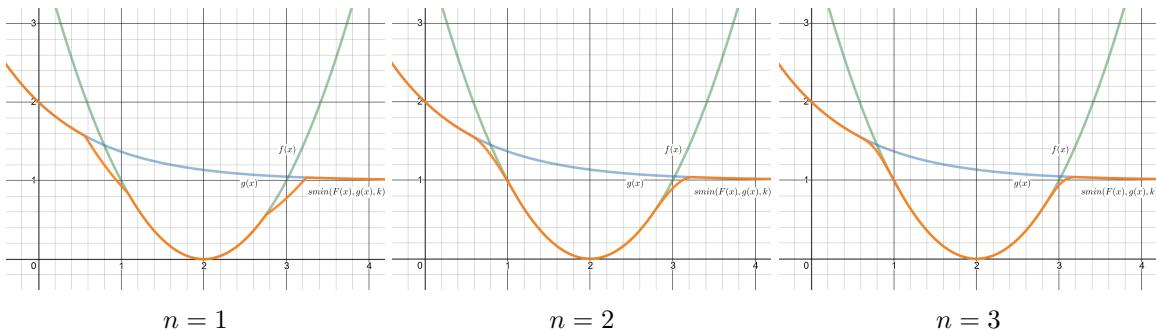
erfüllt ist. Man kann dies nun nach s auflösen und erhält

$$s = -\frac{k}{2n}$$

Damit lässt sich also eine in c C^1 -stetige Gleichung für $smin$ erzeugen.

Um allerdings Beleuchtungsartefakte zu vermeiden, kann es von Vorteil sein eine C^2 -stetige Version $smin$ zu verwenden. Dafür müsste analog die zweite Ableitung des linken Teils im Punkt c mit der zweiten Ableitung des rechten Teils in Punkt c übereinstimmen.

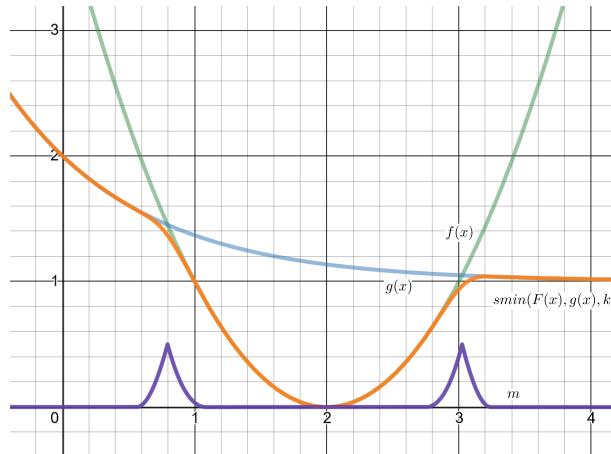
Laut der [Herleitung von Ingo Quilez](#) ist die zweite Ableitung der w -Teile nur für $n > 2$ definiert. Da n allerdings ab der ersten Ableitung nur noch als Faktor im Zähler, bzw. als Exponent einer Basis = 1 auftaucht, ist für mich an dieser Stelle der Einfluss von n auf die zweite Ableitung nicht nachvollziehbar. Trotzdem ist auch in der Praxis ersichtlich, dass die Funktion für größere n stetigere Anschlüsse zu schaffen scheint:



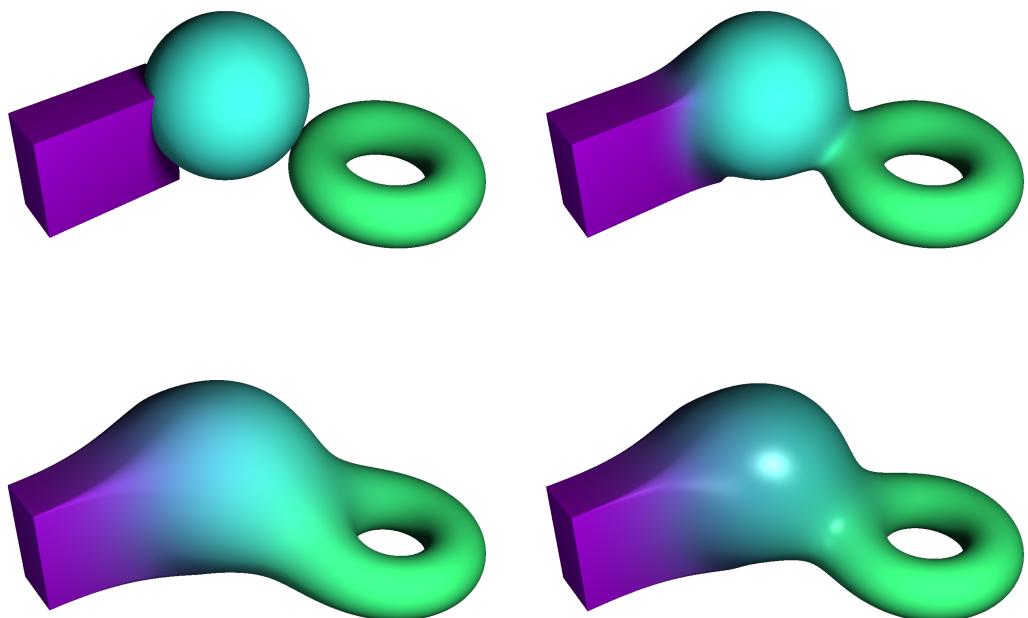
In der Anwendung kann die *smin* Funktion etwas umgeformt werden:

```
float2 cubicSmoothMin(float dstA, float dstB, float k){
    float h = max((k-abs(dstA-dstB)),0)/k;
    float m = h*h*0.5;
    return float2(min(dstA, dstB) - (k/6)*h*h*h, m);
}
```

