Architektur und Programmierung von Grafik- und Koprozessoren

Rendering Algorithmen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Rasterisierung Algorithmus 1

```
function Rasterisierung(V,M,L,f,MV,PR,VP)
    for all v1, v2, v3 ∈ V do
        Transformationen(v1,v2,v3,MV,PR)
        Clipping(v1,v2,v3,PR)
        T ←ErzeugeDreiecke(v1,v2,v3)
        for all t ∈ T do
            Fragmente ←ScanKonvertierung(t,VP)
            for all F ∈ Fragmente do
                for all Lj ∈ Lichtquellen do
                    Beleuchte(F,M,Lj,f)
                end for
                TiefenTest(F, I)
                AlphaBlending(F, I)
            end for
        end for
    end for
end function
```

Rasterisierung - Vertex Phase

function RASTERISIERUNG(V,M,L,f,MV,PR,VP)
  for all v_1, v_2, $v_3 \in V$ **do**
    TRANSFORMATIONEN(v,MV,PR)
    CLIPPING(v,PR)
    ...
  end for
end function

1. Wende Viewing- und Perspektivische auf jedes Vertex an.
2. Clippe Vertices am Sichtbaren Frustum.

Rasterisierung - Primitive Assembly

function RASTERISIERUNG(V,M,L,f,MV,PR,VP)

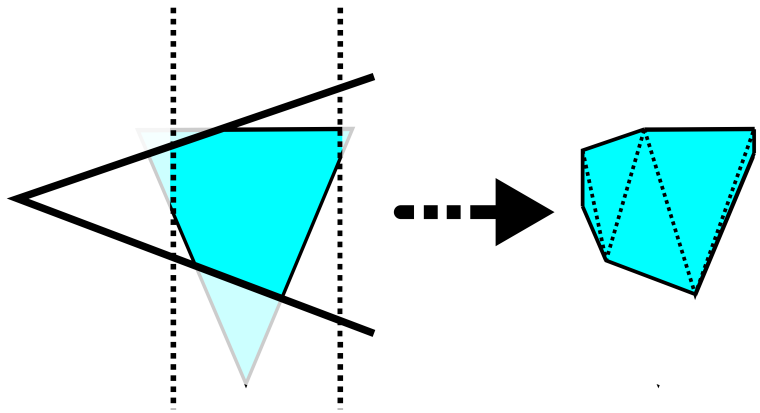
...

$T \leftarrow$ERZEUGEDREIECKE(v_1,v_2,v_3)

...

end function

3. Für jedes Eingabedreieck $\{v_1, v_2, v_3\}$, erzeuge Dreiecke, die sich *nach Clipping* ergeben.



Rasterisierung - Fragment Phase

```
function RASTERISIERUNG(V,M,L,f,MV,PR,VP)
    ...
    for all t ∈ T do
        Fragmente ←SCANKONVERTIERUNG(t,VP)
        for all f ∈ Fragmente do
            for all L_j ∈ Lichtquellen do
                BELEUCHTE(F,M,L_j,f)
            end for
            TIEFENTEST(F, I)
            ALPHABLENDING(F, I)
        end for
    end for
    ...
end function
```

4. Fragmente ergeben sich nach Scan Konvertierung.
5. Jedes Fragment wird beleuchtet, erst dann Tiefentest!

Rasterisierung Algorithmus 1 - Bemerkungen

- Manche Implementierungen führen Beleuchtungsberechnungen auf den Vertices durch und interpolieren später die resultierenden Farben.
- Der Algorithmus unterstützt sowohl opake als auch teiltransparente Geometrie. I. d. R. wird man den Tiefentest nur für opake, und Alpha-Blending nur für teiltransparente Fragmente ausführen.
- APIs (Direct3D, OpenGL, Vulkan) garantieren, dass Reihenfolge während Fragment Phase Eingabereihenfolge entspricht.

Rasterisierung Algorithmus 1 - Worst-Case Komplexität

Vorabüberlegungen

- Aus der *Primitive Assembly Phase* ergeben sich durch Clipping bis zu vier Dreiecke. Die Schleife über alle Dreiecke T lässt sich also durch $O(4)$ abschätzen.
- Die Anzahl an Fragmenten, die nach Scan Konvertierung *eines* Dreiecks entstehen, ist durch $O(VP)$ beschränkt, wobei VP die Bildschirmauflösung bezeichnet.
- Wir nehmen an, dass es für die Subroutinen Transformationen und Clipping, Beleuchte, TiefenTest, AlphaBlending und ErzeugeDreiecke jeweils $O(1)$ Algorithmen gibt. ScanKonvertierung ist durch $O(VP)$ beschränkt.

Rasterisierung - Worst-Case Komplexität

Es ergibt sich also im schlechtesten Fall für den Algorithmus Rasterisierung (1) die Komplexität:

$$O(V) \times O(4) \times (O(VP) \times O(L) + O(VP)) = O(V \times VP \times L) \quad (20)$$

Wir wollen im folgenden vereinfachend annehmen, dass die Anzahl Lichtquellen L konstant ist. Damit ergibt sich die Laufzeitkomplexität

$$O(V \times VP). \quad (21)$$

Wenn wir uns mit “Deferred Shading” beschäftigen, werden wir wieder von variabel vielen Lichtquellen ausgehen.

