Real Time Ray Tracing

Endpräsentation

Stephan Beck Andreas Bernstein Daniel Danch

Bauhaus Universität Weimar

WS 2004/05

Gliederung

Einführung

- Ray-Tracing ist eine der ältesten Techniken zur Erzeugung photorealistischer Bilder
- Weist gegenüber konventioneller Rasterisierung markante Vorteile auf
- Nachteil: hoher Rechenaufwand resultiert in geringerer Performance

Beschleunigungsansätze

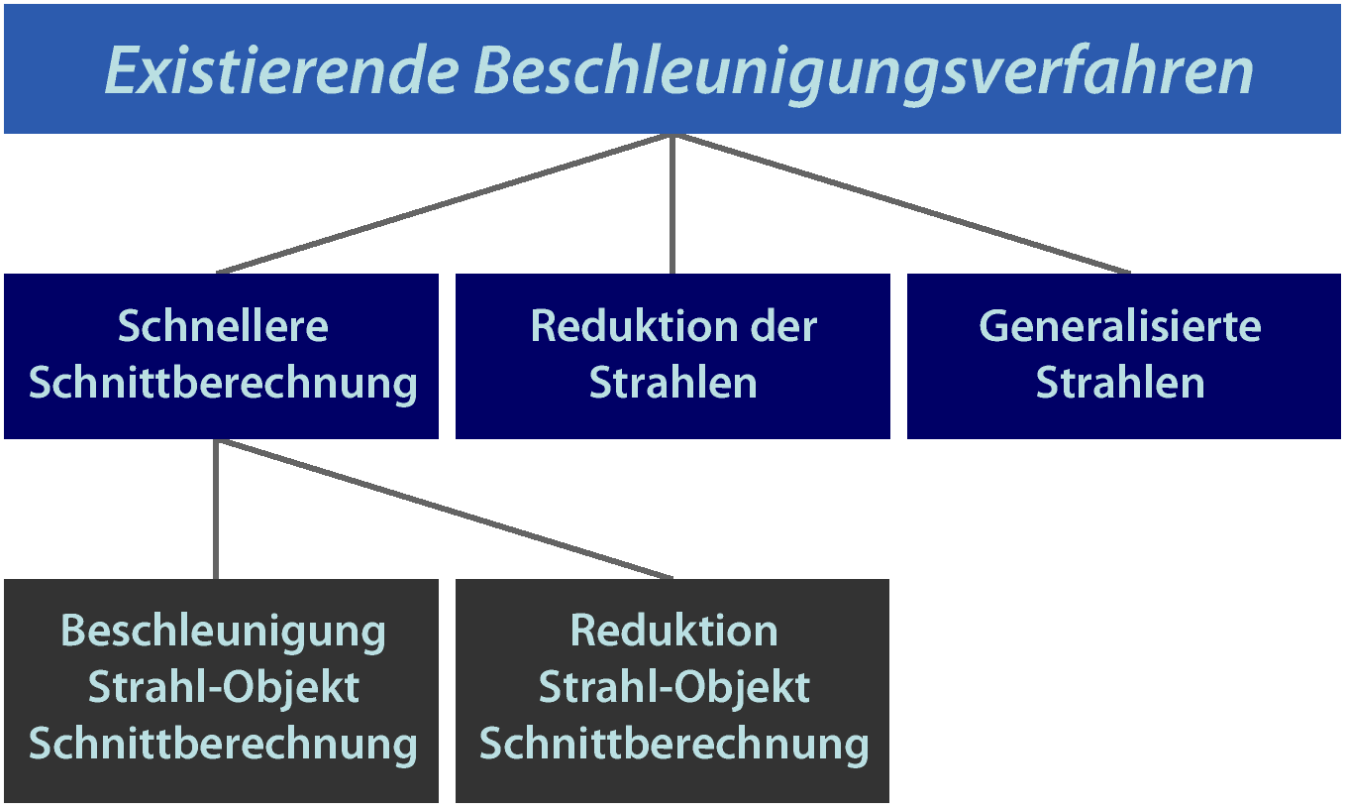
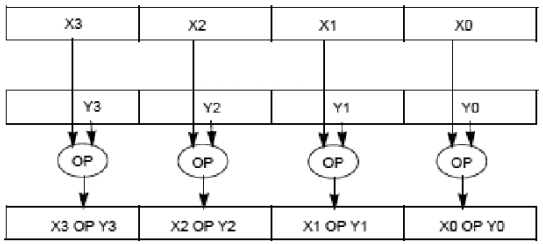


Abbildung: Beschleunigungsansätze

Features

- Iteratives Ray-Tracing
  - Jede Strahlgeneration wird vollständig konsekutiv abgearbeitet
  - Farbbeiträge werden auf entsprechendes Pixel aufaddiert
- SAH basierter Kd-Baum
- Verwendung von Dreiecken
- hochoptimierte Schnittroutine
- Nutzung von SSE
- Alignment der Datentypen
- Multithreaded
- C++ unter Benutzung der Boost-Libraries
- .OBJ Loader

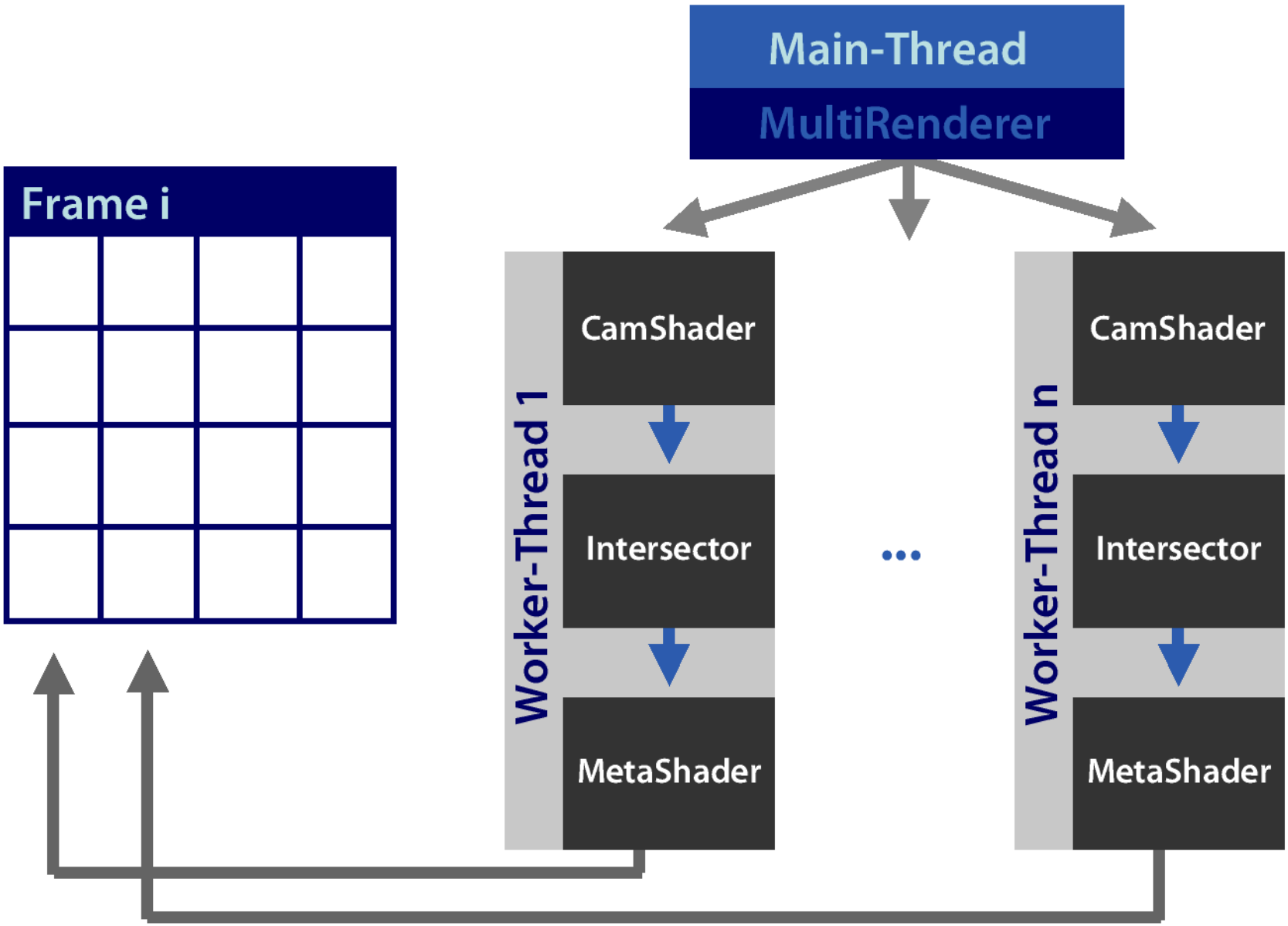
SIMD-Erweiterung



SSE-Erweiterung

- Vektorisierung von arithmetischen Operationen
- Simultane Ausführung 4 identischer Operationen in 128-Bit Registern
- SSE-Datentypen setzen 16-Byte alignment vorraus

Multithreading



Stand zur Zwischenpräsentation

Pinocchio

- Real Time Raytracing von komplexen Modellen auf der CPU
- Speadboat
  - besitzt ca. 16000 Dreiecke
  - 5-10 fps bei 512x512

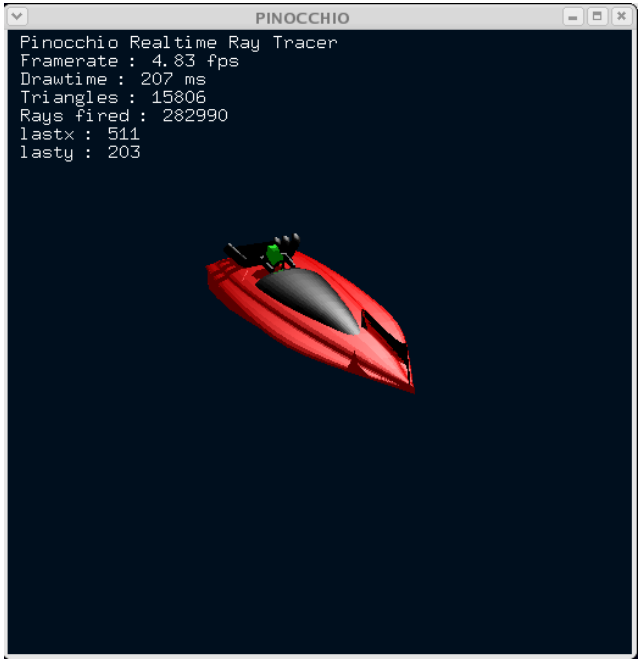


Abbildung: speedboat

Pinocchio goes GPU

- the story continues ...

Verwendung der GPU innerhalb von Pinocchio - Übersicht

- Vorbetrachtungen
- Pinocchios Model Abstraktion
- Shading auf der GPU
- Schatten auf der GPU
- Primärstrahlen auf der GPU - ID-Shading
- Aufgaben der GPU innerhalb des Frameworks - resultierende Renderpasses
- Verschränkung des Rendering Ablaufes

Vorbetrachtungen

- Warum die OpenGL Rendering Pipeline hinzuznehmen?
  - OpenGL rendert Polygone extrem schnell (mind. 300 Mio Vertices/s)
  - Pinocchio arbeitet mit Dreiecken - OpenGL auch:

```
glBegin(GL_TRIANGLES);
{
  glNormal3f(0.0,0.0,1.0);
  glVertex3f(0.0,0.0,0.0);
  glVertex3f(1.0,0.0,0.0);
  glVertex3f(0.0,1.0,0.0);
}
glEnd();
```

- Eine Pinocchio Szene kann somit generell auch in einem OpenGL Renderpass verarbeitet werden!

