

Computergrafik

Universität Osnabrück, Henning Wenke, 2012-07-02

Noch Kapitel XV:



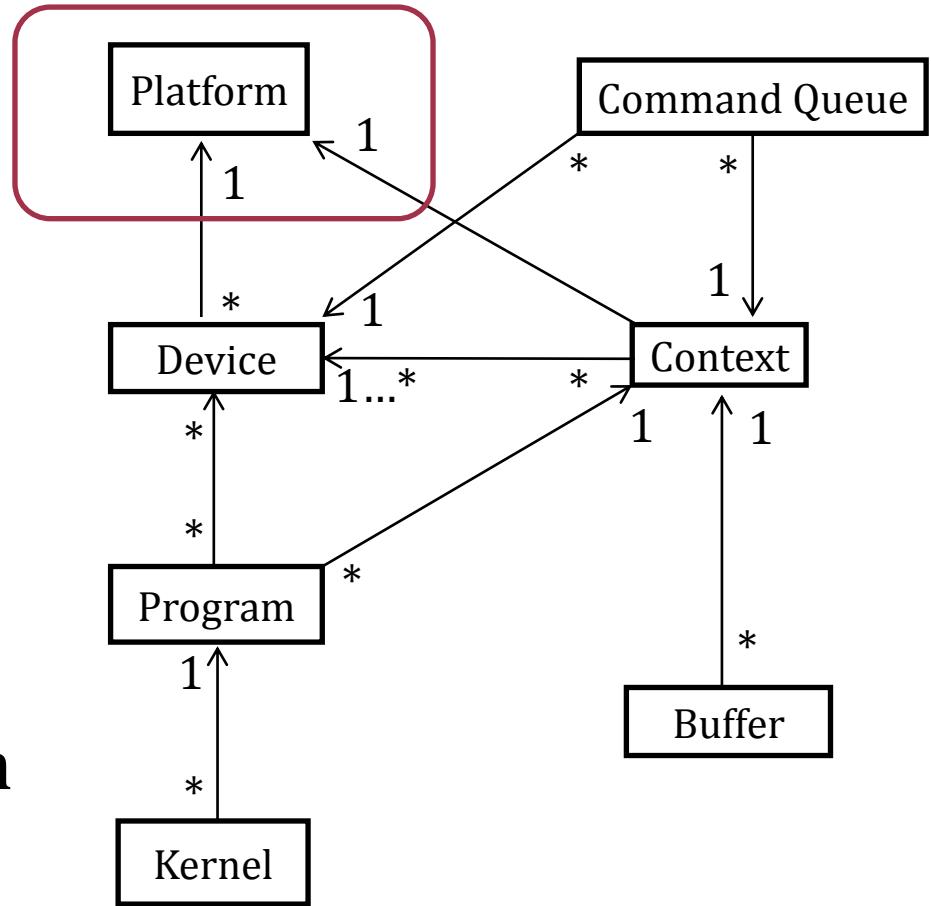
Parallele Algorithmen mit OpenCL

15.9

Programmierung mit OpenCL (Auszug)

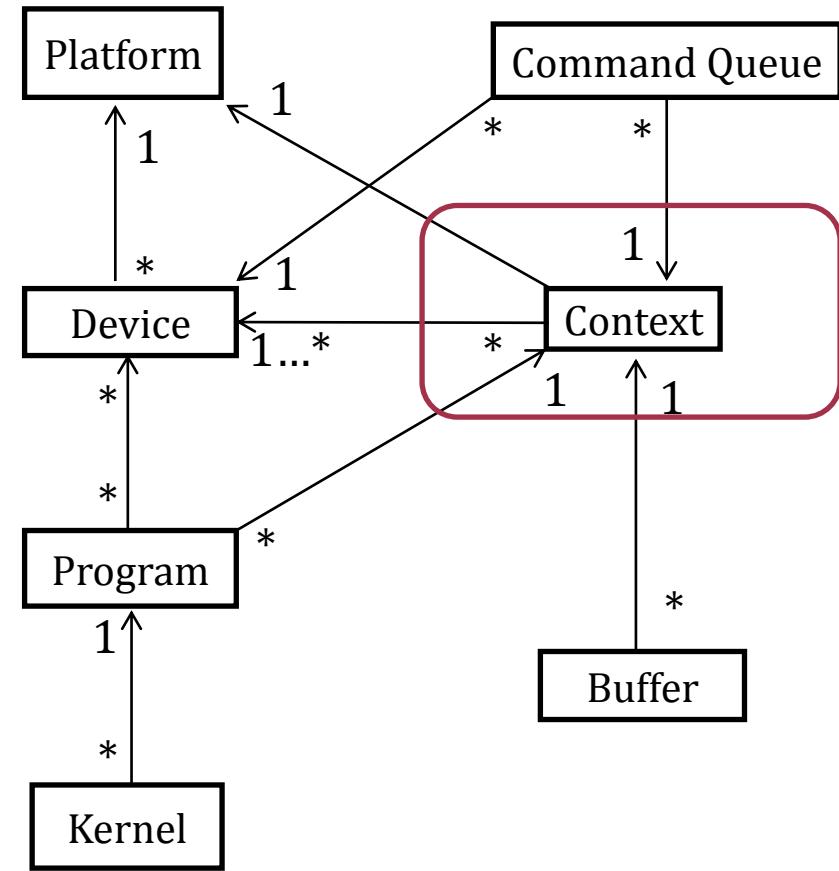
Host / Platform

- Es gibt immer genau einen Host
 - Interface zum restlichen Programm
 - Erzeugt mindestens eine Plattform
- Beispiel AMD, kann Devices GPU und CPU enthalten
- Verschiedene Plattformen kommunizieren über Host



