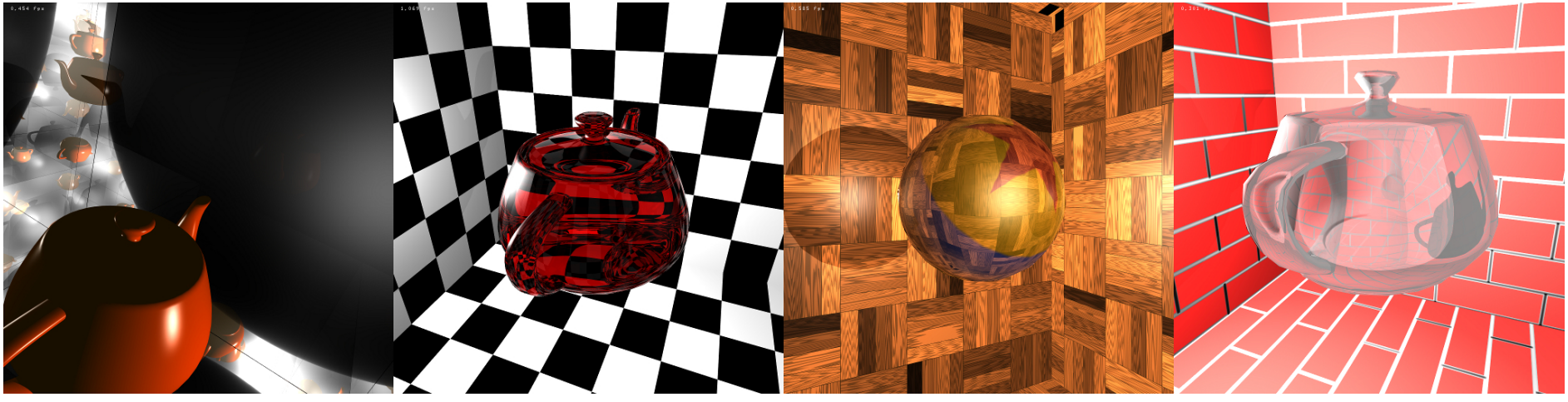
Automatische Paketisierung

Übersetzer in der Computergraphik

Ralf Karrenberg, Sebastian Hack, Philipp Slusallek



SAARLAND UNIVERSITY

COMPUTER SCIENCE

Shader

- Berechnen u.A. Farbe eines Pixels
- Prozedurale Shader für hochqualitative Bilder
  - Werden für jeden Pixel aufgerufen
  - Ray-Tracing: Bei jedem Schnitt
  - Volume Rendering: Für jeden Integrationsschritt eines Strahls
  - In der innersten Schleife des Algorithmus
  - Jeder Taktzyklus zählt
- Shader werden in Shadersprachen geschrieben
  - C Dialekte
  - Angereichert um Spezialkonstrukte für die Grafik
  - Beispiele: RenderMan, Cg, HLSL, GLSL, MetaSL, ...

