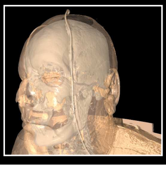
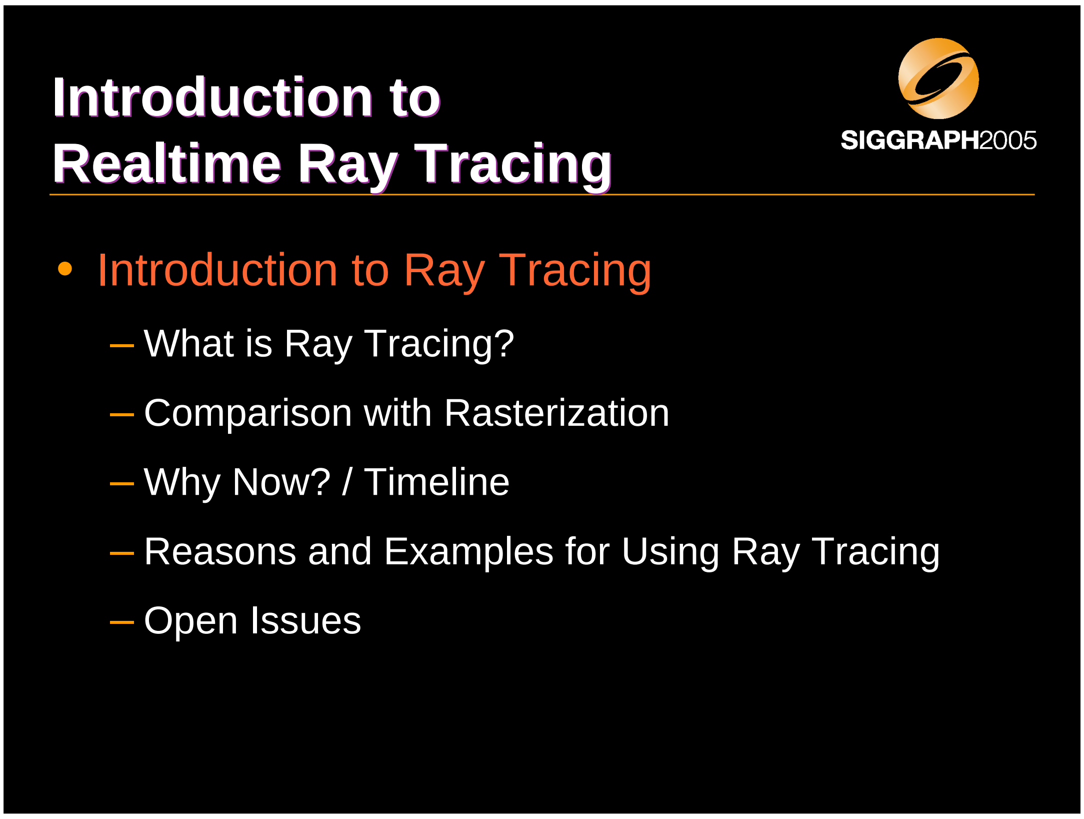
SIGGRAPH2005

Introduction to Realtime Ray Tracing

Course 41

Philipp Slusallek Peter Shirley
Bill Mark Gordon Stoll Ingo Wald

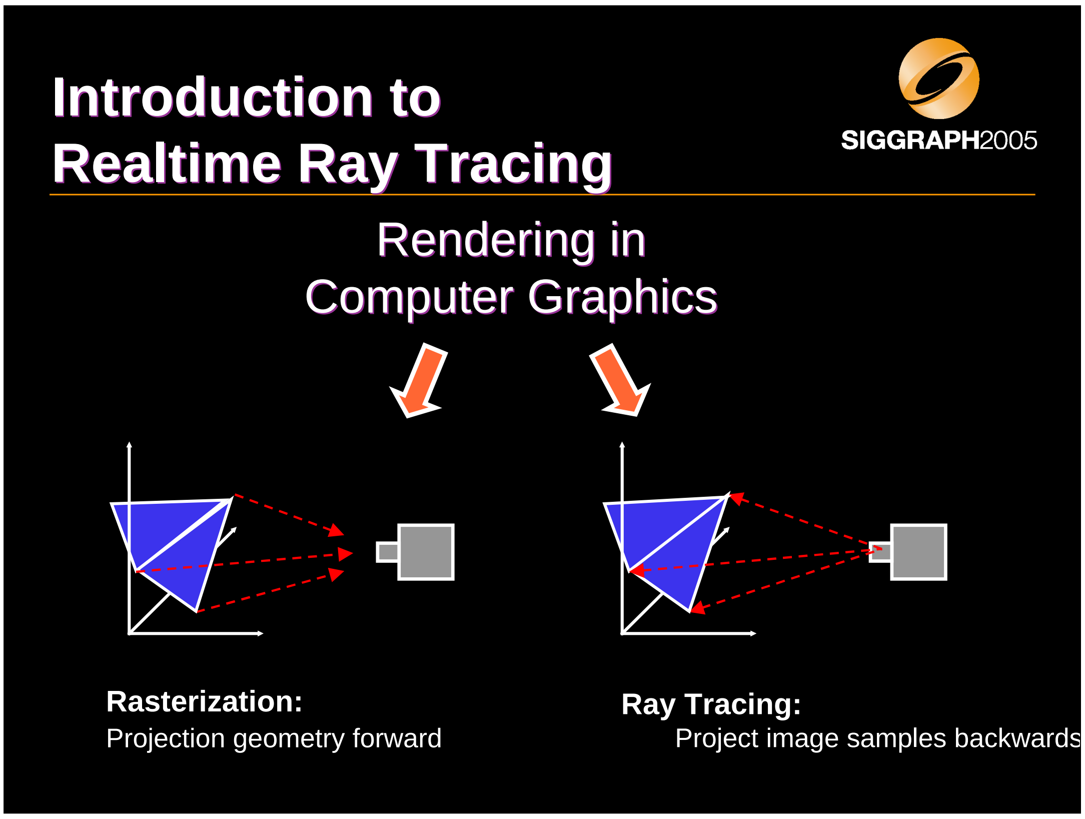




Introduction to Realtime Ray Tracing

SIGGRAPH2005

- Introduction to Ray Tracing
  - What is Ray Tracing?
  - Comparison with Rasterization
  - Why Now? / Timeline
  - Reasons and Examples for Using Ray Tracing
  - Open Issues



Introduction to Realtime Ray Tracing

SIGGRAPH2005

Rendering in Computer Graphics

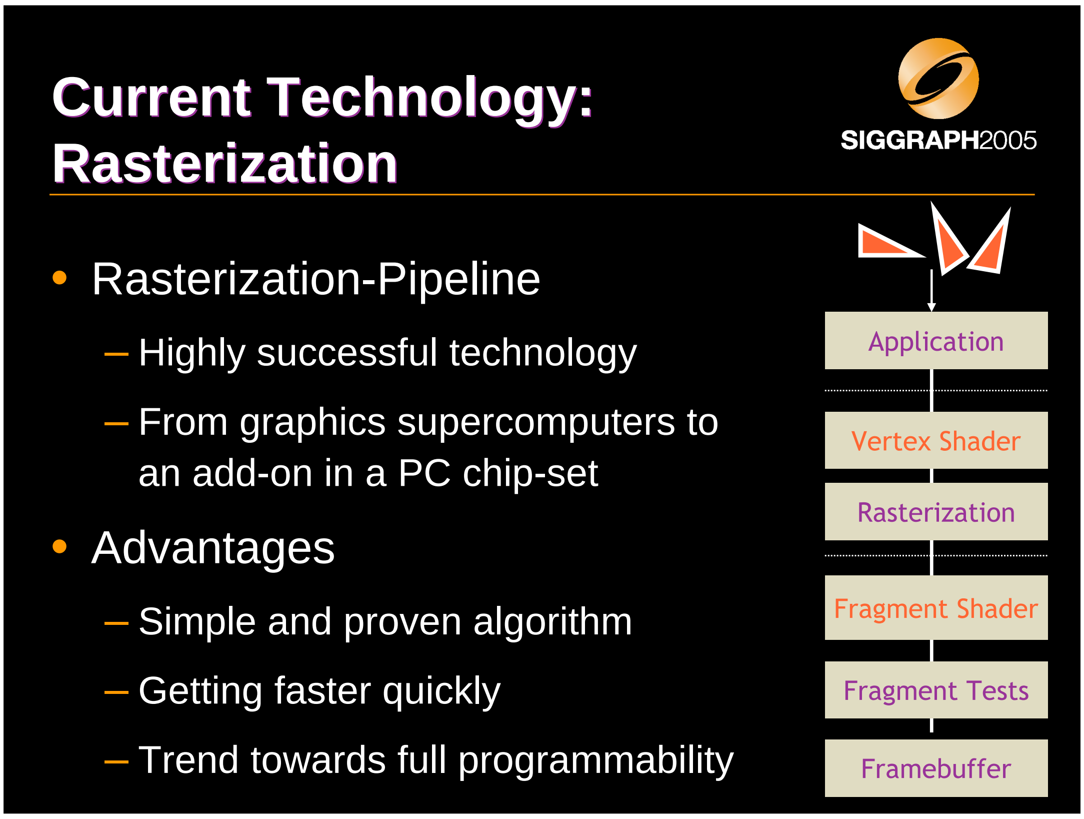
Rasterization:
Projection geometry forward

Ray Tracing:
Project image samples backwards

Computer graphics has only two basic algorithms for rendering 3D scenes to a 2D screen. The dominant algorithms for interactive computer graphics today is the rasterization algorithm implemented in all graphic chips. It conceptually takes a single triangle at a time, projects it to the screen and paints all covered pixels (subject to the Z-buffer and other test and more or less complex shading computations).

Because the HW has no knowledge about the scene it must process every triangle leading to a linear complexity with respect to scene size: Twice the number of triangles leads to twice the rendering time. While here are options to optimize this, must be done in the application separate from the HW.

The other algorithm – ray tracing – works in fundamentally different ways. It starts by shooting rays for each pixel into the scenes and uses advanced spatial indexes (aka. acceleration structures) to quickly locate the geometric primitive that is being hit. Because these indexes are hierarchical they allow for a logarithmic complexity: Above something like 1 million triangles the rendering time hardly changes any more.



Current Technology: Rasterization

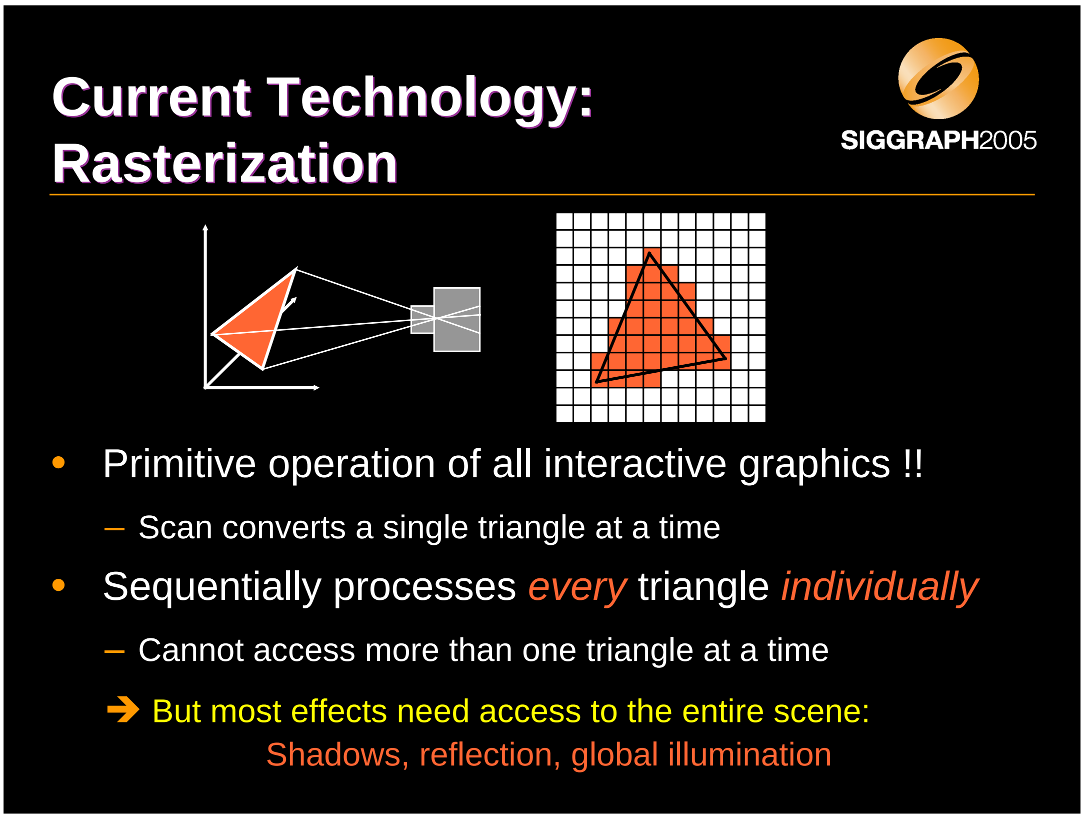
SIGGRAPH2005

- Rasterization-Pipeline
  - Highly successful technology
  - From graphics supercomputers to an add-on in a PC chip-set
- Advantages
  - Simple and proven algorithm
  - Getting faster quickly
  - Trend towards full programmability

Application
Vertex Shader
Rasterization
Fragment Shader
Fragment Tests
Framebuffer

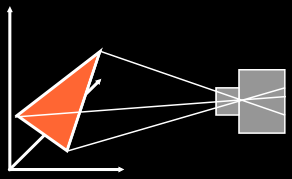
Computer graphics knows two different technologies for generating a 2D image from 3D scene description: rasterization and ray tracing.

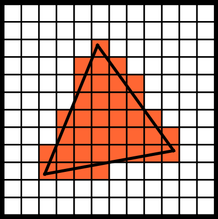
Virtually all interactive graphics today uses the rasterization technique. Because it was easier to implement in hardware during early days of interactive computer graphics (early 1980s), it took over the world. Mainly driven by companies such as SGI, Nvidia, ATI, and some others rasterization hardware developed rapidly to the point that this graphics technology is already embedded in many motherboard chip sets. And the technology is still developing at an astonishing pace. In particular, significant programmability is being added to the rasterization pipeline in every new generation.



Current Technology: Rasterization

SIGGRAPH2005





- Primitive operation of all interactive graphics !!
  - Scan converts a single triangle at a time
- Sequentially processes *every* triangle *individually*
  - Cannot access more than one triangle at a time

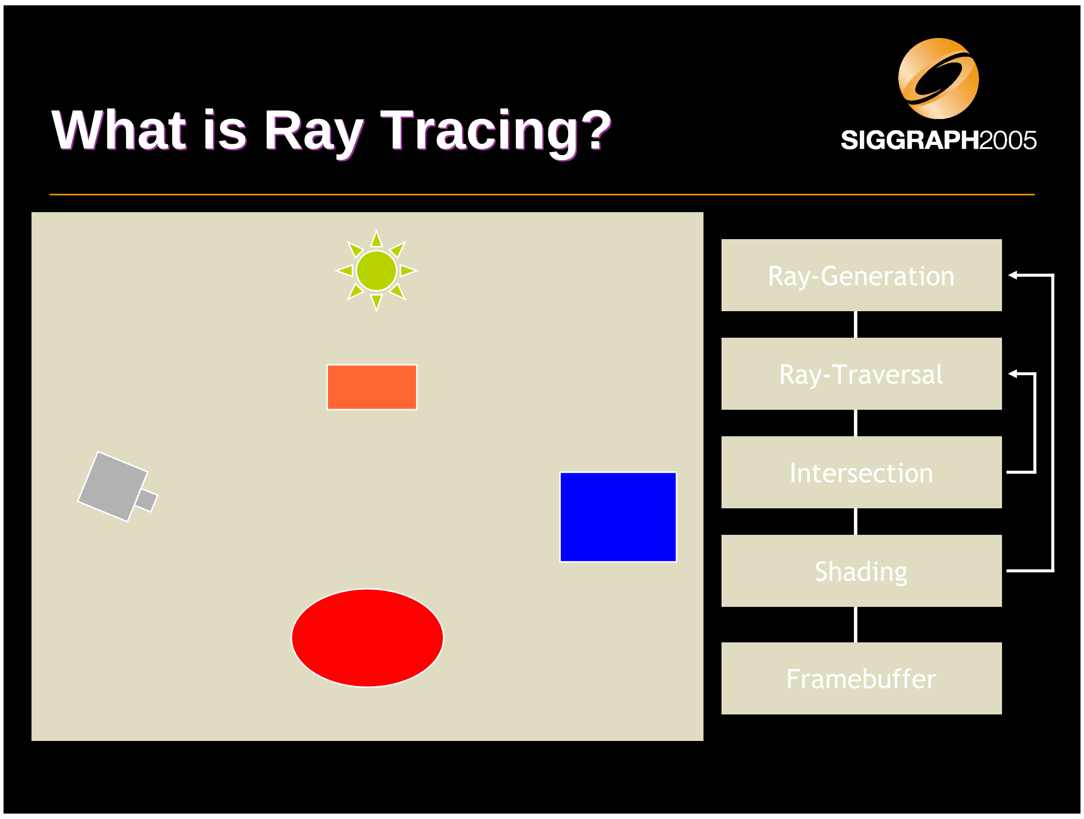
➔ But most effects need access to the entire scene:
Shadows, reflection, global illumination

The basic operation of rasterization is to sequentially project each triangle sent by the application and projecting it to the 2D screen. Then the pixel covered by the triangle are computed and each one is colored according to some programmable shader functions.