Rasterisierung - Algorithmus 2

Bemerkung

Unter praktischen Gesichtspunkten ist die Laufzeitkomplexität dieses Algorithmus sehr hoch. Praktische Arbeitslasten verarbeiten Millionen von Dreiecken und erzeugen Bilder mit Millionen von Pixeln.

Wir wollen eine alternative Formulierung des Algorithmus RASTERISIERUNG betrachten. Ziel: besser verstehen, welche *Umstrukturierungsmaßnahmen* auf GPUs vorgenommen werden, um den *Durchsatz* zu erhöhen.

Rasterisierung - Algorithmus 2

```
function Rasterisierung(V,M,L,f,MV,PR,VP)
    for all v ∈ V do
        Transformationen(v,MV,PR)
        Clipping(v,PR)
    end for
    T ←ErzeugeDreiecke(V)
    for all t ∈ T do
        Fragmente ←ScanKonvertierung(t, VP)
    end for
    for all F ∈ Fragmente do
        for all L_j ∈ Lichtquellen do
            Beleuchte(F,M,L_j,f)
        end for
        TiefenTest(F, I)
        AlphaBlending(F, I)
    end for
end function
```

Rasterisierung - Algorithmus 2

Struktur von Algorithmus 2:

function RASTERISIERUNG(V,M,L,f,MV,PR,VP)
    for all v $\in$ V **do** ▷ Vertex Phase
        ...
    end for
    ... ▷ Pufferung von Dreiecken
    for all t $\in$ T **do** ▷ Scan Konvertierung
        ... ▷ Pufferung aller Fragmente
    end for
    for all F $\in$ Fragmente **do** ▷ Fragment Phase
        ...
    end for
end function

Rasterisierung - Algorithmus 2

Algorithmus Rasterisierung 2 hat offensichtlich die gleiche obere **Laufzeitschranke** wie Algorithmus 1

$$O(V) \times O(VP) \times O(L). \tag{22}$$

Allerdings: Dreiecke und Fragmente werden gepuffert $\Rightarrow$ **Speicherbedarf**:

$$O(V) \times O(VP) \tag{23}$$

Bemerkung: nach der Vertex Phase müssen auch Dreiecke gepuffert werden. Asymptotisch hängt dies von V ab. Da es aber konstant mal so viele Ausgabe- wie Eingabe Vertices gibt, vernachlässigen wir dies und nehmen einfach an, dass die neuen Vertices "in der Nähe" der Eingabe Vertices gepuffert werden können.

Rasterisierung - Algorithmus 2

Bemerkungen

- Wir haben jetzt also zwei alternative Formulierungen des Rasterisierungsalgorithmus. Die zweite Formulierung hat augenscheinlich den Nachteil sehr hohen Speicherbedarfs.
- Den Ansatz, erst Vertices und dann Fragmente zu verarbeiten, nennt man in der Computergrafik “deferred”: die Verarbeitung der Fragmente wird hinausgezögert, bis (genügend) Vertices verarbeitet wurden.

Rasterisierung - PRAM Formulierung

Wir betrachten im folgenden PRAM Formulierungen des Rasterisierungsalgorithmus.

Wir wollen verstehen, an welchen Stellen die beiden Formulierungen Parallelismus exponieren und welche Auswirkungen das auf den *Durchsatz* hat.

Rasterisierung Paralleler Algorithmus 1

```
function Rasterisierung(V,M,L,f,MV,PR,VP)
    for all v1, v2, v3 ∈ V do in parallel                    ▷
        Transformationen(v1,v2,v3,MV,PR)
        Clipping(v1,v2,v3,PR)
        T ←ErzeugeDreiecke(v1,v2,v3)
        for all t ∈ T do
            Fragmente ←ScanKonvertierung(t,VP)
            for all F ∈ Fragmente do
                for all Lj ∈ Lichtquellen do
                    Beleuchte(F,M,Lj,f)
                end for
                TiefenTest(F, I)
                AlphaBlending(F, I)
            end for
        end for
    end for
end function
```

Rasterisierung Paralleler Algorithmus 1

Komplexität

- Der Parallele Algorithmus 1 ist auf der CRCW PRAM lauffähig (synchronisierte Zugriffe für TiefenTest und AlphaBlending).
- Arbeitskomplexität: $W(n) = O(V \times VP)$.
- Zeitkomplexität: $S(n) = O(1) \times O(4) \times O(VP)$.
  - $\Rightarrow$ die Zeitkomplexität beträgt $O(VP)$.

