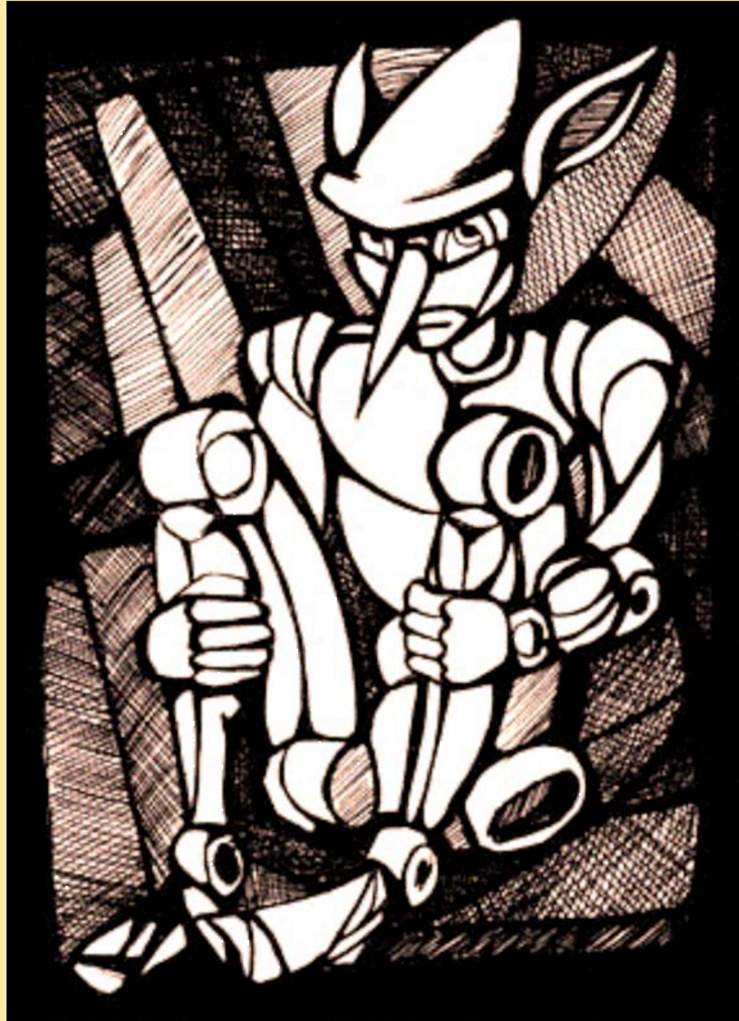
Pinocchio

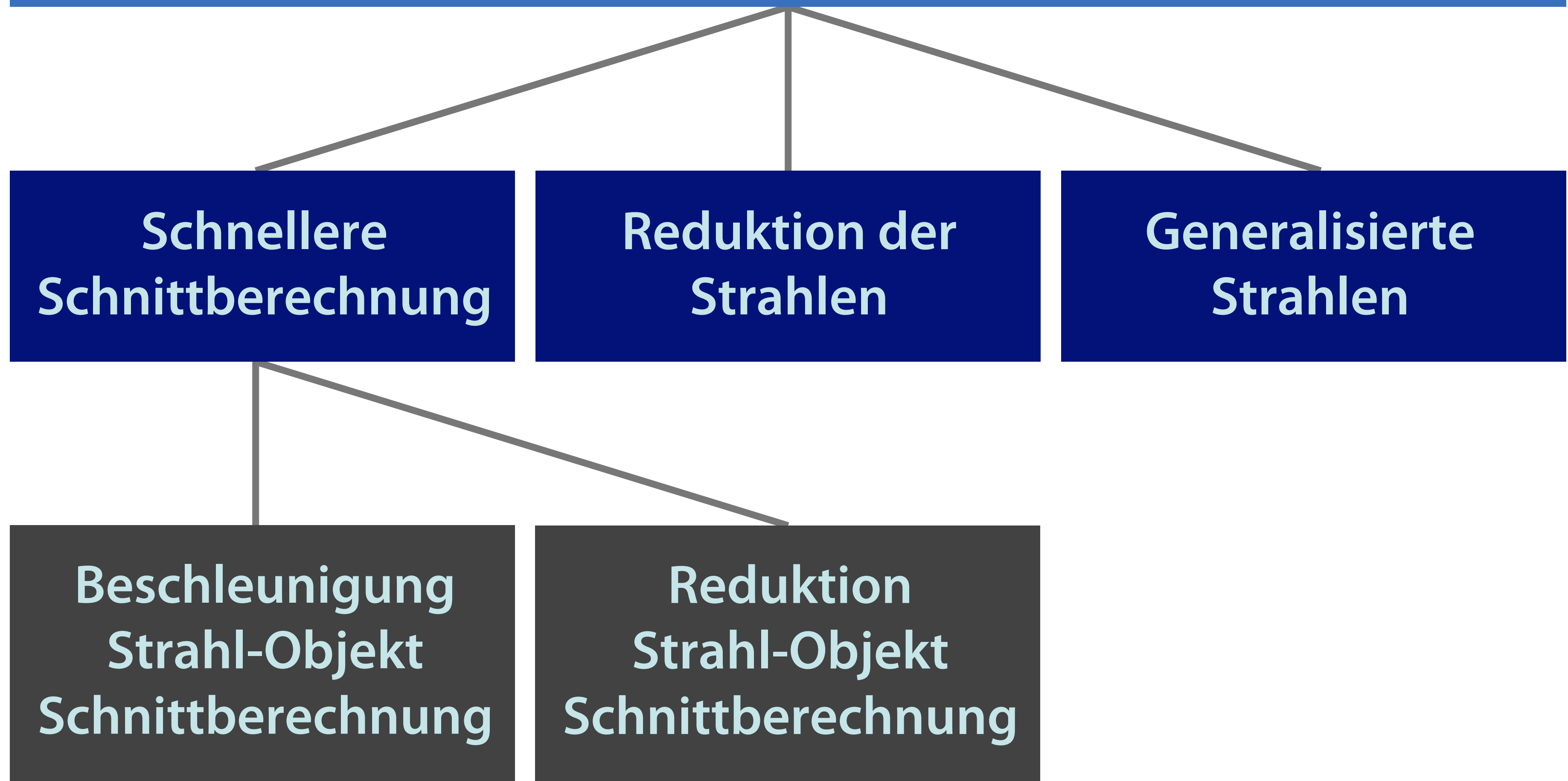


Bauhaus-Universität
Weimar

➔ HINTERGRUND

- Ray-Tracing: Eine der ältesten und hochwertigsten Techniken zur Erzeugung photorealistischer Bilder
- Weist gegenüber konventioneller Rasterisierung markante Vorteile auf *(Occlusion-Culling)*
- Nachteil: hoher Rechenaufwand resultiert in geringer Performance
- Verschiedenste Techniken wurden entwickelt um Ray-Tracing zu beschleunigen

➔ HINTERGRUND



→ HINTERGRUND

- Ray-Tracing ist in Kombination mit Beschleunigungs-verfahren und effizienter Nutzung heutiger Prozessoren Echtzeit-fähig
- Herausragende Stellung nimmt RTRT der Uni-Saarbrücken ein
- Mittlerweile auch kommerziell vertrieben und an Kunden wie VW ausgeliefert