Vorbetrachtungen

- Aber logisch ist auch:
- (Standard-) OpenGL ist nicht gleich Raytracing!
- Deshalb: Analyse der Möglichkeiten die OpenGL (+ Shading Language) für unser Framework bietet.
- in Betracht kommen:
  - Phong-Shading mit Einsatz von Vertex + Fragment Shadern.
  - Techniken zur Schattengenerierung
  - Verfahren zum Image Processing (Filtering)
  - Abarbeiten der Primärstrahlen
- Klar ist: Unser Framework soll die Rechenpower der GPU für viele Aufgaben verwenden.

Pinocchios Model Abstraktion

- Wie organisieren wir unsere Szene?
- Ein Szene wird z.B. in blender erstellt und dann als obj Datei exportiert.
- Derzeit werden nur Dreiecke unterstützt - somit Triangulierung notwendig.
- Eine obj (+mtl) Datei entspricht somit einem Pinocchio Model.
- Deshalb: Kapselung der Geometriedaten in der Klasse Model und bereitstellen diverser Methoden und Datenstrukturen.

Pinocchios Model Abstraktion

- Erzeugung eines Models:

```
Model* myModel = new Model(``cow.obj'');
```

  - ...obj + mtl Datei wird geparst.
  - ...Datenstrukturen für Raytracing und OpenGL werden aufgebaut:
    - Container von Eckpunkten, Normalen, Materialien
    - Struktur zum Referenzieren der Dreiecksdaten
    - Diplay-Listen im OpenGL Server Speicher
  - ...ein Dreieck ist dann unter seiner eindeutigen Nummer (triID) zu erreichen:

```
// the Vertex A of Triangle #7777
const Point3D& vertex =
   Triangles_[7777].vertices_[0];

// the Material of Triangle #7777
const Material& =
   Triangles_[7777].material_;
...
```

Pinocchios Model Abstraktion

- Die Klasse Model stellt u.a. folgende Methoden zur Verfügung:

```
...
// Methods used by OpenGL
void drawRAW();  // calls Display List for ShadowMap Generation
void drawID();   // calls Display List for ID Shading
void draw();     // calls Display List for Shading

// Methods used by Raytracer
const Point3D& getVertex(const uint32_pco_t triID,
                         const uint8_pco_t index);

const Vector3D& getVertexNormal(const uint32_pco_t triID,
                                const uint8_pco_t index);

const Material& getMaterial(const uint32_pco_t triID);

void calculateIntNormal(const uint32_pco_t triID,
                        Vector3D &result,
                        const float 32_pco_t u,
                        const float32_pco_t v);
...
```

Gouraud-Shading auf der GPU

- Einfache Verwendung einer Instanz der Klasse Model:

```
...
do_some_setup();
glShadeModel(GL_SMOOTH);
// send the Geometrie
// down the Pipeline
myModel->draw();
...
```

Abbildung: Gouraud-Shading

- Fazit:
  - Gouraud-Shading ist EckpunktShading
  - Das ist nicht für unser Framework geeignet:
    - keine Highlights innerhalb Primitive, Highlights zittern bei Bewegungen.
  - Wir brauchen Phong-Shading auf der GPU!

Phong-Shading auf der GPU

- Um Phong-Shading zu ermöglichen bedarf es der OpenGL SL.
- Vertex- und Fragment-Programme (Shader) ersetzen Teile der "Fixed-Funktionality" innerhalb der OpenGL Rendering Pipeline.
- Mit Unterstützung des GPU-Programming Projektes wurde eine Abstraktion für Vertex- und Fragment-Programme geschaffen.
- Kapselung der nötigen Initialisierung und Kommunikation zu OpenGL SL in der Klasse GLSLProgram.
- Dadurch einfache Verwendung von Vertex- und Fragment-Shadern innerhalb unseres Frameworks möglich.

GLSLProgram

- Erzeugen eines GLSLPrograms:

```
GLSLProgram* myShader = new GLSLProgram("REQUIRED_EXTENSIONS",
                                        "relative_path_to_src/myShader.vs",
                                        "relative_path_to_src/myShader.fs", );

// parse, compile and link the Shaders
myShader->link();
```

- Aktivieren und Deaktivieren der Shader innerhalb des C++ Programmes:

```
myShader->start();
// we use our shader programs for now
send_something_down_the_pipeline();
myShader->stop();
// we use Fixed-Functionality of OpenGL again
send_something_down_the_pipeline();
```

OpenGL SL Programme für Phong-Shading

- Vertex Shader - arbeitet Fragment Shader zu
  - Ziel: im Fragment Shader Normale und Oberflächenpunkt im Augen-(Camera) Koordinatensystem für die Beleuchtungsberechnung zur Verfügung haben.

```
varying vec3 normal;
varying vec3 ecPosition3;

void main(void)
{
// builtin varying gl_Position must be written to
        gl_Position   = gl_ModelViewProjectionMatrix * gl_Vertex;
// normal in EC calculated from builtins
        normal     =  gl_NormalMatrix * gl_Normal;
// Vertex Position in HEC calculated from builtins
        vec4 ecPosition  =  gl_ModelViewMatrix * gl_Vertex;
// Vertex Position in EC
        ecPosition3 =  (ecPosition / ecPosition.w).xyz;
}
```

- Die varying Variablen normal und ecPosition3 werden durch den Rasterisierungsprozess über die Primitive interpoliert und an das Fragment Programm weitergereicht.

OpenGL SL Programme für Phong-Shading

- Fragment Shader - berechnet Beleuchtung (Half-Vector Methode)

```
varying vec3 normal;
varying vec3 ecPosition3;
void main (void)
{
        vec3 NNormal, NLightVec, NEyeVec, NHalfVec; vec4 MyColor;
        NNormal   = normalize(normal);
        NLightVec = normalize(vec3(gl_LightSource[0].position.xyz));
        NEyeVec   = normalize(-ecPosition3);
        NHalfVec  = normalize(NLightVec + NEyeVec);
// calculate Illumination of the Pixel with Half-Vector Method.
        MyColor  = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
        MyColor += gl_LightSource[0].diffuse *
             gl_FrontMaterial.diffuse * max( dot(NLightVec,NNormal), 0.0 );
        MyColor += gl_LightSource[0].specular *
             gl_FrontMaterial.specular *
     pow(max( dot(NNormal, NHalfVec), 0.0 ), gl_FrontMaterial.shininess);
        MyColor.a = 1.0;
        gl_FragColor = MyColor;
}
```

- Damit kann die Beleuchtung für ein Pixel im Fragment Shader nach einer eigens bestimmten Methode berechnet werden.

OpenGL C++ Programm für Phong-Shading

- Verwendung der Shader zusammen mit einem Model:

```
do_some_setup();
...
myShader->start();
myModel->draw();
myShader->stop();
...
```

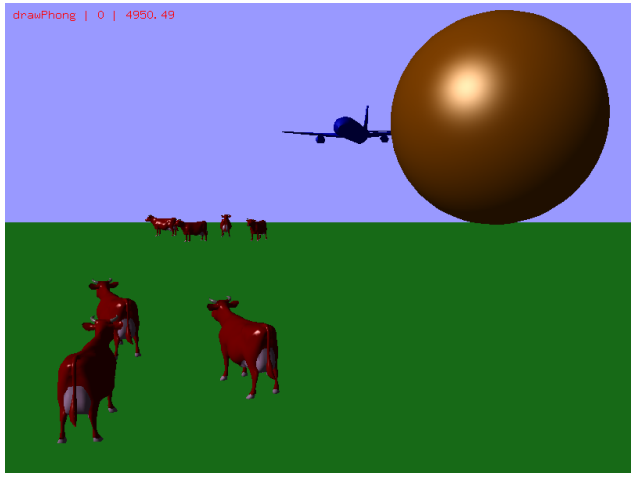


Abbildung: Phong-Shading

- Fazit:
  - Phong-Shading mit Klasse GLSLProgram + kleinen Shader Programmen einfach zu realisieren.
  - Beleuchtungsberechnung für die “Primärstrahlen” kann auf der GPU stattfinden!

Schatten auf der GPU

- Es besteht die Möglichkeit auch den “primären” Schatten auf der Graphikkarte zu generieren!
- unterschiedliche Techniken vorhanden:
  - e.g. Shadowmapping, Shadow-Volumes (Stencil-Shadows)
- Wir benutzen Shadowmapping weil:
  - Shadowmapping ist relativ einfach zu implementieren
  - Shadowmapping ist unabhängig von der Komplexität der Szene, da es im Bildraum arbeitet.
- Es gibt jedoch auch Nachteile:
  - perspectivisches und projektionsbedingtes Aliasing!
- Diverse Vorschläge um diese Technik weiter zu verbessern und zumindest das perspectivisch bedingte Aliasing zu reduzieren sind bekannt:
  - Perspective Shadow Maps (Stamminger et. al.).
  - Light Space Perspective Shadow Maps (Wimmer et. al.).

Idee des Shadowmappings

- "Shadow Mapping is a kind of Depth Test"
- Shadowmapping ist ein Two-Pass-Algorithmus:
- Schritt 1: Rendere die Szene zuerst aus Sicht der Lichtquelle
- Schritt 2: Rendere die Szene aus Sicht der Camera und tranformiere die (projectiven) Texture-Koordinaten jedes Punktes in den Lichtraum.
  Damit erhält man für jeden Punkt einen Tiefenwert Zp.
  Vergleiche diesen Tiefenwert Zp mit dem schon vorhandenen Wert in der Shadow Map Zs:
  - $Zp \leq Zs$: der Punkt liegt nicht im Schatten!
  - $Zp > Zs$: es befindet sich ein anderes Objekt zwischen dem Punkt und der Lichtquelle - Schatten!
- ein Bild sagt mehr als tausend Worte...