Rasterisierung Paralleler Algorithmus 2

(1/2)

```
function RASTERISIERUNG(V,M,L,f,MV,PR,VP)
    for all v ∈ V do in parallel                                    ▷
        TRANSFORMATIONEN(v,MV,PR)
        CLIPPING(v,PR)
    end for
    for all v1, v2, v3 ∈ V do in parallel                           ▷
        T ←ERZEUGEDREIECKE(v1,v2,v3)
    end for
    for all t ∈ T do in parallel                                    ▷
        Fragmente ←SCANKONVERTIERUNG(t,VP)
    end for
    ...
end function
```

Rasterisierung Paralleler Algorithmus 2

(2/2)

function RASTERISIERUNG(V,M,L,f,MV,PR,VP)
    ...
    for all F $\in$ Fragmente **do in parallel** ▷
        for all $L_j \in$ Lichtquellen **do**
            BELEUCHTE(F,M,L_j,f)
        end for
        TIEFENTEST(F, I)
        ALPHABLENDING(F, I)
    end for
end function

Rasterisierung Paralleler Algorithmus 2

Komplexität

- Arbeitskomplexität: $W(n) = O(V \times VP)$.
- Zeitkomplexität:
  $S(n) = O(1) + O(4)O(VP) + O(1) = O(VP)$.
- Die Speicherkomplexität bleibt ggü. dem seriellen Algorithmus unverändert.
- Wir können die einzelnen Phasen des Algorithmus *separat* parallelisieren.

Rasterisierung Paralleler Algorithmus 2

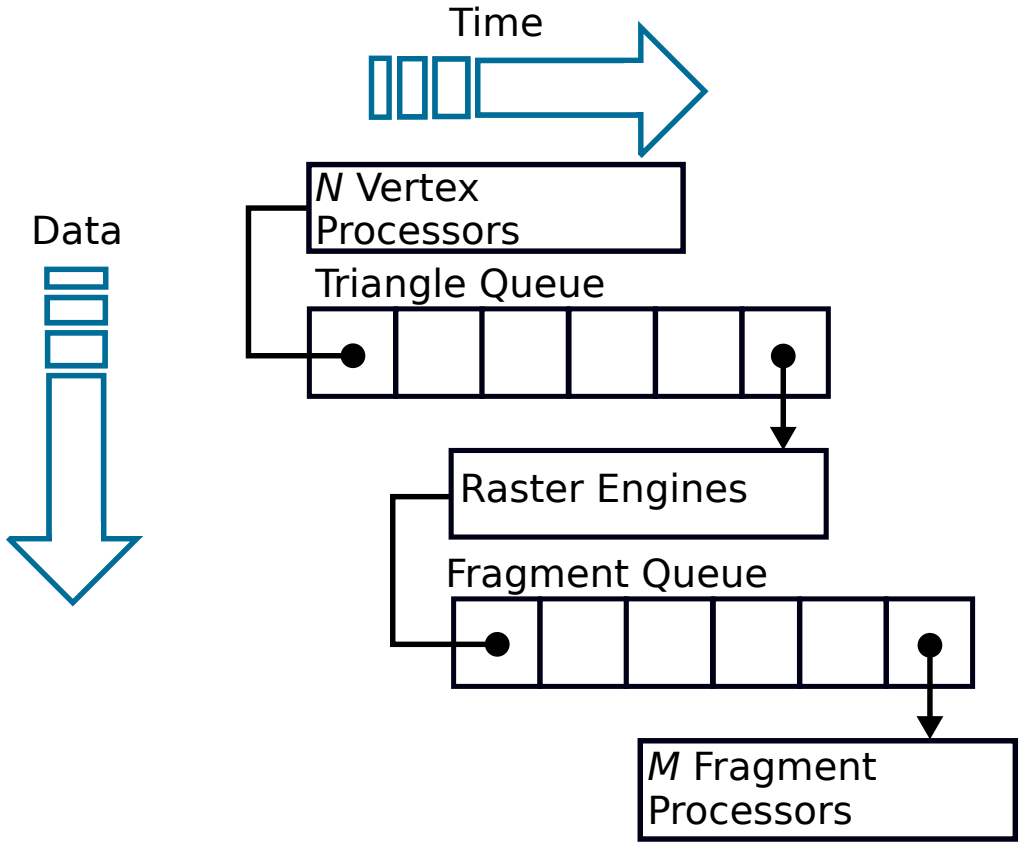
Bemerkungen

- Die Umformulierung führt asymptotisch nicht zu einer Veränderung von Zeit- und Arbeitskomplexität. Der Algorithmus teilt sich nun jedoch in mehrere parallele Arbeitsphasen auf.
- Durch die Operation TiefenTest ergeben sich Speicherzugriffe, die synchronisiert werden müssen ($\Rightarrow$ CRCW), dies ist jedoch bei beiden Algorithmen der Fall.
- Es bleibt der hohe Speicherbedarf ($O(V \times VP)$), der sich mit Hilfe des PRAM Modells nicht auflösen lässt.

Rasterisierung Paralleler Algorithmus 2

Überlegungen

Allgemeines GPU Modell (AGPU): Überlappe die $O(V)$ und $O(V \times VP)$ Arbeitsphasen zeitlich. PRAM für jede Arbeitsphase. Schlange für Dreiecke und Fragmente.