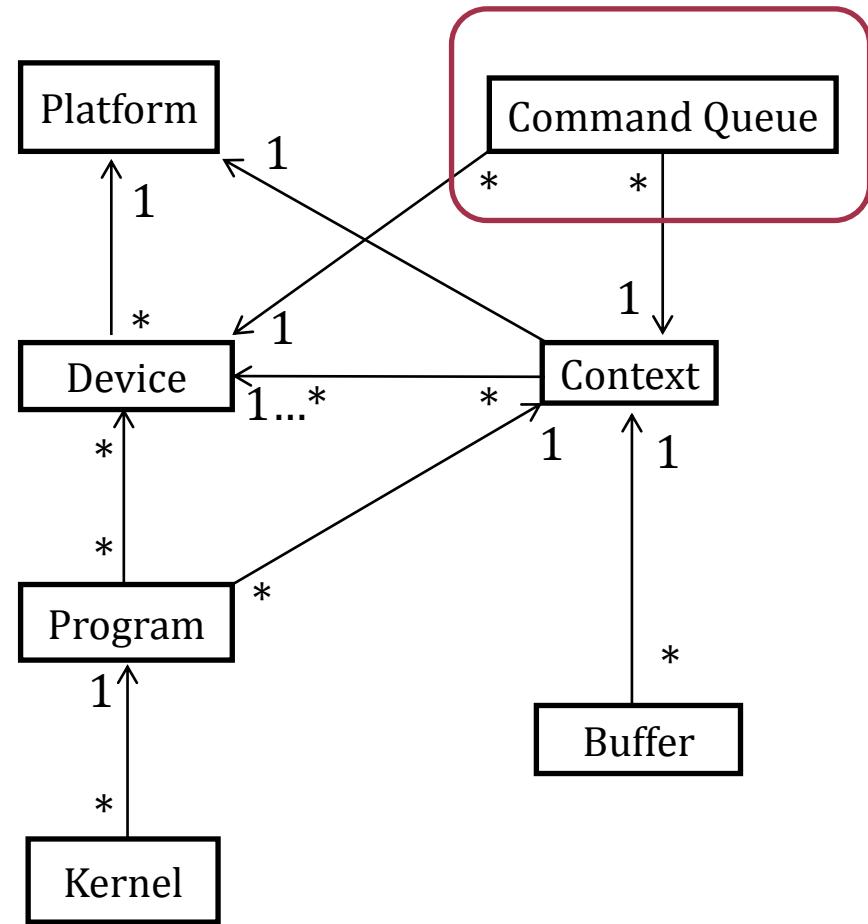
Context

- Context definiert gemeinsame Umgebung innerhalb einer Plattform für:
 - Devices
 - Queues
 - Buffer
 - Kernels
 - Program Objects
- Kann optional zur Verwendung mit OpenGL konfiguriert werden



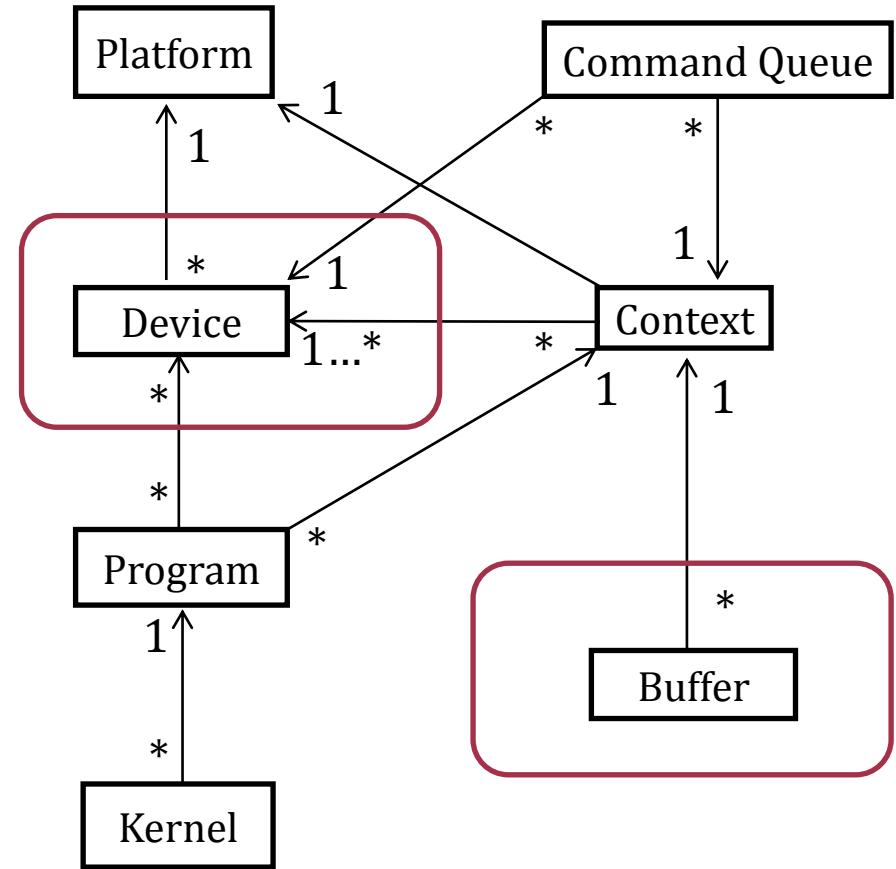
Command Queue

- Wird für bestimmten Context & Device erzeugt
- Regelt Interaktion mit Host
- Befehle werden asynchron zum Host und anderen Queues ausgeführt
- Drei Arten Befehle
 - Ausführung der Kernel
 - Datenoperationen
 - Synchronisation
- Queue kann In-Order oder Out-Of-Order ausgeführt werden