It is important to note that the rasterization pipeline derives its basic efficiency from the fact that it conceptually looks only at a single triangle at a time in the order they ar submitted by the application. At no point does the graphics chip have the ability to look at the rest of the scene.

Yet, this ability is key to some of the most basic and simple optical effects that are required for faithfully rendering 3D scenes. Computing the shadows cast on a triangle requires knowing about the triangles casting the shadow, computing the reflection requires the ability to find the triangles being reflected, and computing indirect illumination on a triangle requires access to the entire scene.

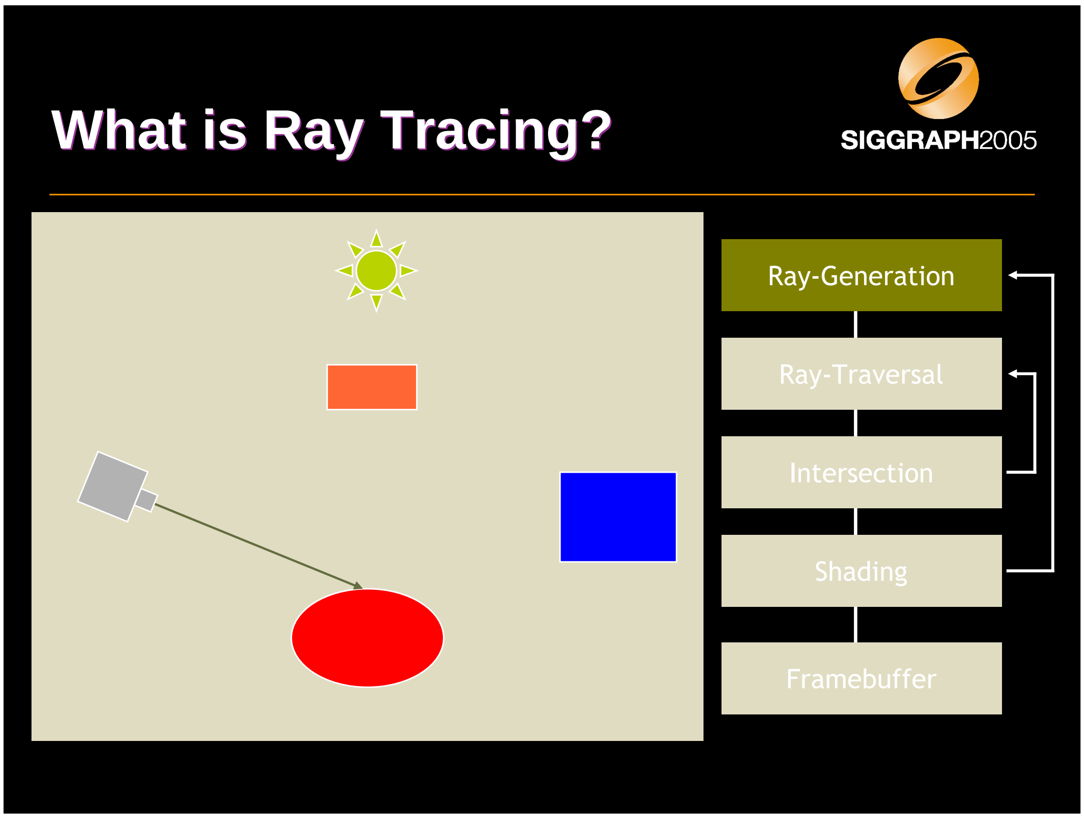
With rasterization such functionality cannot be computed accurately and approximations and fakes must be used (e.g. shadow maps, reflection maps). However, they necessarily have inaccuracies and artifacts and are generally less efficient.



What is Ray Tracing?

SIGGRAPH2005

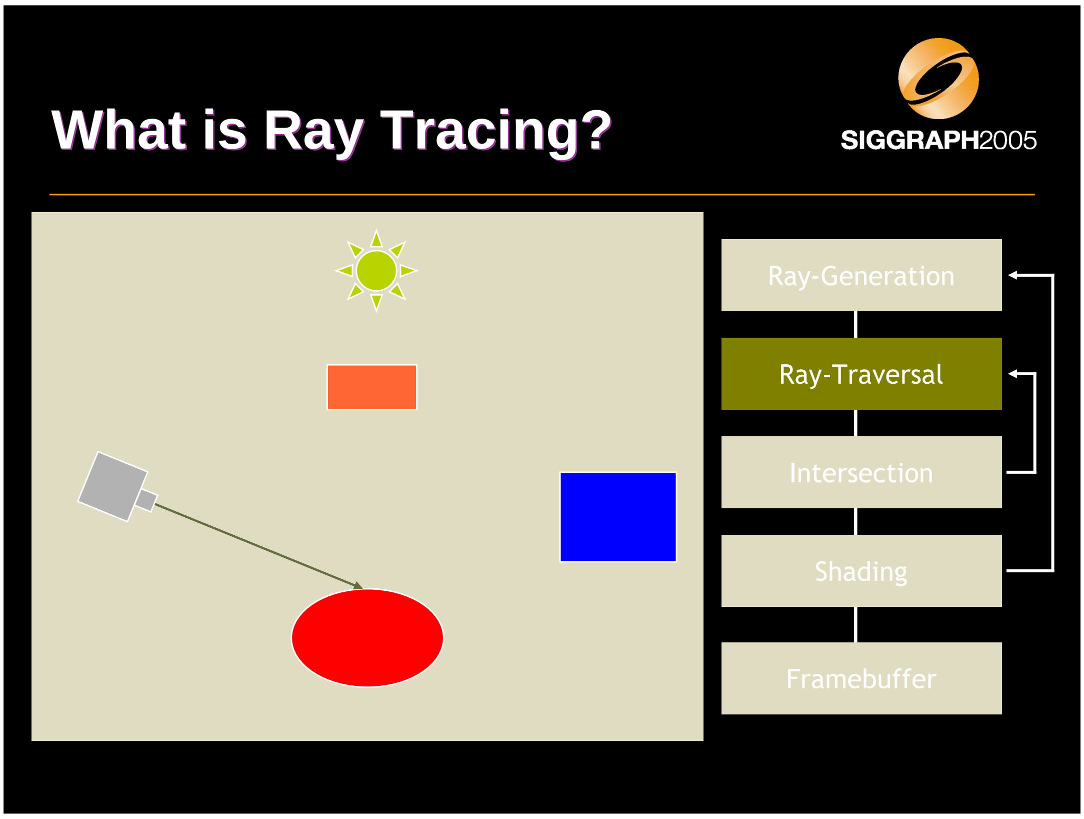
Ray tracing can be formulated very much like the pipeline known from rasterization (an with similar efficiency). A significant change, however, is the feedback loops in the pipeline, which are the key to its ability to compute global information in the scene.



What is Ray Tracing?

SIGGRAPH2005

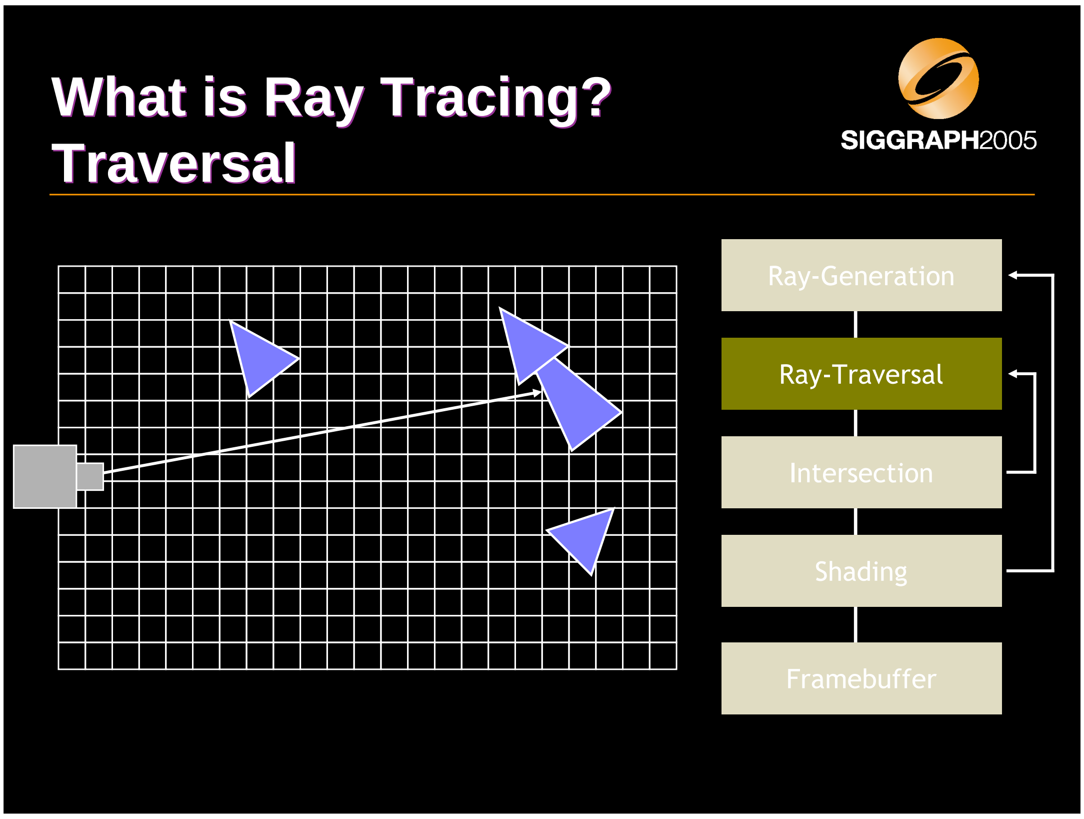
The first stage in the pipeline computes a rays from the camera's parameter and a 2D sample location on the screen.



What is Ray Tracing?

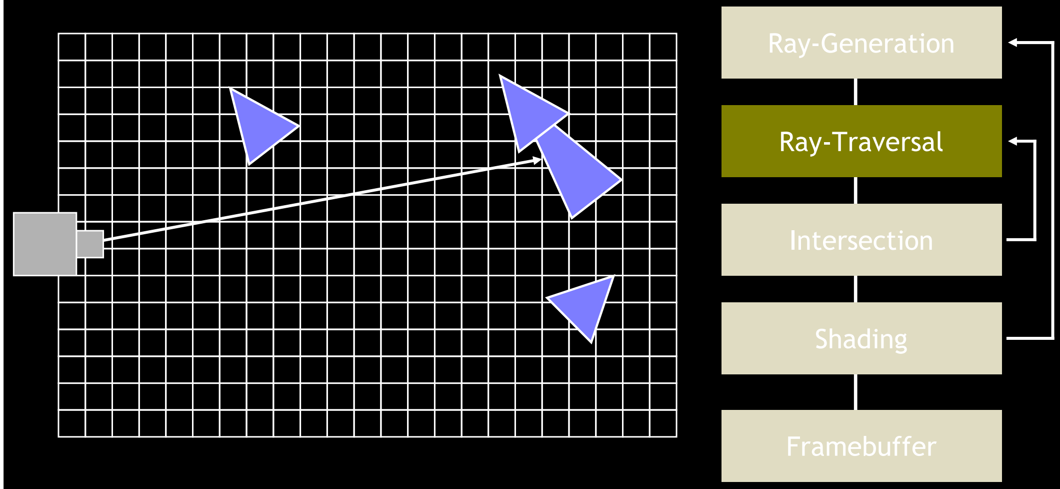
SIGGRAPH2005

The traversal stage takes the ray and traverses the spatial index to locate the first hit point of the ray with a geometric object.



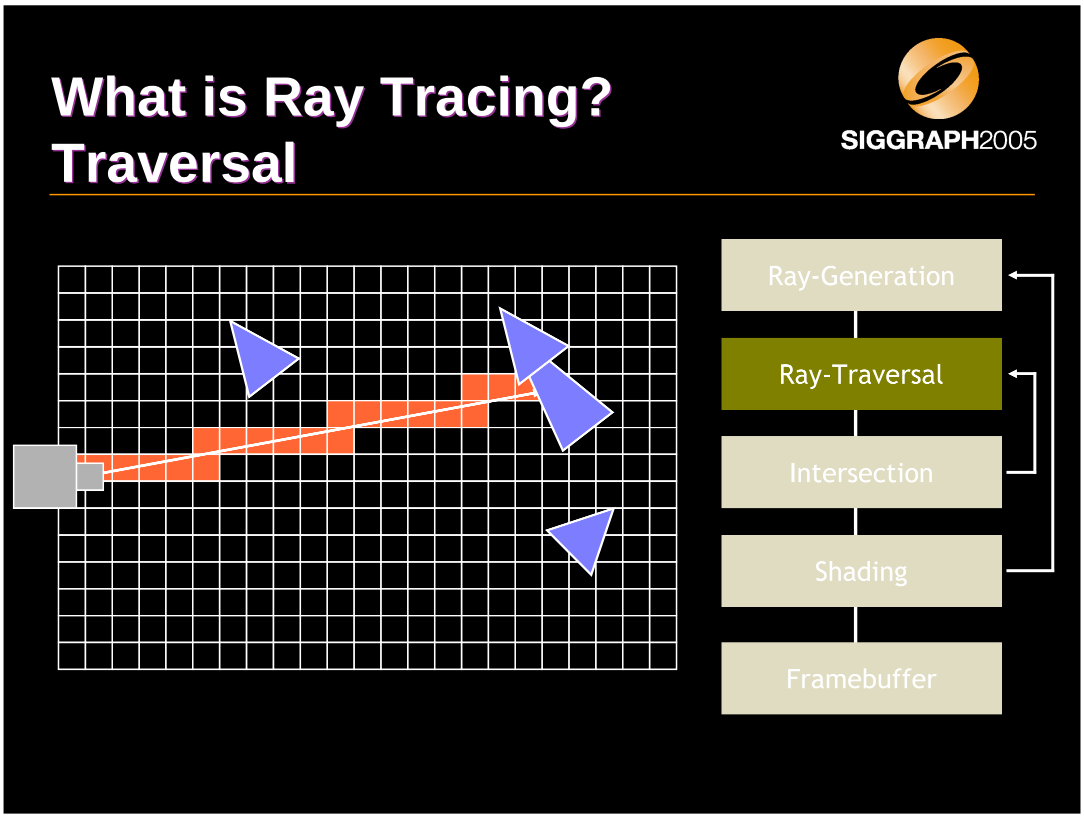
What is Ray Tracing? Traversal

SIGGRAPH2005



Here we visualize the spatial index as a (2D-) grid. Obviously the index is a 3D structure. However, 3D-grids are not often used (except in some HW projects) as they waste much memory and cannot adapt to local changes, such as a highly dense region with many close-by triangles. The grid only symbolizes the spatial index here.

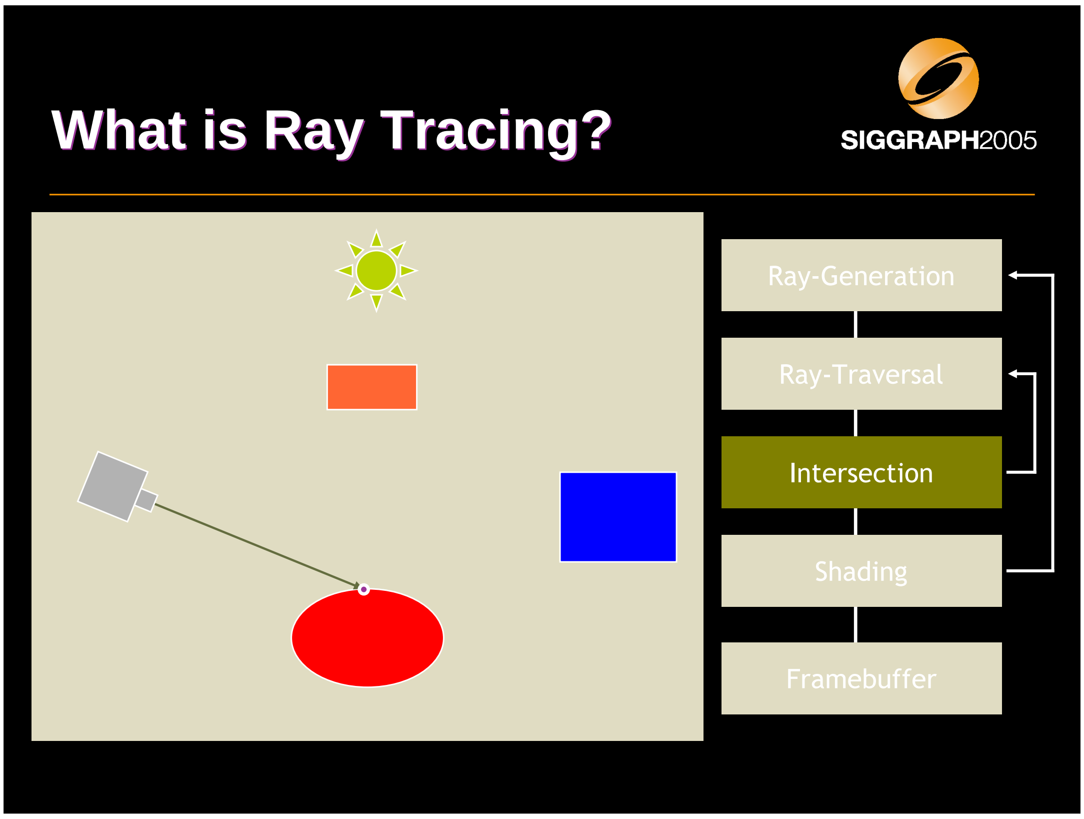
Objects and primitives are spatially sorted and pointers to them are inserted into all 3D grid cells, that they overlap. This is done in a preprocessing step. During runtime, we can quickly traverse all cells hat are pierced by a ray and only compute intersections with primitives that are in these cells and are thus close to the rays and like to get intersected.