Rasterisierung Paralleler Algorithmus 2

Überlegungen

- Mit Hilfe des AGPU Modells lässt sich das Speicherbedarfsproblem durch *Pufferung* abschwächen.
- Es ergeben sich ggf. Wartezeiten dadurch, dass entweder Vertex Prozessoren, Raster Engines oder Fragment Prozessoren keine Arbeit zu verrichten haben.
- Das AGPU Modell für den Parallelen Algorithmus 2 ist ein theoretisches Modell, das der Implementierung tatsächlicher GPUs sehr nahe kommt.

Deferred Shading

Deferred Shading

Vorabüberlegungen

- Der zweite parallele Rasterisierungsalgorithmus kommt dem tatsächlich in Hardware implementierten Algorithmus auf GPUs sehr nahe. Wir gehen im folgenden davon aus, dass das Speicherbedarfsproblem durch Methoden wie Pipelining und Queueing gelöst ist.
- Bei unseren Überlegungen gingen wir von einer konstanten, geringen Anzahl an Lichtquellen aus. Für bestimmte 3D Beleuchtungsszenarien ist dies jedoch nicht adäquat. Wir betrachten daher im folgenden den Fall, dass wir viele Lichtquellen haben, und dass die Anzahl an Lichtquellen asymptotischen Einfluss hat.

Rasterisierung

Wir betrachten wieder den Algorithmus RASTERISIERUNG, diesmal in vereinfachter Form:

function RASTERISIERUNG(M,L,f)
    Fragmente ← VERTEXPHASE
    for all F ∈ Fragmente **do in parallel**
        for all L_j ∈ Lichtquellen **do**
            BELEUCHTE(F,M,L_j,f)
        end for
        TIEFENTEST(F, I)
        ALPHABLENDING(F, I)
    end for
end function

Vertex Transformation und Scan Konvertierung analog zu "Parallele Rasterisierung 2" - wir interessieren uns für Fragment Phase, die asymptotisch von der Anzahl Lichtquellen abhängt.

Beleuchtung von Fragmenten

Problem: Beleuchtung muss für alle Fragmente durchgeführt werden, da Geometrie evtl. teiltransparent. Weitere Vereinfachung daher: nur opake Geometrie.

function RASTERISIERUNG(M,L,f)
  Fragmente ← VERTEXPHASE
  for all F ∈ Fragmente **do in parallel**
    for all L_j ∈ Lichtquellen **do**
      BELEUCHTE(F,M,L_j,f)
    end for
    TIEFENTEST(F, I)
  end for
end function

In der Praxis: separate *Render Passes* für opake und für *tiefensortierte*, transparente Geometrie. 2. Pass: berücksichtige Tiefenpuffer aus erstem Pass.

Beleuchtung von Fragmenten

Unter diesen Annahmen kann man auch die Fragment Phase "deferred" formulieren.

function RasterisierungDeferred(M,L,f)
    Fragmente $\leftarrow$ VertexPhase
    for all F $\in$ Fragmente **do in parallel**
        Pixel(*F*) $\leftarrow$ TiefenTest(F, I)
    end for
    for all $L_j \in$ Lichtquellen **do**
        for all p $\in$ Pixel **do in parallel**
            Beleuchte(p,M,L_j,f)
        end for
    end for
end function

Deferred Shading Komplexität

Durch die Umformulierung verringert sich die Arbeitskomplexität des Algorithmus.
Ursprünglich:

$$W(n) = O(V) \times O(VP) \times O(L) \tag{24}$$

Deferred:

$$W(n) = O(V) \times O(VP) + O(L) \times O(VP) \tag{25}$$

Wie auch schon bei der Umformulierung von Rasterisierung ergibt sich weiterer Speicherbedarf, nämlich dadurch, dass VP "Pixel" gespeichert werden müssen.

Deferred Shading Komplexität

Der erhöhte Speicherbedarf wird asymptotisch ammortisiert, wenn man davon ausgeht, dass wir Deferred Shading auf Basis von Parallele Rasterisierung 2 implementieren:

$$O(V \times VP) + O(VP) = O(V \times VP) \tag{26}$$

Deferred Shading in der Praxis

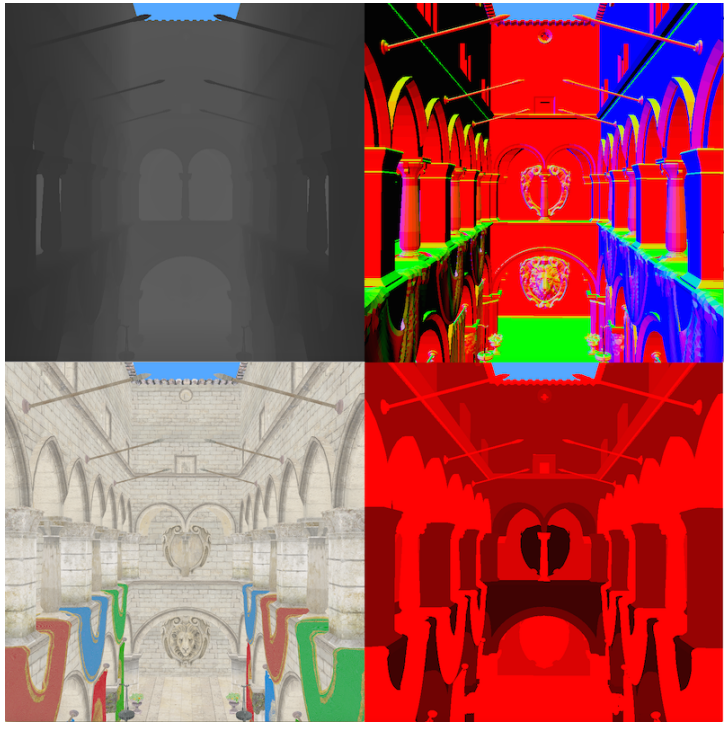
Wir wollen betrachten, wie man Deferred Shading in der Praxis auf GPUs implementiert. Wir gehen vom AGPU Modell von weiter oben aus. Wir nehmen weiter an, dass wir drei 2D Texturen verwalten, die die gleiche Auflösung wie das zu rendernde Bild haben:

1. Tiefentextur (DB): Tiefeninformation für jedes Pixel aus Tiefentest (i. Allg. kann man auf den Tiefenpuffer von GPUs nicht direkt zugreifen, sondern muss diesen kopieren).
2. Normalentextur (NB): Shading Normale für jedes Pixel bzgl. der Geometrie, die den Tiefentest bestanden hat.
3. Diffuse Textur (TB): Diffuse Texturinformation für jedes Pixel.
4. Materialindextextur (MB): Materialindex für jedes Pixel.

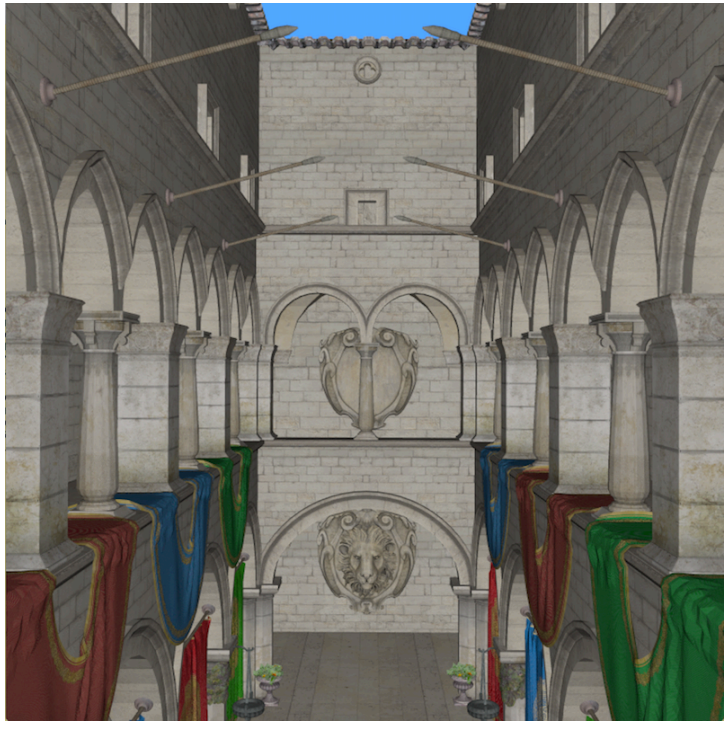
Die sich ergebende Datenstruktur nennt man *Geometry Buffer* (g-Buffer).

Deferred Shading in der Praxis

g-Buffer



Ergebnis



Crytek Sponza 3D Modell: Frank Meinl, (CC-BY 3.0), Rendering: Stefan Zellmann

Deferred Shading in der Praxis

Beim Deferred Shading auf der GPU wird die Fragment Phase von weiter oben leicht angepasst. Eine Implementierung sieht strukturell etwa so aus:

```
parallel_for_each(f in fragmente) {
    tiefen_test(f);
    gbuffer[f.x][f.y] = { f.tiefe, f.normale,
                          f.diffus, f.material_index };
}

for_each(l in lichter) {
    parallel_for_each(p in pixel) {
        beleuchte_pixel(l, p, gbuffer[p.x][p.y]);
    }
}
```

Deferred Shading in der Praxis

Implementiert als “Multi Render Pass” Verfahren. Erster Render Pass baut g-Buffer auf.

```
parallel_for_each(f in fragmente) {
    tiefen_test(f);
    gbuffer[f.x][f.y] = { f.tiefe, f.normale,
                          f.diffus, f.material_index };
}
```

Dann folgen *#Lichtquellen* viele Render Passes, die die *Pixel* (nicht mehr Fragmente!) beleuchten.

```
parallel_for_each(p in pixel) {
    beleuchte_pixel(l, p, gbuffer[p.x][p.y]);
}
```

Auf GPUs ist die Fragment Phase frei programmierbar. Die Render Passes (auch die über Pixel) erfolgen in einem *Fragment Programm* (werden im Vorlesungsteil zu GPU Architekturen besprochen).