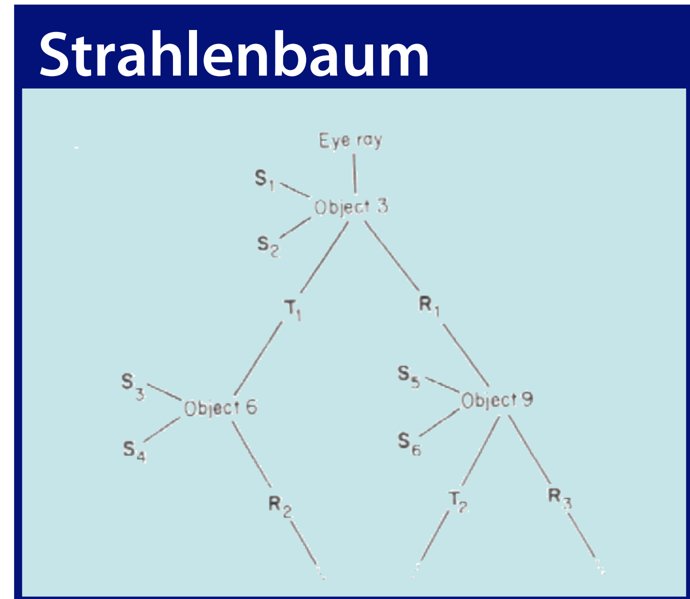
Intel
Developer
Forum

→ PINOCCHIO → FEATURES

- Iteratives Ray-Tracing Verfahren
- C++ unter Zuhilfenahme der Boost-Libraries
- Multithreaded
- Nutzt SIMD-Erweiterungen der Athlon und Intel Prozessoren
- SAH basierter BSP-Baum
- Ausschliessliche Verwendung von Dreiecksprimitiven
- Effiziente Schnittroutine

→ PINOCCHIO → ITERATIVES RAY-TRACING

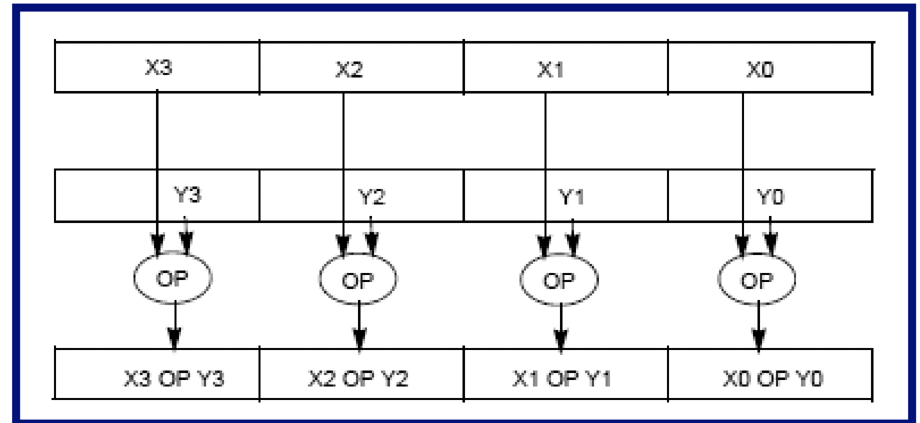
- Intensität eines Pixels:
  $I = I_a + I_d(L) + I_s(L) + I_r(R) + I_t(T)$
- Jede Strahlgeneration wird vollständig abgearbeitet, bevor nächste Generation betrachtet wird



- Beiträge jeder Generation werden auf entsprechendes Pixel aufakkumuliert

- Heutige Prozessorgenerationen ermöglichen Vektorisierung arithmetischer Operationen

- SSE-Erweiterung:
  Simultane Ausführung 4 identischer Operationen in 128-Bit Registern



- Definition eines Datentypen in gcc:

```
typedef float v4sf __attribute__((mode(V4SF)));
```

→ PINOCCHIO → SIMD-ERWEITERUNG

- Extrahierung einzelner Werte aus SSE-Datentyp prinzipiell nicht möglich

- Lösung:

```
union
f4Vector
{
        public:
        v4sf v;
        float32_pco_t values[4];
};
```

- Zwei Sichten existieren:
  → SSE-Einheit betrachtet `union` als 128-Bit Datentyp
  → Applikation betrachtet `union` als Array

➜ PINOCCHIO ➜ ALIGNMENT

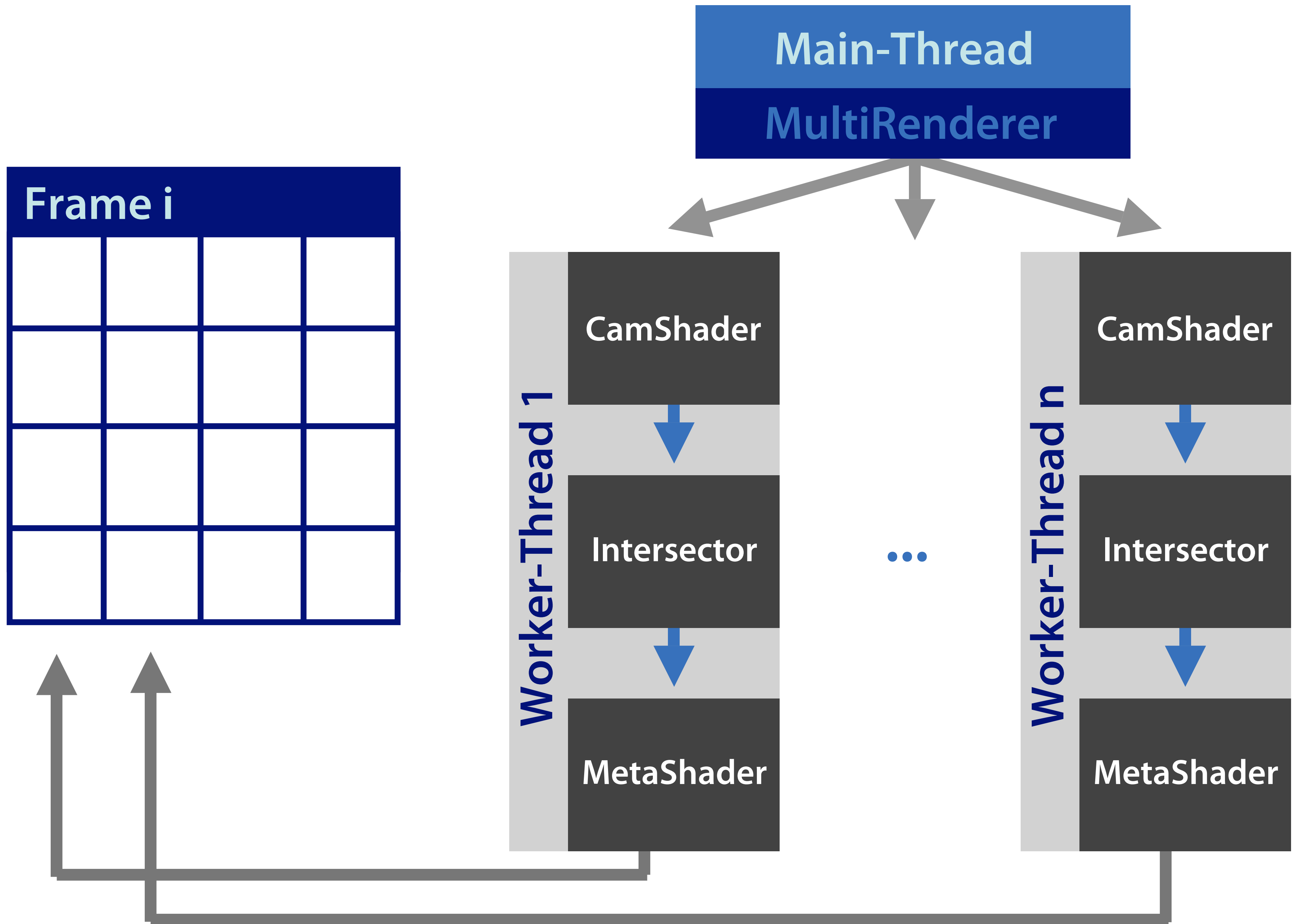
- SSE-Datentypen setzen 16-Byte alignment voraus
- Weiterer Aspekt: Effiziente Nutzung des Cache
- Daten werden blockweise *(Cacheline)* in Cache geladen
- Für verwendete Datentypen wird sichergestellt:
  ➜ Alignment
  ➜ Einhaltung der Cacheline-Grösse