What is Ray Tracing? Traversal

SIGGRAPH2005

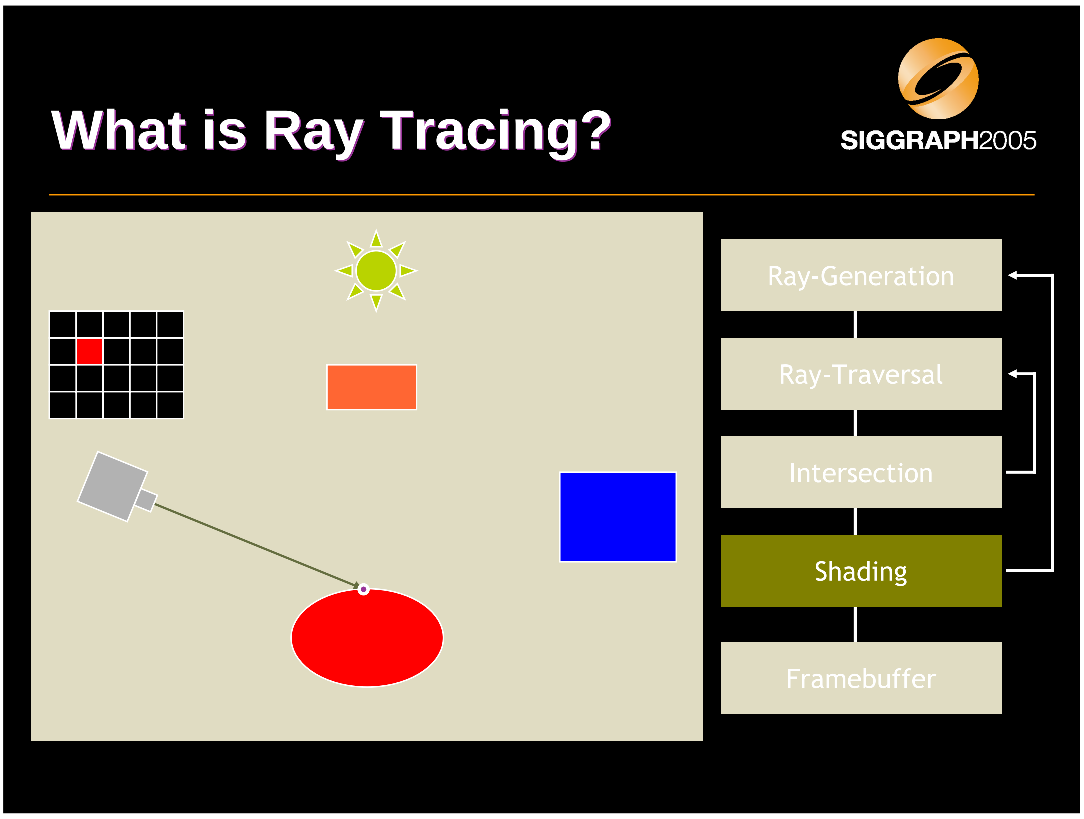
Traversal is done is a simple algorithm that will be explained in another part of the course.



What is Ray Tracing?

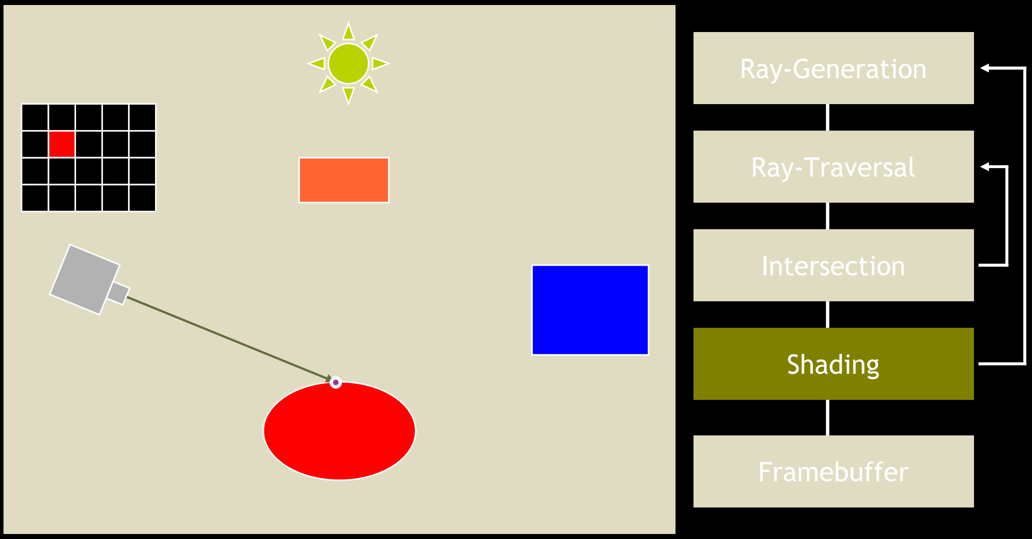
SIGGRAPH2005

When possible objects are located the intersection stage computed the exact intersection of the ray with the geometric primitive (e.g. triangle). It not hit point is found with any of the objects in the cell, we go back to the traversal stage and continue traversal.



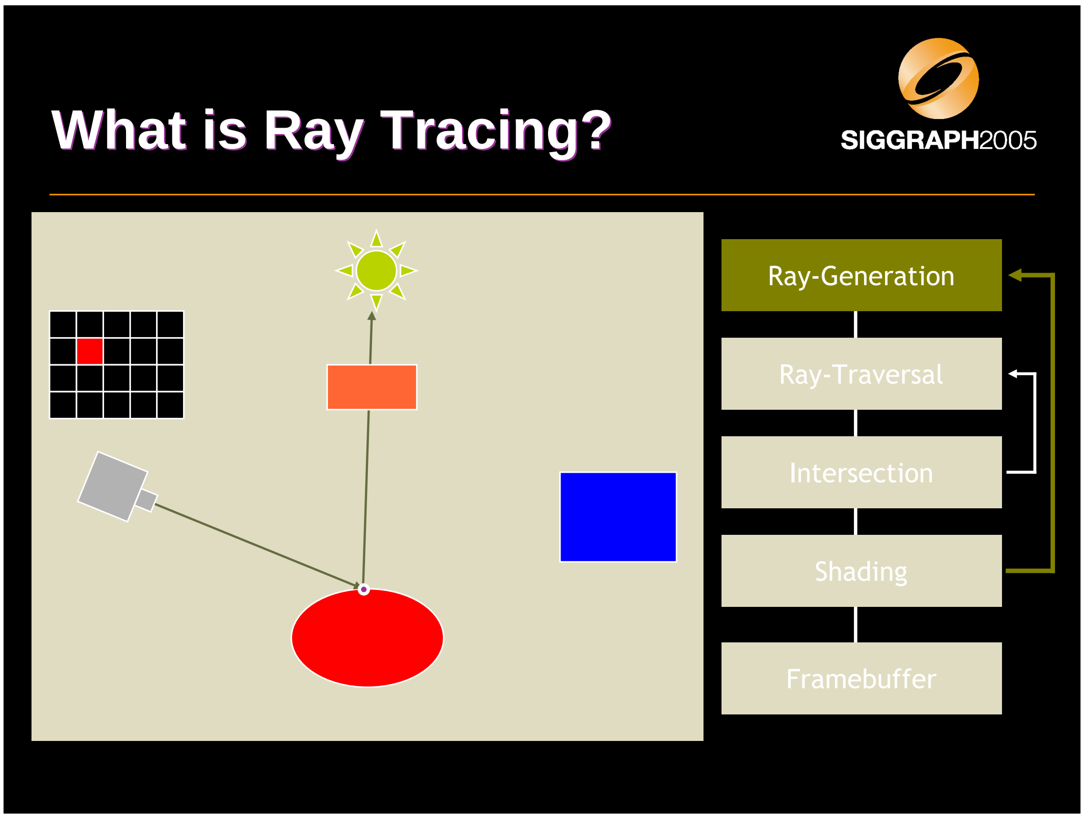
What is Ray Tracing?

SIGGRAPH2005



Once a hit point has been found it is forwarded to the shading stage, which job it is to compute the color of the returned light, which can then be used to color the corresponding pixel. In order to know how much light is reflected from the intersected location towards the camera, we must first know how much light arrives at this location.

We can do this by shooting some more rays.

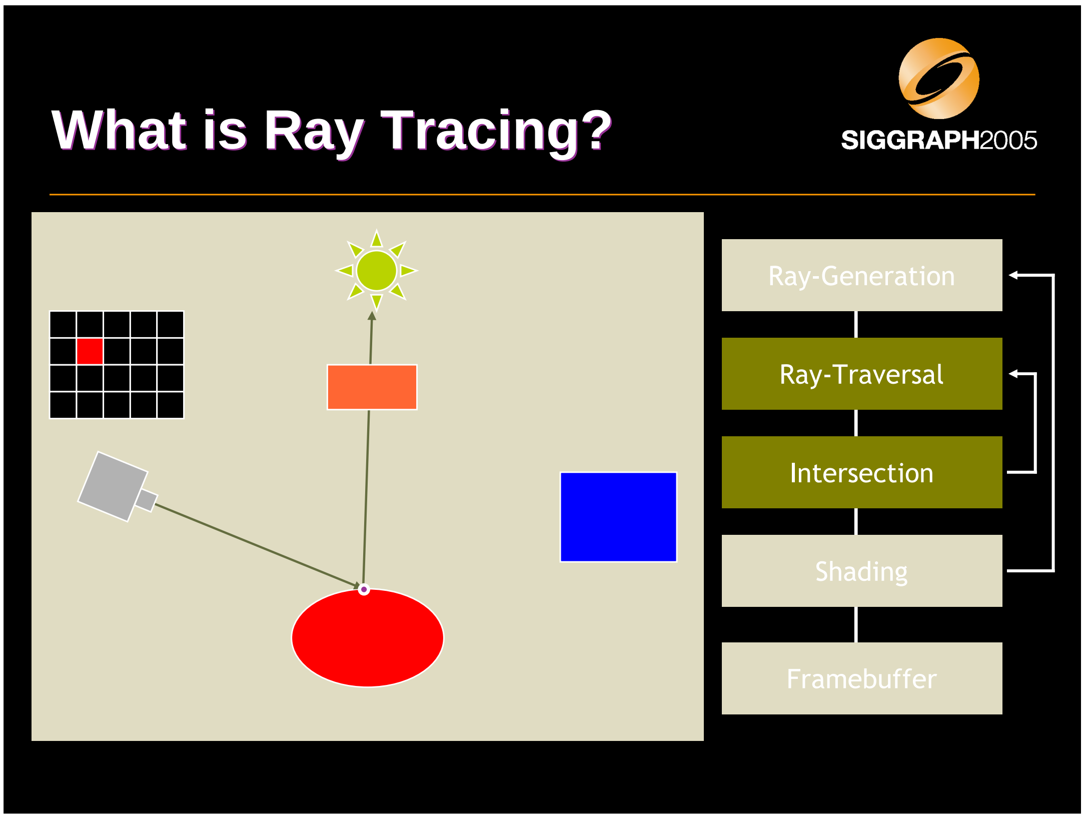


What is Ray Tracing?

SIGGRAPH2005

For example, we can send a “shadow ray” to all light sources in order to find out if there is a free path between the point and the light. This is the case if there is no intersection with any object from ray segment between the hit point and the light source. This elegantly, accurately, and efficiently solves the shadow computation problem in graphics once and for all.

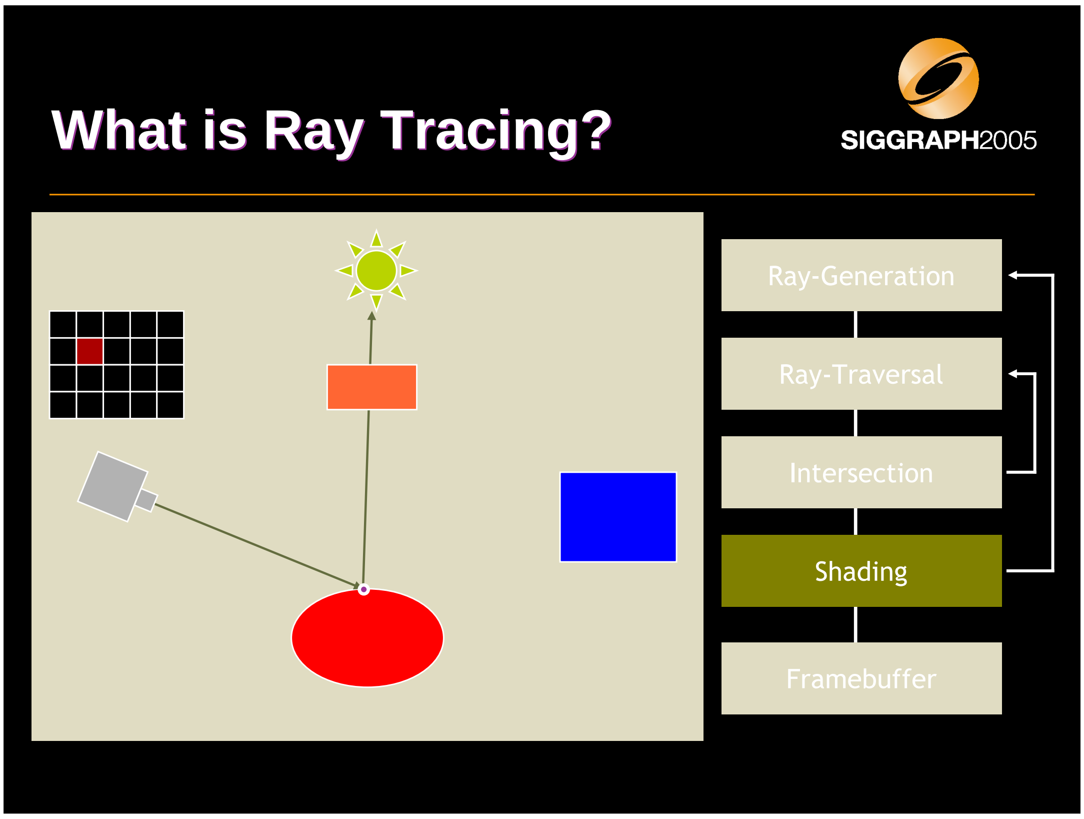
The shadow ray is fed into the pipeline just as a normal ray but some optimizations can be done because we do not need to find the **first** intersection because **any intersection** would be fine.



What is Ray Tracing?

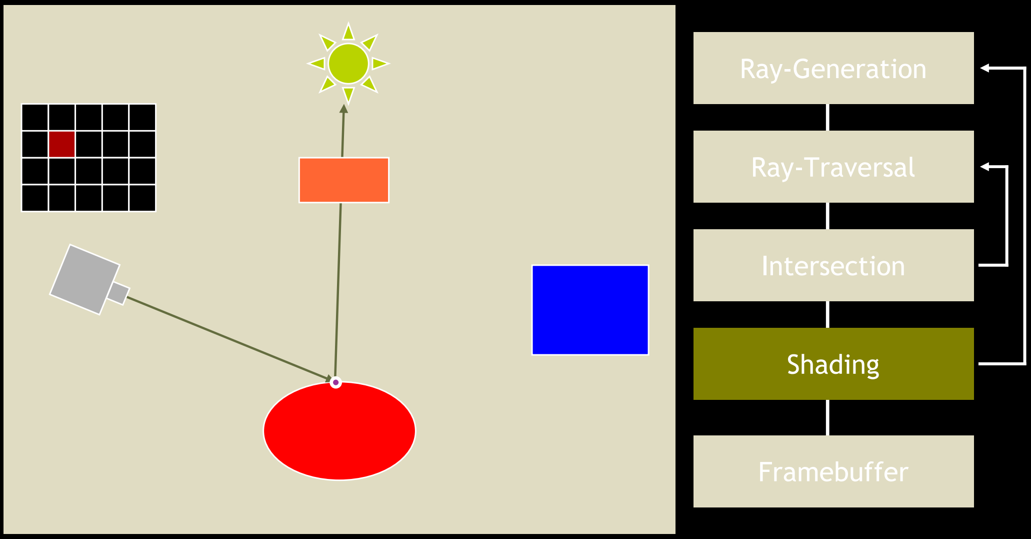
SIGGRAPH2005

The show rays are then traversed and intersected as normal, but no shading computations need to be done for them.