Shader

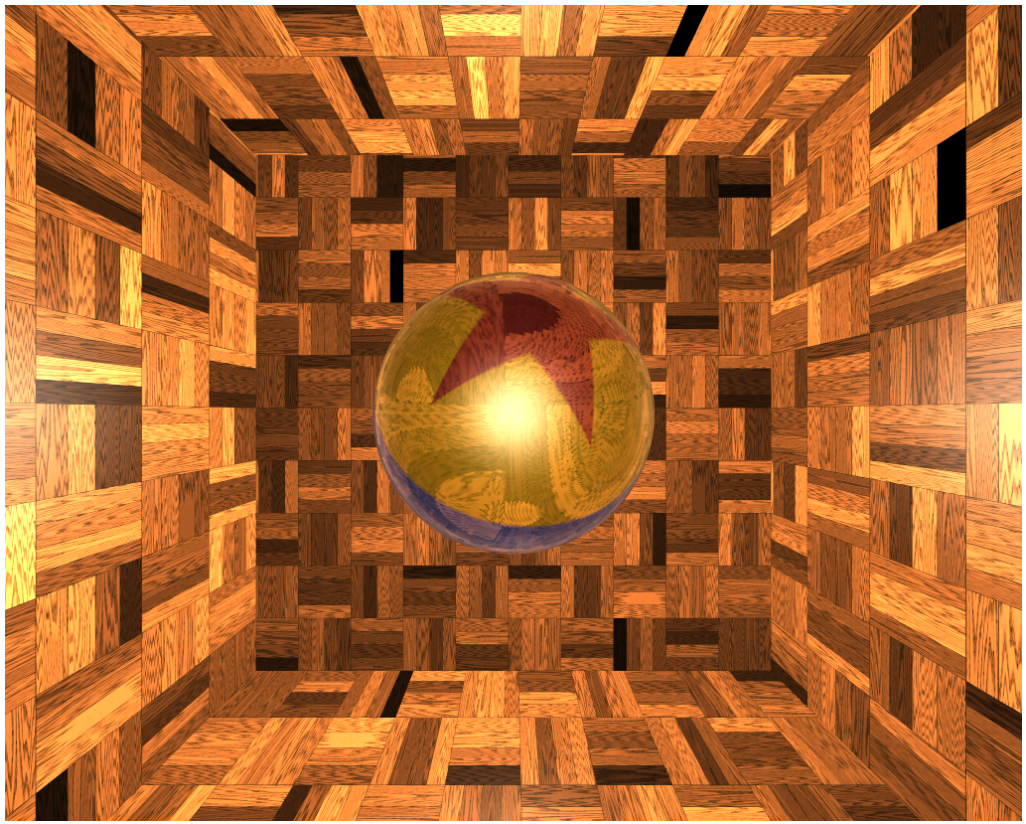
ein Beispiel: Wood

```
surface
wood(float ringscale = 10;
     color lightwood = color(0.3, 0.12, 0.03),
           darkwood  = color(0.05, 0.01, 0.005);
     float Ka = 0.2,
           Kd = 0.4,
           Ks = 0.6,
           roughness = 0.1)
{
  point NN, V, PP;
  float y, z, r;

  NN = faceforward(normalize(N), I);
  V = -normalize(I);
  PP = transform("shader", P);
  PP += noise(PP);

  y = ycomp(PP);
  z = zcomp(PP);
  r = sqrt(y * y + z * z);
  r *= ringscale;
  r += abs(noise(r));
  r -= floor(r);
  r = smoothstep(0, 0.8, r)
    - smoothstep(0.83, 1.0, r);
  Ci = mix(lightwood, darkwood, r);

  Oi = Os;
  Ci = Oi * Ci * (Ka * ambient() + Kd * diffuse(NN))
     + (0.3 * r + 0.7) * Ks * specular(NN, V, roughness);
}
```



Ziele von Shadersprachen

- Portabilität
  - Verstecke Details des Renderers vor dem Programmierer
- Bequemlichkeit
  - Bieten Grafik-spezifische Sprachkonstrukte (illuminance loop, vector/matrix data types)
- Beschränkungen
  - Verbiete Sprachkonstrukte, die schlecht übersetzbar sind
- Leistung
  - Erzeugter Code kann an das Renderingsystem angepasst werden

. . . in der Praxis

- Portabilität
  - Semantik abhängig vom Rendering Modell
  - Daher oft nicht portabel auf ein anderes Rendering-System
- Bequemlichkeit
  - Alle Spezialkonstrukte können mit „Bordmitteln“ herkömmlicher OO-Sprachen ausgedrückt werden
  - Operatorüberladung, Klassen, Schablonen, . . .
- Einschränkungen
  - Moderne Shadersprachen weichen diese sowieso auf
  - Warum also C++ neu erfinden?
- Leistung
  - oft interpretiert; man will keinen optimierenden Übersetzer schreiben
  - oder nach C übersetzt und dynamisch geladen

. . . in der Praxis

- Pro Renderer eine Shadersprache
- Das Bauen eines neuen Renderers . . .
- . . . läuft auf das Bauen eines neuen Übersetzers hinaus