Funktionsweise des Shadowmappings

- Für einen Punkt P wird der (transformierte) Tiefenwert Zp mit dem Wert Zs in der Shadowmap verglichen:

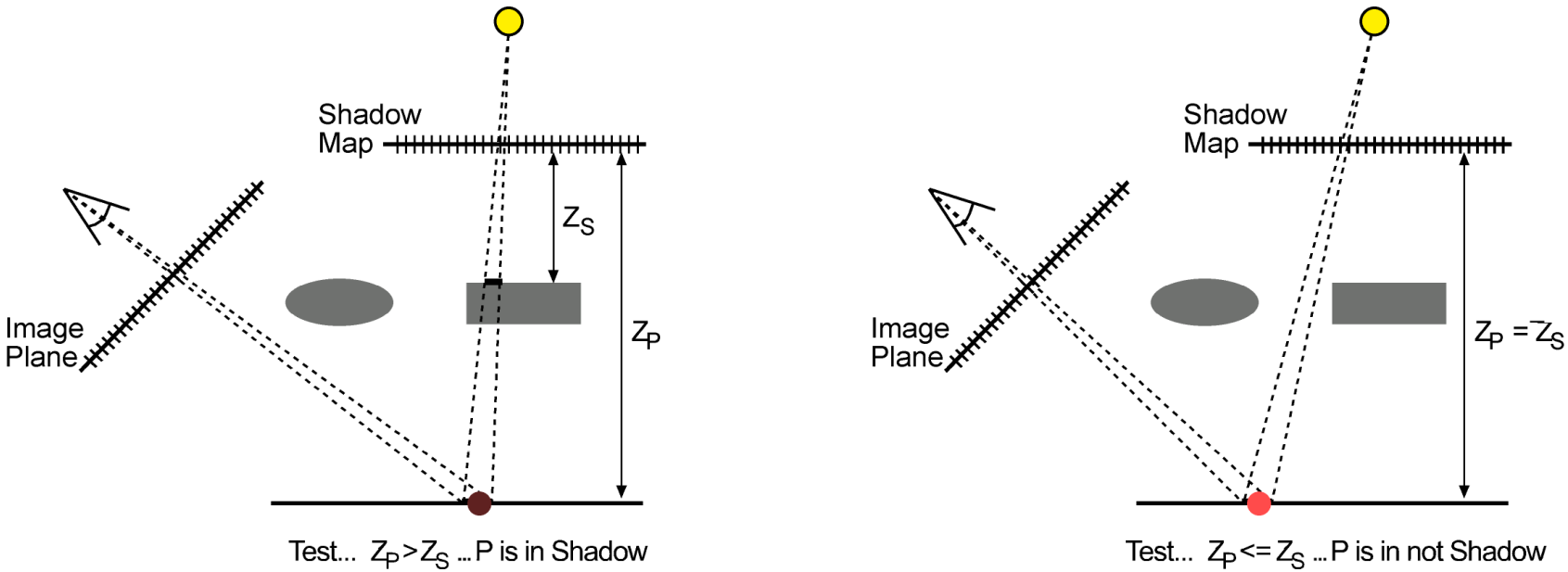


Abbildung: Shadowmapping als Tiefen Test

Visualisierung einer Shadowmap

- Eine Shadowmap ist nichts anderes als eine Tiefentextur.
- Je näher ein Objekt an der Lichtquelle ist, desto dunkler erscheint es in der Visualisierung - der Grauwert entspricht der Entfernung:

Abbildung: Uniform Shadowmap

Klasse Shadowmap

- Unser Framework stellt Uniform- und Light Space Perspective Shadowmapping zur Verfügung.
- Kapselung der notwendigen Berechnungen sowie setzen entsprechender OpenGL States für das Shadowmapping in eine Klasse Shadowmap.
- Hierzu viel Code aus einem Beispiel von Wimmer, TU-Wien, übernommen...
- Erstellen einer Shadowmap (für eine Lichtquelle):

```
myShadowmap = new Shadowmap(width,height);
```

Shadowmapping im Framework - 2 Schritte

- Schritt 1: Erzeugen der Shadowmap:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
mySM->beginShadowmapGeneration(cameraPos,
                               viewDir,
                               lightDir,
        myModel->getAABoundingBox());

{
  myModel->drawRAW(); // only Vertices are needed
}
mySM->endShadowmapGeneration();
```

- Schritt 2: Rendern der Szene mit aktiviertem Shadowmapping:

```
glClear(GL_DEPTH_BUFFER_BIT);
setupCamera();
mySM->beginShadowmapping();
{
  myModel->draw();
}
mySM->endShadowmapping();
```

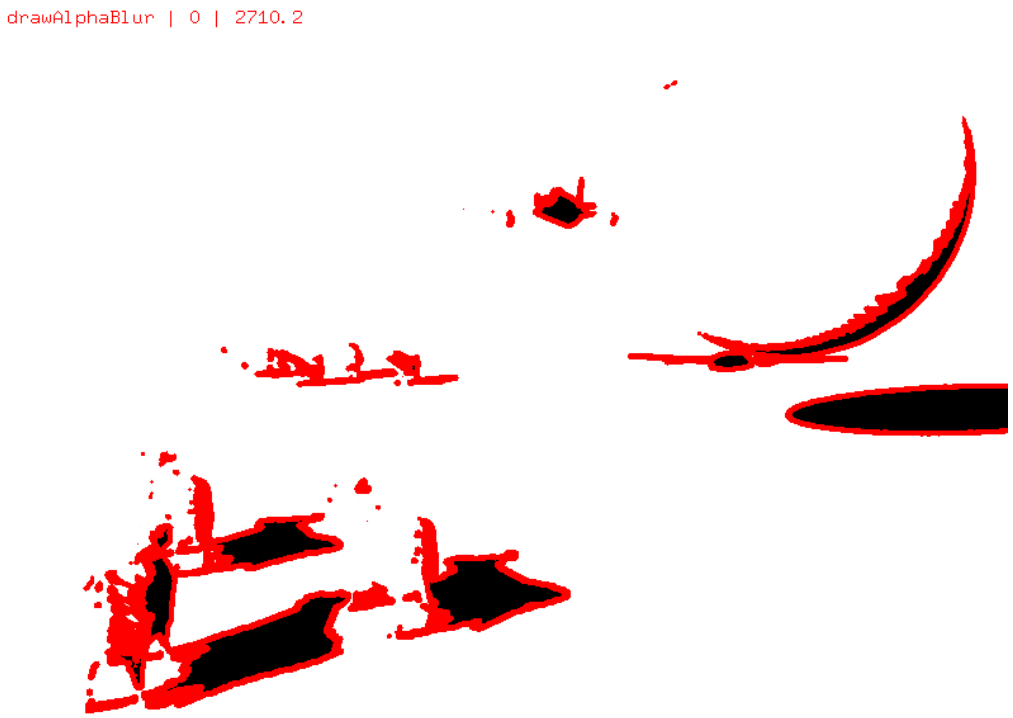
Shadowmapping + Phong Shading

Abbildung: Shadow Mapping + Phong Shading

- Fazit:
  - Shadowmapping in Kombination mit Phong-Shading liefert ein gutes Bild für die Primärstrahlen...

Verbesserung

- Problem:
  - Der Schatten ist an den Rändern nicht ganz perfekt und es treten teilweise Artefakte auf.
  - Idee: “markiere” die Schattenränder und Artefakte mit einem “Blur-Shader” (im Bildraum) und entscheide (später) an diesen Stellen mit einem richtigen Schattenstrahl “ob da wirklich Schatten ist”!



Primärstrahlen auf der GPU - ID-Shading

- Die Primärstrahlen können auf der GPU mit der Szene geschnitten werden - Dem eigentlichen Raytracing wird eine Menge Arbeit abgenommen.
- Wie ist das möglich?
  - kodiere die eindeutige Nummer jedes Dreiecks als Farbwert:

```
unsigned int triID = 1;
glBegin (GL_TRIANGLES);
for(every triangle of in the scene){
  GLubyte id[4];  unsigned int tmpID = 0;
  tmpID = triID;
  memcpy(&id,&tmpID,3);
  glColor4ubv(id);
  glVertex3f(Point A of this Triangle);
  glVertex3f(Point B of this Triangle);
  glVertex3f(Point C of this Triangle);
  ++triID;
}
glEnd ();
```

  - ...und rendere dann diesen Geometrie-Datensatz mit OpenGL:

```
myModel->drawID();
```

ID-Shading auf der GPU

- Ein interessantes Ergebnis:
  - jedes Pixel enthält nun versteckt im RGB-Kanal eine Nummer als Verweis auf das geschnittene Dreieck...
  - ...oder es bleibt schwarz - und somit ist kein Dreieck vom Primärstrahl “getroffen” worden.

Abbildung: ID-Shading

Fazit: Aufgaben der GPU

- Schatten - generiere die Shadowmap und führe Shadowmapping durch.
- ID-Shading - schneide die Primärstrahlen gegen die Szene.
- Blur-Alpha - “markiere” die Schattenränder und Schatten-Artefakte.
- “Verpackung” - die Dreiecks-IDs sollen im RGB Kanal, die Information über Schatten im Alpha Kanal kodiert sein.
- Phong-Shading - berechne die Beleuchtung für die Primärstrahlen unter Berücksichtigung des Schattens.
- Kombination - addiere die Anteile der Reflektion auf das Bild.
- ...zeige das fertige Bild an:)

Resultierende Renderpasses

- Aufgaben als Renderpasses dargestellt - manches kann in einem Pass geschehen:

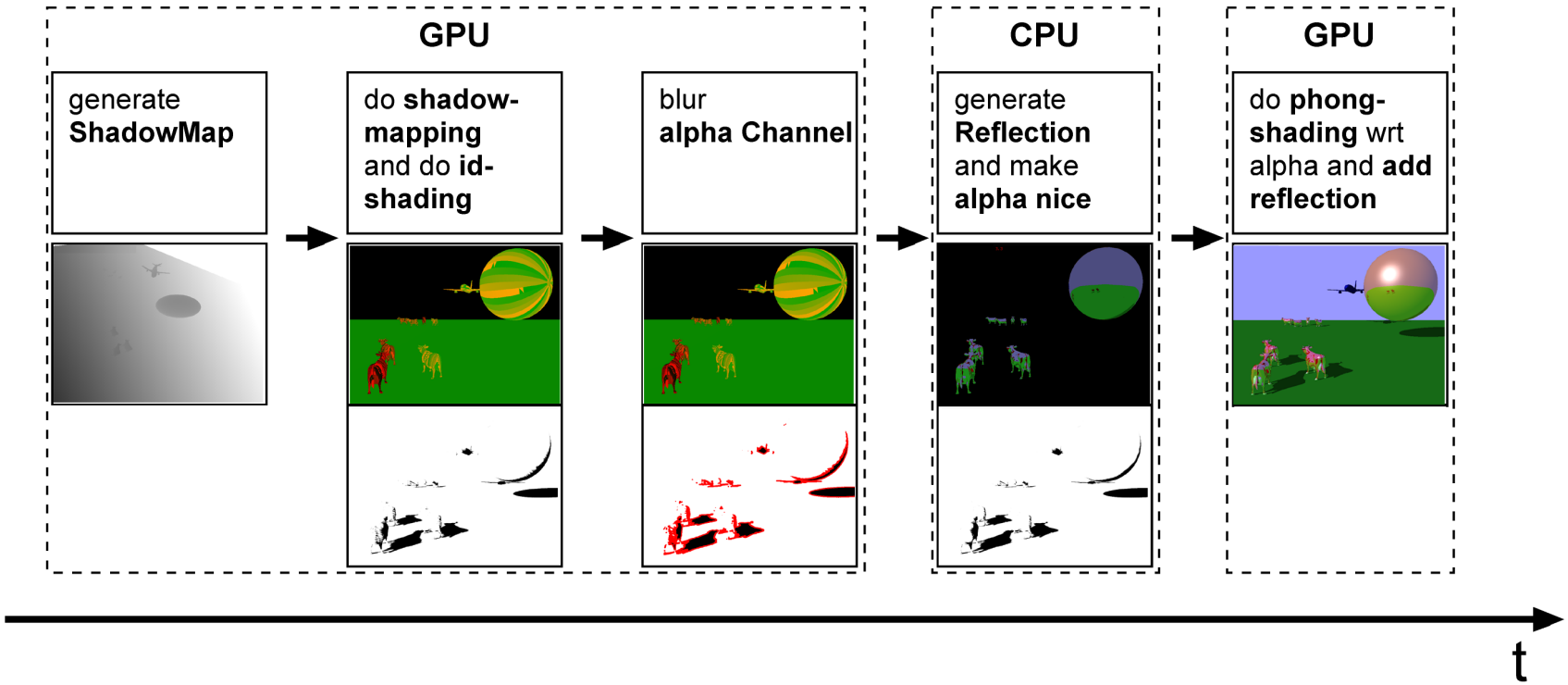


Abbildung: Renderpasses der GPU

Verschränkung des Rendering Ablaufes

- Die GPU und CPU sind zwei unabhängige Recheneinheiten.
- Die Aufgaben lassen sich zeitlich und "räumlich" trennen.
- Der Ablauf kann somit verschränkt werden:

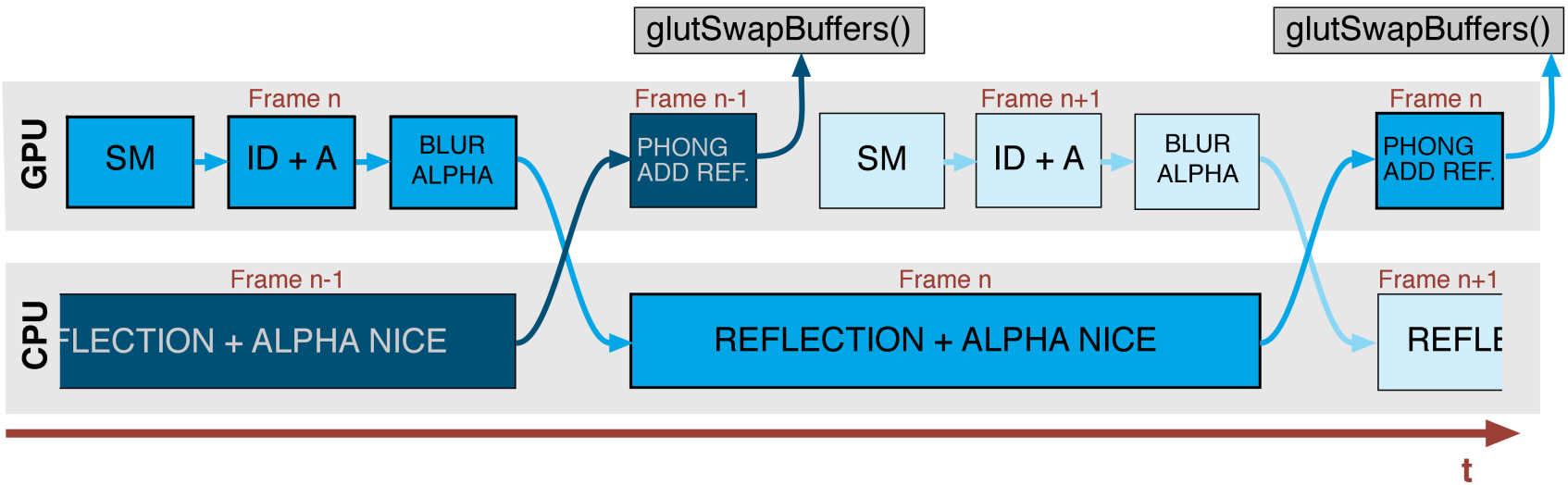


Abbildung: Interleaving

Was die CPU machen muss:

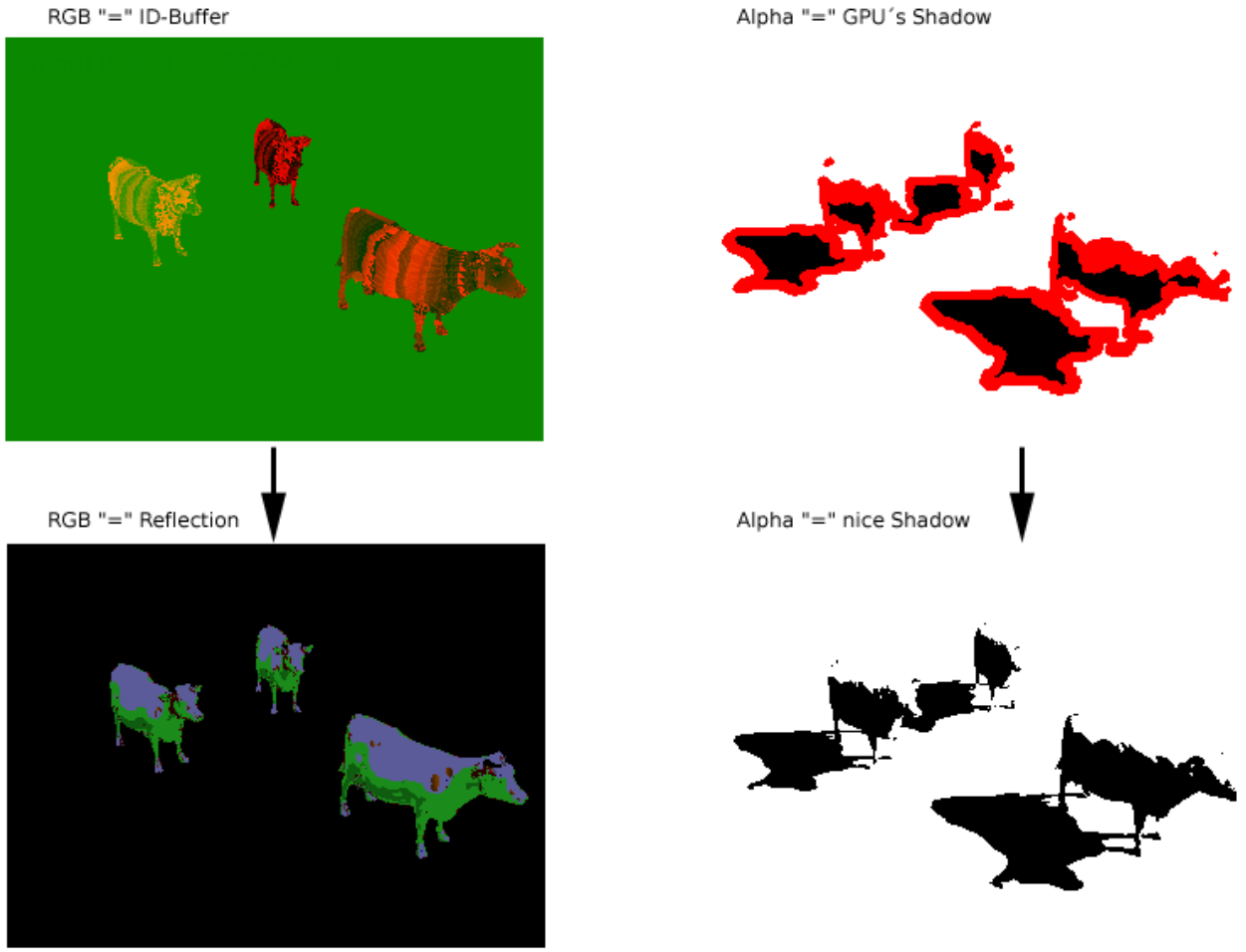


Abbildung: CPU Input / Output

Anforderungen

- Erzeugung von Reflektionsstrahlen erster Generation
- Eliminierung bzw. Reduzierung von Schattenartefakten, die mittels Shadowmappings auf GPU erzeugt wurden
- Wiederverwendung bestehender Konzepte, Datenstrukturen und Algorithmen des zur Zwischenpräsentation bestehenden Pinocchio-Systems
- Wie wurden Zielsetzungen erreicht?

Datenstrukturen zur Kommunikation

- CPU und GPU müssen Zustände und Ergebnisse kommunizieren können
- Entwicklung dreier Datenstrukturen
  - Framebuffer
  - Kamera
  - Lichtquelle
- Um Verschränkung handhabbar zu machen, wurde Double-Buffering Konzept übernommen ⇒ Datenstrukturen verfügen über Back- und Front-State

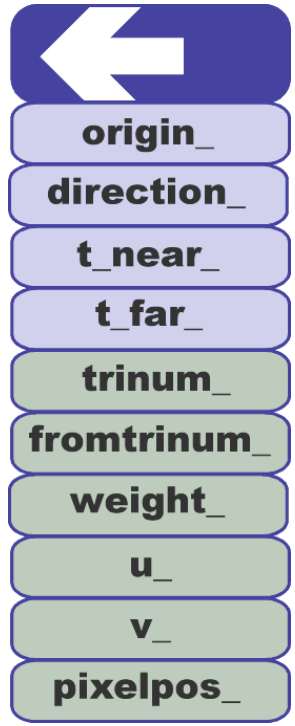
```
myDB->swap();

DoubleBuffer::value_type* temp;
temp = this->front_;
this->front_ = this->back_;
this->back_ = temp;
```

Ray.h

```
#ifndef RAY_H
#define RAY_H
...
Ray{
  public:
  ...
  // 32 byte
  Point3D origin_;
  Vector3D direction_;

  // 16 byte
  float32_pco_t t_near_;
  float32_pco_t t_far_;
  uint32_pco_t trinum_;
  uint32_pco_t fromtrinum_;

  // 16 byte
  float32_pco_t weight_;
  float32_pco_t u_;
  float32_pco_t v_;
  Pixel* pixelpos_;
}
...
#endif // #ifndef RAY_H
```

RenderClient

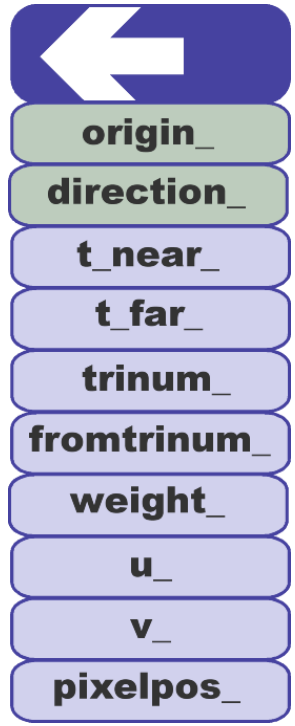
```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();

    if (!triID) continue;
    weight = scene_->getRefl(triID);

    if(weight != 0.0){
      secondaryRaysEnd_->trinum_     = triID;
      secondaryRaysEnd_->fromtrinum_ = triID;
      secondaryRaysEnd_->weight_     = weight;
      secondaryRaysEnd_->u_          = x;
      secondaryRaysEnd_->v_          = y;
      secondaryRaysEnd_->pixelpos_   = pixel;
      ++secondaryRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
```