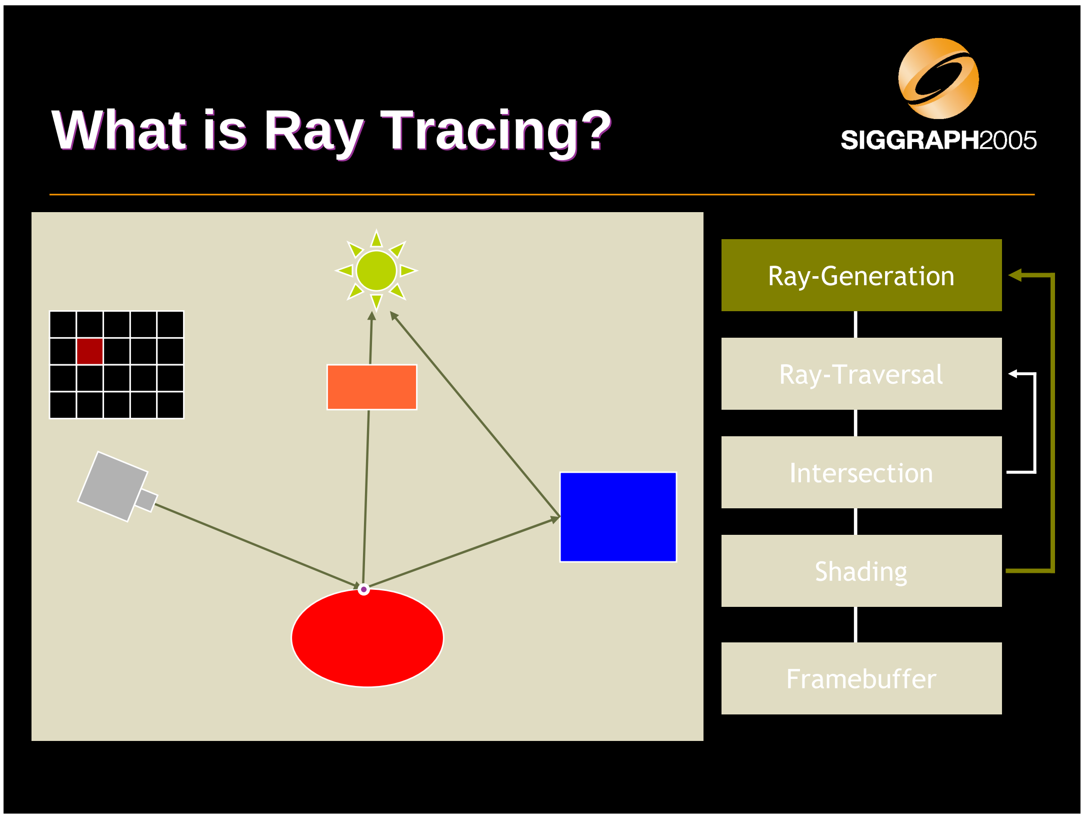


What is Ray Tracing?

SIGGRAPH2005

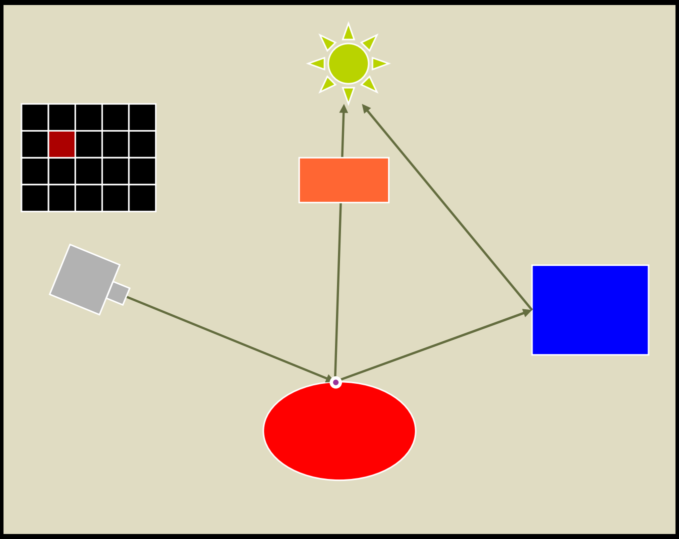


When the original shader get the information from the shadow ray it can adapt its shading results accordingly. Note that this operation requires full recursion in the shader, as we must wait until the results of tracing the shadow ray returns.

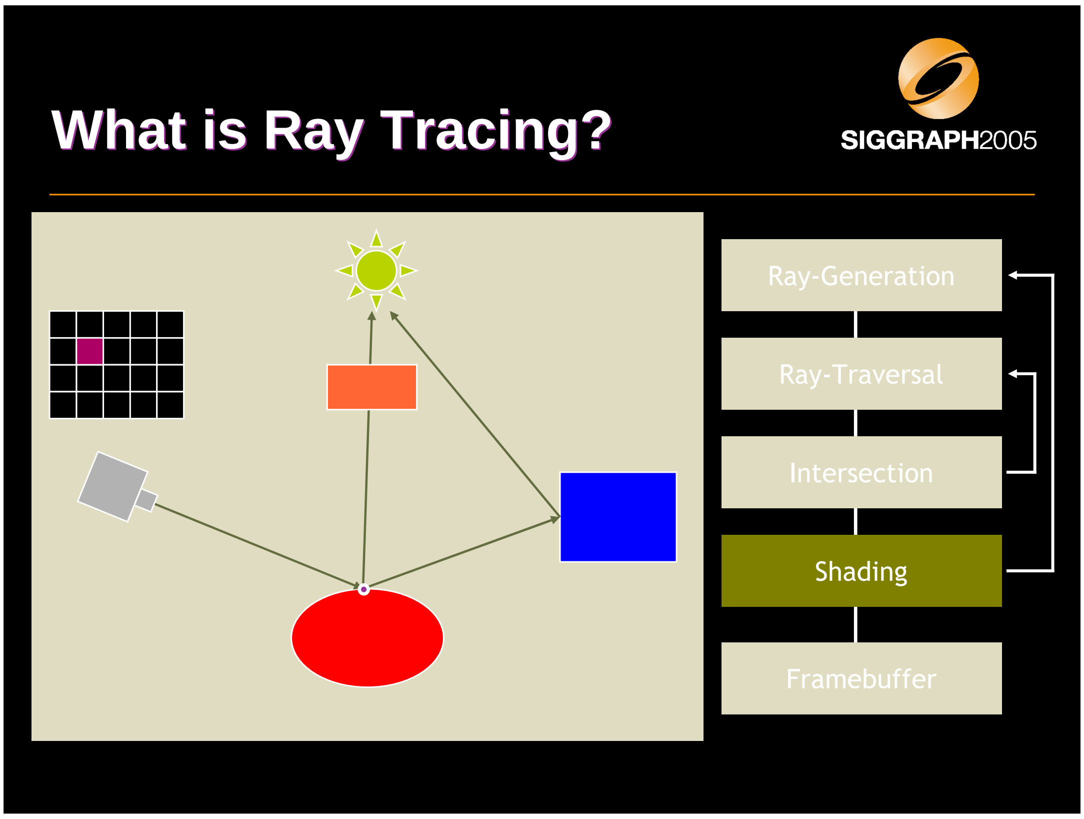


What is Ray Tracing?

SIGGRAPH2005

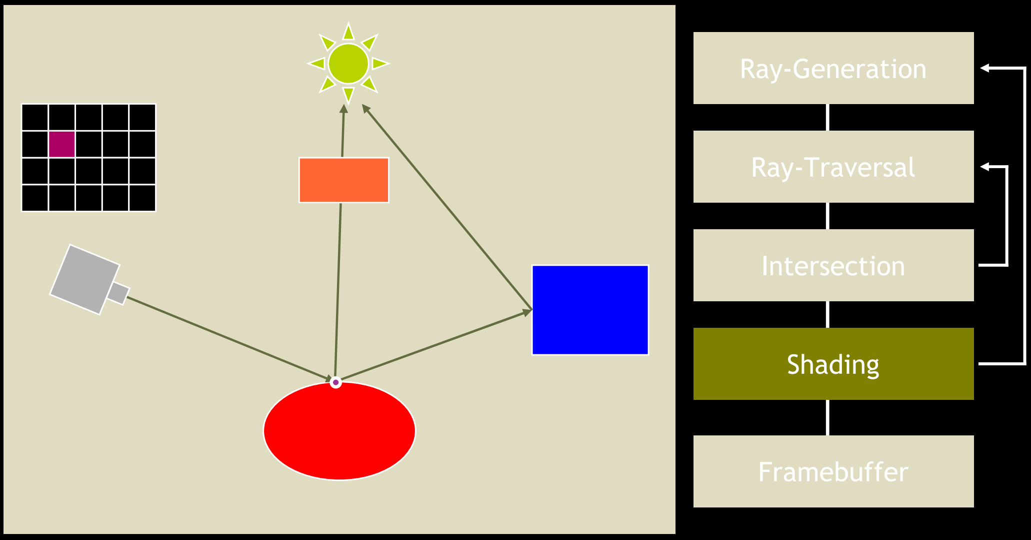


Similarly the shader may trace additional rays for querying the incoming illumination from other directions. For refraction or reflection, new rays are send to find out how much light arrived from that particular directions. This is a full recursive ray tracing procedure as the new hit point may in turns start new rays (which a shadow ray would not do).

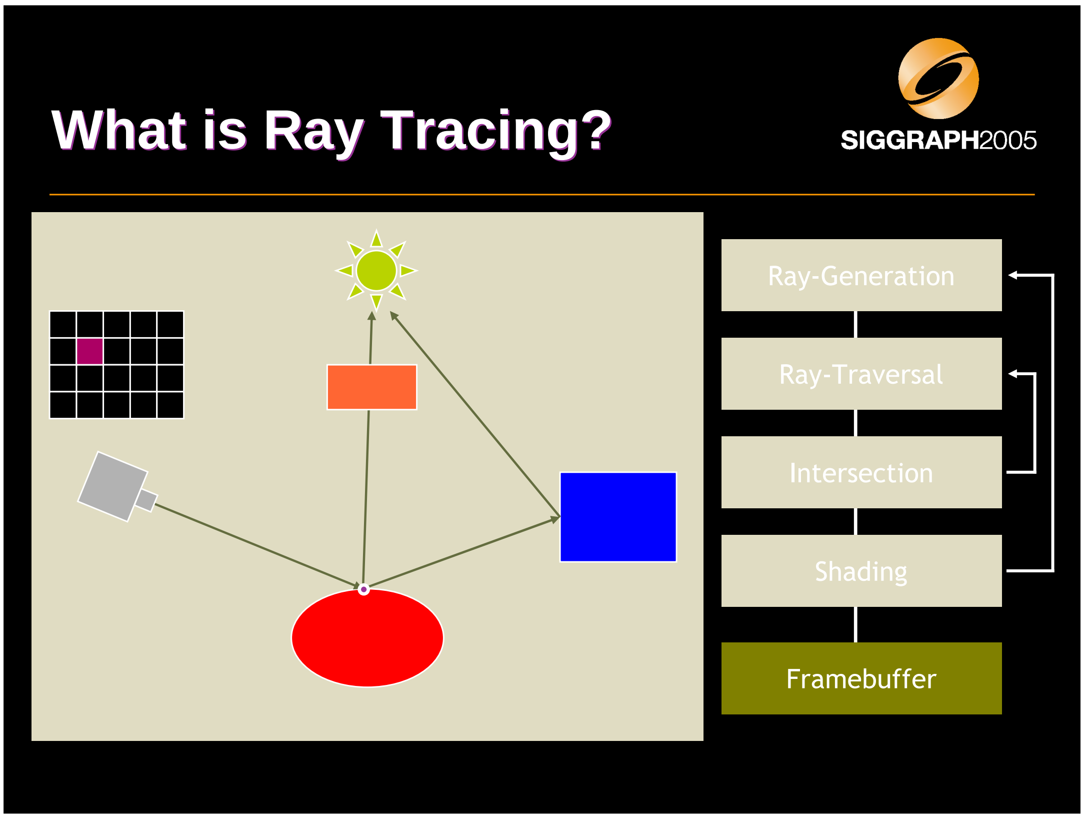


What is Ray Tracing?

SIGGRAPH2005



A fraction of the light from these additional rays is then added to the pixels color based on the reflection propertied of the primary hit point …



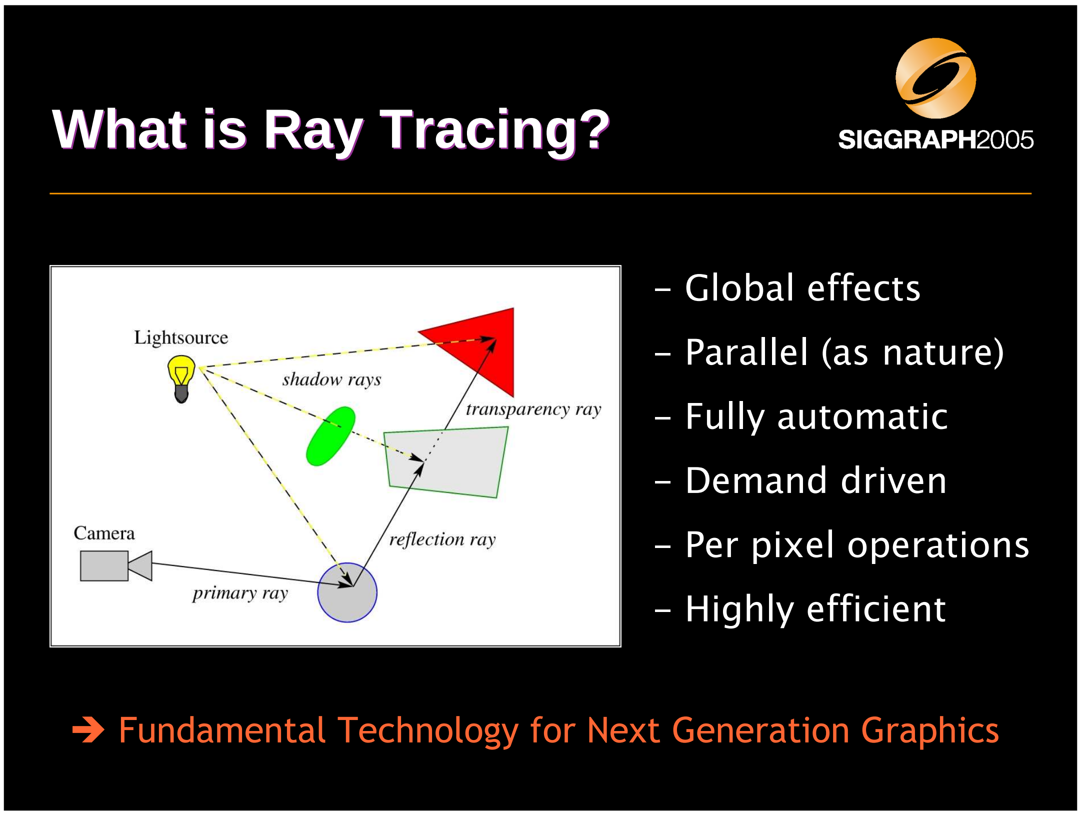
What is Ray Tracing?

SIGGRAPH2005

… and finally the results of the rays tracing process is a single color that can be directly written to the frame buffer.

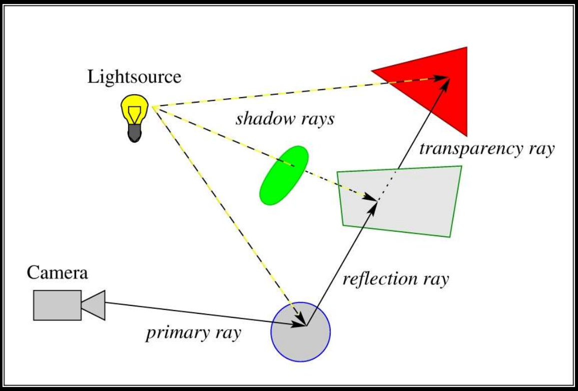
No z-buffer, accumulation buffer, stencil buffer, alpha buffer are needed as these computations are performed by the surface and other shaders or are not even needed with ray tracing at all.

This property of ray tracing to only write the final result to memory is one of its major strengths and leads to significantly reduced memory bandwidth.



What is Ray Tracing?

SIGGRAPH2005



- Global effects
- Parallel (as nature)
- Fully automatic
- Demand driven
- Per pixel operations
- Highly efficient

➔ Fundamental Technology for Next Generation Graphics

This slide summarized the most important reasons for using ray tracing:

-IT supports computing global effects because global information can be queried from a scene by tracing rays into its environment.

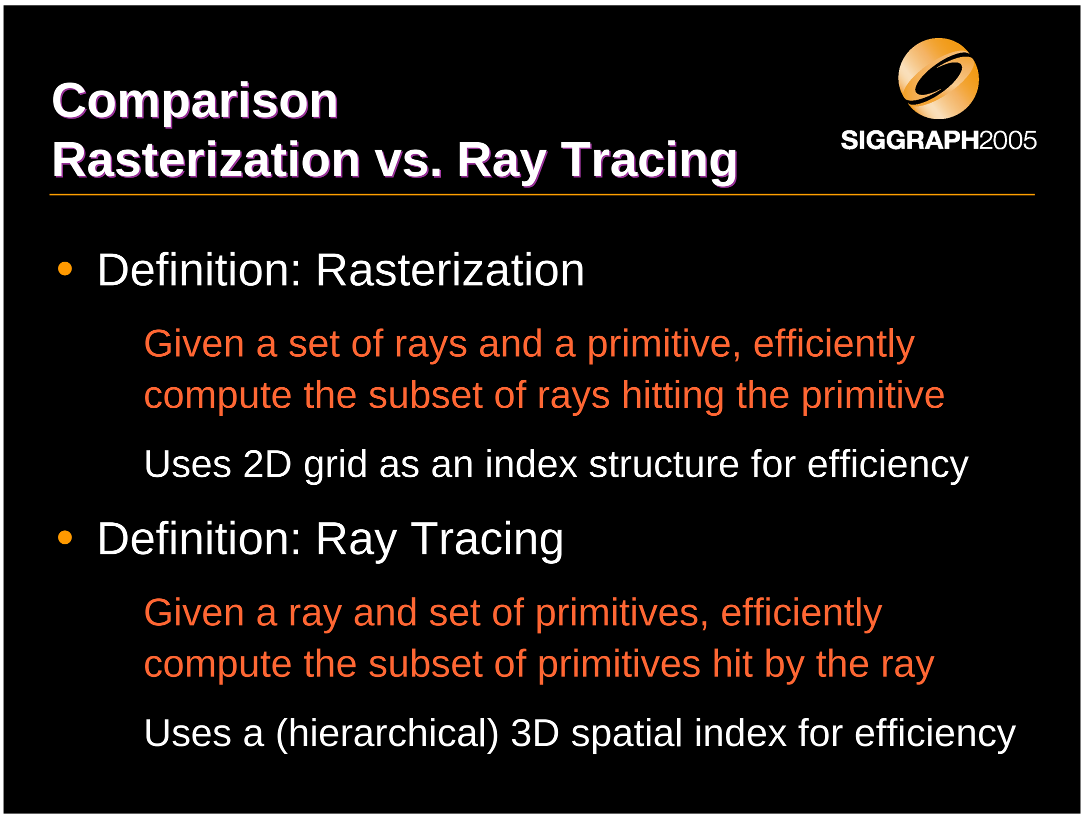
- Each primary ray is completely independent from all other primary rays, which would ultimate make it possible to assign one processor per primary ray. Ray tracing has also been called “embarrassingly parallel”.