Deferred Shading in der Praxis

Wir wollen einschätzen, wie hoch der Speicherbedarf für den g-Buffer beim Deferred Shading, etwa in MB, ist. Nehmen wir an, dass wir die g-Buffer Einträge wie folgt speichern:

DB	24-bit
NB	$2 \times$ 32-bit (Polarkoordinaten)
TB	$3 \times$ 32-bit (RGB)
MB	16-bit

Für eine Bildschirmauflösung von 1024×1024 Pixel ergibt sich dann etwa ein Speicherverbrauch von 25 MB.

Deferred Shading in der Praxis

Überlegt man sich, dass um 2005 herum Videospeicher häufig bei etwa 100 MB bemessen war, versteht man, dass g-Buffer Datenstrukturen bzgl. des Speicherbedarfs optimiert werden mussten. Dies rückt angesichts mehrerer GB Speicher heutiger GPUs in den Hintergrund.

Unsere Überlegungen zum Speicherverbrauch in der Praxis über die Jahre bestätigen also, dass der zusätzliche Speicherbedarf durch eine verzögerte Beleuchtungsphase asymptotisch ammortisiert wird ($O(V \times VP) + O(VP) = O(V \times VP)$).

Deferred Shading in der Praxis

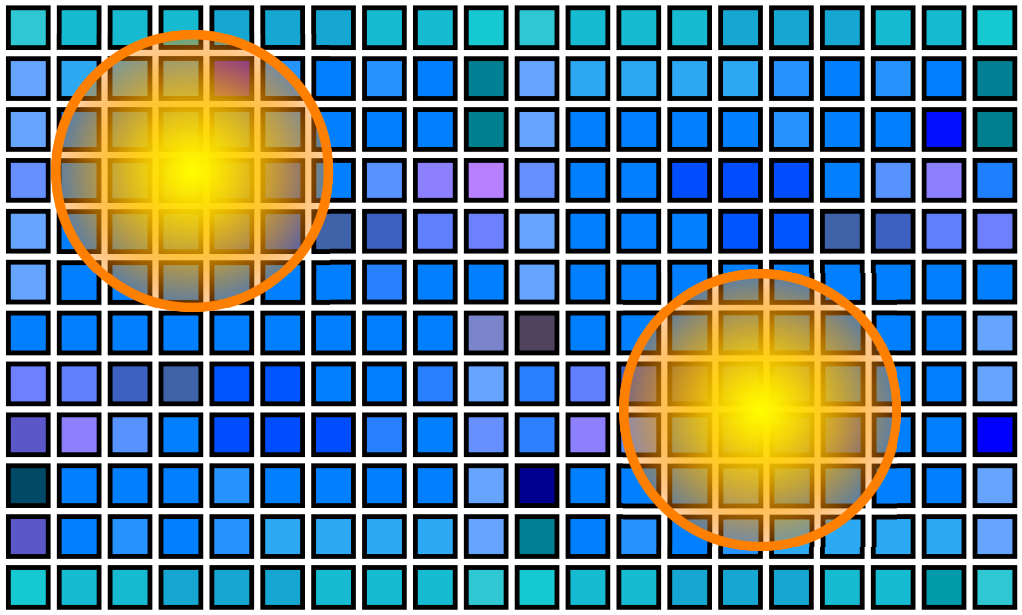
g-Buffer Latenz

- Problem allerdings: g-Buffer liegt in GDDR3 Speicher $\Rightarrow$ hohe Zugriffslatenz.
- Großes Problem auf alten GPUs.
- Moderne GPUs sind darauf optimiert, während Speicherzugriffen unabhängige Berechnungen durchzuführen. Positiv beim Deferred Shading.
- Mehr dazu im Vorlesungsteil zu GPU Architekturen.

Deferred Shading in der Praxis

Beleuchtungsphase

Wenn g-Buffer aufgebaut, L Passes über Pixel (Tiefentest ist also bereits entschieden, i. Allg. gilt: #Pixel $<<$ #Fragmente.



Punktlichtquellen: radialer Fall-Off ("inverse square"), man kann Umkugel für Lichtquellen berechnen $\Rightarrow$ Passes gehen i. d. R. nicht über alle Pixel.

Deferred Shading in der Praxis

- Deferred Shading wurde als Erweiterung der Rasterisierungs Pipeline älterer GPUs implementiert - maßgeblich in den Software-Industriezweigen, die sich mit realistisch anmutender Echtzeitgrafik beschäftigen (Spiele).
- Game Engines wie Unreal oder Unity verfügen über Rendering Pfade, die auf g-Buffer Algorithmen aufsetzen.
- Es wurden über die Jahre Erweiterungen oder alternative Verfahren entwickelt, die auf g-Buffer aufsetzen, u. a. *Deferred Lighting*, *Screen Space Ambient Occlusion*, aber auch Image-Filter Verfahren, die die Tiefeninformation ausnutzen.