→ PINOCCHIO → ALIGNMENT

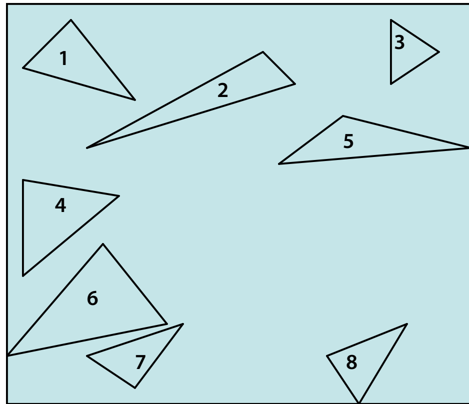
```
template <uint8_pco_t alignment = 64>
struct AlignedPoolAllocator{
...
  static char* malloc(const size_type bytes){
   void* memptr;
   posix_memalign(&memptr, alignment, bytes);
   return reinterpret_cast<char*>(memptr);}

  static void free(char* const block){
   std::free(block);}};
```

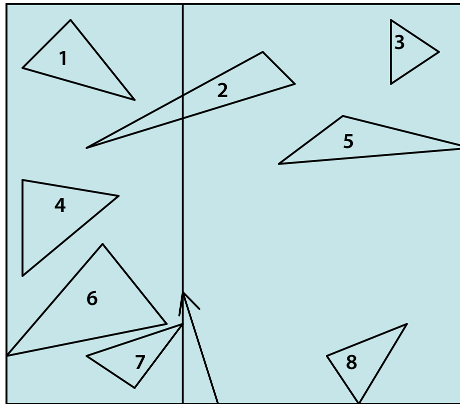
```
struct
#if defined __INTEL_COMPILER
__declspec(align(32))
#endif
PhongAccel{
  PhongAccel();
  ~PhongAccel();

  void* operator new(size_t size);
  void* operator new[](size_t size);
  void* operator new(size_t, void* ptr);
  void  operator delete(void* ptr, size_t size);
  void  operator delete[](void* ptr, size_t size);

  float kd_[3];
  float ks_[3];
  float glossi_;
  float pad_;
}
#if defined __GNUCPP__
__attribute__((aligned(32)))
#endif;

typedef boost::singleton_pool<PoolTagDummy, sizeof(PhongAccel),\
AlignedPoolAllocator<32> > phongAccelPool;
```

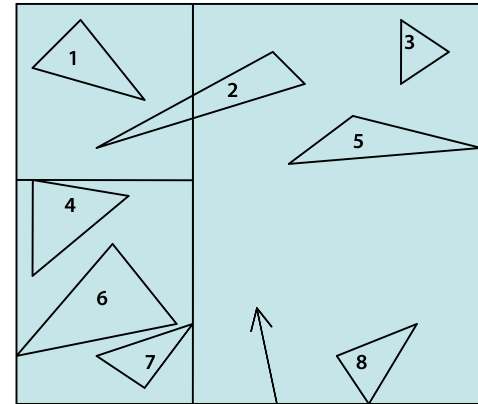
→ PINOCCHIO → MULTITHREADING



- Binary Space Partition Tree







Axis Aligned Bounding Box

Splitting Plane

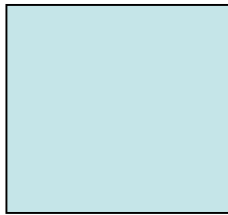
Voxel

→ PINOCCHIO → BSP-TREE

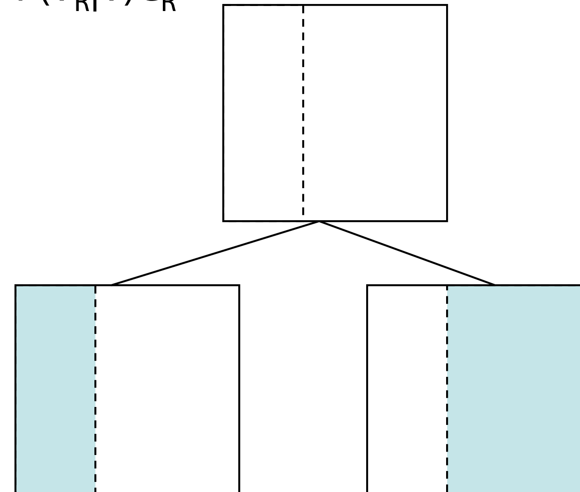
- Konstruktion
  - Platzierung der Schnittebene
    - Unterteilung im Median des Voxels
    - Unterteilung im Objekt Median
    - Kostenfunktion
  - Abruchkriterien
    - maximale Baumtiefe (20-25)
    - Primitive pro Blatt (2-4)
    - mittels Kostenvergleich

- Konstruktion mit Surface Area Heuristic
- $P(V_L|V) = SA(V_L)/SA(V)$

$Cost_{LEAF} = C_{iters} * N_{LEAF}$



$Cost_{INNER} = C_{TRAV} + P(V_L|V)C_L + P(V_R|V)C_R$



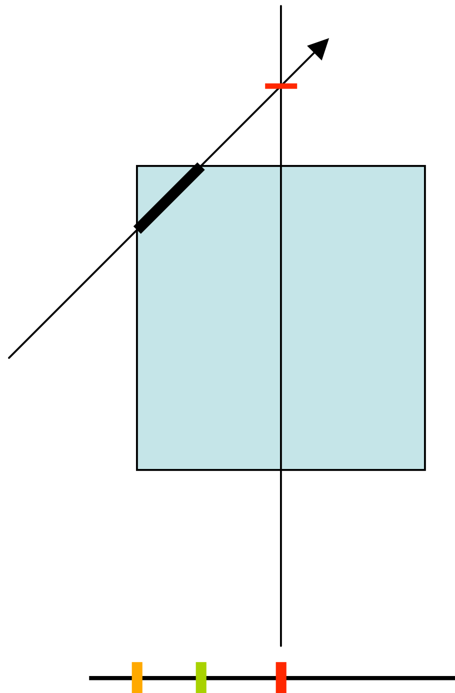
- finde beste split position (minimale kosten)
- Terminierungskriterium: Split Kosten > $Cost_{LEAF}$