- Ray tracing can deal with “declarative scene descriptions” that specify how a scene should look like, without specifying how this effect should be achieved. This includes full orthogonal descriptions of the geometry, its appearance (surface shaders), the camera, as well as the lighting environment and any light sources. This scene description can then be rendered fully automatically without the help of the creating application.

- Ray tracing is demand driven, meaning it only ever accesses something if that something is hit by a ray. This means that there might be gigabytes of stuff hidden behind a wall, yet a ray tracer might not even load it into memory.

- All operations are performed per pixel, including occlusion culling, interpolations, illumination etc. This leads to a high image quality.

- In summary of the above points ray tracing is highly efficient and in many cases more efficient than rasterization. For instance, it does not need to build a complete reflection map but simply computes the reflection where relevant.



Comparison
Rasterization vs. Ray Tracing

SIGGRAPH2005

- Definition: Rasterization

  Given a set of rays and a primitive, efficiently compute the subset of rays hitting the primitive

  Uses 2D grid as an index structure for efficiency

- Definition: Ray Tracing

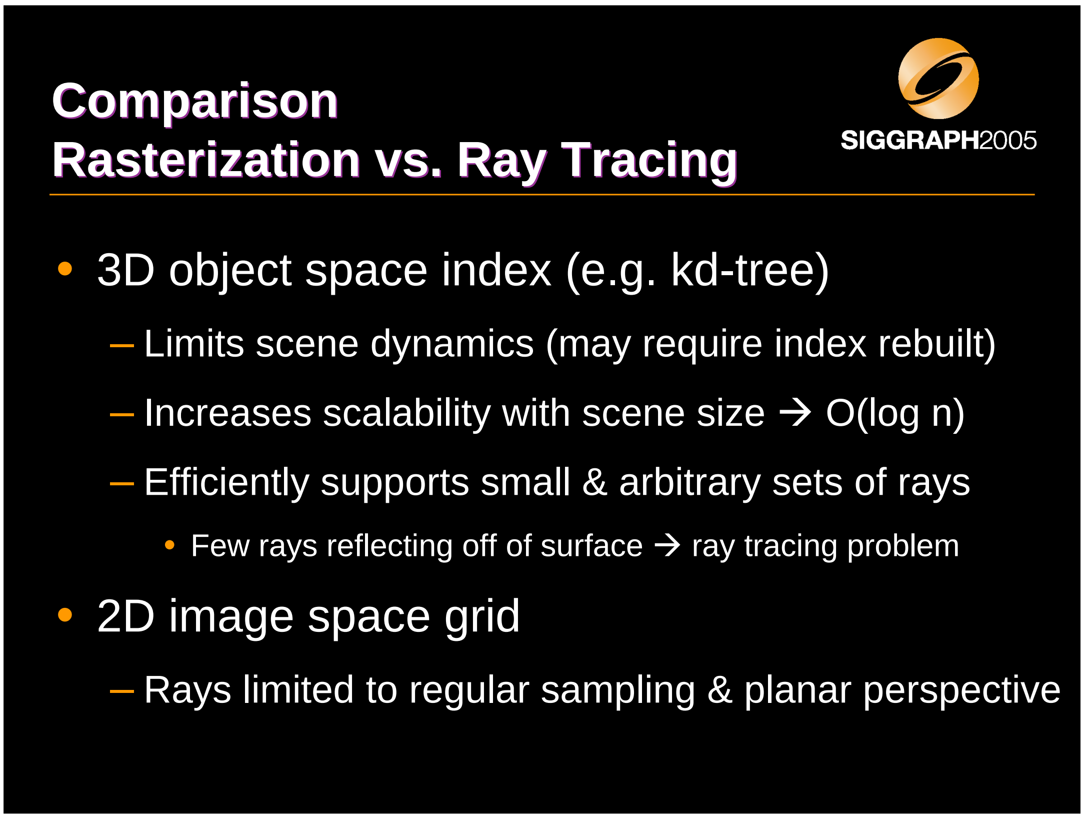
  Given a ray and set of primitives, efficiently compute the subset of primitives hit by the ray

  Uses a (hierarchical) 3D spatial index for efficiency

The two definitions show that the two algorithms are quite related but start at different end of the spectrum. Rasterization only uses a 2D grid as an index structure in image space. This limits the set of rays to those, that start at a single point and go through a regular set of sample points on a plane. This is a severe limitation already. No 3D index structure in object space is supported, even though this can be added by the application. This, however, means that it cannot be supported in hardware and there is always a communication overhead.

In contrast ray tracing is flexible in the number and the set of traced rays but needs a hierarchical spatial index for efficient computations. Ray tracing does not need to look at all scene objects but only deals with those that are visible. It uses an efficient 3D spatial index structure to quickly find any primitive that may be hit by a ray. A hierarchical index structure leads to logarithmic scalability in terms of scene size, which is a significant advantage over rasterization. These spatial index structure can also be reused for other purposes such as collision detection and many others. However, this index structure must be built to great detail, which poses challenges for dynamic and interactive scenes.

Because of the spatial 3D index, ray tracing can efficiently answer queries for individual or small groups of rays, which is handy for tracing just the necessary rays for a small reflective object or such.

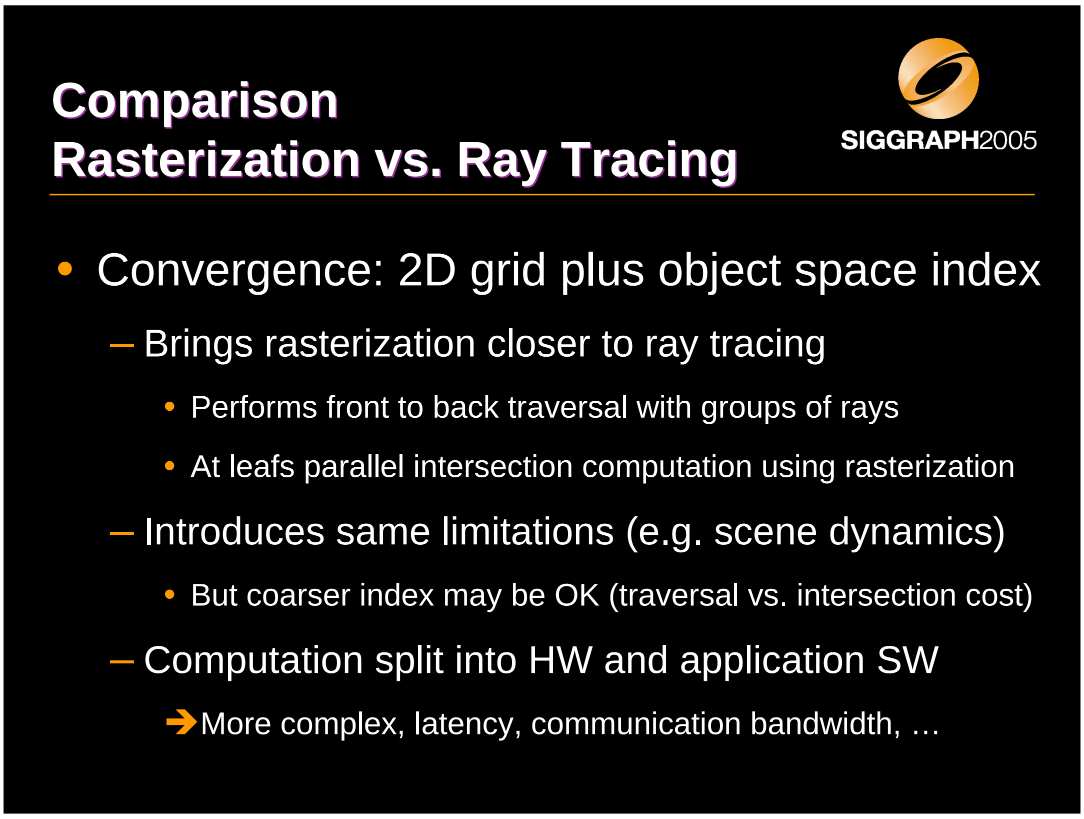


Comparison Rasterization vs. Ray Tracing

SIGGRAPH2005

- 3D object space index (e.g. kd-tree)
  - Limits scene dynamics (may require index rebuilt)
  - Increases scalability with scene size → O(log n)
  - Efficiently supports small & arbitrary sets of rays
    - Few rays reflecting off of surface → ray tracing problem
- 2D image space grid
  - Rays limited to regular sampling & planar perspective

See previous slide.



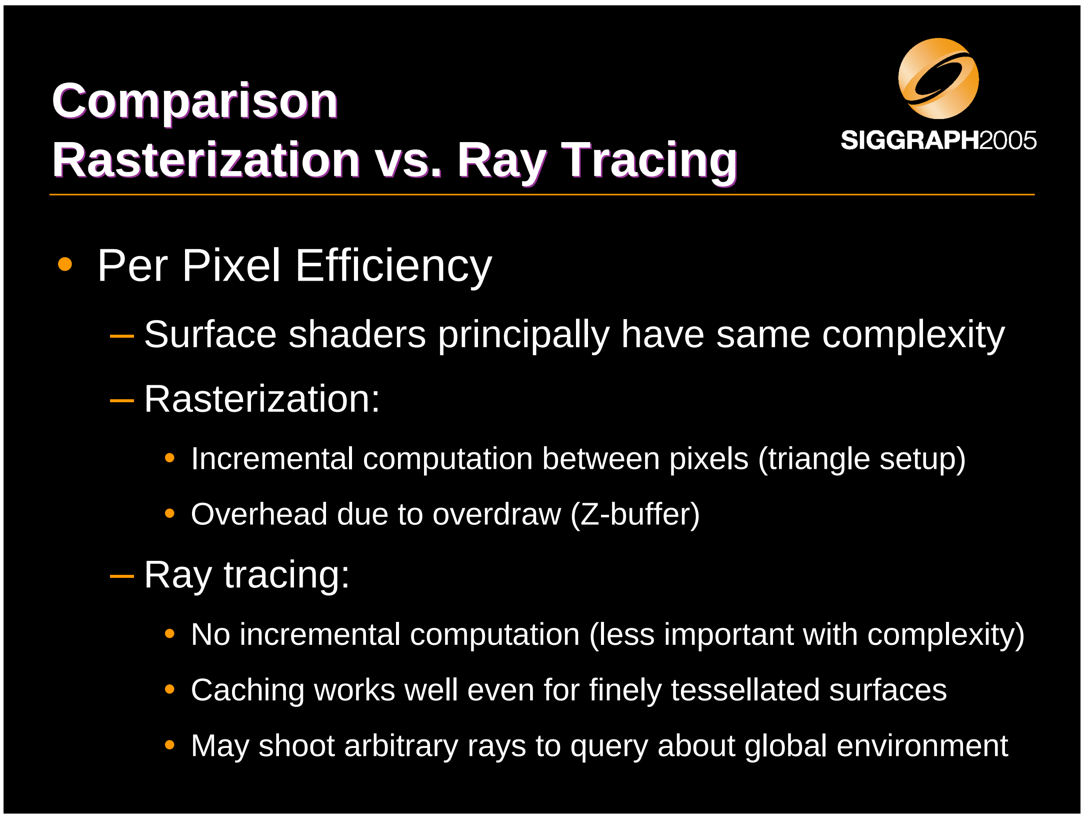
Comparison
Rasterization vs. Ray Tracing

SIGGRAPH2005

- Convergence: 2D grid plus object space index
  - Brings rasterization closer to ray tracing
    - Performs front to back traversal with groups of rays
    - At leafs parallel intersection computation using rasterization
  - Introduces same limitations (e.g. scene dynamics)
    - But coarser index may be OK (traversal vs. intersection cost)
  - Computation split into HW and application SW
    - ➔More complex, latency, communication bandwidth, …

Object space 3D spatial indices can be used with rasterization. It brings rasterization close to being ray tracing as it then performs a front to back traversal operation (for larger packets of rays) and uses the rasterization engine for doing the ray triangle intersection test.

However, this approach imposes the same limitations on rasterization such as the complexity to support dynamic scenes. It also splits the computations between the HW and the application software, which adds overhead and complications.



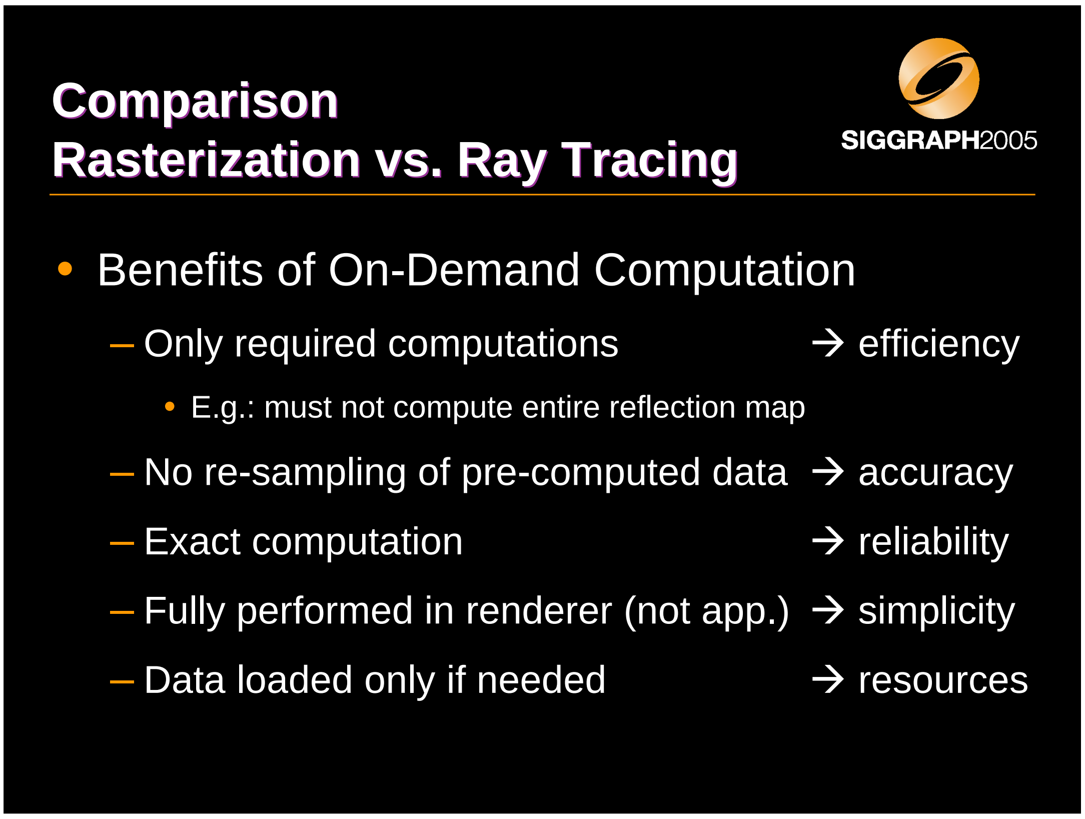
Comparison
Rasterization vs. Ray Tracing

SIGGRAPH2005

- Per Pixel Efficiency
  - Surface shaders principally have same complexity
  - Rasterization:
    - Incremental computation between pixels (triangle setup)
    - Overhead due to overdraw (Z-buffer)
  - Ray tracing:
    - No incremental computation (less important with complexity)
    - Caching works well even for finely tessellated surfaces
    - May shoot arbitrary rays to query about global environment

Shaders in GPUs and RPUs are fundamentally similar and have the same complexity for their basic operations. The main difference is that ray tracing can just shoot rays to query about global information in a scene (reflection, indirect lighting, …), where rasterization must fall back to inefficient multi-pass methods.

Rasterization has the option of doing incremental operations between pixels of the same triangle, but looses is the scene is too finely tesselated where this advantage can turn into a disadvantage. Ray tracing must re-compute all properties of the hit geometry for every ray but can take advantage of caching and SIMD computations to optimize for this case (see HW section).