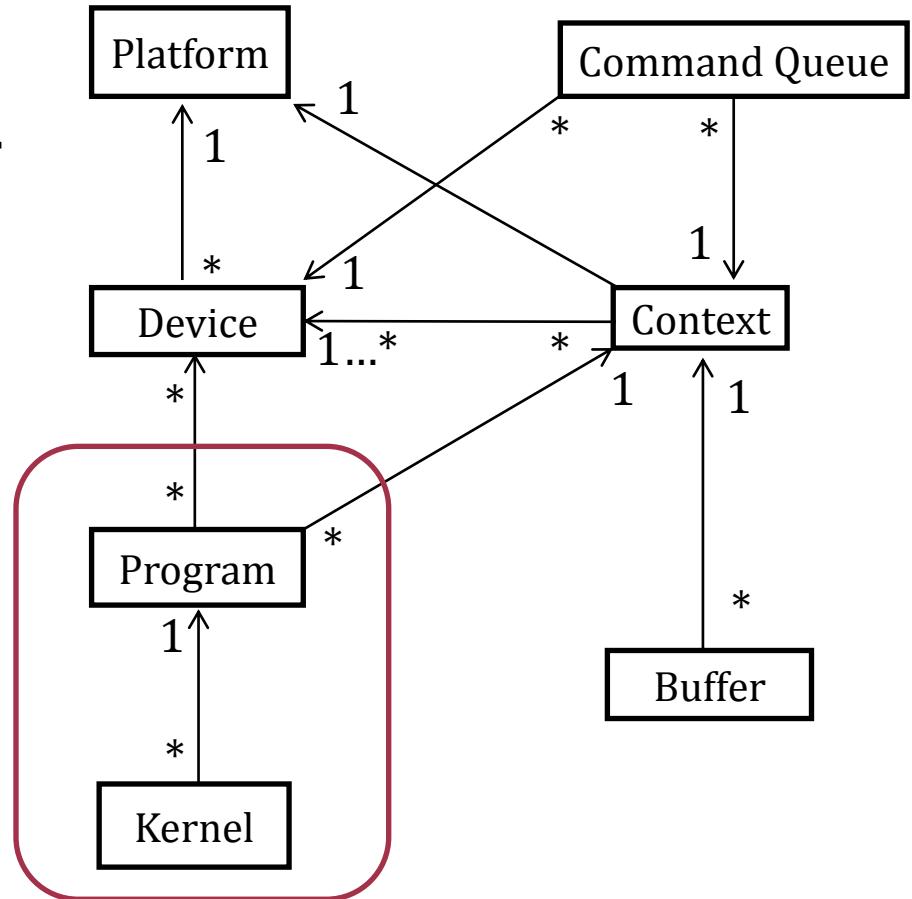
Device & Buffer

- Devices Repräsentieren Gerät zur Ausführung der Befehle einer Queue
- Gehören zu einer Platform
- Buffer repräsentieren lineare Byte Arrays zur Verarbeitung durch Kernel
- CLMem Objects: Superklasse der Buffer



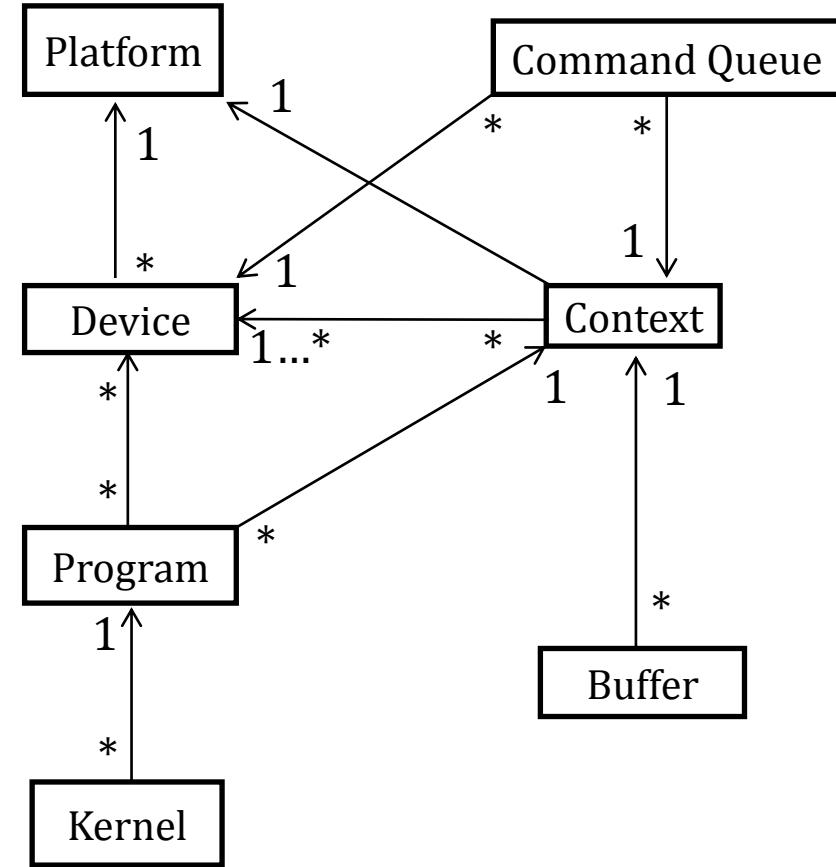
Program Object & Kernel

- Kernel Object kapselt eine Funktion, die zur parallelen Ausführung auf einem OpenCL-Device geeignet ist
- Program Object enthält mindestens einen Kernel und zusätzlich ggf. Funktionen & Konstanten
- Wird zur Laufzeit für best. Devices übersetzt



Diese Veranstaltung

- Eine Platform
- Ein Context
- Ein Device
- Eine Queue (In-Order)
- Ein Program
- Ein oder mehrere Kernel
- Mehrere Buffer
- Initialisierungsprozess
gegeben
- Beispiel dazu: Webseite
der Veranstaltung



Erzeugen einer Command Queue

```
// Liefert „echtes“ Java oder C++ Objekt, keine id.  
// Aber auch nur zum bequemeren arbeiten mit dem entsprechenden  
// Objekt des Devices  
CLCommandQueue clCreateCommandQueue(  
    CLContext context,           // Queue ist Device und Context zugeordnet  
    CLDevice device,            // Siehe Abhängigkeiten auf vorherige Folien  
                               // Wird nachfolgend ignoriert  
  
    long properties,           // 0 nichts (diese Veranstaltung), oder:  
                               // CL_QUEUE_PROFILING_ENABLE  
                               // CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE  
  
    IntBuffer errcode_ret) // liefert Fehlercode. Wird durch  
                           // unseren Wrapper behandelt.  
                           // Argument verschwindet daher.  
);
```

Erzeugen eines Kernels

```
// Erzeugt ein CLProgram Objekt basierend auf den in "sourceCode"  
// enthaltenen Kernen, Funktionen und Konstanten.  
CLProgram clCreateProgramWithSource("sourceCode", ...);  
  
// Kompiliert und linked „program“. Liefert eine Fehlerkonstante  
int clBuildProgram(CLProgram program, ...);  
  
// Erzeugt ein CLKernel Objekt, welches einen Kernel repräsentiert  
CLKernel clCreateKernel(  
    CLProgram program,           // Ausführbares CLProgram Objekt, welches  
                                // den Kernel enthält  
    String kernel_name,         // Name des Kernels im Sourcecode des  
                                // CLPrograms  
    ...  
);
```