```
camShader_  ->shade              (secondaryRays_, secondaryRaysEnd_);
intersector_->intersectID        (secondaryRays_, secondaryRaysEnd_);
metaShader_ ->createNewGeneration(secondaryRays_, secondaryRaysEnd_);
intersector_->intersect          (secondaryRays_, secondaryRaysEnd_,
                                  hitsEnd_, missesEnd_);
metaShader_ ->shade              (secondaryRays_, secondaryRaysEnd_,
                                  hits_, hitsEnd_,
                                  misses_, missesEnd_);

secondaryRaysEnd_ = secondaryRays_->begin();
```

CamShader

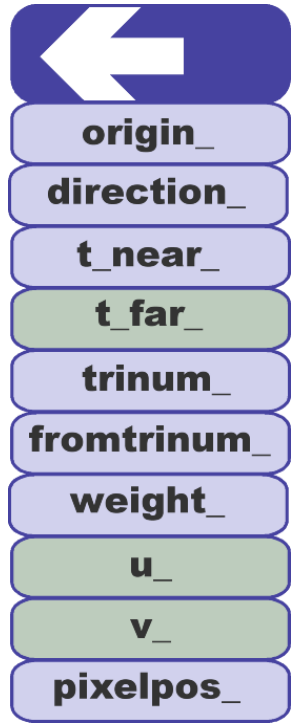
```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();

    if (!triID) continue;
    weight = scene_->getRefl(triID);

    if(weight != 0.0){
      secondaryRaysEnd_->trinum_     = triID;
      secondaryRaysEnd_->fromtrinum_ = triID;
      secondaryRaysEnd_->weight_     = weight;
      secondaryRaysEnd_->u_          = x;
      secondaryRaysEnd_->v_          = y;
      secondaryRaysEnd_->pixelpos_   = pixel;
      ++secondaryRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
```




```
camShader_  ->shade              (secondaryRays_, secondaryRaysEnd_);
intersector_->intersectID        (secondaryRays_, secondaryRaysEnd_);
metaShader_ ->createNewGeneration(secondaryRays_, secondaryRaysEnd_);
intersector_->intersect          (secondaryRays_, secondaryRaysEnd_,
                                  hitsEnd_, missesEnd_);
metaShader_ ->shade              (secondaryRays_, secondaryRaysEnd_,
                                  hits_, hitsEnd_,
                                  misses_, missesEnd_);

secondaryRaysEnd_ = secondaryRays_->begin();
```

KdTreeIntersector

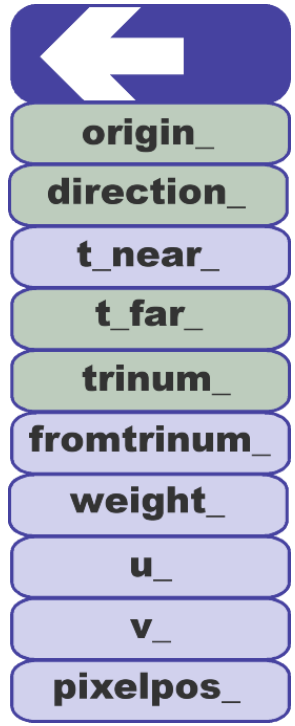
```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();

    if (!triID) continue;
    weight = scene_->getRefl(triID);

    if(weight != 0.0){
      secondaryRaysEnd_->trinum_     = triID;
      secondaryRaysEnd_->fromtrinum_ = triID;
      secondaryRaysEnd_->weight_     = weight;
      secondaryRaysEnd_->u_          = x;
      secondaryRaysEnd_->v_          = y;
      secondaryRaysEnd_->pixelpos_   = pixel;
      ++secondaryRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
```




```
camShader_  ->shade              (secondaryRays_, secondaryRaysEnd_);
intersector_->intersectID        (secondaryRays_, secondaryRaysEnd_);
metaShader_ ->createNewGeneration(secondaryRays_, secondaryRaysEnd_);
intersector_->intersect          (secondaryRays_, secondaryRaysEnd_,
                                  hitsEnd_, missesEnd_);
metaShader_ ->shade              (secondaryRays_, secondaryRaysEnd_,
                                  hits_, hitsEnd_,
                                  misses_, missesEnd_);

secondaryRaysEnd_ = secondaryRays_->begin();
```

MetaShader

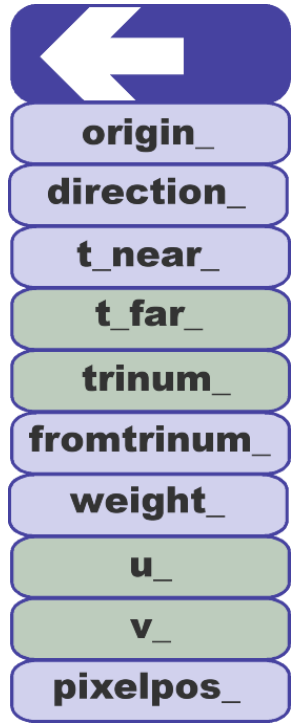
```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();

    if (!triID) continue;
    weight = scene_->getRefl(triID);

    if(weight != 0.0){
      secondaryRaysEnd_->trinum_     = triID;
      secondaryRaysEnd_->fromtrinum_ = triID;
      secondaryRaysEnd_->weight_     = weight;
      secondaryRaysEnd_->u_          = x;
      secondaryRaysEnd_->v_          = y;
      secondaryRaysEnd_->pixelpos_   = pixel;
      ++secondaryRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
```




```
camShader_  ->shade              (secondaryRays_, secondaryRaysEnd_);
intersector_->intersectID        (secondaryRays_, secondaryRaysEnd_);
metaShader_ ->createNewGeneration(secondaryRays_, secondaryRaysEnd_);
intersector_->intersect          (secondaryRays_, secondaryRaysEnd_,
                                  hitsEnd_, missesEnd_);
metaShader_ ->shade              (secondaryRays_, secondaryRaysEnd_,
                                  hits_, hitsEnd_,
                                  misses_, missesEnd_);

secondaryRaysEnd_ = secondaryRays_->begin();
```

KdTreeIntersector

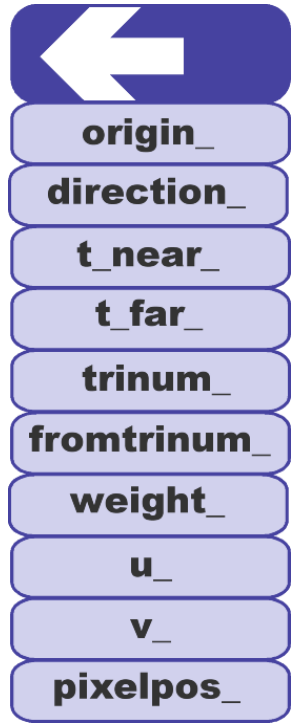
```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();

    if (!triID) continue;
    weight = scene_->getRefl(triID);

    if(weight != 0.0){
      secondaryRaysEnd_->trinum_     = triID;
      secondaryRaysEnd_->fromtrinum_ = triID;
      secondaryRaysEnd_->weight_     = weight;
      secondaryRaysEnd_->u_          = x;
      secondaryRaysEnd_->v_          = y;
      secondaryRaysEnd_->pixelpos_   = pixel;
      ++secondaryRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
```




```
camShader_  ->shade              (secondaryRays_, secondaryRaysEnd_);
intersector_->intersectID        (secondaryRays_, secondaryRaysEnd_);
metaShader_ ->createNewGeneration(secondaryRays_, secondaryRaysEnd_);
intersector_->intersect          (secondaryRays_, secondaryRaysEnd_,
                                  hitsEnd_, missesEnd_);
metaShader_ ->shade              (secondaryRays_, secondaryRaysEnd_,
                                  hits_, hitsEnd_,
                                  misses_, missesEnd_);

secondaryRaysEnd_ = secondaryRays_->begin();
```

MetaShader

```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();

    if (!triID) continue;
    weight = scene_->getRefl(triID);

    if(weight != 0.0){
      secondaryRaysEnd_->trinum_     = triID;
      secondaryRaysEnd_->fromtrinum_ = triID;
      secondaryRaysEnd_->weight_     = weight;
      secondaryRaysEnd_->u_          = x;
      secondaryRaysEnd_->v_          = y;
      secondaryRaysEnd_->pixelpos_   = pixel;
      ++secondaryRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
```




```
camShader_  ->shade              (secondaryRays_, secondaryRaysEnd_);
intersector_->intersectID        (secondaryRays_, secondaryRaysEnd_);
metaShader_ ->createNewGeneration(secondaryRays_, secondaryRaysEnd_);
intersector_->intersect          (secondaryRays_, secondaryRaysEnd_,
                                  hitsEnd_, missesEnd_);
metaShader_ ->shade              (secondaryRays_, secondaryRaysEnd_,
                                  hits_, hitsEnd_,
                                  misses_, missesEnd_);

secondaryRaysEnd_ = secondaryRays_->begin();
```

Resultat

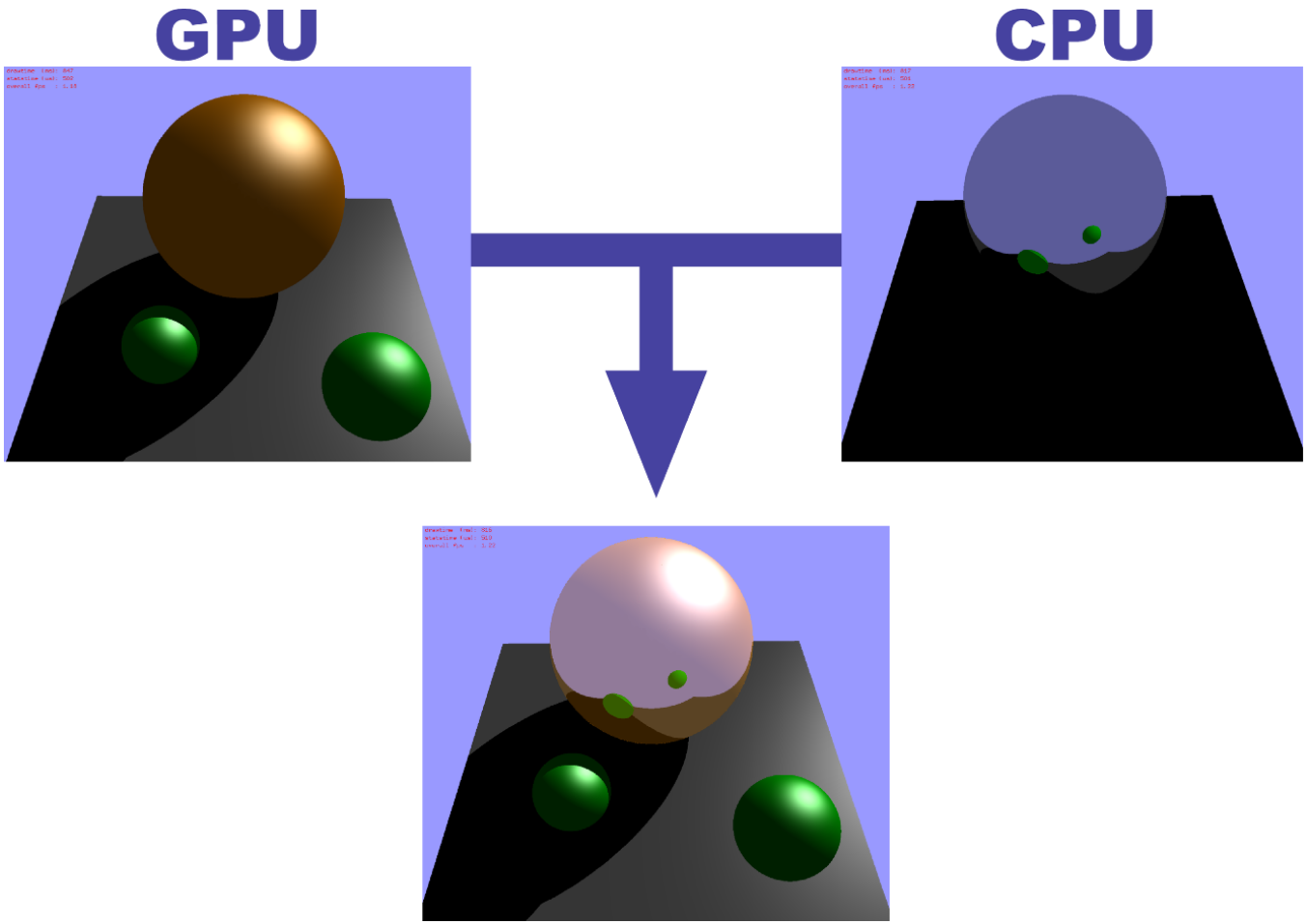
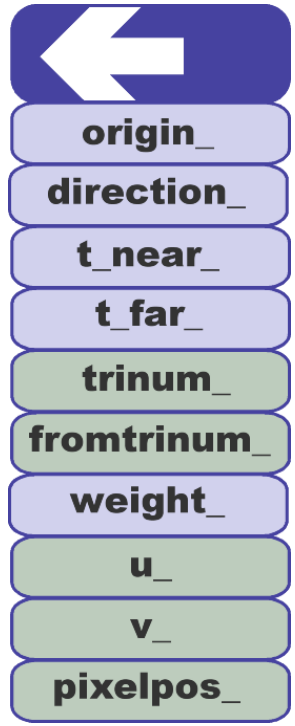


Abbildung: GPU-CPU Arbeitsteilung

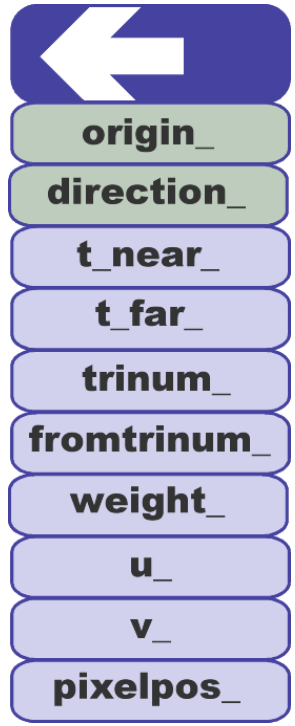
RenderClient