Der Faktor m kann dabei genutzt werden um im Übergangsbereich der beiden Funktionen bzw. Primitive linear zwischen den Materialeigenschaften, wie beispielsweise der Farbigkeit zu interpolieren:



Durch Anpassen des k -Faktors können so unterschiedliche Grade der Glättung zwischen den Primitiven sowie deren Materialeigenschaften eingestellt werden:



7 Einsatzbereiche des Raymarchings

Auch wenn es mit Raymarching prinzipiell möglich ist klassische Polygon-Meshes zu rendern, ist der interessanter Anwendungsfall das Rendering von mathematisch definierten SDFs. Dies erlaubt nicht nur das Erstellen komplexer Szenen mit minimalen Speicheranforderungen, was beispielsweise bei Demoscenes, also möglichst komplexen Szenen, generiert durch möglichst kleine Executables, wie diesem beeindruckenden [sieben Kilobyte großen Beispiel](#), eine zentrale Rolle spielt, sondern bietet auch unbeschränkte Möglichkeiten der prozeduralen Generierung. So können mit SDFs von einfachen Primitiven komplexe Objekte geschaffen werden, indem diese mit den verschiedenen Mengenoperationen kombiniert oder anderen Funktionen verzerrt werden. Es können aber auch wesentlich anspruchsvollere SDFs, beispielsweise für Fraktale, bzw. deren Orbit Traps, also den Abständen der Fraktalfunktion zu einer beliebigen geometrischen Form, definiert werden. Besonders die Kombination aller dieser Techniken beschert Raymarching ein extrem breites Anwendungsgebiet vom klassischen Modelling, über prozedurale Generierung, bishin zur dreidimensionalen Darstellung von Fraktalen.

8 Ausblick

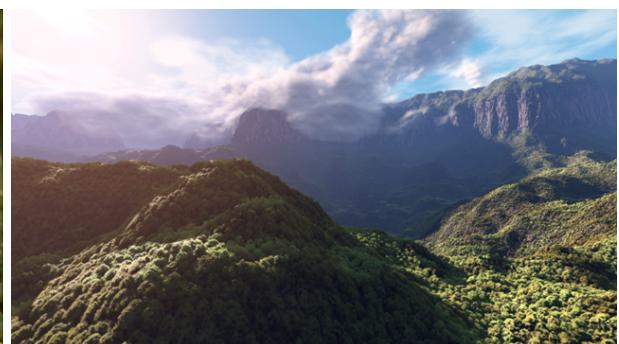
Wie aus dem obigen Abschnitt hervorgeht sind die Erweiterungsmöglichkeiten einer Raymarching Anwendung nahezu unbegrenzt.

Neben Verbesserungen beim Shading oder der globalen Lichtverteilung, wie beispielsweise das Hinzufügen globaler Beleuchtung oder Reflexionen, sind selbstverständlich auch bei der Verdeckungsberechnung mit Raymarching noch einige Erweiterungen der Arbeit denkbar. Auch wenn mit den implementierten Grundobjekten in Kombination mit den verschiedenen Mengenoperationen sowie der Smooth-Minimum-Funktion bereits das Modellieren einfacher zusammengesetzter Objekte möglich ist, wären das Untersuchen verschiedener Funktionen zur Verzerrung der Primitive und die daraus entstehenden Möglichkeiten der prozeduralen Generierung sehr interessant. Auch die Implementierung von SDFs verschiedener Fraktale wäre ein enorm spannendes Gebiet der Erweiterung.

Als Beispiel für den Umfang der Möglichkeiten der Raymarching-Technik sei an dieser Stelle erneut auf den großartigen [Blog von Ingo Quilez](#) verwiesen, in dem sich zu allen diesen Themen Beschreibungen, Codebeispiele und beeindruckende Renderings finden lassen:



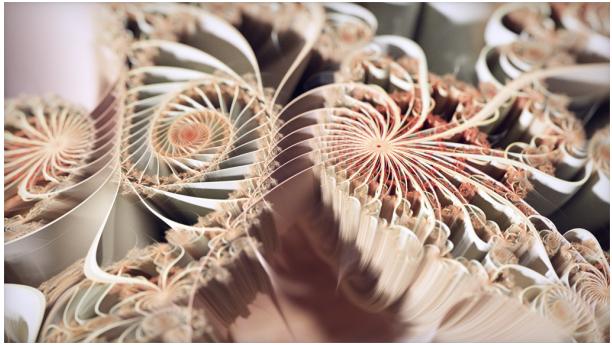
"klassisches" Modelling



prozedural generierte Landschaft



Zylinder-Julia-Mengen-Orbit-Traps



Ebenen-Julia-Mengen-Orbit-Traps

9 Links der Anwendungen

Raymarching in Unity
Interaktive Visualisierung
(Visualisierung Git-Respository)

10 Quellen

Blogs und Websites

<http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/>
<https://iquilezles.org/www/articles/distfunctions/distfunctions.htm>
<https://iquilezles.org/www/articles/smin/smin.htm>
<https://iquilezles.org/www/articles/rmshadows/rmshadows.htm>
<https://iquilezles.org/www/articles/normalsSDF/normalsSDF.htm>
<https://de.wikipedia.org/wiki/Bildsynthese>
<https://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell>
https://en.wikipedia.org/wiki/Blinn%CE%80%93Phong_reflection_model
https://en.wikipedia.org/wiki/Orbit_trap
<https://www.scratchapixel.com/>
<https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF>

Videos

Deriving the SDF of a Box
Computergrafik: 8.2 – Beleuchtung: Phong und Blinn-Phong (SoSe 2020)
Coding Adventure: Ray Marching
Ray Marching for Dummies!

Code-Repositories

<https://github.com/SebLague/Ray-Marching>
<https://www.shadertoy.com/view/XIGBW3>