Daten

```
// clCreateBuffer erzeugt einen Buffer (lineare Bytesequenz).  
// Wird durch ein CLMem Objekt repräsentiert.  
CLMem clCreateBuffer(...);  
  
// Fordert den für data benötigten Speicher an und kopiert die Daten  
CLMem clCreateBuffer(CL_MEM_COPY_HOST_PTR, FloatBuffer data,...);  
  
// Fordert nur den benötigten Speicher (in bytes) an  
CLMem clCreateBuffer(long host_ptr_size, ...);  
  
// Erzeugt Objekt zum Zugriff auf ein OpenGL Buffer Object. Dieses muss  
// mit glBufferData bereits eine Größe zugewiesen bekommen haben  
CLMem clCreateFromGLBuffer(  
    int bufobj, // id des OpenGL Buffer Objects  
...);  
// Sonstige, z.B.:  
CLMem clCreateFromGLTexture2D(...);  
CLMem clCreateFromGLTexture3D(...);
```

Kopieren

```
// Kopiert Daten aus einem Java Buffer Objekt in einen OpenCL Buffer.  
// Wird asynchron durch eine Queue ausgeführt  
int clEnqueueWriteBuffer(  
    CLCommandQueue command_queue,  
    CLMem buffer, // OpenCL Buffer, welcher die Daten erhalten soll  
    <java.nio.Buffer>, // Die zu kopierenden Daten  
    ...  
) ;  
  
// Auslesen der Daten aus einem OpenCL Buffer analog  
int clEnqueueReadBuffer(  
    CLCommandQueue command_queue,  
    CLMem buffer,  
    <java.nio.Buffer>  
    ...  
) ;
```

Daten anbinden

```
// Gegeben: Kernelobjekt myKernel
CLKernel myKernel = clCreateKernel(...,"akernel");
CLMem aBuffer = clCreateBuffer(...);

// Bindet ein Buffer Object an globale Variable.
myKernel.setArg(0,      // Index der Variable. Hier: Die Erste (a)
                aBuffer // Anzubindende Daten.
);
// Konstanten werden analog angebunden
myKernel.setArg(1,      // Index der Variable. Hier: Die Zweite (b)
                90000   // Übergebener Wert
);

// Für Local Memory muss lediglich Speicher angefordert werden
myKernel.setArgSize(2,  // Index der Variable. Hier: Die Dritte (c)
                    1024 // Benötigter Speicher in bytes
);

kernel void aKernel(
    global float4* a,
    const int      b,
    local float4*  c;
) { ... }
```

Ausführen eines Kernels

```
// Hängt den OpenCL Kernel kernel zur Ausführung in die Command Queue
// command_queue ein. Wird asynchron ausgeführt.
// org.lwjgl.PointerBuffer kapselt Werte des Typs long (≈LongBuffer)
int clEnqueueNDRangeKernel(
    CLCommandQueue command_queue,
    CLKernel kernel,
    int work_dim, // Dimension der Kernel-Indizierung
    PointerBuffer global_work_offset, // Globaler Offset pro Dimension
    PointerBuffer global_work_size, // Globale Anzahl Work Items pro Dim
    PointerBuffer local_work_size, // Lokale Anzahl Work Items pro Dim
    PointerBuffer event_wait_list, // Events, die completed sein müssen
                                    // bevor dieser Befehl ausgeführt wird
    PointerBuffer event // Kann Id eines Events liefern.
                        // Damit kann z.B. bestimmt werden, ob die
                        // Ausführung bereits abgeschlossen ist
                        // Auch lässt sich so die Ausführungszeit
                        // messen
) ;
```

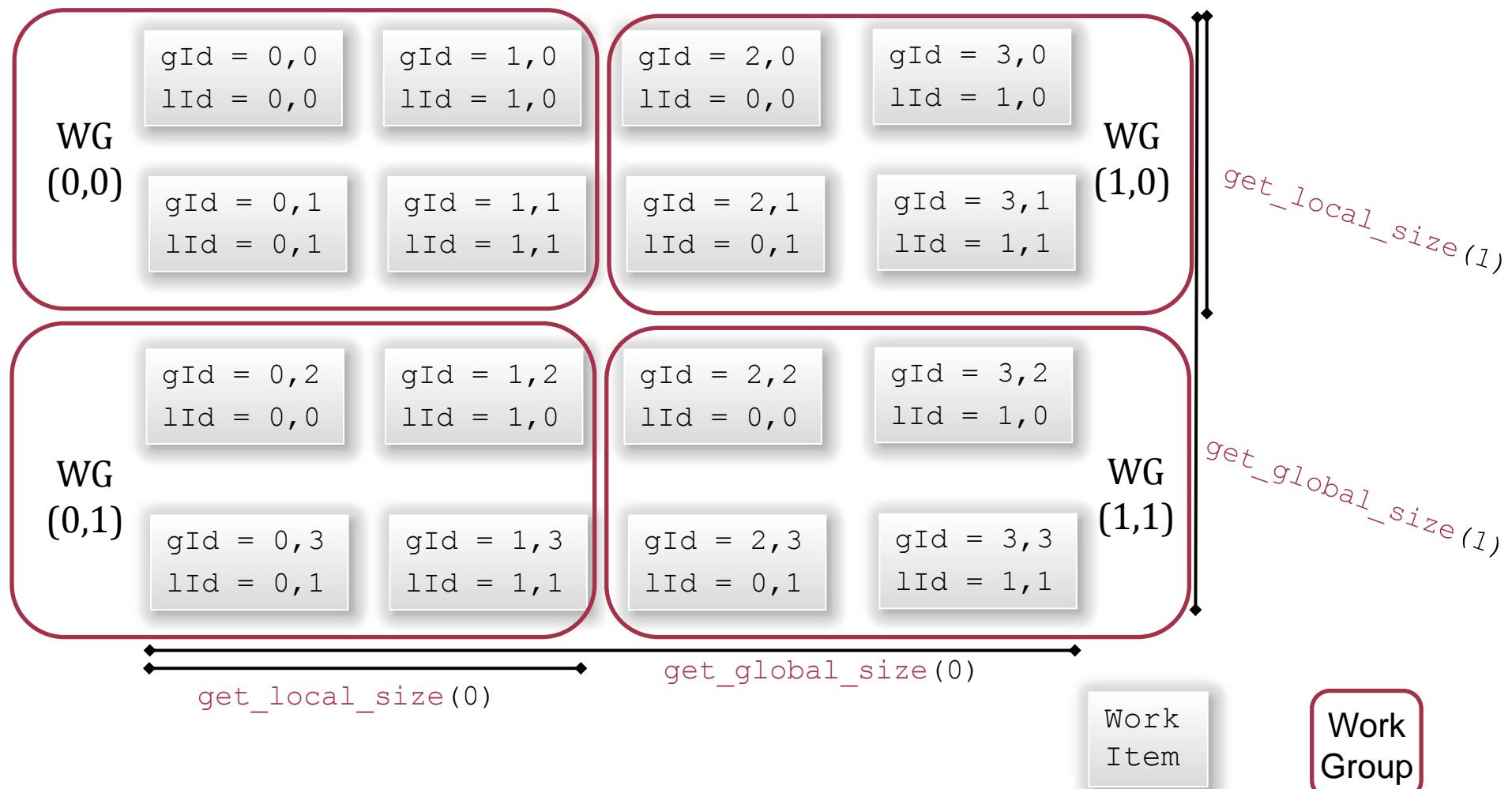
Ausführen eines Kernels: Beispiel

```
PointerBuffer globalSize = BufferUtils.createPointerBuffer(1);
globalSize.put(0, 256000); // Anzahl der zu erzeugenden Kernelinstanzen

PointerBuffer localSize = BufferUtils.createPointerBuffer(1);
localSize.put(0, 256); // Muss ganzzahliges Teiler von globalSize sein
                      // außerdem durch Hardware begrenzt
PointerBuffer globalOffset = BufferUtils.createPointerBuffer(1);
globalOffset.put(0, 0); // Globale Indices bei 0 beginnen. Hier: Immer

clEnqueueNDRangeKernel(
    myQueue,
    myKernel,
    1, // Verwende eindimensionale Indizierung
    globalOffset,
    globalSize,
    localSize,
    null, null // Events verwenden wir nicht, alles in-Order
);
```