```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();
    shadow = (*pixel)[3];

    if (!triID) continue;

    if((shadow != 0) && (shadow != 255)){
      shadowRaysEnd_->trinum_     = triID;
      shadowRaysEnd_->fromtrinum_ = triID;
      shadowRaysEnd_->u_          = x;
      shadowRaysEnd_->v_          = y;
      shadowRaysEnd_->pixelpos_   = pixel;
      ++shadowRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
camShader_  ->shade             (shadowRays_, shadowRaysEnd_);
intersector_->intersectID       (shadowRays_, shadowRaysEnd_);
metaShader_ ->createShadow      (shadowRays_, shadowRaysEnd_);
intersector_->intersectShadow   (shadowRays_, shadowRaysEnd_);

RayIt shadowIt = shadowRays_->begin();
for(; shadowIt != shadowRaysEnd_; ++shadowIt){
  (shadowIt->trinum_ == 0) ?
     shadowIt->pixelpos_->setA(255) :
     shadowIt->pixelpos_->setA(0);
}
shadowRaysEnd_ = shadowRays_->begin();
```



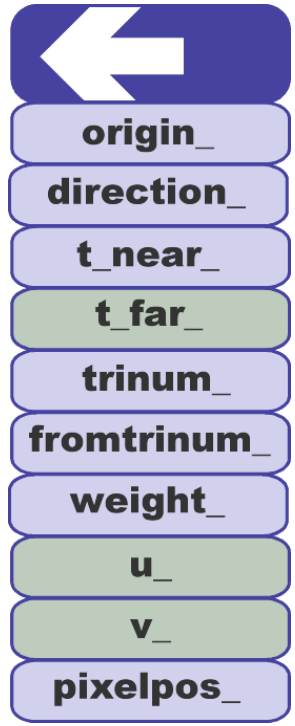
CamShader

```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();
    shadow = (*pixel)[3];

    if (!triID) continue;

    if((shadow != 0) && (shadow != 255)){
      shadowRaysEnd_->trinum_     = triID;
      shadowRaysEnd_->fromtrinum_ = triID;
      shadowRaysEnd_->u_          = x;
      shadowRaysEnd_->v_          = y;
      shadowRaysEnd_->pixelpos_   = pixel;
      ++shadowRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
camShader_  ->shade             (shadowRays_, shadowRaysEnd_);
intersector_->intersectID       (shadowRays_, shadowRaysEnd_);
metaShader_ ->createShadow      (shadowRays_, shadowRaysEnd_);
intersector_->intersectShadow   (shadowRays_, shadowRaysEnd_);

RayIt shadowIt = shadowRays_->begin();
for(; shadowIt != shadowRaysEnd_; ++shadowIt){
  (shadowIt->trinum_ == 0) ?
     shadowIt->pixelpos_->setA(255) :
     shadowIt->pixelpos_->setA(0);
}
shadowRaysEnd_ = shadowRays_->begin();
```



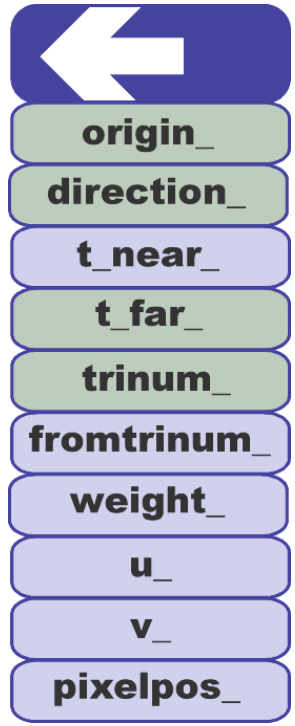
KdTreeIntersector

```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();
    shadow = (*pixel)[3];

    if (!triID) continue;

    if((shadow != 0) && (shadow != 255)){
      shadowRaysEnd_->trinum_     = triID;
      shadowRaysEnd_->fromtrinum_ = triID;
      shadowRaysEnd_->u_          = x;
      shadowRaysEnd_->v_          = y;
      shadowRaysEnd_->pixelpos_   = pixel;
      ++shadowRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
camShader_  ->shade             (shadowRays_, shadowRaysEnd_);
intersector_->intersectID       (shadowRays_, shadowRaysEnd_);
metaShader_ ->createShadow      (shadowRays_, shadowRaysEnd_);
intersector_->intersectShadow   (shadowRays_, shadowRaysEnd_);

RayIt shadowIt = shadowRays_->begin();
for(; shadowIt != shadowRaysEnd_; ++shadowIt){
  (shadowIt->trinum_ == 0) ?
     shadowIt->pixelpos_->setA(255) :
     shadowIt->pixelpos_->setA(0);
}
shadowRaysEnd_ = shadowRays_->begin();
```



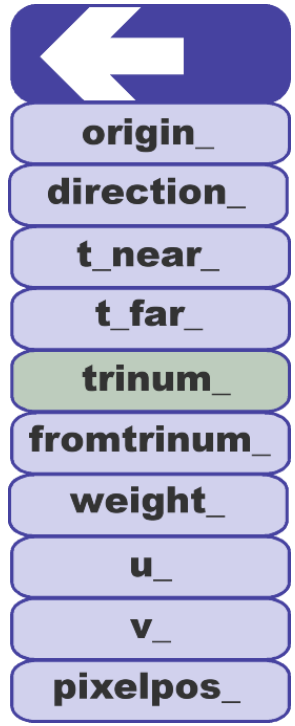
MetaShader

```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();
    shadow = (*pixel)[3];

    if (!triID) continue;

    if((shadow != 0) && (shadow != 255)){
      shadowRaysEnd_->trinum_     = triID;
      shadowRaysEnd_->fromtrinum_ = triID;
      shadowRaysEnd_->u_          = x;
      shadowRaysEnd_->v_          = y;
      shadowRaysEnd_->pixelpos_   = pixel;
      ++shadowRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
camShader_  ->shade             (shadowRays_, shadowRaysEnd_);
intersector_->intersectID       (shadowRays_, shadowRaysEnd_);
metaShader_ ->createShadow      (shadowRays_, shadowRaysEnd_);
intersector_->intersectShadow   (shadowRays_, shadowRaysEnd_);

RayIt shadowIt = shadowRays_->begin();
for(; shadowIt != shadowRaysEnd_; ++shadowIt){
  (shadowIt->trinum_ == 0) ?
     shadowIt->pixelpos_->setA(255) :
     shadowIt->pixelpos_->setA(0);
}
shadowRaysEnd_ = shadowRays_->begin();
```



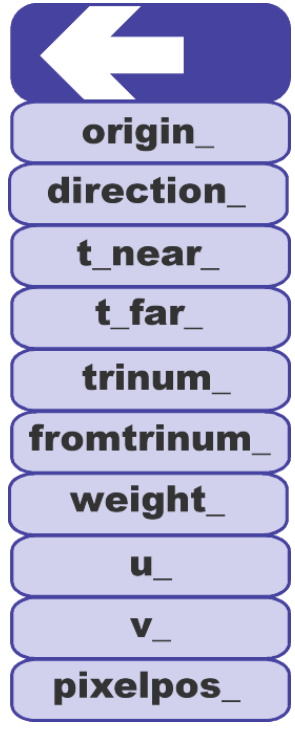
KdTreeIntersector

```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();
    shadow = (*pixel)[3];

    if (!triID) continue;

    if((shadow != 0) && (shadow != 255)){
      shadowRaysEnd_->trinum_     = triID;
      shadowRaysEnd_->fromtrinum_ = triID;
      shadowRaysEnd_->u_          = x;
      shadowRaysEnd_->v_          = y;
      shadowRaysEnd_->pixelpos_   = pixel;
      ++shadowRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
camShader_  ->shade             (shadowRays_, shadowRaysEnd_);
intersector_->intersectID       (shadowRays_, shadowRaysEnd_);
metaShader_ ->createShadow      (shadowRays_, shadowRaysEnd_);
intersector_->intersectShadow   (shadowRays_, shadowRaysEnd_);

RayIt shadowIt = shadowRays_->begin();
for(; shadowIt != shadowRaysEnd_; ++shadowIt){
  (shadowIt->trinum_ == 0) ?
     shadowIt->pixelpos_->setA(255) :
     shadowIt->pixelpos_->setA(0);
}
shadowRaysEnd_ = shadowRays_->begin();
```



RenderClient

```
for(unsigned int y = clientnum_; y < sizeY; y += numclients_){
  for(unsigned int x = 0; x != sizeX; ++x){
    pixel  = &(buffer_->getFront()(x, y));
    triID  = pixel->getTriID();
    shadow = (*pixel)[3];

    if (!triID) continue;

    if((shadow != 0) && (shadow != 255)){
      shadowRaysEnd_->trinum_     = triID;
      shadowRaysEnd_->fromtrinum_ = triID;
      shadowRaysEnd_->u_          = x;
      shadowRaysEnd_->v_          = y;
      shadowRaysEnd_->pixelpos_   = pixel;
      ++shadowRaysEnd_;
    }
    memset(&(pixel[0]), 0, 3);
  }
}
camShader_  ->shade             (shadowRays_, shadowRaysEnd_);
intersector_->intersectID       (shadowRays_, shadowRaysEnd_);
metaShader_ ->createShadow      (shadowRays_, shadowRaysEnd_);
intersector_->intersectShadow   (shadowRays_, shadowRaysEnd_);