Comparison
Rasterization vs. Ray Tracing

SIGGRAPH2005

- Benefits of On-Demand Computation
  - Only required computations → efficiency
    - E.g.: must not compute entire reflection map
  - No re-sampling of pre-computed data → accuracy
  - Exact computation → reliability
  - Fully performed in renderer (not app.) → simplicity
  - Data loaded only if needed → resources

The on-demand computation offers many benefits:

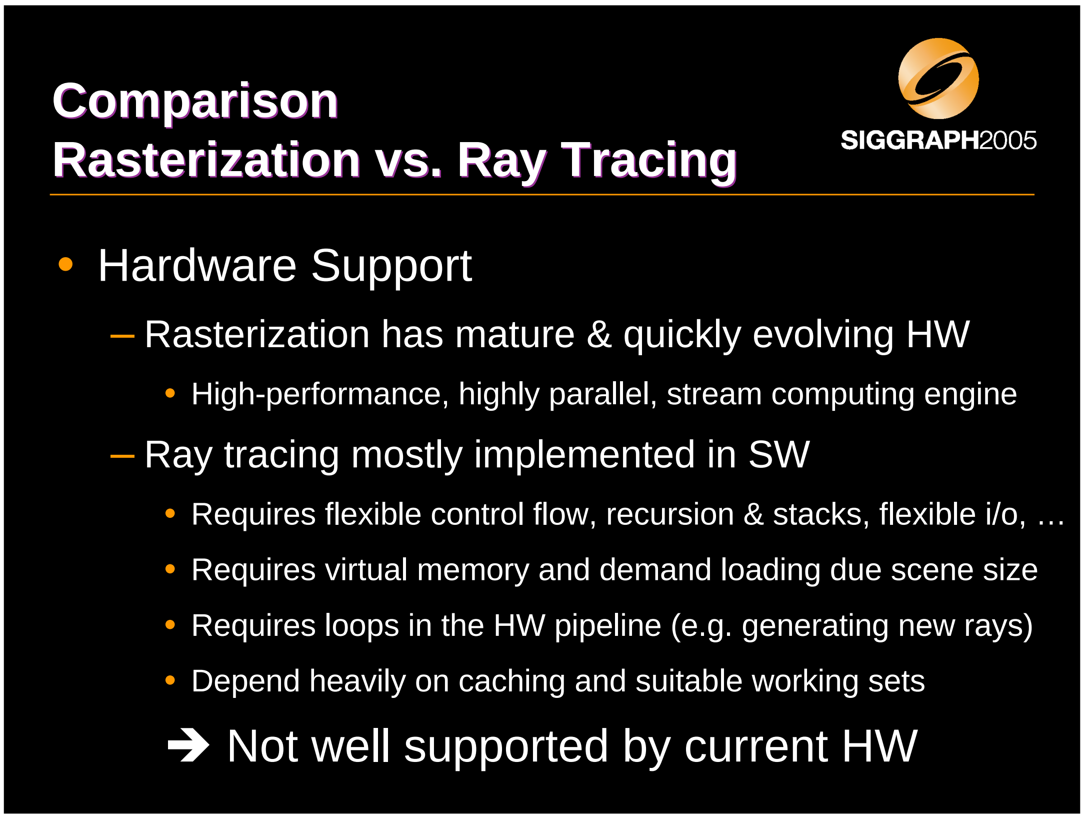
-It only computes information that is known to contribute to the image, which leads to increased efficiency. It also offers tighter control over the computer work load.

- Because data is computed on-the-fly when it is needed, ray tracing rarely stores temporary results in memory. An example are shadow: It traces the rays as needed instead of storing a discretely sampled representation all possible rays, which must be re-sampled when queried. This re-sampling can significantly reduce the accuracy and lead to artifacts.

- Because the computations are performed accurately and physically correct for exactly the necessary rays, the results have less artifacts and are much more reliable.

- because ray tracing supports a full declarative scene description, the entire rendering computation can be implemented in hardware or at least with in a separate rendering engine. The application is only needed to update the scene between frames.

- Due to the on-demand approach necessary data is only loaded when needed, which can greatly reduce the working set and thus the resources needed for a given computation.



Comparison
Rasterization vs. Ray Tracing

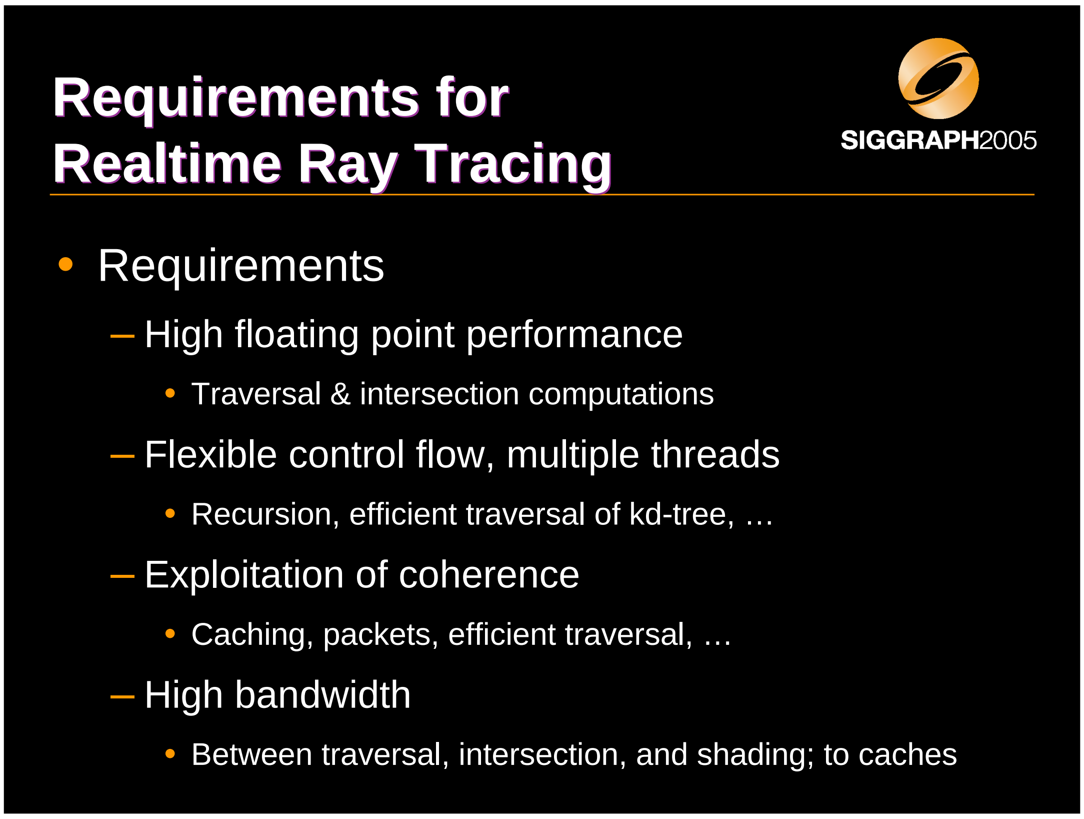
SIGGRAPH2005

- Hardware Support
  - Rasterization has mature & quickly evolving HW
    - High-performance, highly parallel, stream computing engine
  - Ray tracing mostly implemented in SW
    - Requires flexible control flow, recursion & stacks, flexible i/o, …
    - Requires virtual memory and demand loading due scene size
    - Requires loops in the HW pipeline (e.g. generating new rays)
    - Depend heavily on caching and suitable working sets

  ➔ Not well supported by current HW

Ray tracing is still lacking significantly with respect to hardware support. While rasterization has seen more than twenty years of intense development, hardware for ray tracing is only just appearing. Nonetheless, it has been shown that ray tracing can be implemented highly efficiently in hardware (see RPU section).

However, ray tracing requires significantly more flexibility from a hardware architecture, which necessitates extensions and new approaches compared to today’s graphics architectures.



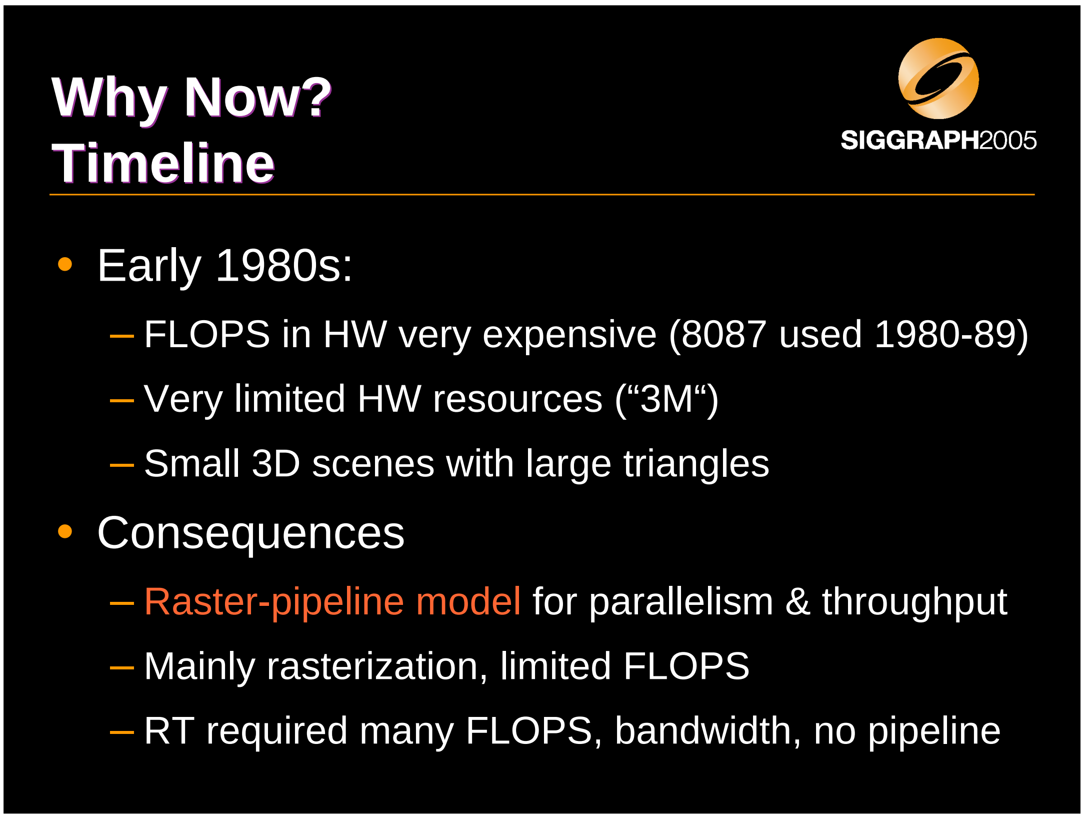
Requirements for Realtime Ray Tracing

SIGGRAPH2005

- Requirements
  - High floating point performance
    - Traversal & intersection computations
  - Flexible control flow, multiple threads
    - Recursion, efficient traversal of kd-tree, …
  - Exploitation of coherence
    - Caching, packets, efficient traversal, …
  - High bandwidth
    - Between traversal, intersection, and shading; to caches

Ray tracing can only be implemented efficiently with floating point computations. Traversal, intersection, and shading all require many FLOPS per ray. Because of the traversal of hierarchical tree structures and the more-or-less general purpose nature of shading computations, a flexible hardware support is required.

However, ray tracing inherently has a high degree of coherence that can be used to reduce the computational and memory bandwidth requirements of a naïve implementation.



Why Now? Timeline

SIGGRAPH2005

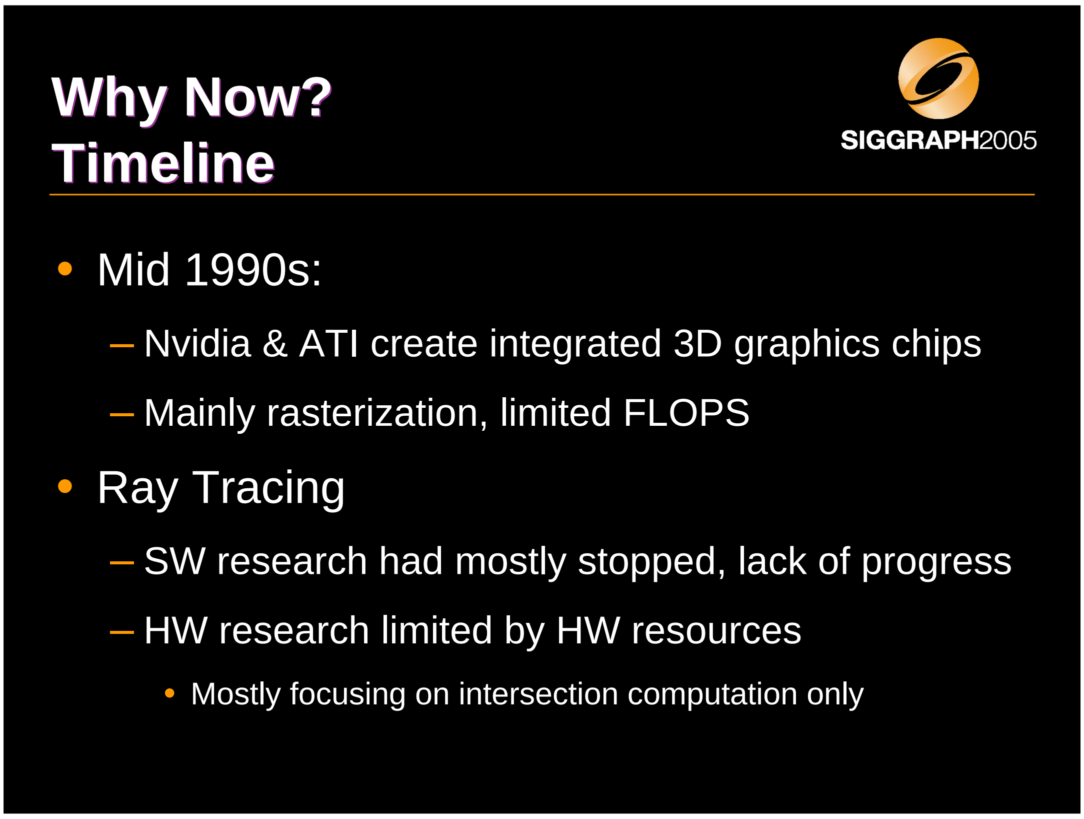
- Early 1980s:
  - FLOPS in HW very expensive (8087 used 1980-89)
  - Very limited HW resources (“3M“)
  - Small 3D scenes with large triangles
- Consequences
  - Raster-pipeline model for parallelism & throughput
  - Mainly rasterization, limited FLOPS
  - RT required many FLOPS, bandwidth, no pipeline

An interesting question to ask is: Why has realtime ray tracing not been done before?

It turns out that many researchers have repeatedly stated very early that ray tracing would eventually become faster than rasterization because of its logarithmic complexity in terms of scene size. However, these claims have not been fulfilled for more than twenty years, and research on ray tracing essentially stopped in the late 1980s / early 1990s. There had been **no research** that has explored ***WHY*** ray tracing has been so dramatically slower than rasterization despite other expectations.

Even though, there are several reasons why realtime performance could not be realized in the early days. FLOPS have been very expensive in these days, scenes usually had few large polygons, and hardware resources were very scarce. Due to its purely local computational model, rasterization is much better suited for such an environment.

Since then a complete generation of researchers, developers, and users grew up exclusive in a rasterization based world. Now everyone just “knew” that ray tracing is slow and cannot be implemented in hardware.

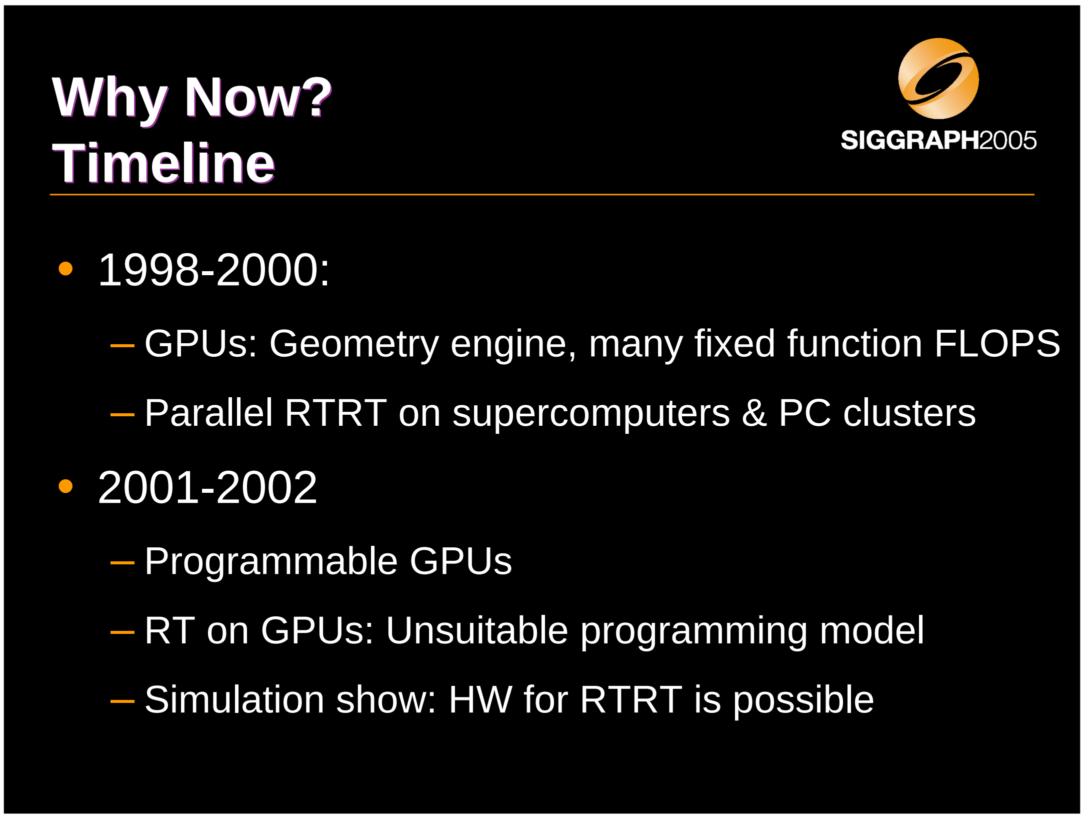


Why Now?
Timeline

SIGGRAPH2005

- Mid 1990s:
  - Nvidia & ATI create integrated 3D graphics chips
  - Mainly rasterization, limited FLOPS
- Ray Tracing
  - SW research had mostly stopped, lack of progress
  - HW research limited by HW resources
    - Mostly focusing on intersection computation only

In the mid 1990s VLSI graphics chips mainly accelerated the rasterization part of OpenGL, which mainly consists of fixed point arithmetic. FLOPS were still expensive to realize in hardware. Thus the rasterization took off in the mass market while ray tracing still could not be realized on a competitive basis.