. . . in der Praxis

- Pro Renderer eine Shadersprache
- Das Bauen eines neuen Renderers . . .
- . . . läuft auf das Bauen eines neuen Übersetzers hinaus

Unser Ziel ist ein Shader Rahmenwerk mit folgenden Eigenschaften:

- Einfach in einen Renderer zu integrieren
- Hochqualitativer Code, maximale Leistung
- Unabhängigkeit von der Eingabesprache

AnySL

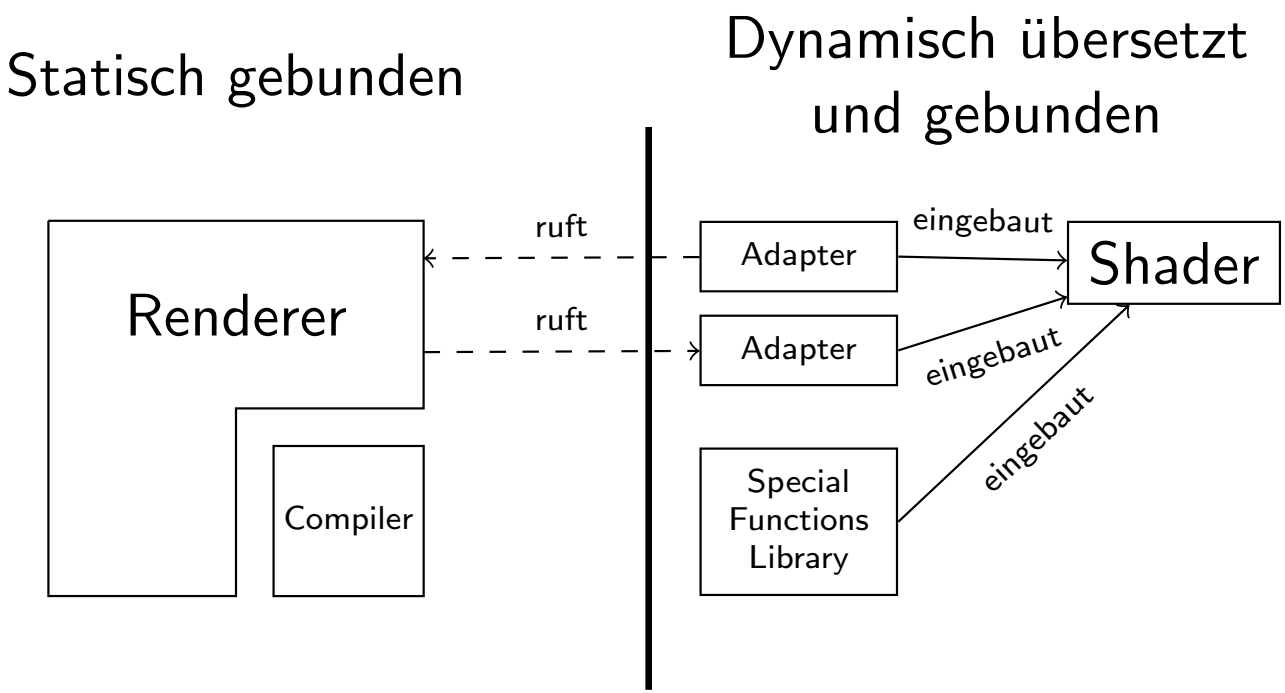
- Verwende eine Zwischendarstellung als Shadingsprache
- Verwende eine Übersetzer-Bibliothek im Renderer
- Lade Shader zur Laufzeit
  - adaptiere sie an den Renderer
  - übersetze und optimiere sie für die spez. Plattform
- Jede Shadingsprache kann unterstützt werden

Probleme:

- Effiziente Portabilität:
  - API zum Renderer
- Spezial-Codeerzeugung im Graphikbereich
  - Hier: Paketisierung für Echtzeit-Ray-Tracing

Portabilität

- Einfache Integration in den Renderer
- Renderer stellt API Klebecode in der Zwischendarstellung zur Verfügung
- Wird zur Laufzeit „wegoptimiert“, wenn der Shader geladen wird