Deferred Shading in der Praxis

"Pitfalls"

- Keine teiltransparente Geometrie (es gibt Verfahren wie etwa "Depth Peeling", die verwendet werden können, üblich ist aber ein separater "forward pass" ohne g-Buffer).
- Anti-aliasing nicht ganz einfach.
  - Man kann *nicht* einfach "supersamplen", also das Bild mit höherer Auflösung rendern und dann kleiner skalieren. Normalen aus der Normalentextur können beim Skalieren nicht einfach linear interpoliert werden.
- Anzahl an *Materialtypen* nicht variabel, g-Buffer kodiert Attribute.
- Es ist ungünstig, dass Tiefenwerte umkopiert werden müssen, obwohl die Hardware sie bereits speichert.

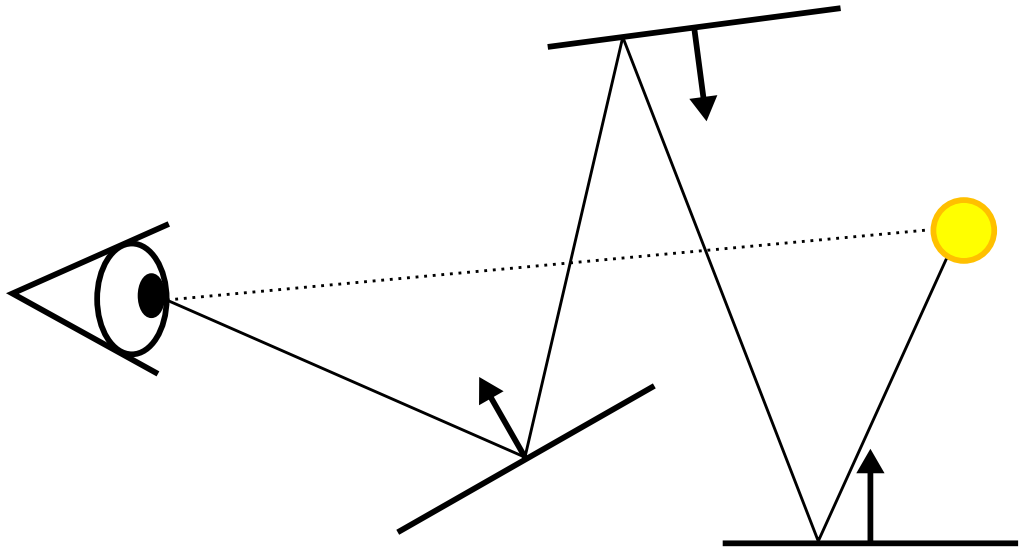
Strahlverfolgung

Strahlverfolgung

- Ansatz basiert auf Grundprinzip, dass sich Lichtpartikel (*Photonen*), die von einer Lichtquelle aus *emittiert* werden, entlang gerader Linien durch den Raum bewegen, und dass Lichtpartikel nicht mit anderen Lichtpartikeln interagieren.
- Jedoch Interaktion mit *massebehafteten* Partikeln $\Rightarrow$ Streuung und Absorption. Vielmehr geht man davon aus, dass Lichtpartikel mit massebehafteten Partikeln interagieren und es zu Phänomenen wie Streuung und Absorption kommt.
- Es ergeben sich *Pfade* - i. Allg. von Interesse sind diejenigen vom Licht zum Betrachter.
- Häufig ist das zu aufwendig - daher paart man Strahlverfolgung oft mit lokaler Beleuchtung.

Symmetrie

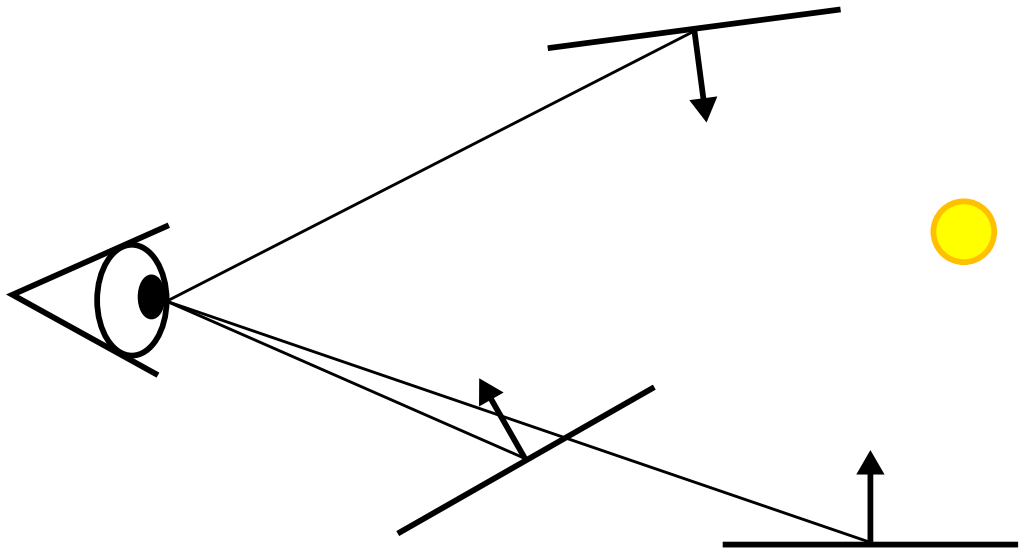
Aufgrund von Symmetrieeigenschaften darf man das Problem umformulieren, sodass Pfade vom Betrachter zum Licht gesucht werden (beginne Strahlverfolgung vom Betrachter aus).



