Raytracing mit CUDA

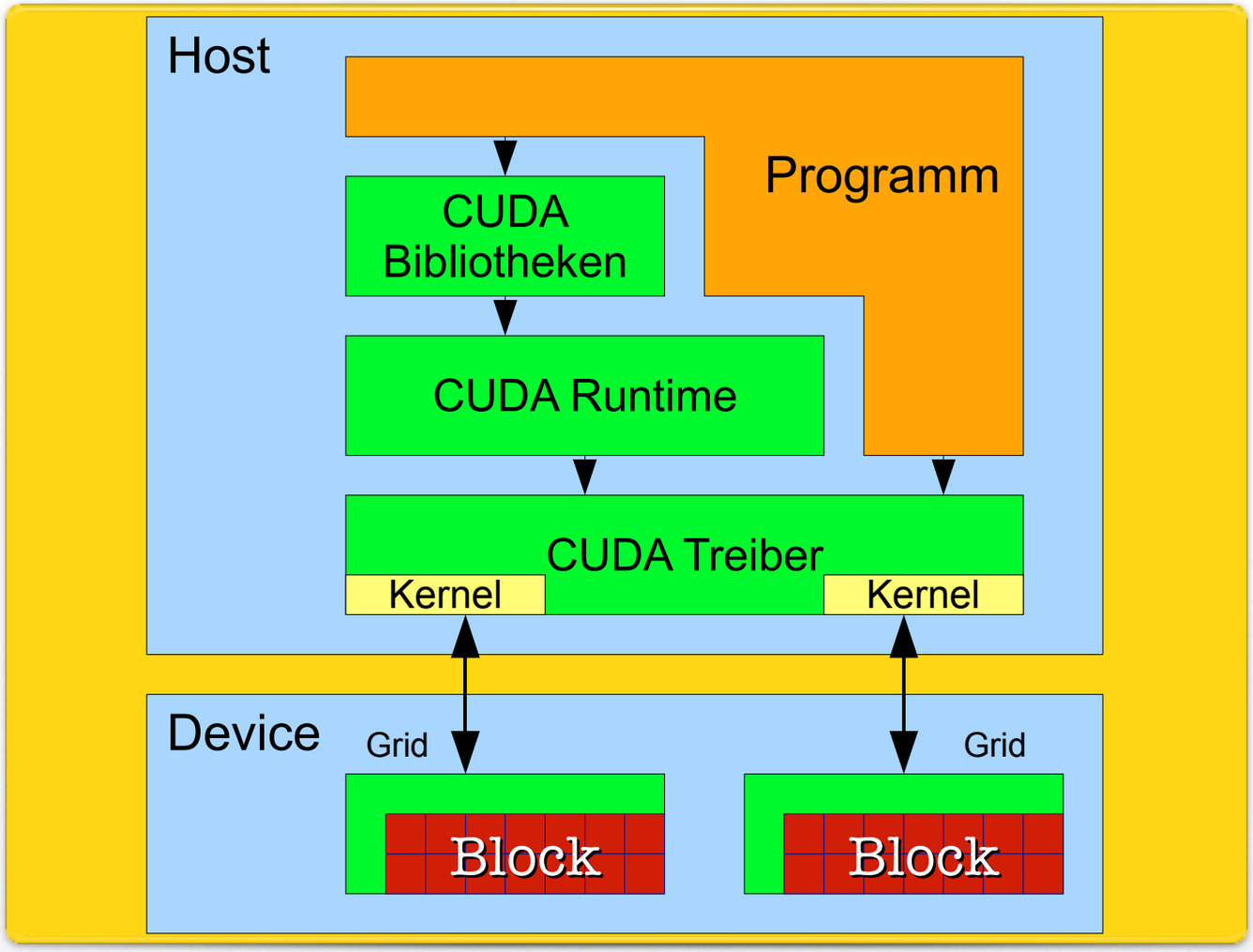
Dorian Weber

Gliederung

- kurze Einführung in CUDA
- Funktionsweise einer traditionellen Grafikengine (z.B. OpenGL, Direct3D)
- Design- und Architekturentscheidungen
- Implementation
- Demonstration

CUDA

- „Compute Unified Device Architecture“ von NVIDIA
- nutzt die Grafikkarte als parallelen Co-Prozessor
- Architektur ist stark auf grafische Berechnungen optimiert