Paket-basiertes Ray-Tracing

- Moderne Ray Tracer schießen „Pakete“ von Strahlen

Einzelner Strahl

Base

x
y
z

Dir

x
y
z

Strahlenpaket der Größe n

Base

x_1	x_2	x_3	$\cdots$	x_n
y_1	y_2	y_3	$\cdots$	y_n
z_1	z_2	z_3	$\cdots$	z_n

Dir

x_1	x_2	x_3	$\cdots$	x_n
y_1	y_2	y_3	$\cdots$	y_n
z_1	z_2	z_3	$\cdots$	z_n

- Ausnutzen der Vektorinstruktionen moderner Prozessoren:
  Führen Instruktion auf $k \leq n$ Werten gleichzeitig aus

- Gegenwärtige Architekturen:
  SSE: $k = 4$ AVX: $k = 8$ Larrabee: $k = 16$

Paket-basiertes Ray-Tracing

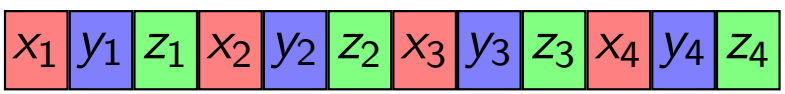
- Das Schreiben von Paket Code ist mühselig
- Hauptgrund: Hohe Leistung erfordert Änderung der Datenanordnung:
  $\Longrightarrow$ Lokalität (caches), Vektorinstruktionen
- Reihungen von Verbunden (AOS)
  werden zu Verbunden von Reihungen (SOA)

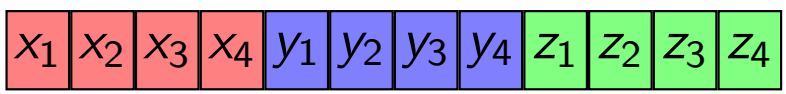
AOS:

```
struct Vector {
  float x, y, z;
};
typedef Vector Packet4[4];
```

SOA:

```
struct Packet4 {
  float x[4];
  float y[4];
  float z[4];
};
```



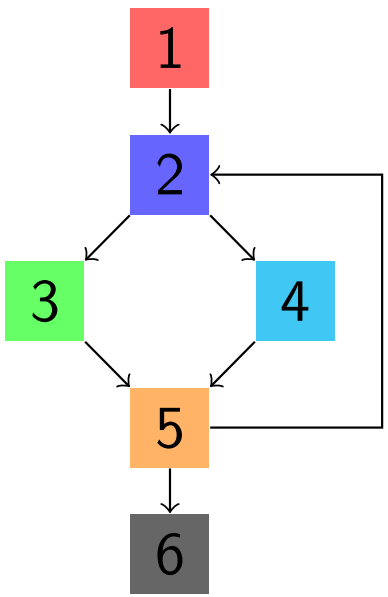


Paketisierte Shader

- Shader-Code gegeben als SSA-Programm mit CFG
- Der paketisierte Shader führt k Instanzen des Originals parallel aus
- Steuerfluss kann zwischen den Instanzen divergieren!

Paketisierte Shader

- Shader-Code gegeben als SSA-Programm mit CFG
- Der paketisierte Shader führt k Instanzen des Originals parallel aus
- Steuerfluss kann zwischen den Instanzen divergieren!



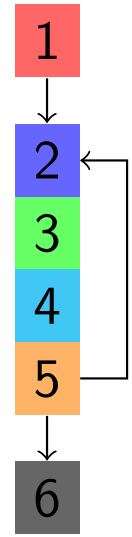
Beispiel: Vier Instanzen

Instanz	Ausgeführte Blöcke
1	1 2 3 5 6
2	1 2 4 5 6
3	1 2 3 5 2 3 5 6
4	1 2 3 5 2 4 5 6