In der Abbildung: ein *direkter Lichtpfad*, und ein Pfad, der aufgrund mehrerer Licht/Oberflächeninteraktionen entsteht.

Primärsichtbarkeit

Für unsere Betrachtungen wollen wir uns zuerst nur für Pfade mit Länge eins hin zu einer Oberfläche interessieren (*Primärstrahlen*).



An den Interaktionspunkten wird lokales Beleuchtungsmodell ausgewertet. Eingabedaten genau wie bei Rasterisierung. Das optische Ergebnis sollte äquivalent zu Rasterisierung sein.

Strahlen

Strahlen definiert man bzgl. *Ursprung* und *Richtung*. Den Ursprung drückt man durch eine 3D Position aus, die Richtung durch einen Einheitsvektor:

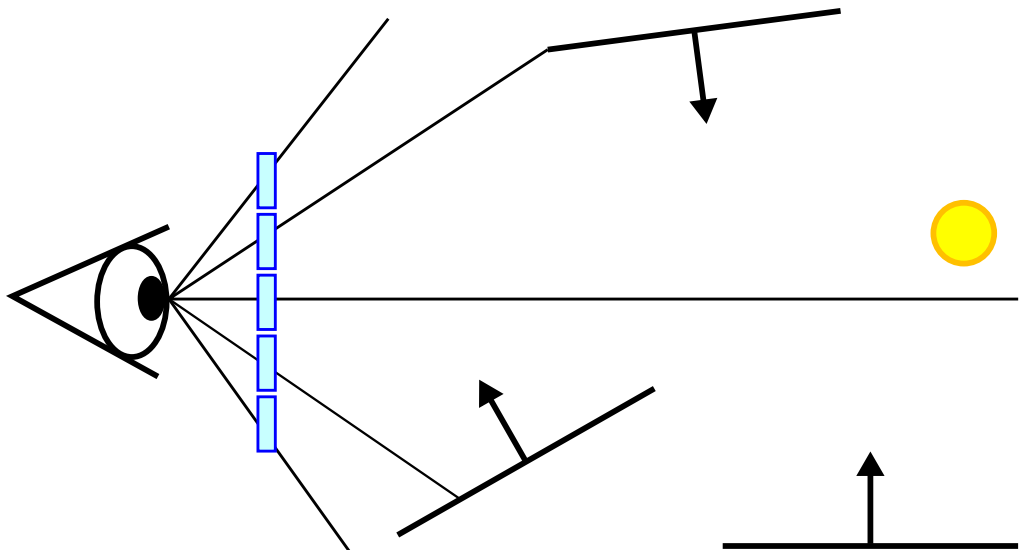
$$R = \{\boldsymbol{o}, \vec{d}\} \tag{27}$$

Die Distanz, die entlang eines Strahls zurückgelegt wurde, parametrisiert man mittels eines Skalars t. Darüber lassen sich z. B. Schnittpunkte mit Geometrie entlang des Strahls berechnen.

$$I = \boldsymbol{o} + \vec{d}t. \tag{28}$$

Primärstrahlen

Primärstrahlen generiert man so, dass ihr Ursprung beim Betrachter liegt, und dass sie die einzelnen Pixel des Rasters *samplen*.



Schnitttests

Die Basisoperation bei der Strahlverfolgung ist es, zu testen, ob ein Strahl eine Oberfläche schneidet. Dazu setzt man die Strahlgleichung in der parametrischen Form (28) in die Oberflächengleichung ein und löst nach t auf.

Strahl/Ebenen Schnitttest

Für den Schnitt zwischen Strahl und Ebene wollen wir dies kurz exemplarisch erläutern. Wir betrachten die Ebenengleichung in Parameterform

$$(\boldsymbol{x} - \boldsymbol{p_0}) \cdot \vec{n} = 0, \tag{29}$$

wobei $\boldsymbol{p_0}$ ein beliebiger Punkt in der Ebene und $\vec{n}$ ein normalisierter Richtungsvektor senkrecht zur Ebene (geometrische Normale) ist. Wir setzen die Strahlgleichung ein und erhalten

$$(\boldsymbol{o} + \vec{d}t - \boldsymbol{p_0}) \cdot \vec{n} = 0. \tag{30}$$

Strahl/Ebenen Schnitttest

$$(\boldsymbol{o} + \vec{d}t - \boldsymbol{p_0}) \cdot \vec{n} = 0$$

Wir lösen nach t auf und erhalten

$$t = \frac{(\boldsymbol{p_0} - \boldsymbol{o}) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}. \tag{31}$$

Ist der Nenner 0, wissen wir, dass der Strahl parallel zur Ebene verläuft. Diesen Fall behandeln wir speziell, andernfalls gibt t die Distanz zwischen Strahlursprung und Ebene an.

Schnitttests mit einfachen, polynomiellen Oberflächen mit wenig Koeffizienten (z. B. Dreiecke, Quadriken etc.) führt man analog durch.