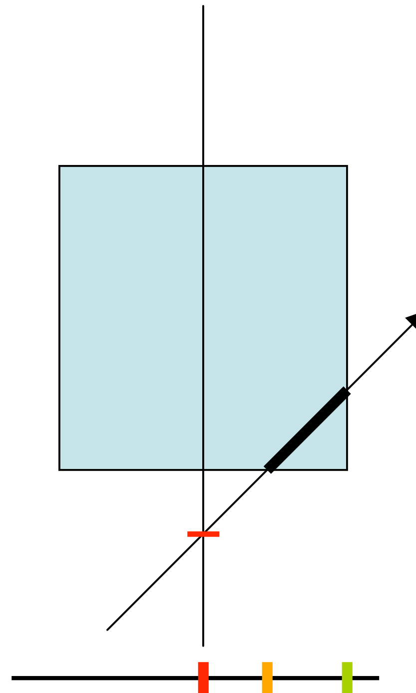
- BspNode

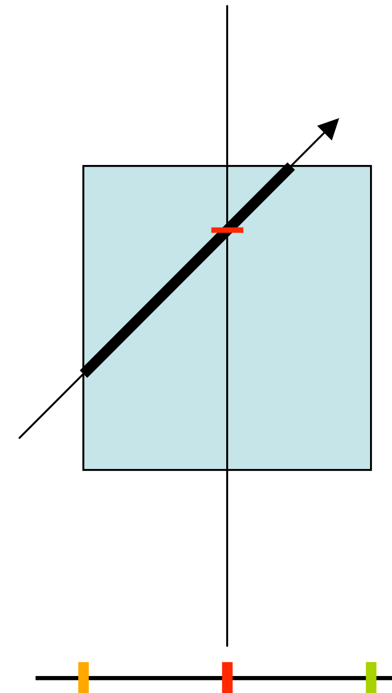
```
class BspNode2
{
public:
    BspNode2() : m_flags(0), m_nr(0)      {};
    void            SetLeaf( bool a_Leaf );
    void            SetAxis( int a_Axis );
    void            SetSplitPos( float a_Pos );
    void            SetLeft( BspNode2* a_Left );
    void            SetList( TriAccel * l);
    int             GetAxis();
    float           GetSplitPos();
    BspNode2*       GetLeft();
    BspNode2*       GetRight();
    bool            IsLeaf();
    TriAccel *      GetList();
    unsigned int    GetNrOfObj();
    void            SetNrOfObj(const unsigned int);

private:
    unsigned int    m_flags;  //!< Inner: Bit 0..1 splitting dim, 2 isleaf 3..31 first child address
                              //!< Leaf : Bit 2 flag isleaf, 3..31 address of object list
    union {
        float          m_split;   //!< split position
        unsigned int   m_nr;      //!< number of triangles in node
    };
};
```

- Traversierung

if (ray.direction_[axis] > 0) { // left to right







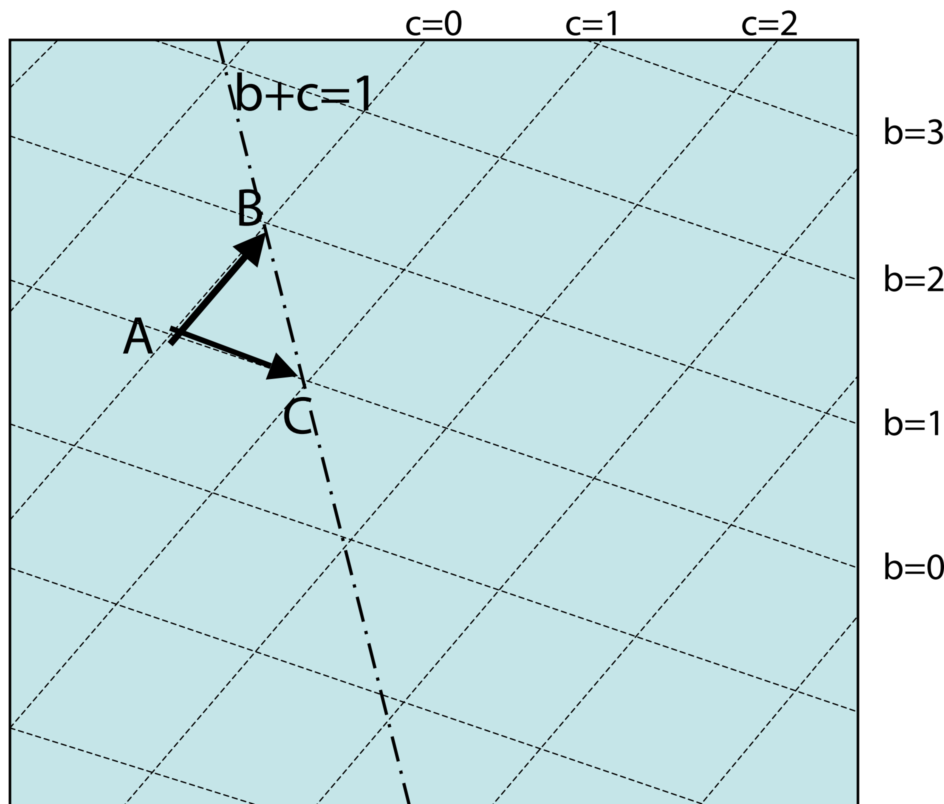
tnear tfar   d

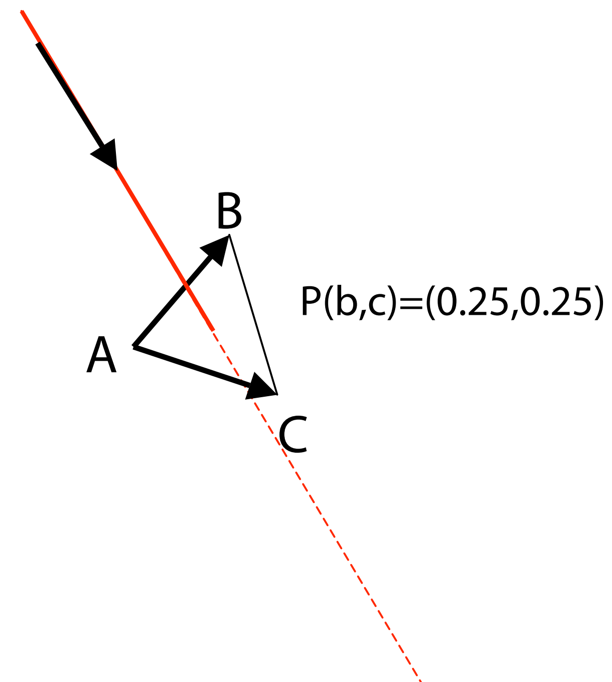
d tnear tfar

tnear   d   tfar

} else { // rigth to left

- Baryzentrische Koordinaten
- P(a,b,c) = aA+bB+cC mit 0 <= a,b,c <= 1 und a+b+c = 1
- P(b,c) = A+b(B-A)+c(C-A) mit a = 1-b-c
- P im Dreieck dann und nur dann wenn: 0 <= b,c und b+c <= 1





→ PINOCCHIO → TRIANGLE INTERSECTION

- o + *t*d = A + b(B-A) + c(C-A)
- Beschleunigung nach Wald [Wald04]
- Projektion von Schnittpunkt und Dreieck in die Ebene
- baryzentrische Koordinaten bleiben erhalten