RayIt shadowIt = shadowRays_->begin();
for(; shadowIt != shadowRaysEnd_; ++shadowIt){
  (shadowIt->trinum_ == 0) ?
     shadowIt->pixelpos_->setA(255) :
     shadowIt->pixelpos_->setA(0);
}
shadowRaysEnd_ = shadowRays_->begin();
```



Resultat

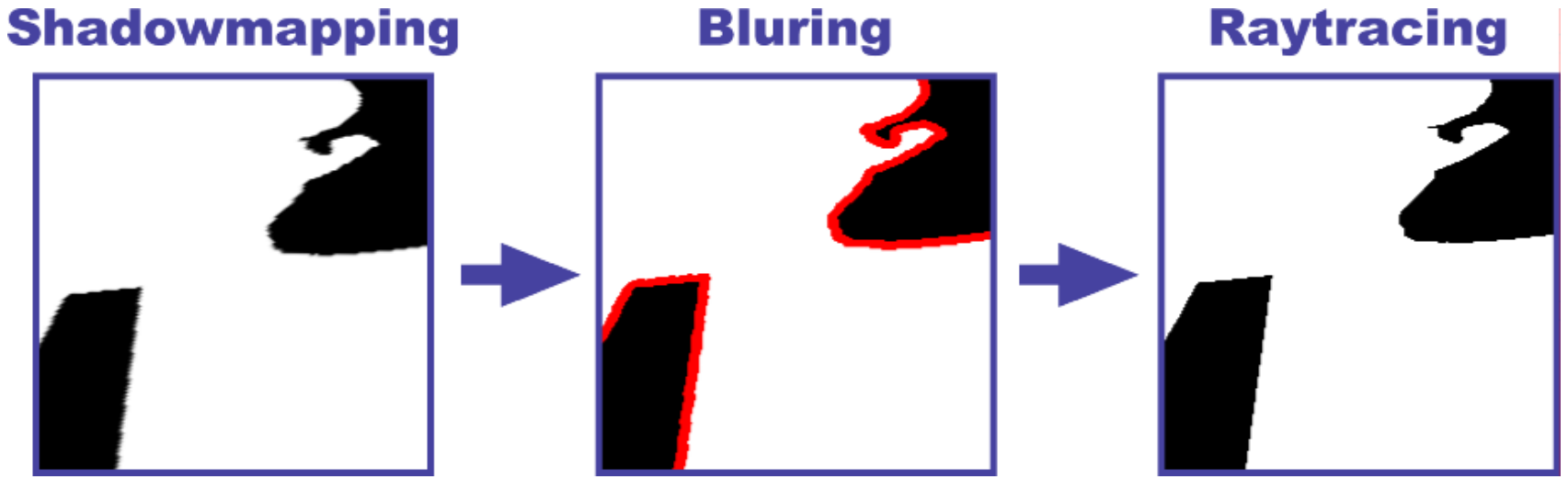


Abbildung: Stufen der Schattengenerierung

CamShader

- GPU liefert zwar Informationen welche Dreiecke geschnitten werden, aber Ursprung und Richtung der Strahlen sind unbekannt
- Schwierigkeit: Erzeugung identischer Primärstrahlen auf GPU und CPU

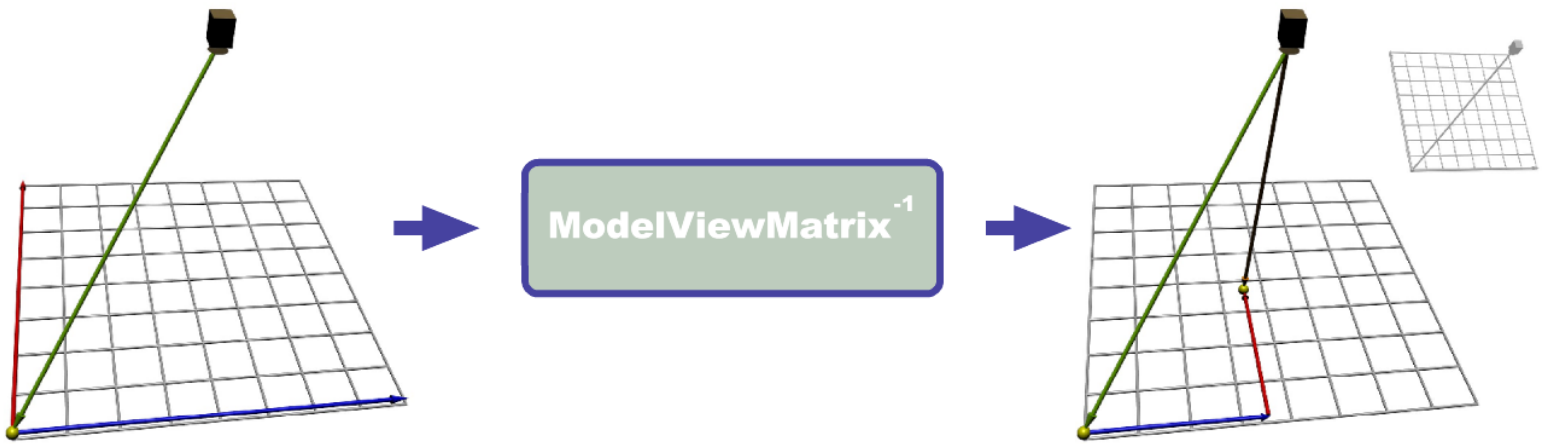


Abbildung: Erzeugung der Primärstrahlen

Resultat

- Verfahren ist nicht in der Lage identische Kopien der GPU-Primärstrahlen zu erzeugen

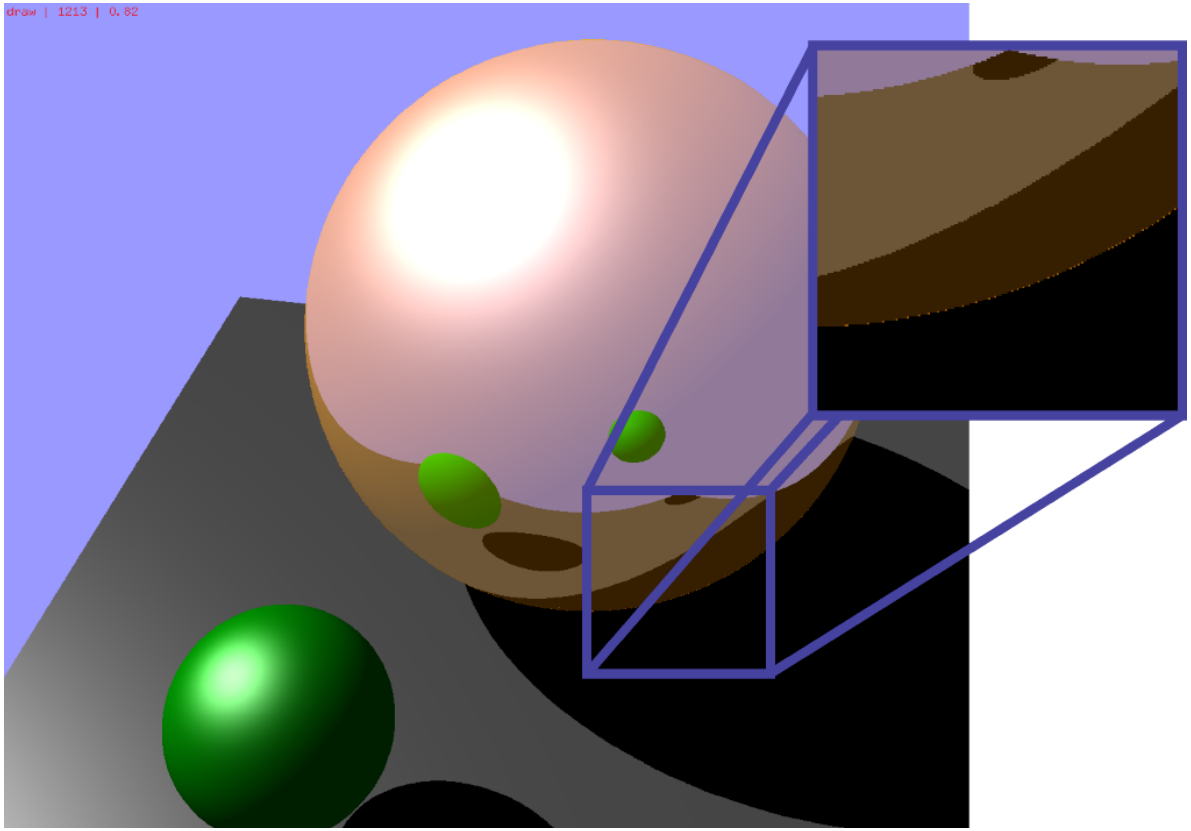


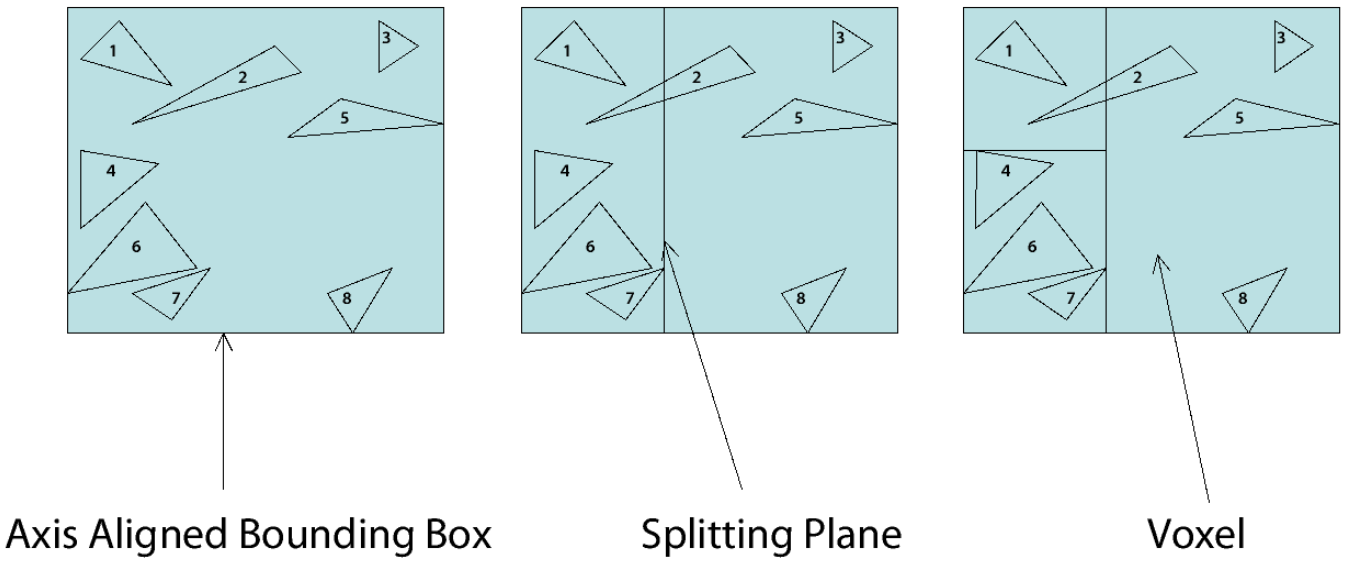
Abbildung: Artefakte an Objekträndern

Alternatives Verfahren

- Wieso iteriert jede Stufe aufs neue über Strahlenvektor?
- Idee: Cache-kohärente Ausführung von Instruktionen
- Ein Seiteneffekt ist die Möglichkeit beliebig viele Strahlgenerationen erzeugen zu können
- Alternatives Raytracing-Verfahren wurde implementiert
  - Arbeitet Strahl für Strahl ab statt Vektoren von Strahlen
- Widersprüchliche Erfahrungen
  - Performancegewinn auf Einprozessorsystemen von $\leq$20%
  - Performanceverlust auf Mehrprozessorsystemen von $\leq$15%

Kd-Tree

- Reduktion der Strahl-Objekt Schnitte



Konstruktion

- Möglichkeiten zur Platzierung der Schnittebene
  - Unterteilung im Median des Voxels
  - Unterteilung im Objekt Median
  - mittels Kostenfunktion
- Abruchkriterien
  - maximale Baumtiefe (20-25)
  - Primitive pro Blatt (2-4)
  - mittels Kostenvergleich

Aufbau in zwei Schritten

- Nodes des zweiten Baumes sind Speicheroptimiert (8 byte)

Kostenfunktion