Architektur

CUDA Schnittstellen

- Application Programming Interface kommt in zwei Geschmacksrichtungen
  - Drivers API (low level)
  - Runtime API (high level)
- APIs schließen sich gegenseitig aus

Drivers API

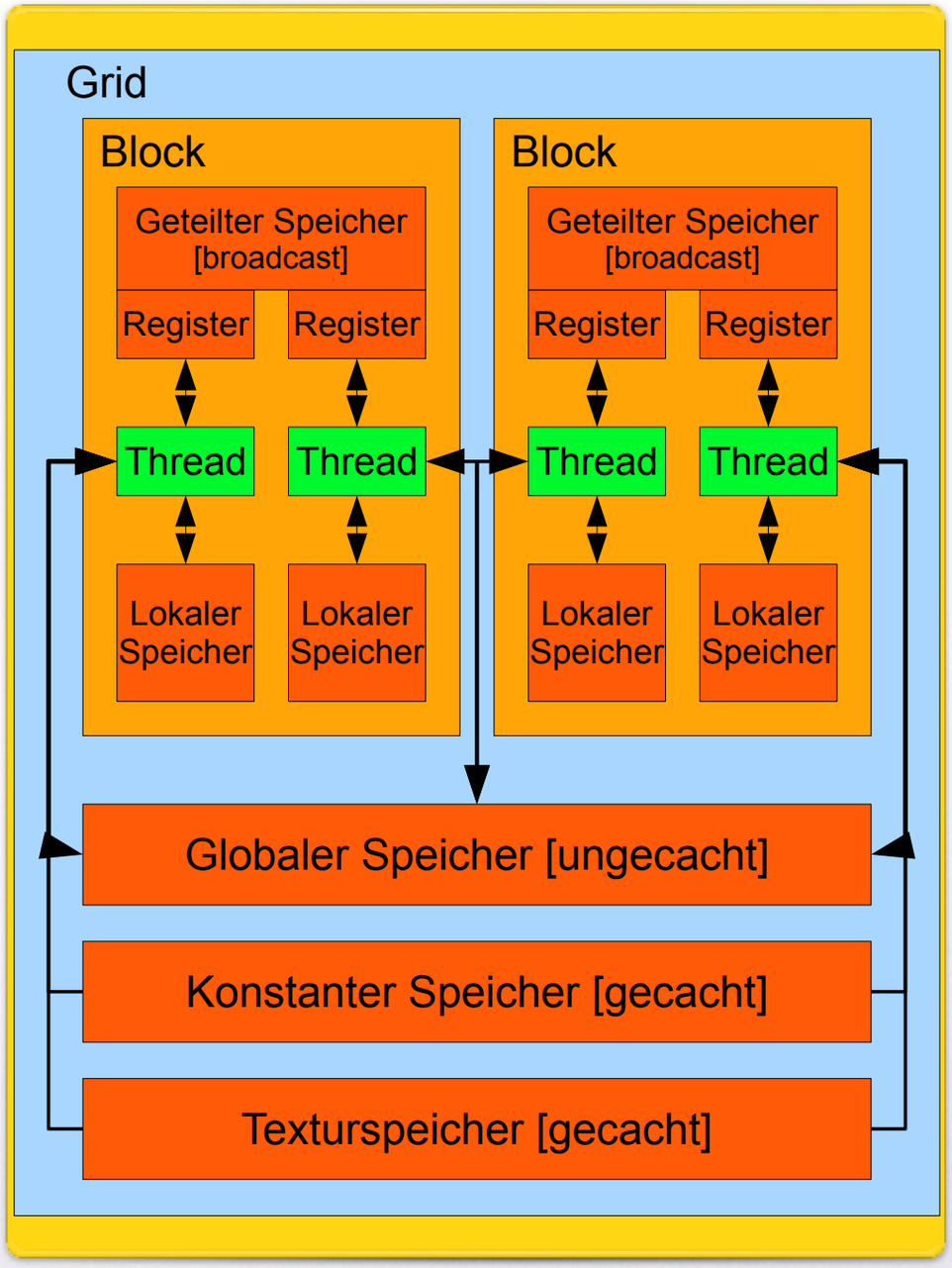
Vorteile	Nachteile
• Null Overhead • Host-Code unabhängig vom NVCC • dynamische Kernelverwaltung • keine CUDA Runtime Library nötig	• low level, imperativ • kein Emulationsmodus • Compilierung von Host- und Devicecode getrennt • explizite Modulverwaltung