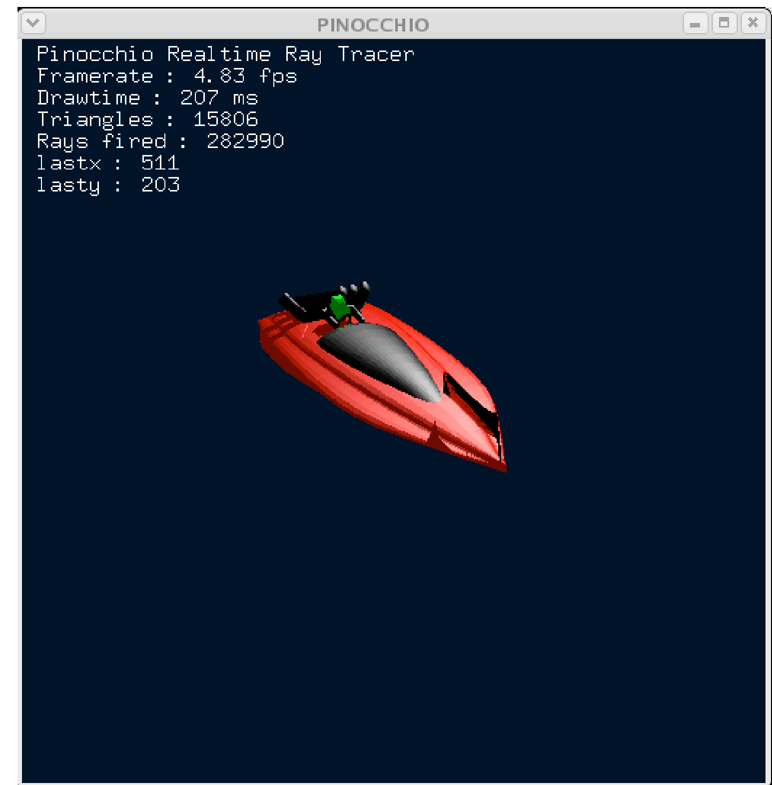
- Durch Vorrausberechnung einiger Terme effiziente Schnittberechnung
- Maximal: 10 Multiplikationen, 1 Division und 11 Additionen

➔ PINOCCHIO ➔ Ausblick

- Status Quo:

  Real Time Raytracing von
  komplexen Modellen
  auf der CPU möglich!

  NB: Speedboat Modell
  hat ca. 16.000 Dreiecke
  - pinocchio packt das
  mit 5-10 Frames/sec@512x512.



→ PINOCCHIO → Ausblick

- CPU alleine reicht nicht aus - Wie geht es weiter?

-> Kombination Real Time Ray Tracing und OpenGL.

-> Rechenleistung von CPU und GPU geschickt vereinen.

-> *“Pinocchio lernt die GPU kennen."*

→ PINOCCHIO → Ausblick→ Ueberblick

- Vorbemerkungen
- CPU / GPU Charakteristik
- *“Analyse eines Bildes”*
- 2 Designvorschläge

- Hohe Anforderungen an den CPU-GPU Datenaustausch:

  Lösung: PCI Express (x16)

  -> theoretisch 4x Bandbreite von 8xAGP, d.h. **8** Gbyte/s down- und upload!

  -> hoffentlich bald Realität...

→ PINOCCHIO → Ausblick→ Vorbemerkungen

- Anforderungen an OpenGL/SL Programmierung:

  Phong Shading auf der GPU (besser als Gouraud).

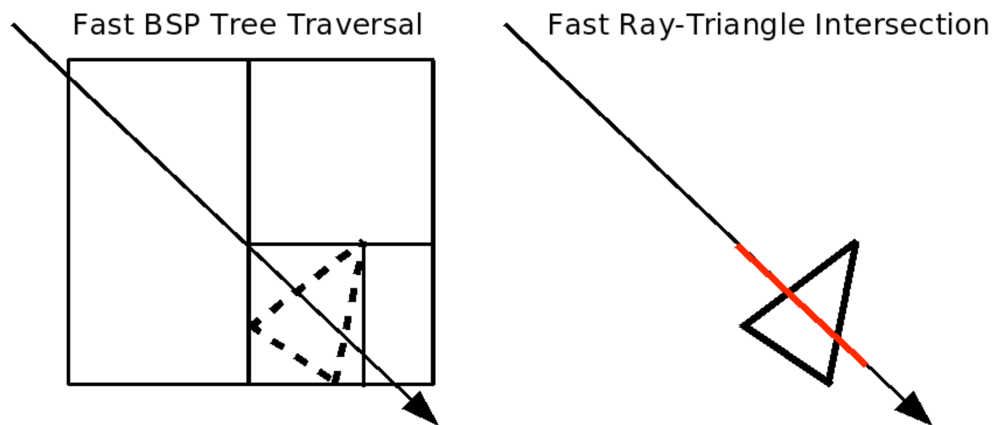
  pBuffer Klasse ermöglicht 128bpp Buffer (-> Designvorschlag 2).

  -> neue Schwerpunkte im Projekt: OpenGL / SL

→ PINOCCHIO → Ausblick→ CPU-Charakteristik

- general purpose computing!
- multithreading, multiprocessing.
- bei der Berechnung von Graphik im wesentlichen perPixel, resp. perRay -Algorithmen (sequentiell / SIMD-SSE).

-> zeigt sich in speziellen Algorithmen + Beschleunigungsstrukturen bei pinocchio.



-> ~10 Millionen R-T Intersections/s

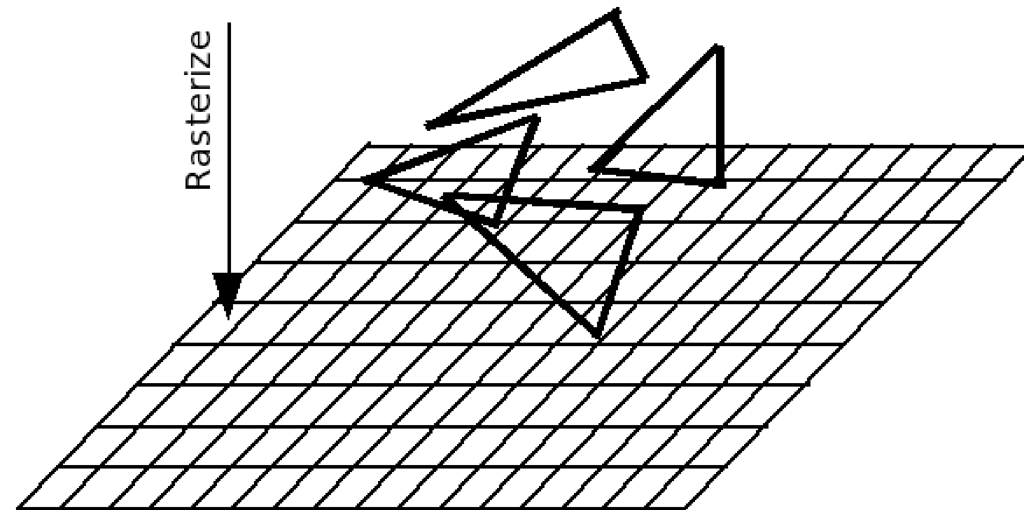
(pentium4 / AMD Athlon XP 2800+)

→ PINOCCHIO → Ausblick→ GPU-Charakteristik

• computing for graphics!