Nachtrag: 2D Indizierungsmodell



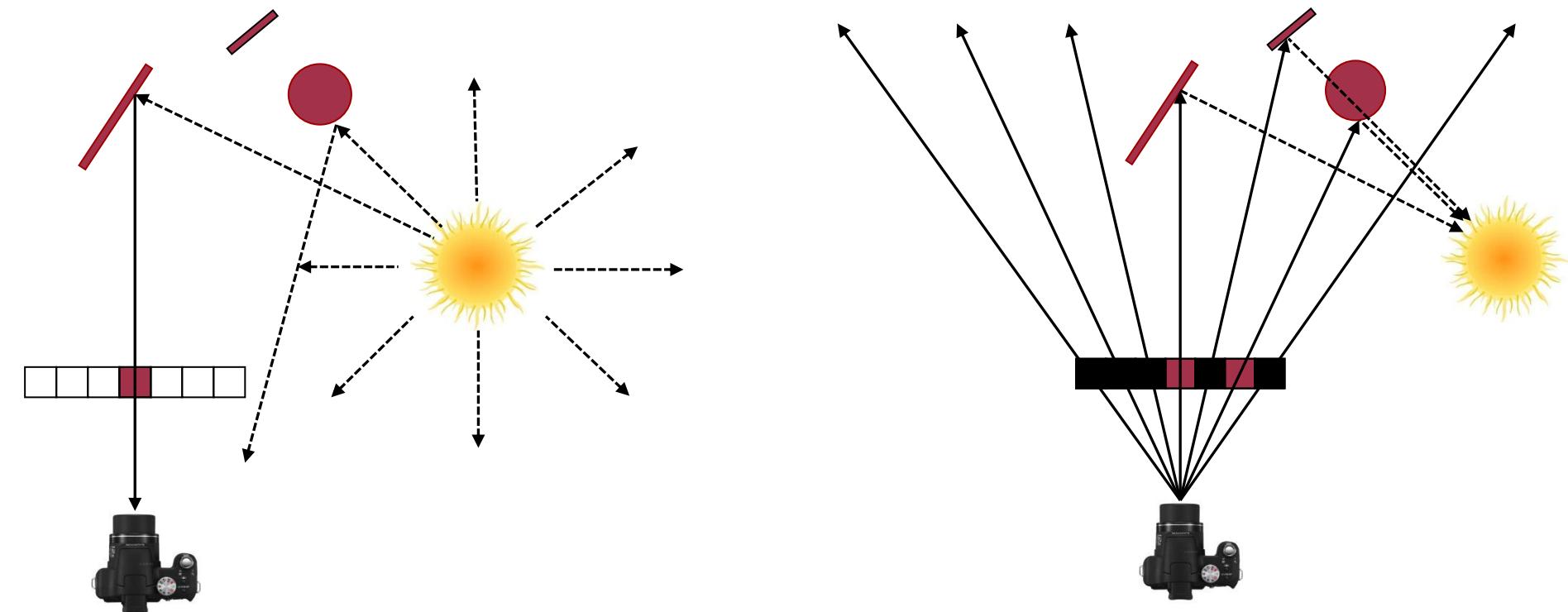
- 'gId': Globale Id eines Work Items: x: `get_global_id(0)`; y: `get_global_id(1)`;
- 'lId': Lokale Id eines Work Items: x: `get_local_id(0)`; y: `get_local_id(1)`; ...
- Hier: global_size: (4, 4), local_size: (2, 2)

Synchronisation

```
// Blockiert bis command_queue abgearbeitet ist.  
// Globaler Synchronisierungspunkt der Queue  
int clFinish(  
    CLCommandQueue command_queue  
) ;
```