Runtime API

Vorteile	Nachteile
• high level, imperativ • automatisches Laden benötigter Module • Funktionssetup und implizites Typecasting • Emulationsbibliothek • Compilerintegration	• Vorhalten aller potentiell benötigten Module • Abhängigkeit des Devicecodes vom Hostcompiler • linkt CUDA Runtime

Speicher-hierarchie

„All programming can be viewed as an exercise in caching.“
(Terje Mathisen)



Klassische 3D Engines

- Rendervorgang mittels Matrizen umgesetzt
- elegante und einheitliche Beschreibung aller Transformationen
- Bewegung, Rotation, Skalierung, Projektion, Clipping
- jede beliebige Kombination (Matrixmultiplikation)

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

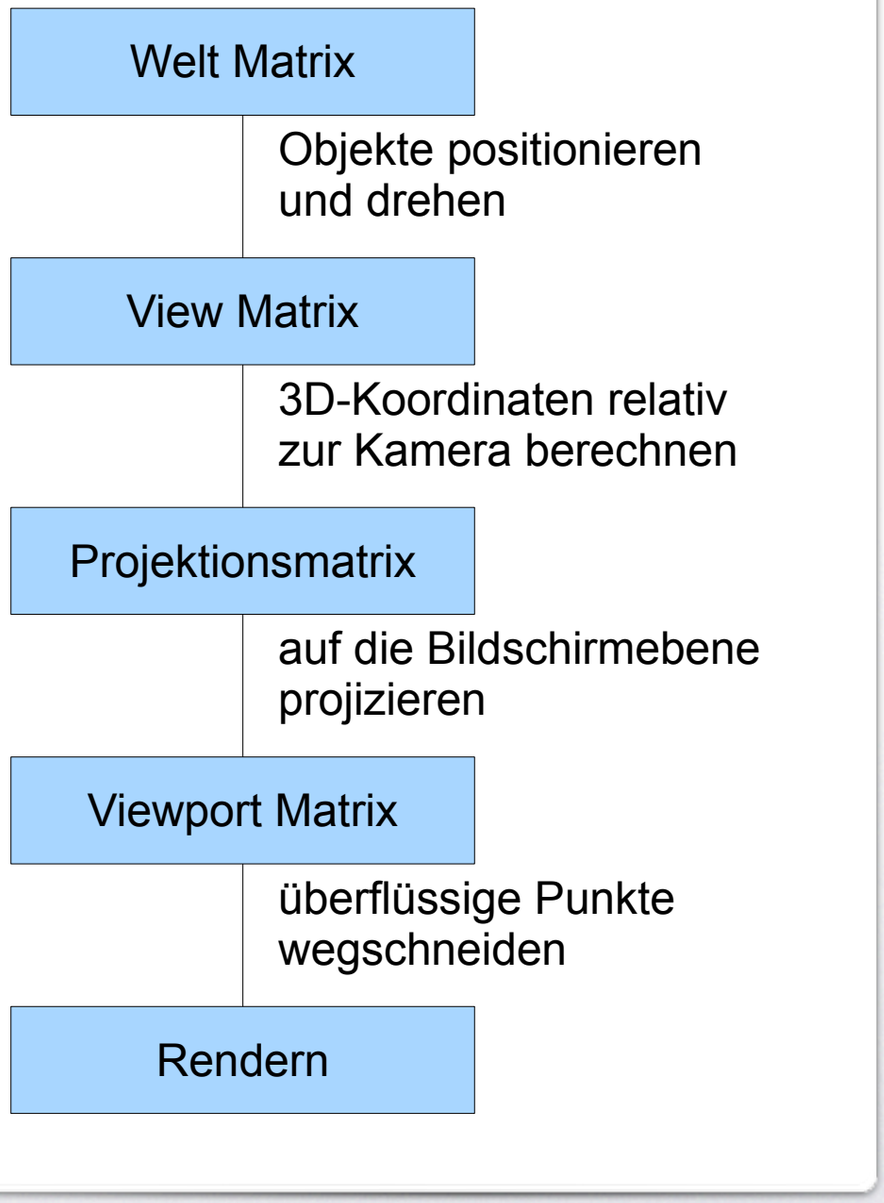
Rotationsmatrix

$$\begin{pmatrix} \cos(90°) & -\sin(90°) & 0 & 0 \\ \sin(90°) & \cos(90°) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

Multiplikation mit einem Vektor

Beispiel: Rotation

Render-vorbereitung



Matrizen für Raytracer?