-> *“Graphikpipeline, Streamprocessing, BruteForce”*

• im wesentlichen perBuffer / perFragment Berechnungen in OpenGL programmiert



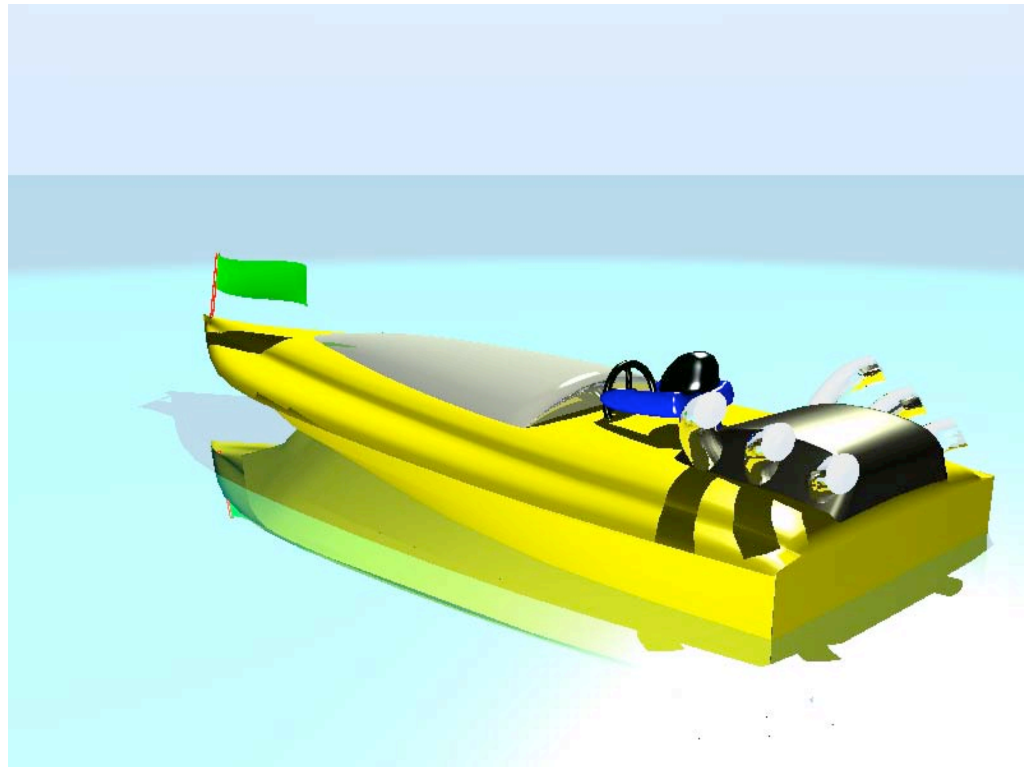
-> 600 Millionen Vertices/sec
(NVIDIA GeForce 6800)

Phongshading

Spiegelung

Schatten

2 Lichtquellen



-> Beleuchtung in der Computergrafik sehr wichtig

(Licht ist eine *"hohe Kunst"* in Film+Photographie)

Bemerkung: Gegenlichtsituation – nur eine Lichtquelle *"castet"* Schatten.

- Bemerkung:

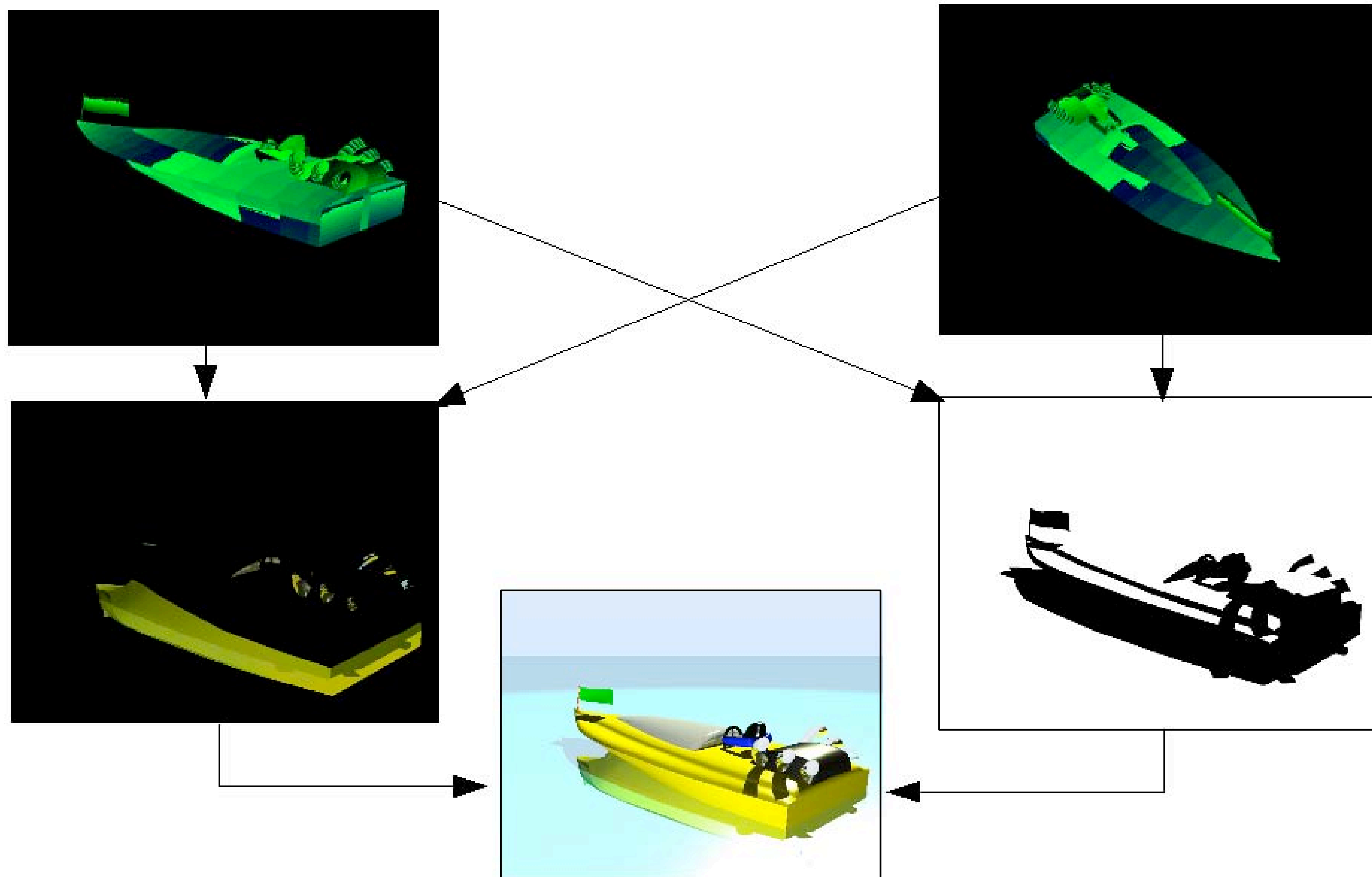
  viele zusätzliche Aspekte wurden hier noch nicht betrachtet.
  (Texturemapping, Bump/Environmentmapping, Radiosity, etc.)

  => nun 2 Vorschläge wie man Spiegelung,
  Schatten und Phongshading berechnen könnte.