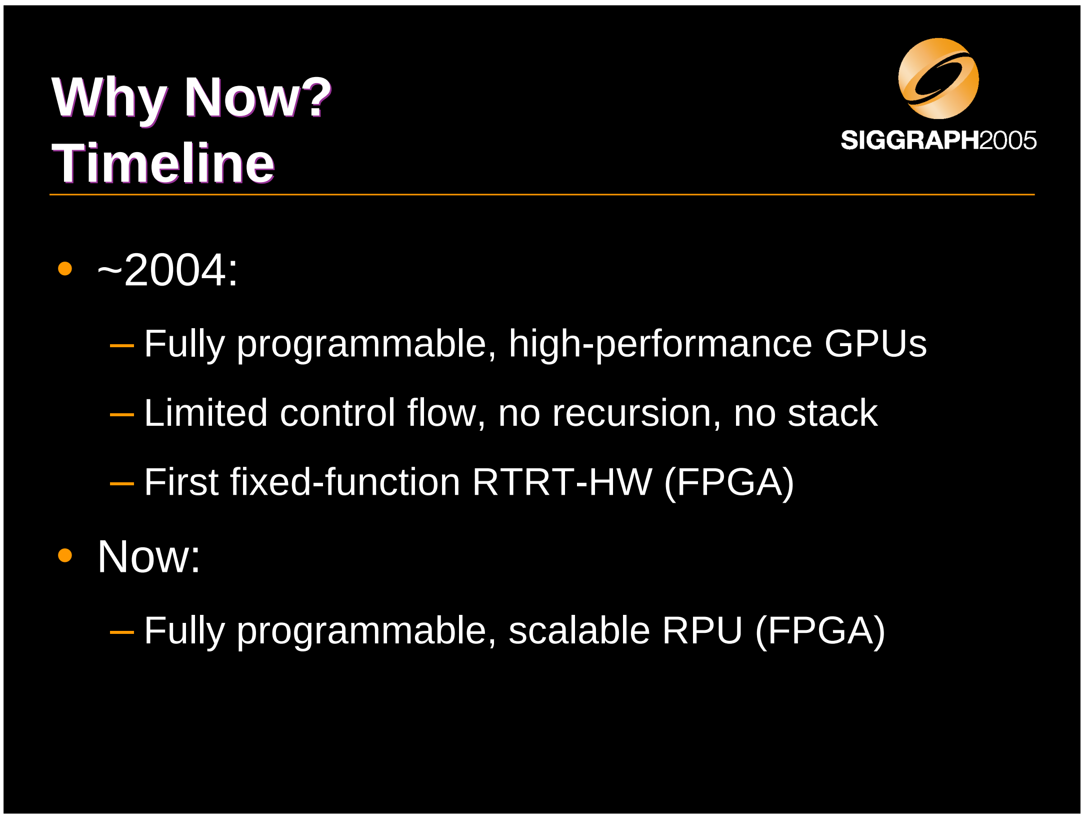
Why Now? Timeline

SIGGRAPH2005

- 1998-2000:
  - GPUs: Geometry engine, many fixed function FLOPS
  - Parallel RTRT on supercomputers & PC clusters
- 2001-2002
  - Programmable GPUs
  - RT on GPUs: Unsuitable programming model
  - Simulation show: HW for RTRT is possible

At the end of the late 1990s the hardware resources became available to perform realtime ray tracing on large supercomputers and a few years later also on clusters of PCs. FLOPS became cheap and were available in every PC through SIMD, high clock rates, long CPU pipelines, etc. However, the new architecture was not well suited for the inner loops of traditional ray tracing algorithms. Only when the algorithms were re-implemented could these new hardware features be exploited effectively.

A few years later large numbers of FLOPS also became available in programmable GPUs. However, until now their programming model is to inflexible to effectively exploit the raw performance for ray tracing.



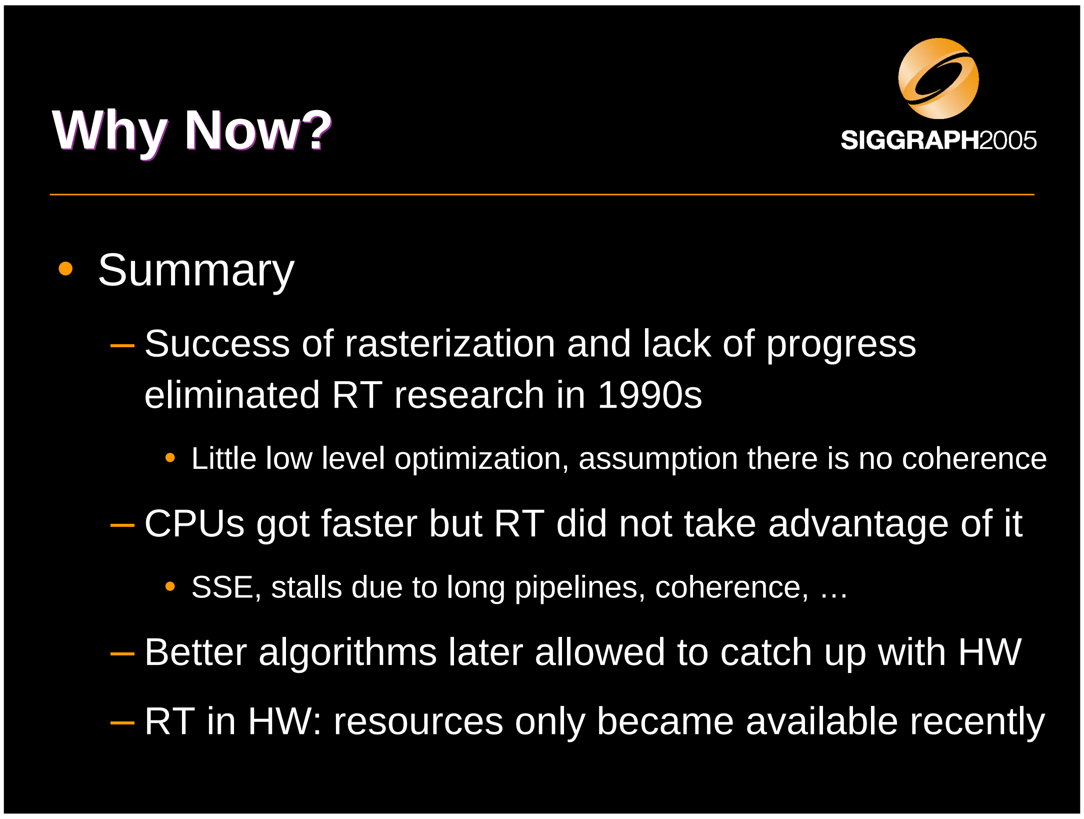
Why Now? Timeline

SIGGRAPH2005

- ~2004:
  - Fully programmable, high-performance GPUs
  - Limited control flow, no recursion, no stack
  - First fixed-function RTRT-HW (FPGA)
- Now:
  - Fully programmable, scalable RPU (FPGA)

The features and flexibility of GPUs has increased significantly since then. But it is still insufficient to implement fast ray tracing on GPUs except in toy examples. It will be interesting to see where GPUs will be moving in the next few years. The dominant stream programming model seems not well suited for ray tracing type algorithms.

However, in the mean time custom hardware has been developed that performs the entire computation highly efficiently in hardware. In 2004 a first fixed-function ray tracing chip was presented. A year later the first fully programmable RPU (ray processing unit) was presented at Siggraph. Interesting enough the latter architecture is based to large degrees on GPUs but extends then in key locations by dedicated hardware units as well as significantly increased programming flexibility.

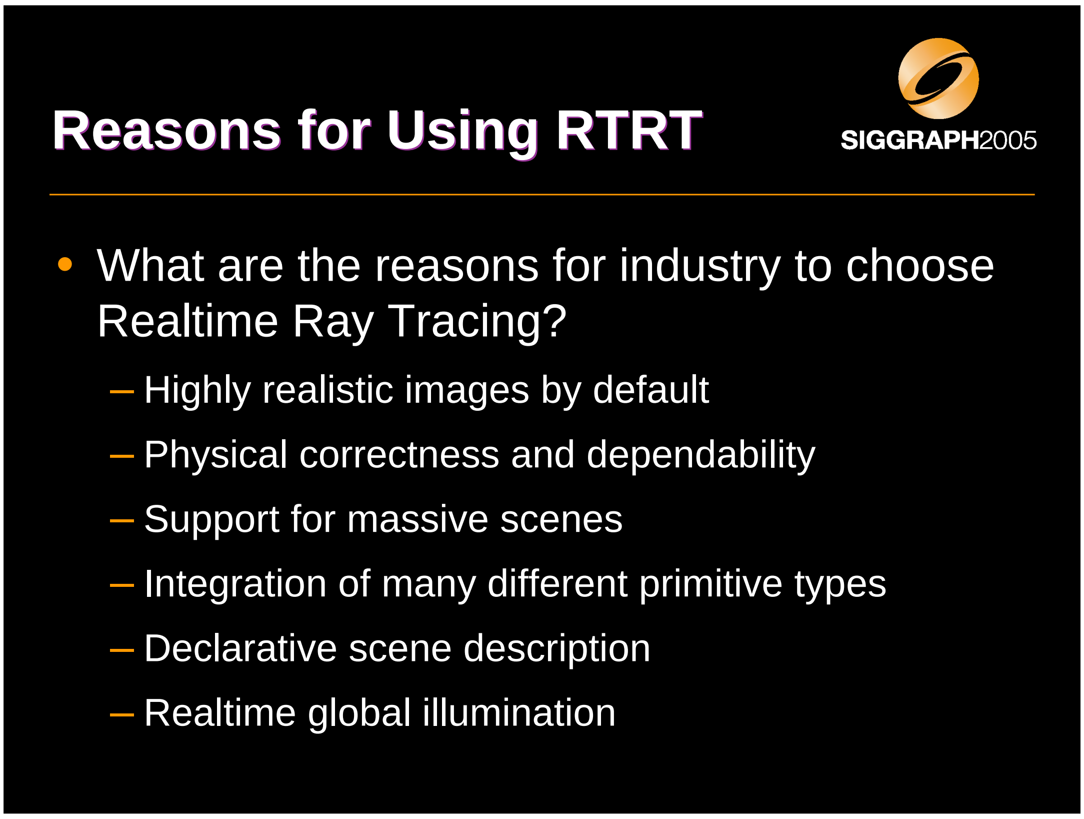


Why Now?

SIGGRAPH2005

- Summary
  - Success of rasterization and lack of progress eliminated RT research in 1990s
    - Little low level optimization, assumption there is no coherence
  - CPUs got faster but RT did not take advantage of it
    - SSE, stalls due to long pipelines, coherence, …
  - Better algorithms later allowed to catch up with HW
  - RT in HW: resources only became available recently

In summary, it becomes clear that ray tracing has a much higher fixed cost in terms of cost of hardware. This has held back ray tracing for long enough to discourage most researchers that by the time the resources became available, nobody was interested any more. However, at this point progress could be made in great leaps because of catching up with the then current hardware and software possibilities.



Reasons for Using RTRT

SIGGRAPH2005

- What are the reasons for industry to choose Realtime Ray Tracing?
  - Highly realistic images by default
  - Physical correctness and dependability
  - Support for massive scenes
  - Integration of many different primitive types
  - Declarative scene description
  - Realtime global illumination

We are providing realtime ray tracing technology to industry through our spin-off company inTrace GmbH. We have seen a significant interest from industry already three years ago. inTrace customers are now all major German car manufactures (Volkswagen, DaimlerChrysler, BMW, Audi) and Airbus with additional projects run at Skoda, Boeing, and other companies.

From this contacts we see the following main reasons for industry to be interested in this new technology. However, the reasons are weighted very differently by different departments even of the same company.

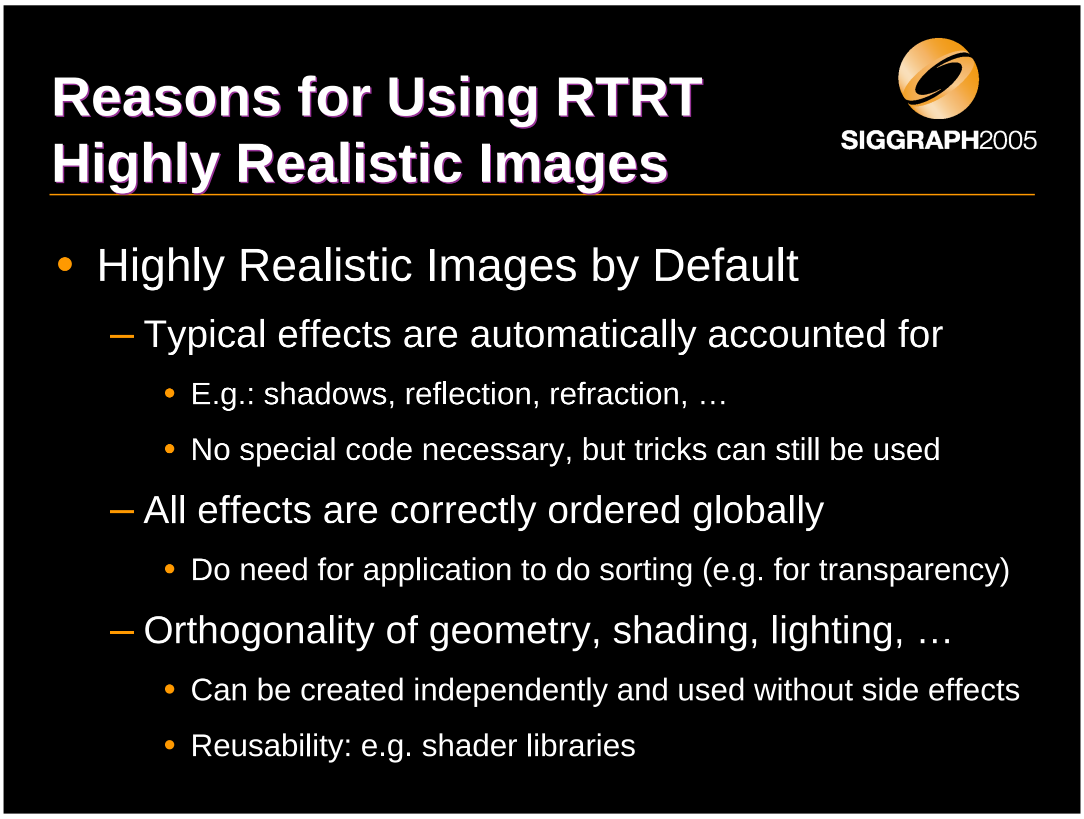
-The better image quality due to accurate shadows, reflections, and refraction even on highly complex models is a major motivation for most departments.

- Often more important is the fact that the viewer can be confident that what he sees on the screen has been computed physically correct and that he can depend of these results. Given that at design reviews major investments are made based on the visual appearance of a virtual model, this dependability if of paramount importance.

- Large CAD models usually had to be simplified significantly for achieving realtime performance even with the best rasterization technology. Due to the logarithmic scaling, companies can now work interactively with full detail models like entire cars or airplanes down to details like individual screws. This greatly simplifies their process pipeline.

- Ray tracing is also able to render spline surfaces and points directly, which has caused significant interest. Similarly, global illumination is interesting for providing the still missing level of realism.