Vorteile	Nachteile
• einheitliche Beschreibung • leicht parallelisierbar • etabliert (also viel Hilfe verfügbar) • intuitive Mathematik	• sehr groß - 4 GPU-Register pro Matrix • viele Multiplikationen • Gimbal Lock

(Einheits-)quaternionen

- leben auf einer 4D Hyperkugel
- ein Realteil, drei Imaginärteile
- Drehungen im Raum sehr elegant zu beschreiben
- passen in ein GPU Register

$$\begin{pmatrix} \sin(\alpha/2)\cdot x\cdot i \\ \sin(\alpha/2)\cdot y\cdot j \\ \sin(\alpha/2)\cdot z\cdot k \\ \cos(\alpha/2) \end{pmatrix}$$

Einheitsquaternion

$$\begin{pmatrix} \sin(45°)\cdot 0\cdot i \\ \sin(45°)\cdot 0\cdot j \\ \sin(45°)\cdot 1\cdot k \\ \cos(45°) \end{pmatrix} * \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} * \begin{pmatrix} -\sin(45°)\cdot 0\cdot i \\ -\sin(45°)\cdot 0\cdot j \\ -\sin(45°)\cdot 1\cdot k \\ \cos(45°) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Multiplikation

Rotation von vorhin

Designentscheidungen

- Drivers API
- Quaternionen
- OpenGL Subsystem
- Phongs Lichtmodell

Implementation (I)

```
Funktion Render()

// berechne deine Position auf dem Bildschirm
pixel.x = getPosX()
pixel.y = getPosY()

// berechne den Primärstrahl, der durch den Pixel geht
primärStrahl.position = kamera.position
primärStrahl.richtung = getPrimaryRay(kamera, pixel)

// ermittle die Farbe des Pixels
pixel.farbe = getPixelColor(primärStrahl, kamera)

return
```

Implementation (II)

```
Funktion getPixelColor(strahl, kamera)

color = hintergrundfarbe

// bestimme den kleinsten Abstand
dist, objekt = getMinDist(strahl)

Wenn objekt ungleich NULL ist
     color = attenuateColor(color, objekt.farbe)

     // schieße wild mit weiteren Strahlen um dich
     Schattenstrahl = getShadowRay(strahl, objekt)
     color = attenuateColor(color, getPixelColor(Schattenstrahl, kamera)

     Reflektionsstrahl = getReflectionRay(strahl, objekt)
     color = attenuateColor(color, getPixelColor(Reflektionsstrahl, kamera)

     Brechungsstrahl = getRefractionRay(strahl, objekt)
     color = attenuateColor(color, getPixelColor(Brechungsstrahl, kamera)

     // (etc, ...)

return color
```

Implementation (III)

```
Funktion getMinDist(strahl)

// setze den kleinsten Abstand auf unendlich
minDist = INFINITY
objekt = NULL

// teste für jedes Objekt, ob es von diesem Strahl geschnitten wurde
für jedes Objekt A:
     Wenn A vom aktuellen Strahl geschnitten wurde
     und dieser Abstand dist kleiner ist als minDist,
     dann
          objekt = A
          minDist = dist

return minDist, objekt
```

Details

- Objekte und Lichtquellen im Texturspeicher
- Renderstruktur im konstanten Speicher
- Rekursionsbehandlung mittels lokalem Stack (a priori beschränkt)
- Ausnutzung der Asynchronität zwischen GPU und CPU

Was ist getan?

- Framework zur komfortablen Bedienung der Drivers API - Template-basiert
- Toolset für die Steuerung des Tracers (insbesondere mathematische Werkzeuge)
- Werkzeuge für das Gerät

Was ist getan?

- Raytracer:
  - Phong Lichtmodell
  - Reflexion & Brechung (allgemein gelöst, physikalisch korrekt)
  - beliebig viele Objekte und Reflektionen
  - Echtzeit

Was fehlt?

- Objekte sitzen nicht im Texturspeicher
- nur eine (frei positionierbare) Lichtquelle
- ausschließlich Kugeln, allerdings Vorbereitung für weitere Objekttypen
- kein Schatten
- Stack für Rekursion unnatürlich langsam

Demo

Vielen Dank für eure Aufmerksamkeit!