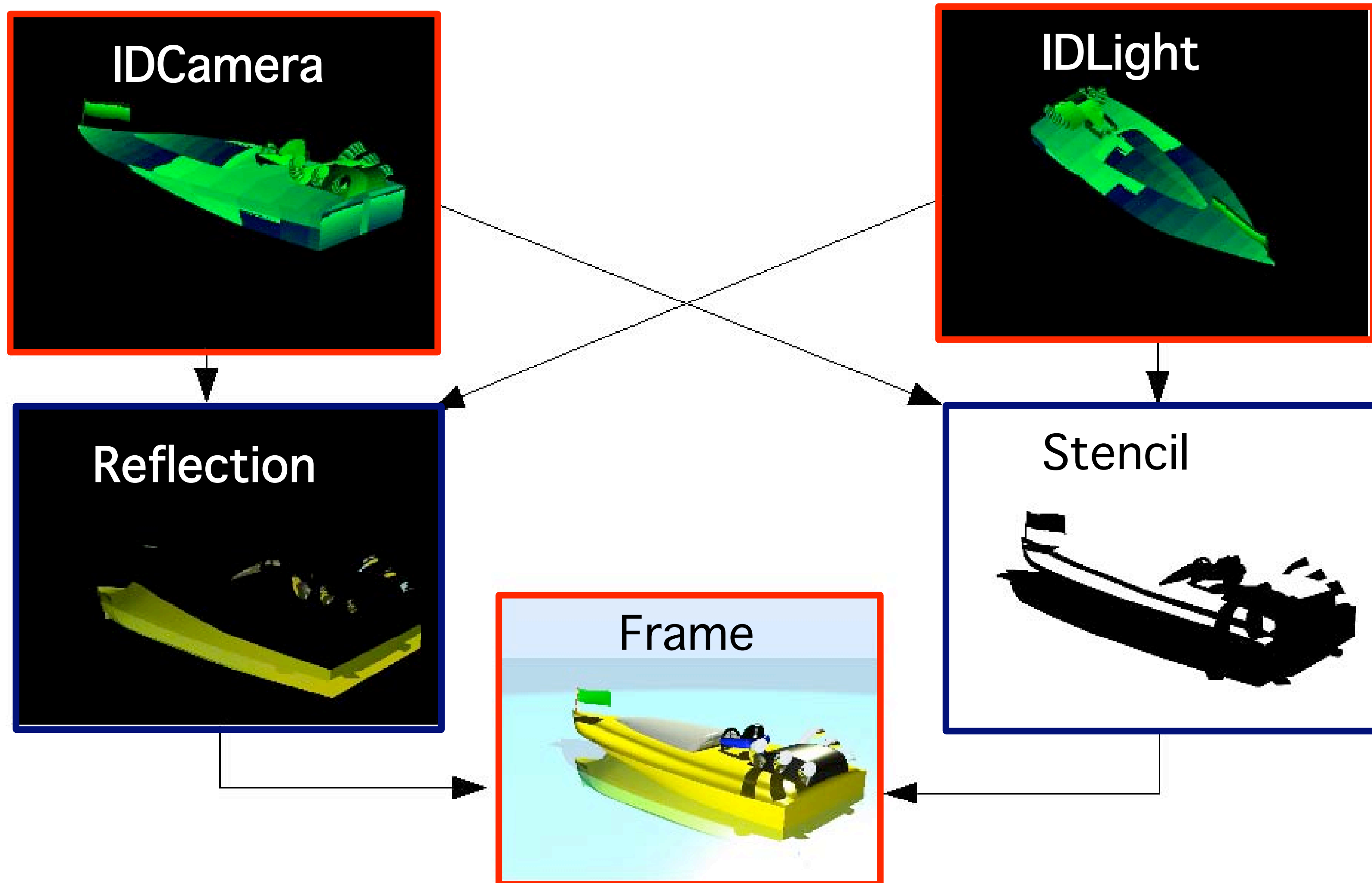
- Fazit:

  realistische Bilder zeichnen sich durch komplexe Eigenschaften aus.
  -> Wie sieht hierfür eine sinnvolle Aufgabenverteilung CPU - GPU aus?

- separiere die Informationen des Bildes:
  -> *“Welche Informationen sind nötig, damit die GPU ein Bild PhongShaden kann sowie Spiegelung und Schatten zu sehen ist?”*



- separiere die Informationen des Bildes:
  -> *“Welche Informationen sind nötig, damit die GPU ein Bild PhongShaden kann sowie Spiegelung und Schatten zu sehen ist?”*



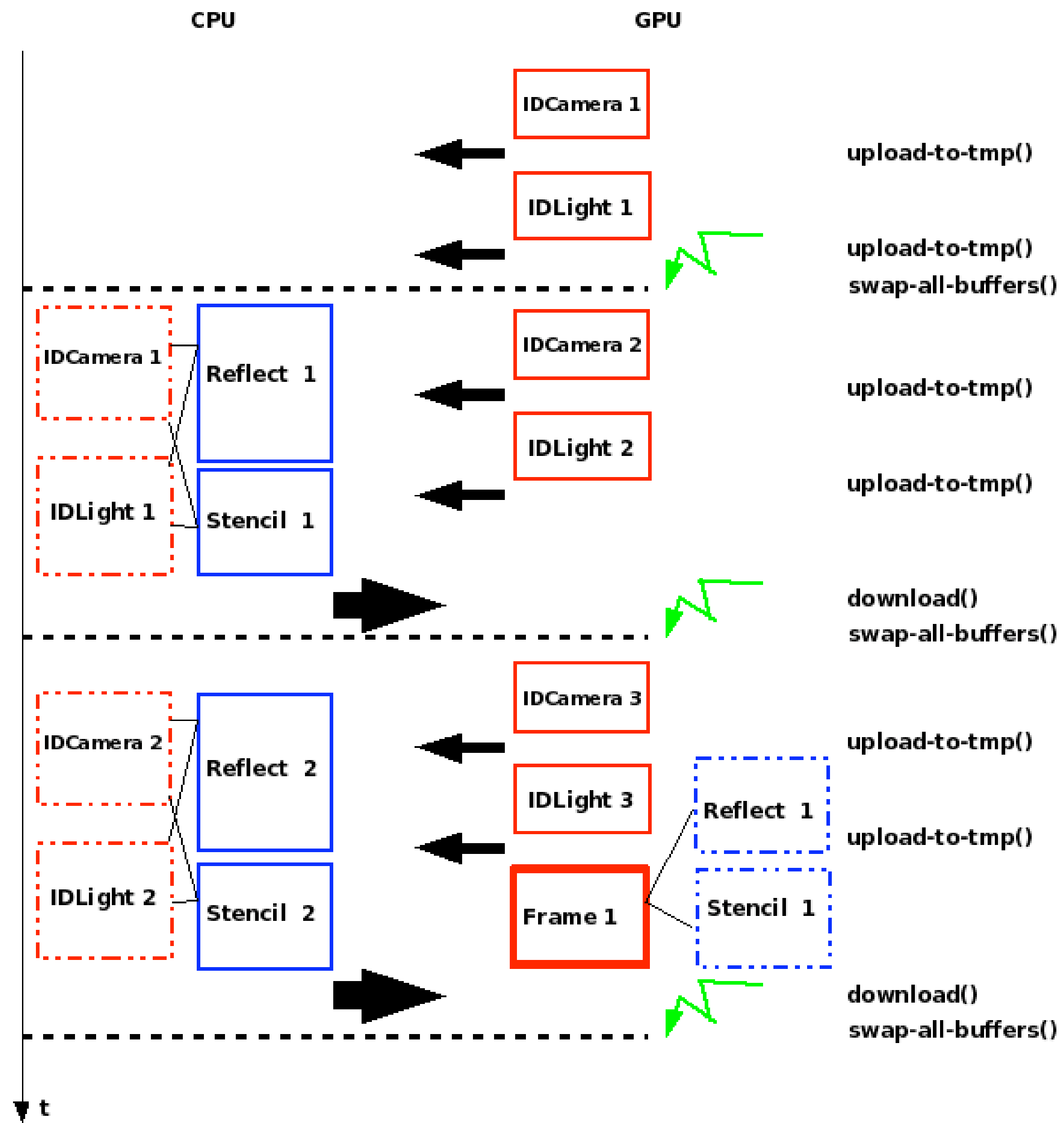
- parallele Renderpasses in der CPU und GPU:

  -> *"CPU und GPU arbeiten sich zu."*

  -> upload/frame: 2x800x600x24bit = **2.88** MByte

  -> download/frame: 800*600*8bit + 800*600*24bit = **1.92** MByte

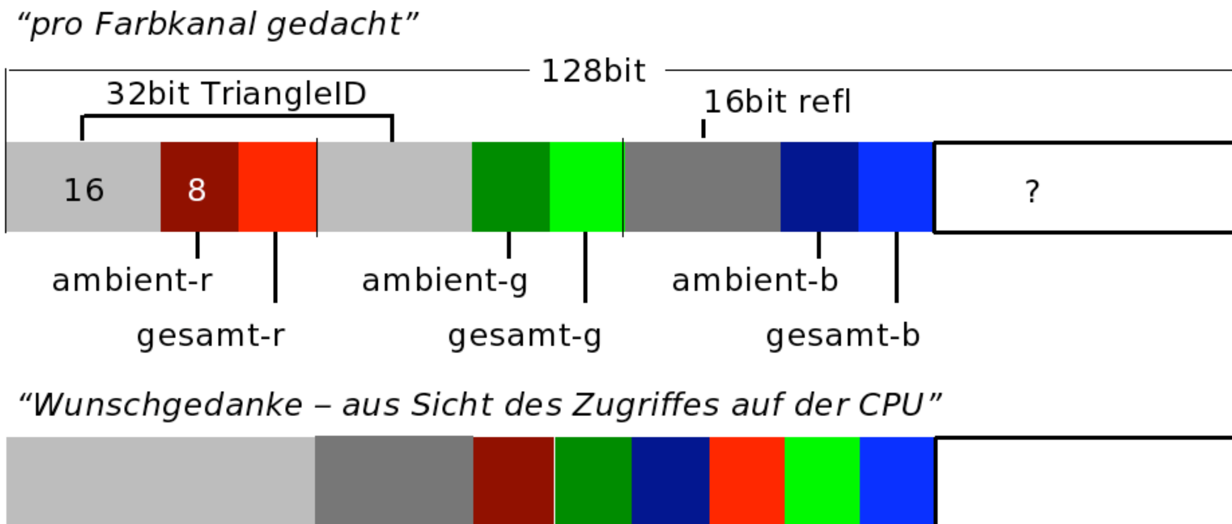
  -> 2 frames delay upload > download



→ PINOCCHIO → Ausblick → Design 2 → Idee

- Verwendung eines 128bpp tiefen pBuffers.
  -> *"codiere sinnvolle Pixeldaten in den breiten Buffer."*

  -> mehrere Möglichkeiten...e.g.:



-> CPU fügt die *"Komponenten"* zusammen (mit Schattentest) und ergänzt das Bild um Spiegelung.

-> Konsequenzen: höhere Anforderungen an die GPU-Programmierung + Graphikhardware - *"a little magic"*.

→ PINOCCHIO → Ausblick→ Design 2→ Rendern

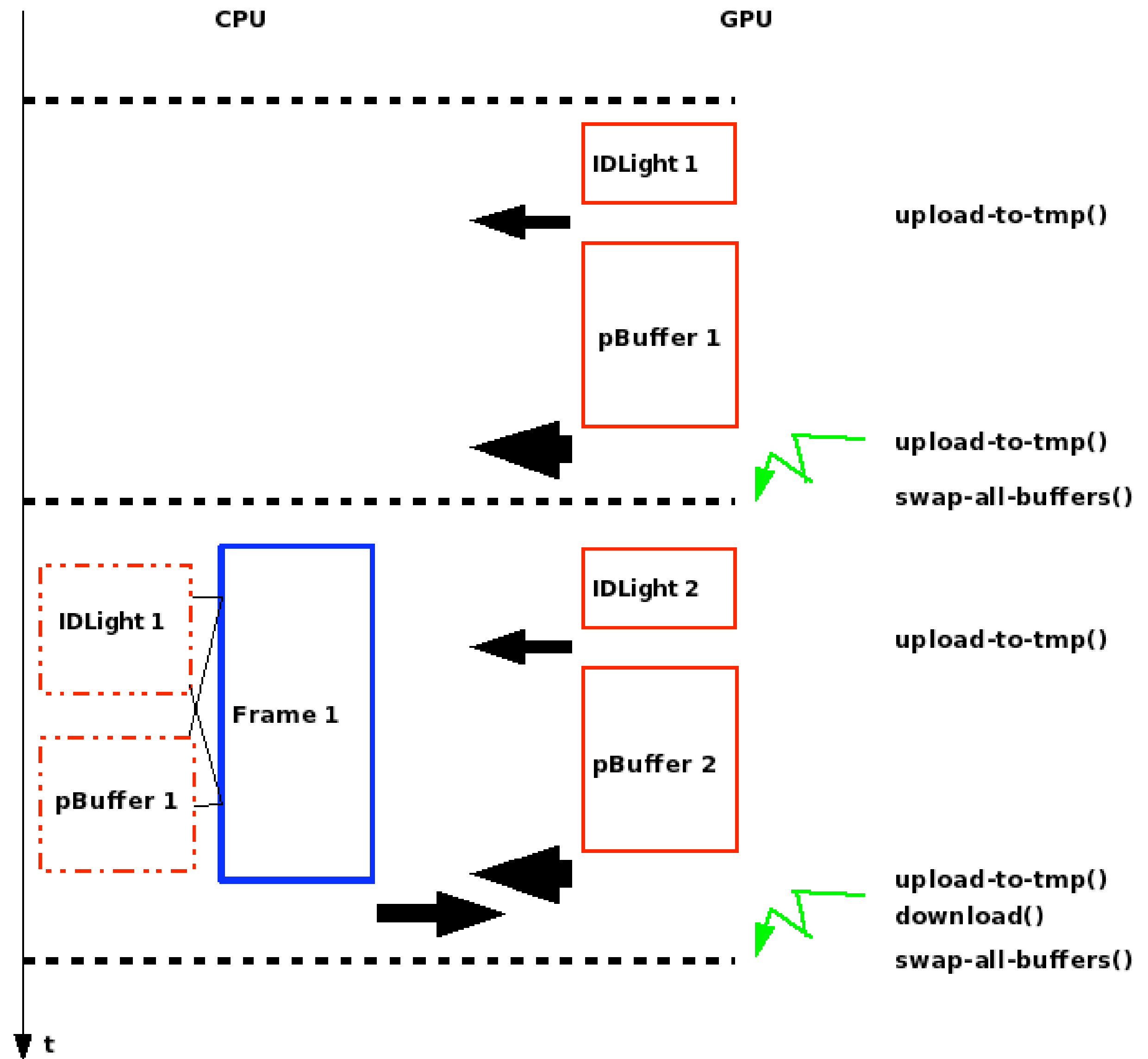
- parallele Renderpasses auf der CPU und GPU.

  -> *“GPU arbeitet der CPU zu.”*

  -> upload/frame:
  800x600x24bit+
  800*600*128bit
  = **9.12** MByte

  -> download/frame:
  800*600*24bit
  = **1.44** MByte

  -> 1 frame delay,
  upload >> download



→ PINOCCHIO → Ende→ Demo

- Vielen Dank für Ihre Aufmerksamkeit :)
- Fragen?
- Demo