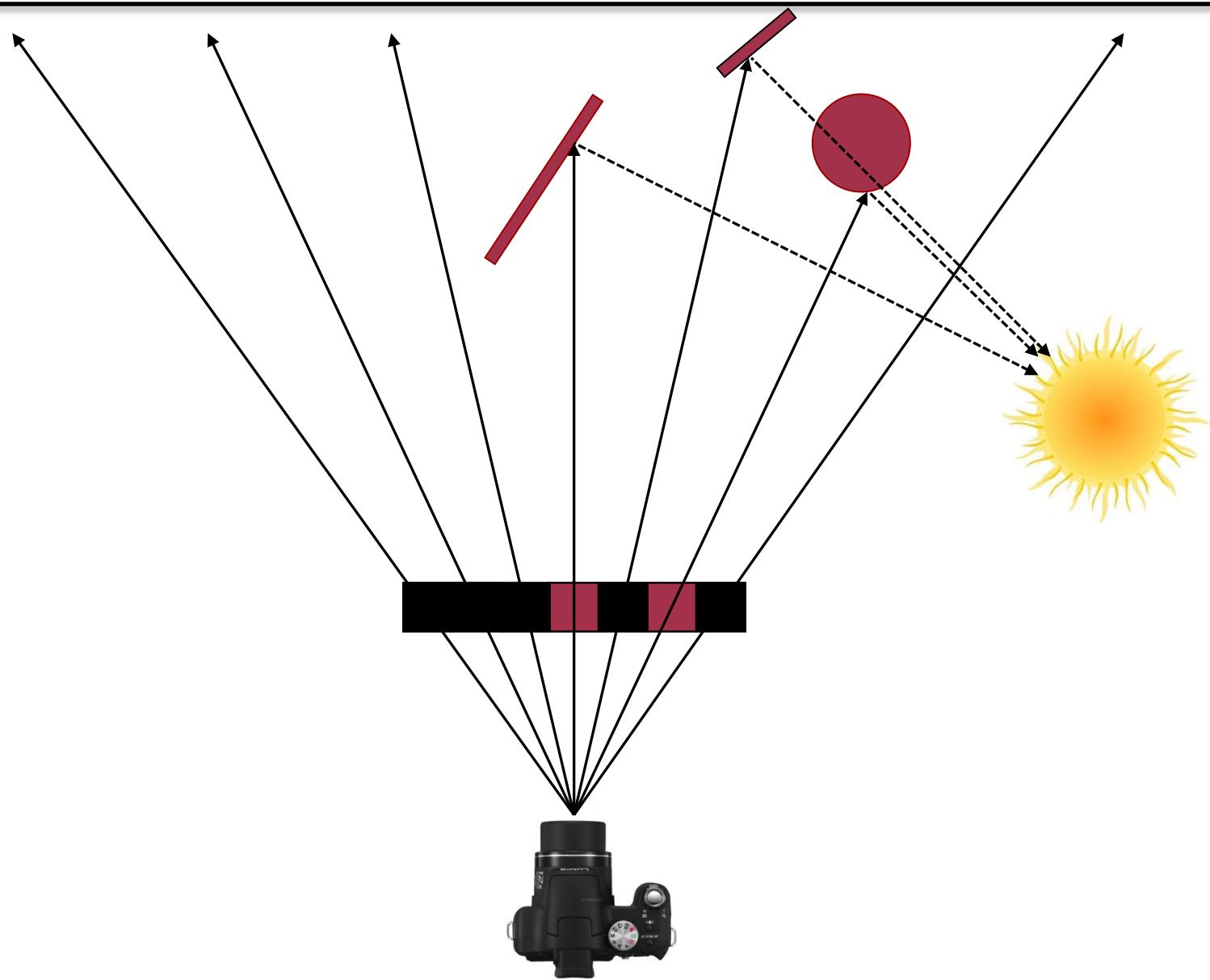
```
// OpenGL Objekte können nicht von OpenGL und OpenCL gleichzeitig  
// verarbeitet werden. Müssen daher vor Verwendung durch OpenCL  
// in Beschlag genommen und anschließend wieder abgegeben werden  
int clEnqueueAcquireGLObjects(CLMem mem_object, ...);  
int clEnqueueReleaseGLObjects(CLMem mem_object, ...);
```