Paketisierte Shader

Erweiterung von Allen, Kennedy, et al. '83

- Programmtransformation
- Plätten des Steuerflusses
  $\Longrightarrow$ Prädikate
- Jede Instanz führt alle Befehle aus
- Falsche Ergebnisse werden ausmaskiert
- Schleifen laufen bis die letzte Instanz herausfällt
- Verlassene Instanzen werden auf korrektes Ergebnis zurückgesetzt
- Simuliert, was GPUs in Hardware machen



Inst	Executed blocks
1	1 2 3 4 5 2 3 4 5 6
2	1 2 3 4 5 2 3 4 5 6
3	1 2 3 4 5 2 3 4 5 6
4	1 2 3 4 5 2 3 4 5 6

Load	26
Utilization	$26/40 = 65\%$
Speedup	$26/10 = 2.6$x

Paketisierte Shader

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Man möchte das nicht von Hand erledigen ...

Skalar

```
if (all(inside(vec2(u.x, v.x), 0.f, 1.f))) {
vec3 p0 = mix(mix(p00, p01, v.x),
          mix(p10, p11, v.x), u.x);
          ...
```

Paketisiert

```
__m128 u_x = _mm_div_ps(_mm_xor_ps(_mm_add_ps(
  _mm_mul_ps(Av1_, v_x), A1_),
  _mm_castsi128_ps(_mm_set1_epi32(0x80000000))),
  _mm_add_ps(_mm_mul_ps(Auv1_, v_x),
  Au1_));
__m128 u_y = _mm_div_ps(_mm_xor_ps(_mm_add_ps(
  _mm_mul_ps(Av1_, v_y), A1_),
  _mm_castsi128_ps(_mm_set1_epi32(0x80000000))),
  _mm_add_ps(_mm_mul_ps(Auv1_, v_y),
  Au1_));
__m128 ia13 = _mm_set1_ps(0);
__m128 ia15 = _mm_set1_ps(1);
__m128 ia16 = u_x;
__m128 ia17 = v_x;
__m128 condmask18 = _mm_and_ps(_mm_and_ps(
  _mm_cmpgt_ps(ia16, ia13), _mm_cmplt_ps(ia16,
  ia15)), _mm_and_ps(_mm_cmpgt_ps(ia17,
  ia13), _mm_cmplt_ps(ia17, ia15
  )));
  ...
```

- Optimierungen können mit geplättetem Steuerfluss nicht umgehen
- Daher: Optimiere skalare Version, paketisiere dann

Paketisierte Shader

Ergebnisse

Scene	Speedup
brick	4.0x
checker	4.5x
checker2	4.8x
glass	5.0x
glass2	5.7x
granite	1.0x
parquet	1.3x
phong	2.7x
screen	4.9x
starball	2.5x
starball2	3.0x
whitted	4.5x
whitted2	4.8x
wood	2.0x

- Paketgröße $k = 4$
- Shader liegen skalar vor
- werden automatisch paketisiert
- Durchschnittlich 3.2x Speedup des gesamten Renderers
- Manchmal superlinear:
  $\Longrightarrow$ Lokalität
- Für daten-parallele Programme generell einsetzbar

Zusammenfassung

- Shader werden in maschinenunabhängigem Zwischencode gehalten
- Als Eingabesprache kann jede beliebige Shader-Sprache verwendet werden
- Oder „gewöhnliche“ Hochsprachen, wie C/C++
- hochoptimierender JIT im Renderer
- API Adapter werden „wegoptimiert“
- Plattform-unabhängige automatische Paketisierung:
  3.2x Speedup auf Benchmarks

Ausblick

- Unterstützung anderer Hardware
  - GPUs (in Arbeit), Cell, Larrabee (set $k = 16$ ☺)
- Graphik-spezifische Code Transformationen:
  - Debugging: Visualisiere Inhalte von Variablen
  - Ableitungen: Automatische Differenzierung auf Code
  - Genauigkeit: Affine Arithmetik
- Datenfluss-Programmierung in der Hochleistungs-Graphik
  - Renderer als Netz von Shadern
  - Ausnutzung von Kohärenz/Lokalität
  - Adaptive Übersetzung
  - Multi-Plattform