```
for each axis in X, Y, Z:
    NrAB = NrA = 0; NrB = number of objects;
    for each object in node:
        if object.max is inside the node:
            put it into sorted list;
        if object.min is inside the node:
            put it into sorted list;
        else:
            ++NrAB; --NrB;
     for each splitposition in list:
         if splitposition is a start location:
             evaluate cost function;
             ++NrAB; --NrB;
         else:
             --NrAB; ++NrA;
             evaluate cost function;
```

Abbruchkriterium

```
if (bestCost > pNode->GetObjNr())
    return false;
```

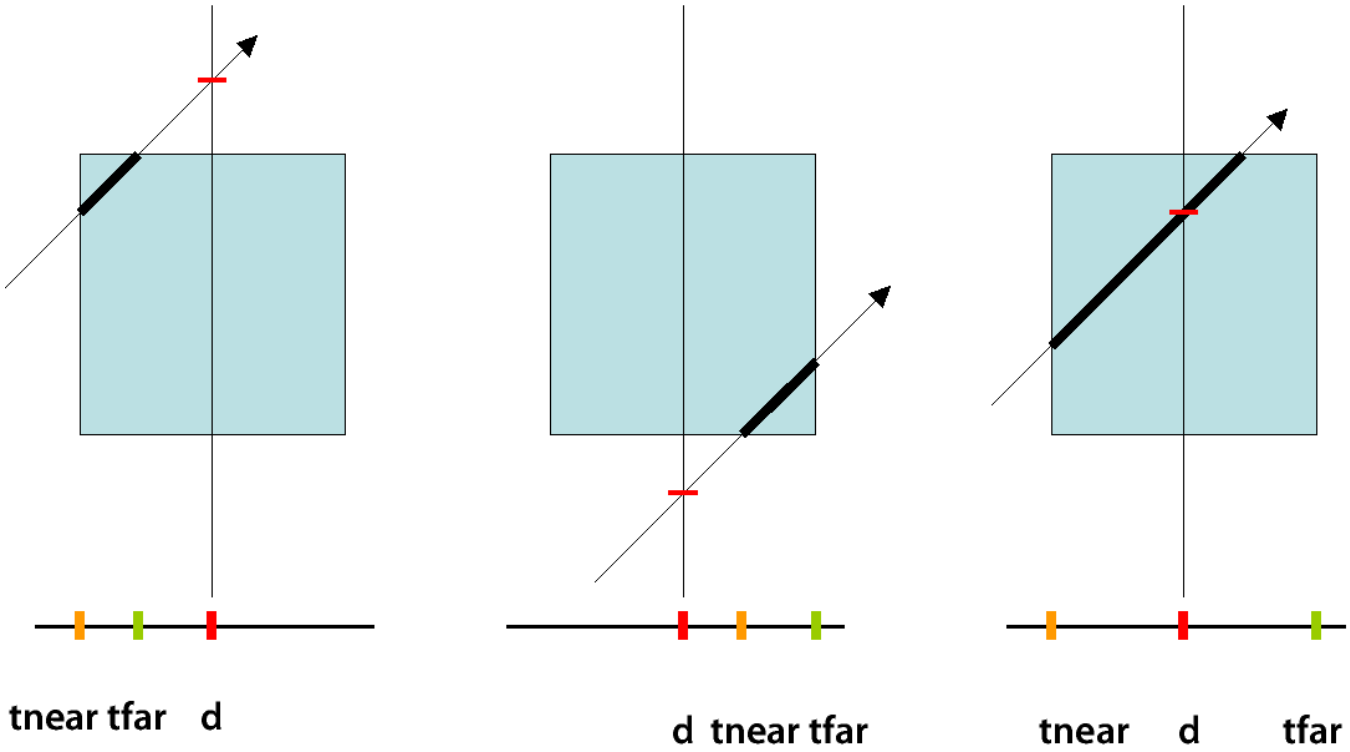
KdTreeIntersector

- Die Klasse KdTreeIntersector stellt die verwendeten Schnittroutinen zur Verfügung

```
void intersectID(RayVec* rays, const RayIt & raysend);
void intersectShadowID(RayVec* rays, const RayIt & raysend);

void intersect(RayVec * secondaryRays, const RayIt & secondaryRaysEnd,
 HitIt & hitsEnd, MissIt & missesEnd);
void intersectShadow(RayVec* shadowRays, const RayIt & shadowRaysEnd);
```

Traversierung

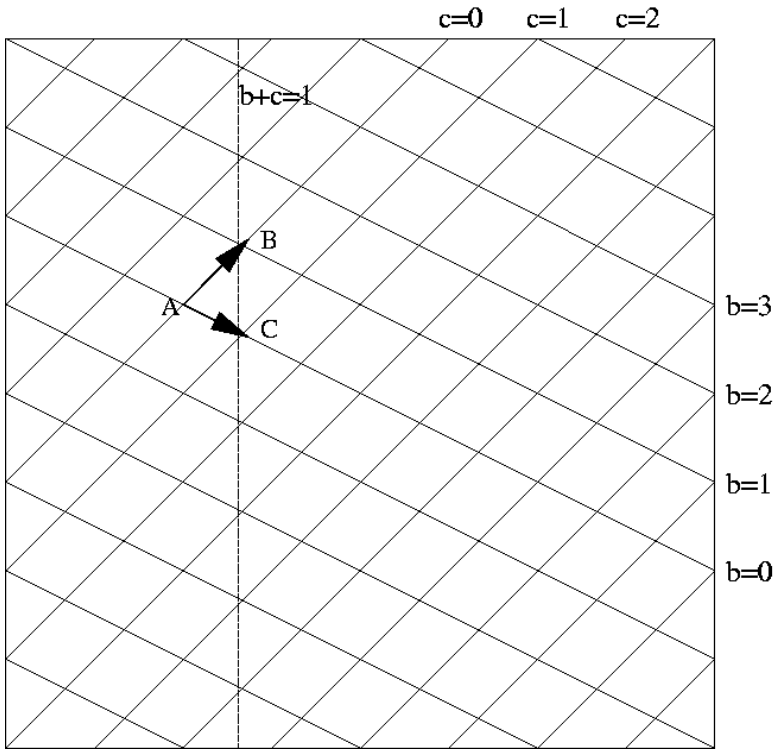
```
void intersect(Ray * ray)
{
    ...
    if ( ray.direction_[axis] > 0) { // left to right
```

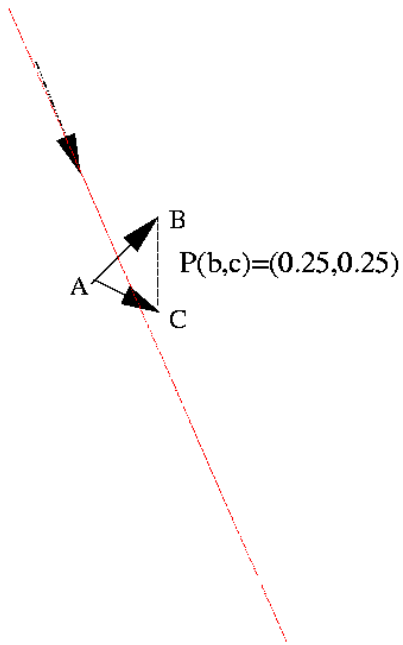



```
    } else { // right to left
    ...
}
```

IntersectTriangle

- Baryzentrische Koordinaten:
- $P(\alpha, \beta, \gamma) = \alpha A + \beta B + \gamma C$ mit $0 \leq \alpha, \beta, \gamma \leq 1$
- $P(\beta, \gamma) = A + \beta(B - A) + \gamma(C - A)$ mit $\alpha = 1 - \beta - \gamma$
- P im Dreieck dann und nur dann wenn: $0 \leq \beta, \gamma$ und $\beta + \gamma \leq 1$





IntersectTriangle

- $o + td = A + \beta(B - A) + \gamma(C - A)$
- Projektion von Schnittpunkt und Dreieck in die Ebene
- baryzentrische Koordinaten bleiben erhalten
- Vorrausberechnungen einiger Terme möglich zur effizienten Schnittberechnung

Features

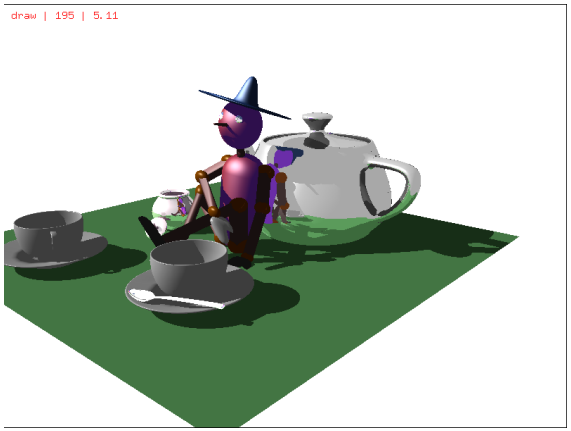


Abbildung: 82322 Faces, 5.11fps,800x600

Features

- Multithreading
- Nutzung von SSE
- Optimierter Beschleunigungsansatz
- Einzelne Stufen des Raytracers arbeiten auf Vektoren von Strahlen
- Blinn-Phong Beleuchtungsmodell auf der GPU als auch auf der CPU
- Primär Strahlen auf der GPU
- Shadowmapping mit Ausbesserung der Ränder
- Intel Xeon 3.20 GHz, GeForce Quadro 4400, 1 GPU-Thread, 4 RT-Threads

Ausblick

- Stereo
- Interaktion, dynamische Szenen
- Anbindung von VRPN
- und nun zur Demonstration

Bibliographie

- E. Haines. BSP Plane Cost Function Revisisted. http://www.acm.org/tog/resources/RTNews/html/rtnv17n1.html#art8, 2004.
- I. Wald. Realtime Ray Tracing and Interactive Global Illumination. Dissertation, 2004.
- Vlastimil Havran. Heuristic Ray Shooting Algorithms. Dissertation Thesis, 2000.