Kapitel XVI

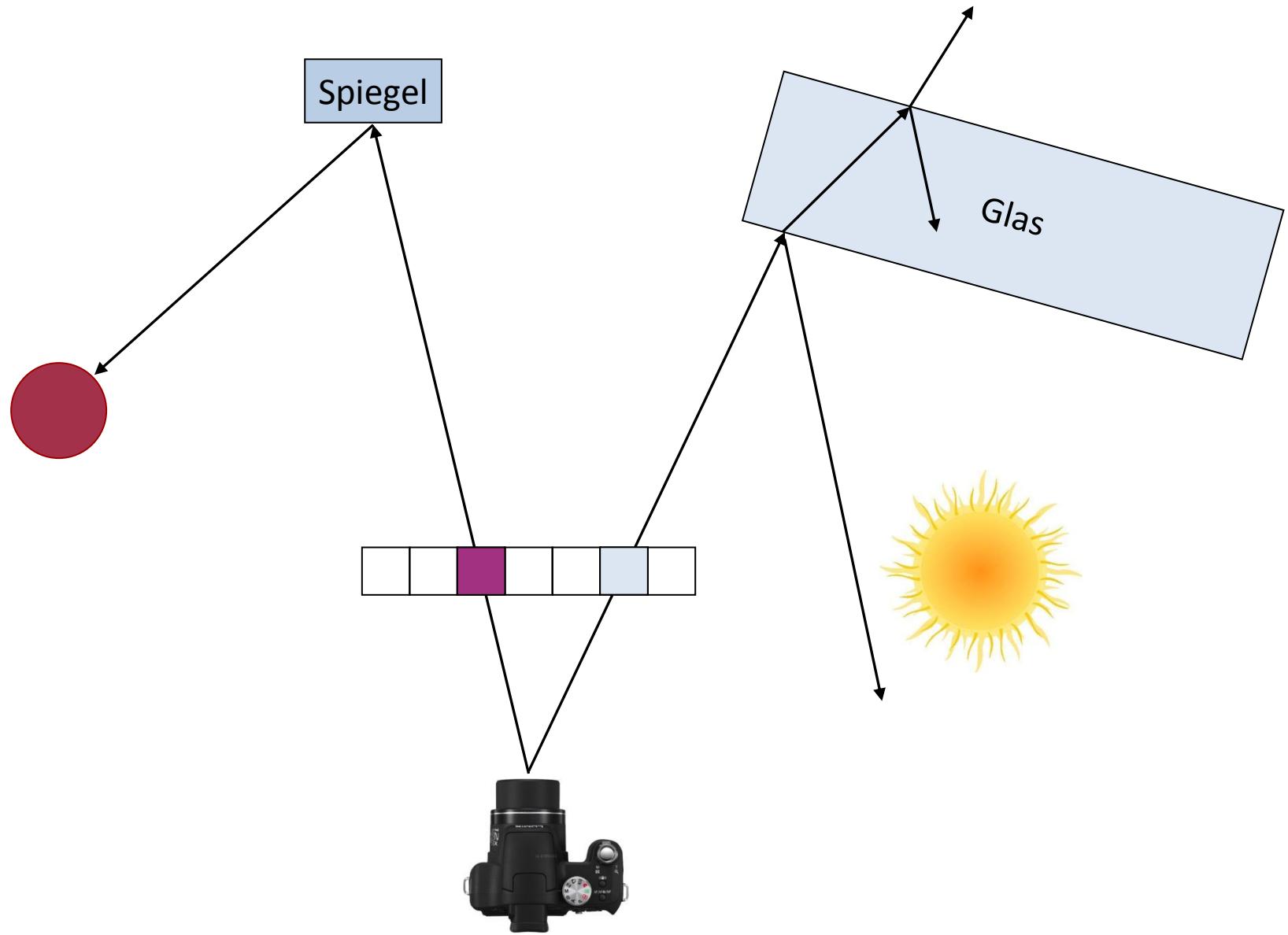


Realtime Ray Tracing

Variante: Ray Tracing mit Shadow Rays



Variante: Rekursives Ray Tracing



Implizites vs. Explizites Rendern

➤ Explizites Rendern: Rastergrafik

- Geht von der Szene aus
- Fragt: Welche Pixel werden von diesem Objekt eingefärbt?
- For all primitives pr do
 - For all Pixels overlaped by pr do
 - calculate Color of Pixel

➤ Implizites Rendern: Z.B. Raytracing

- Geht von Bildebene aus
- Fragt: Welcher Teil der Szene färbt dieses Pixel ein?
- For all pixels pi do
 - For all primitives pr do
 - check influence of pr on pi

16.1

Implizites 2D Rendern (naiver Ansatz)

Implizite Kreisdarstellung

- Für Punkt P auf Rand eines Kreises um M gilt:

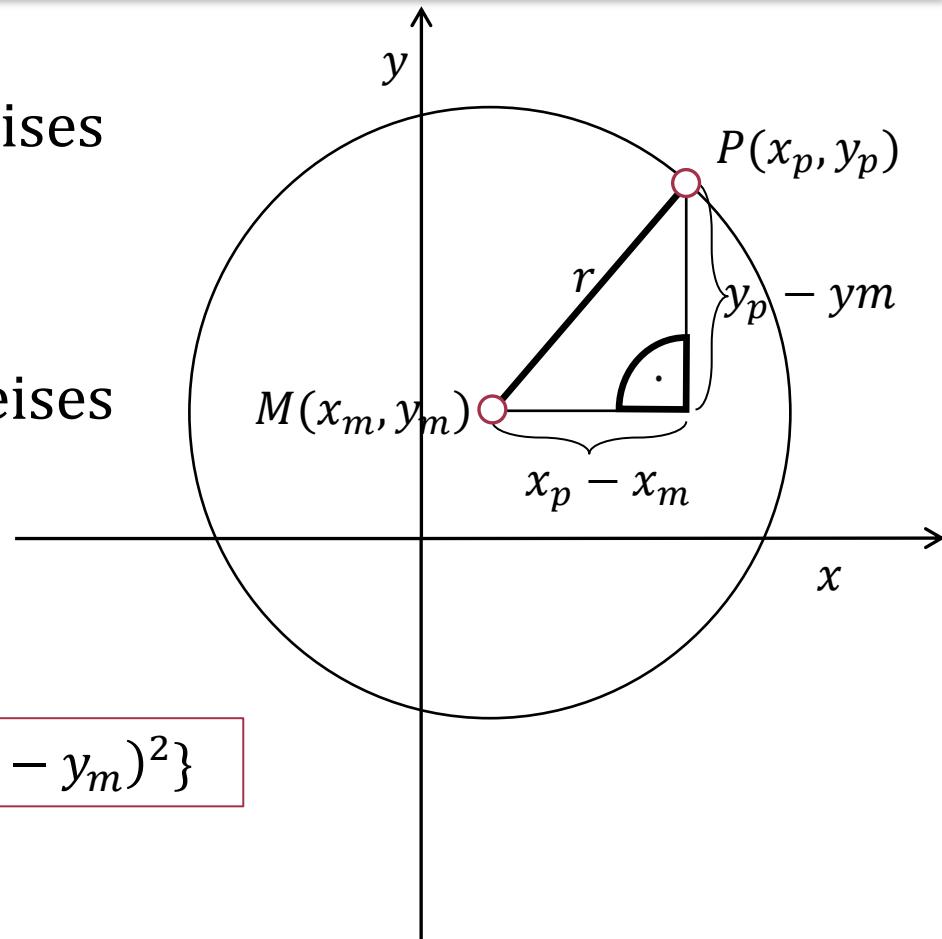
- $r^2 = (x_p - x_m)^2 + (y_p - y_m)^2$

- Für Punkt P im Inneren des Kreises gilt:

- $r^2 > (x_p - x_m)^2 + (y_p - y_m)^2$

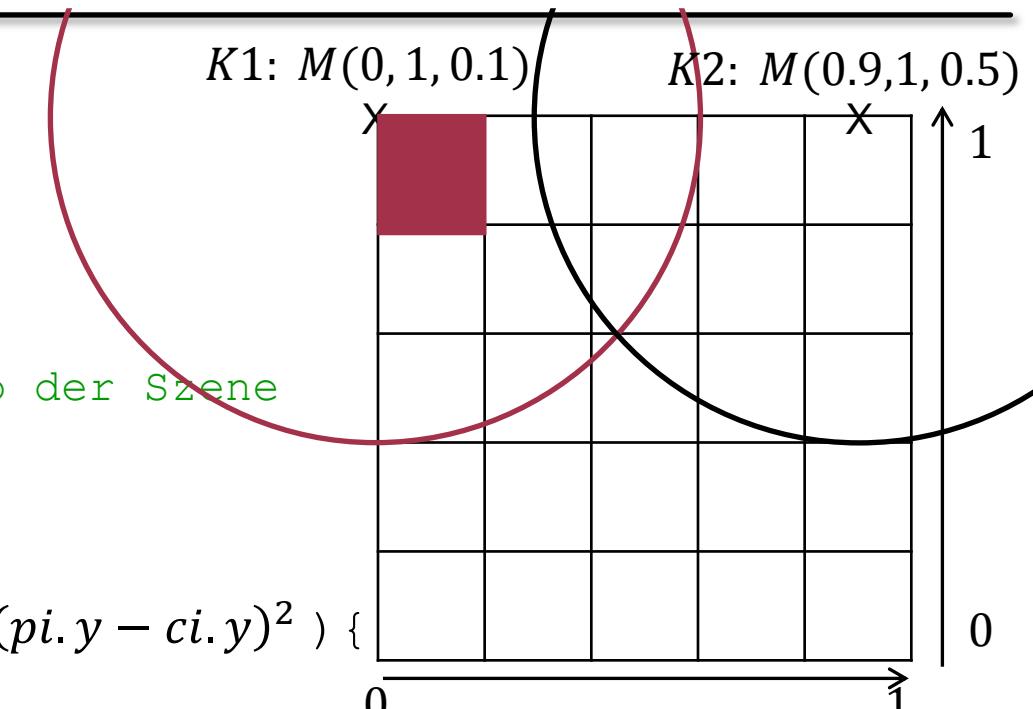
- Implizite Kreisgleichung:

$$K_{r,M} = \{(x, y) \mid r^2 \geq (x - x_m)^2 + (y - y_m)^2\}$$



Implizites Kreisrendern

- 2 Kreise definiert durch:
 - Position (x, y)
 - Tiefe z $\in [0,1]$ für Ordnung
 - Radius = 0.5, Farbe
- Initialisiere alle Pixel
 - color: Weiß
 - minDepth = 2; // Außerhalb der Szene
- For all pixels pi do
 - For all circles ci do
 - if ($ci.z < minDepth \&&$
 $ci.r^2 > (pi.x - ci.x)^2 + (pi.y - ci.y)^2$) {
 - color = ci.color;
 - minDepth = ci.z;



```
    }  
    pix = 0.1; piy = 0.9; // Pixelscoords  
    // Circle 1: cix = 0, ciy = 1, cz = 0.1  
    0.1 < 2 // n. Verdeckt  
    und 0.25 > (0.1-0)^2 + (0.9-1)^2 <=> 0.25 > 0.02 // Drin  
    Farbe <- rot;  
    minDepth <- 0.1;  
    // Circle 2: cix = 0.9, ciy = 0.9, cz = 0.5  
    0.5 < 0.1 // Falsche Aussage => verdeckt
```

Implementations Grundlagen

- Grundprinzip aller Implementationen hier:
1. Erzeuge eine 2D RGB Textur „`renderedImage`“, welche exakt der gewünschten Bildauflösung entspricht
 2. Starte einen OpenCL 2D „Renderkernel“, dessen globale Work Item Konfiguration Breite und Höhe des Fensters in Pixeln entspricht
 3. Jedes Work Item berechnet Farbe eines Pixels und schreibt diese in `renderedImage`
 4. OpenGL rendert ein Viereck, welches exakt den Sichtbereich ausfüllt und verwendet als Farbtextur `renderedImage`

Implementations Grundlagen II

- Bildauflösung
 - Width = 1024
 - Height = 1024
- Float Datenbuffer “circles“ für Structs Circle

```
{x0, y0, r0, depth0, r0, g0, b0, a0,  
 x1, y1, r1, depth1, r1, g1, b1, a1, ...}
```
- Erzeuge Kernel “implicit2DRenderer“ (Folie 31)
- Starte Kernel mit:
 - Angebundenen Daten
 - dim: 2
 - global_work_size (1024, 1024)

Struct für Kreise

- C-Structs vereinen verschiedene Daten zu einem Verbund, um sie bequemer verwenden zu können
- Komponenten werden über Namen angesprochen
- Komponenten liegen nacheinander im Speicher

```
typedef struct Circle { // 8 floats, die einen Kreis repräsentieren
    float x;           // x-Koordinate des Mittelpunkts
    float y;           // y-Koordinate des Mittelpunkts
    float r;           // Radius
    float depth;       // Tiefe, nur für Ordnung
    float4 color;      // Farbe des Kreises
} Circle;
```

Verdeckungs- & Überlapptest

```
// Testet, ob Circle c das Pixel mit pixelPos überlappt und, wenn ja,  
// ob Kreis näher als ein ggf. ebenfalls überlappender anderer Kreis  
// der Tiefe minDepth ist.  
// Wenn das der Fall ist liefert die Funktion true  
bool overlaps_N_Nearer(Circle c, int2 pixelPos, int minDepth) {  
    // Werte implizite Kreisgleichung für Position des Pixels aus  
    float dx = (float) pixelPos.x - c.x;  
    float dy = (float) pixelPos.y - c.y;  
    if(dx * dx + dy * dy < c.r * c.r && minDepth > c.depth)  
        return true;  
    else  
        return false;  
}
```

Funktionsweise des Kernels

- Für jeden Pixel, repräsentiert durch ein Work Item, wird folgendes ausgeführt:
 - Teste alle Kreise mit `overlaps_N_Nearer`
 - Pixel wird von Kreis(en) überlappt?
 - Ja: Schreibe Farbe des Nächsten in Textur an Stelle dieses Pixels
 - Nein: Schreibe Hintergrundfarbe

Kernel

```
kernel void implicit2DRenderer(
global Circle*      circles,    // Siehe Folie 28
const   int          circleCnt,
write_only image2d_t rederedImage) {
    int2 pixelPos = (int2)(get_global_id(0), get_global_id(1));

    float minDepth = 90000; // > maximale Szenentiefe
    float4 color = (float4) (0.0, 0.0, 0.0, 0.0); // Hintergrundfarbe
    for(int i = 0; i < circleCnt; ++i) {
        Circle c = circles[i];
        if(overlaps_N_Nearer(c, pixelPos, minDepth)) { // Folie 29
            // Kreis setzt sich durch. Speichere Werte lokal
            minDepth = c.depth;
            color = c.color;
        }
    }
    // Schreibe Farbe des nächsten Kreises. Keiner? Hintergrundfarbe
    write_imagef(rederedImage, pixelPos, color);
}
```