- Finally, the ability to easily and fully automatically describe an entire highly complex scene with shaders from a predefined library, such that the visualization works on the press of a button is highly interesting to industry

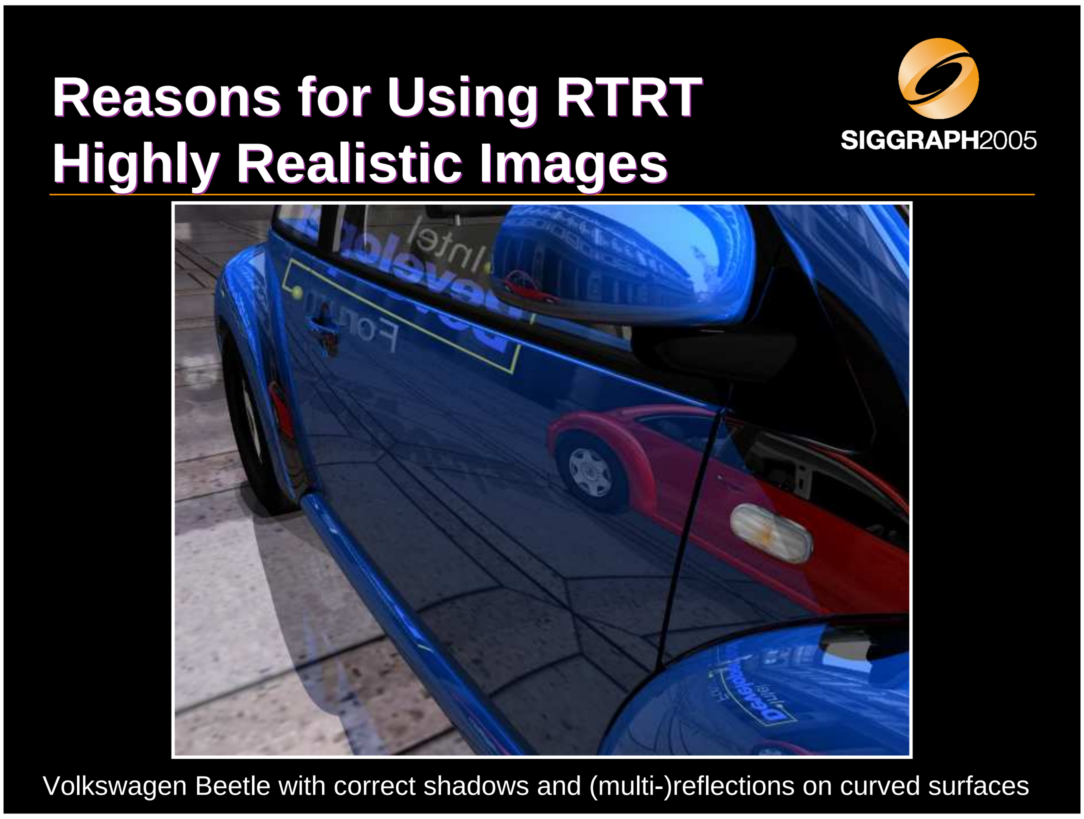


Reasons for Using RTRT Highly Realistic Images

SIGGRAPH2005

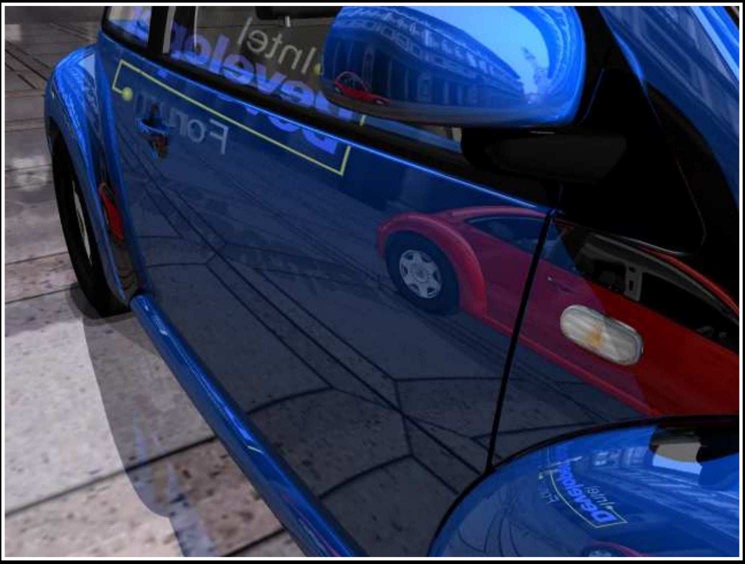
- Highly Realistic Images by Default
  - Typical effects are automatically accounted for
    - E.g.: shadows, reflection, refraction, …
    - No special code necessary, but tricks can still be used
  - All effects are correctly ordered globally
    - Do need for application to do sorting (e.g. for transparency)
  - Orthogonality of geometry, shading, lighting, …
    - Can be created independently and used without side effects
    - Reusability: e.g. shader libraries

See previous slide.



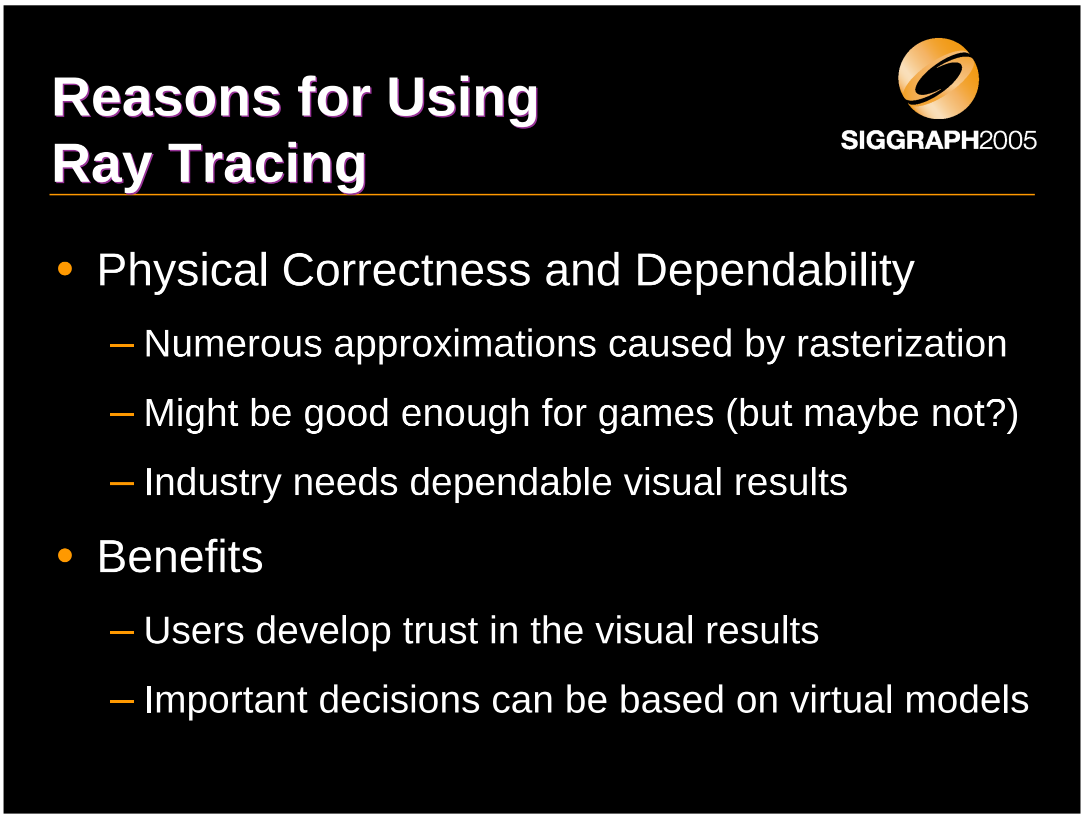
Reasons for Using RTRT Highly Realistic Images

SIGGRAPH2005



Volkswagen Beetle with correct shadows and (multi-)reflections on curved surfaces

This image shows a simple example of the complex optical effects that need to be rendered with a car model, including shadows of curved surfaces, multiple reflections, environment maps, and many more.



Reasons for Using Ray Tracing

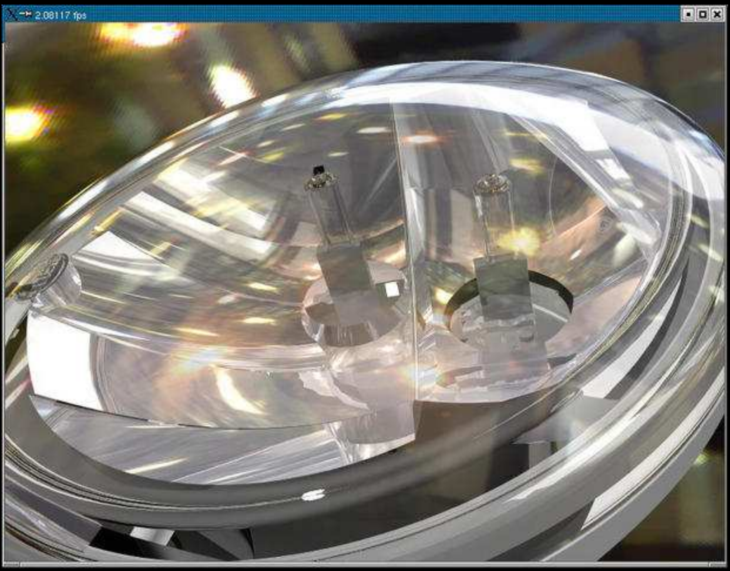
SIGGRAPH2005

- Physical Correctness and Dependability
  - Numerous approximations caused by rasterization
  - Might be good enough for games (but maybe not?)
  - Industry needs dependable visual results
- Benefits
  - Users develop trust in the visual results
  - Important decisions can be based on virtual models

See before.

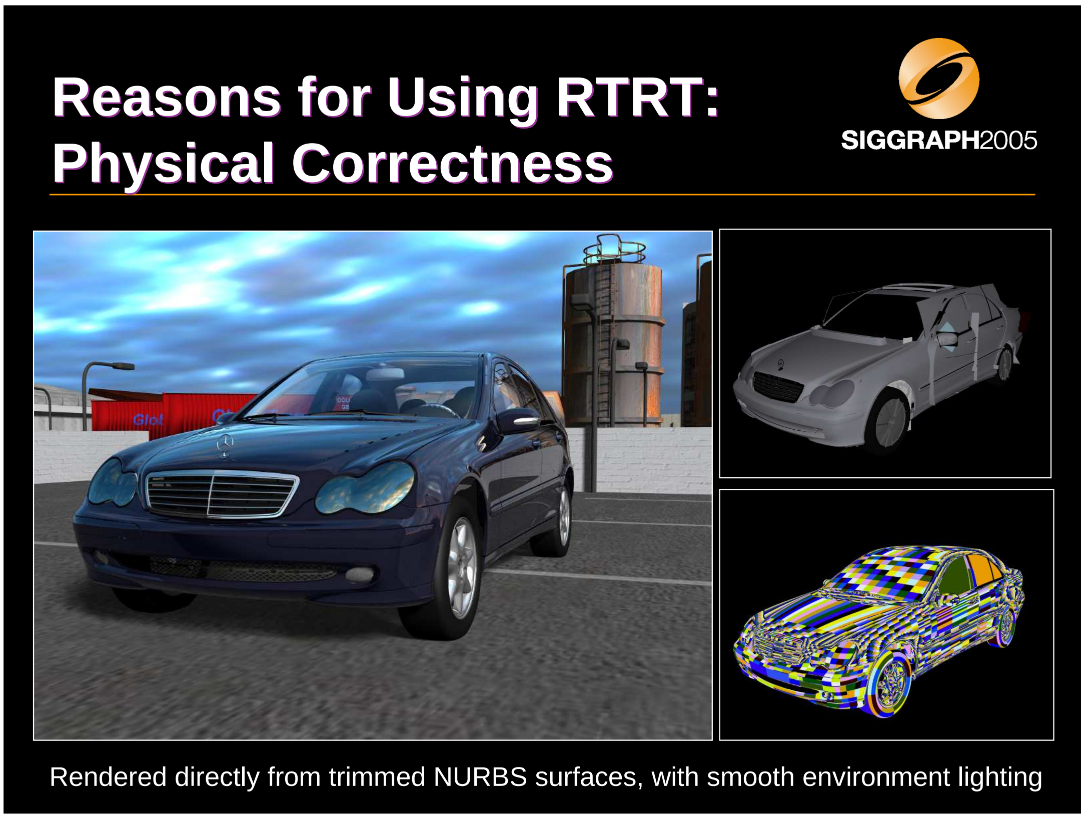
Reasons for Using RTRT: Physical Correctness

SIGGRAPH2005



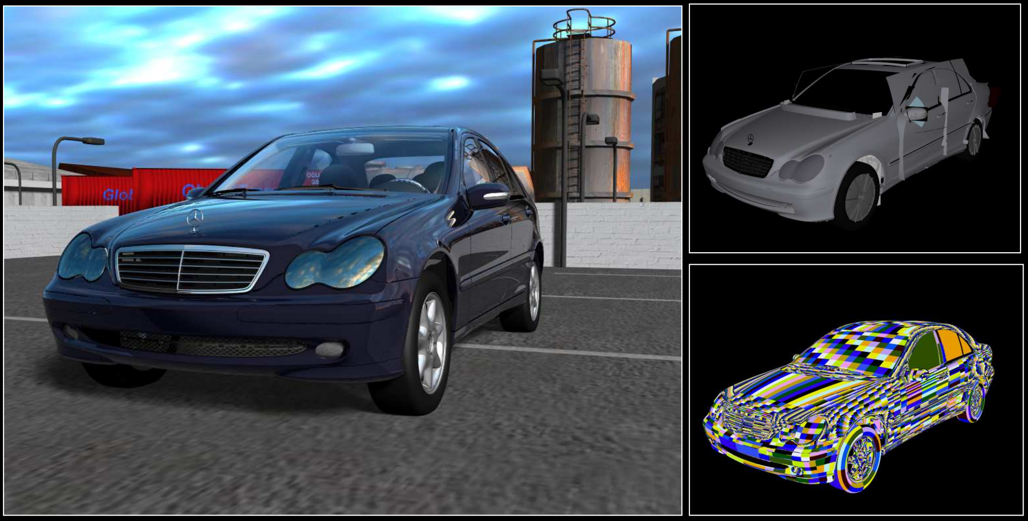
Fully ray traced car head lamp, faithful visualization requires up to 50 rays per pixel

Only ray tracing is able to render a complex image like this. Rays trees of at least depth of 10 and up to 25 and more must be traced to obtain faithful results. In total this adds up to more than 50 rays per pixel. This images runs in 5-7 fps on a small PC cluster.



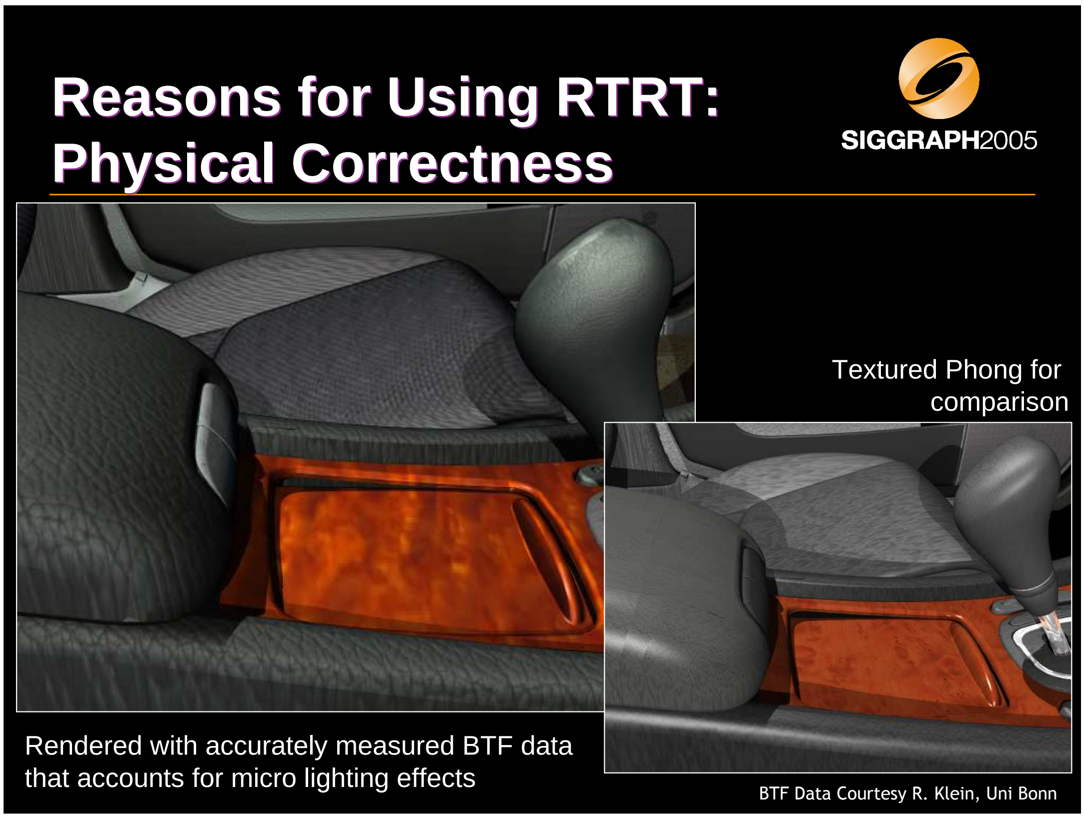
Reasons for Using RTRT: Physical Correctness

SIGGRAPH2005



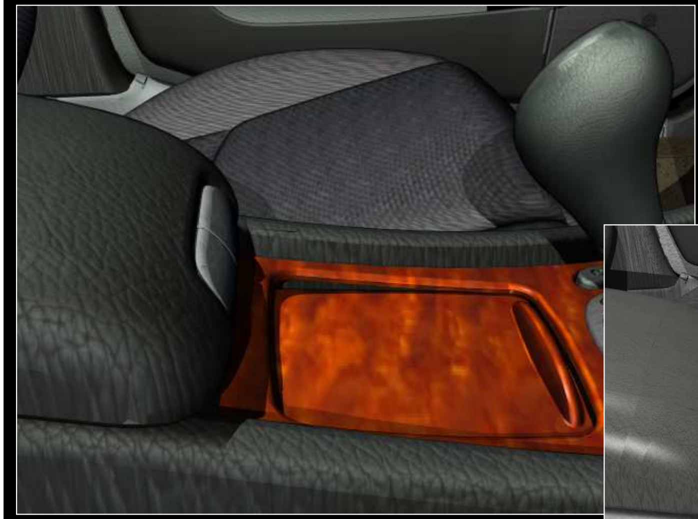
Rendered directly from trimmed NURBS surfaces, with smooth environment lighting

In a recent project an entire car was rendered directly with trimmed NURBS surfaces instead of many triangles. In addition, we used a highly accurate car paint shader and global illumination from a sky dome. For realtime purposes a large PC cluster is required.



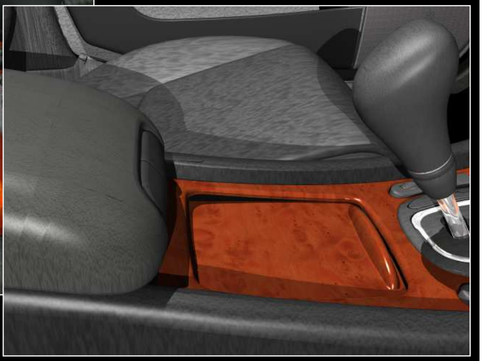
Reasons for Using RTRT: Physical Correctness

SIGGRAPH2005



Rendered with accurately measured BTF data that accounts for micro lighting effects

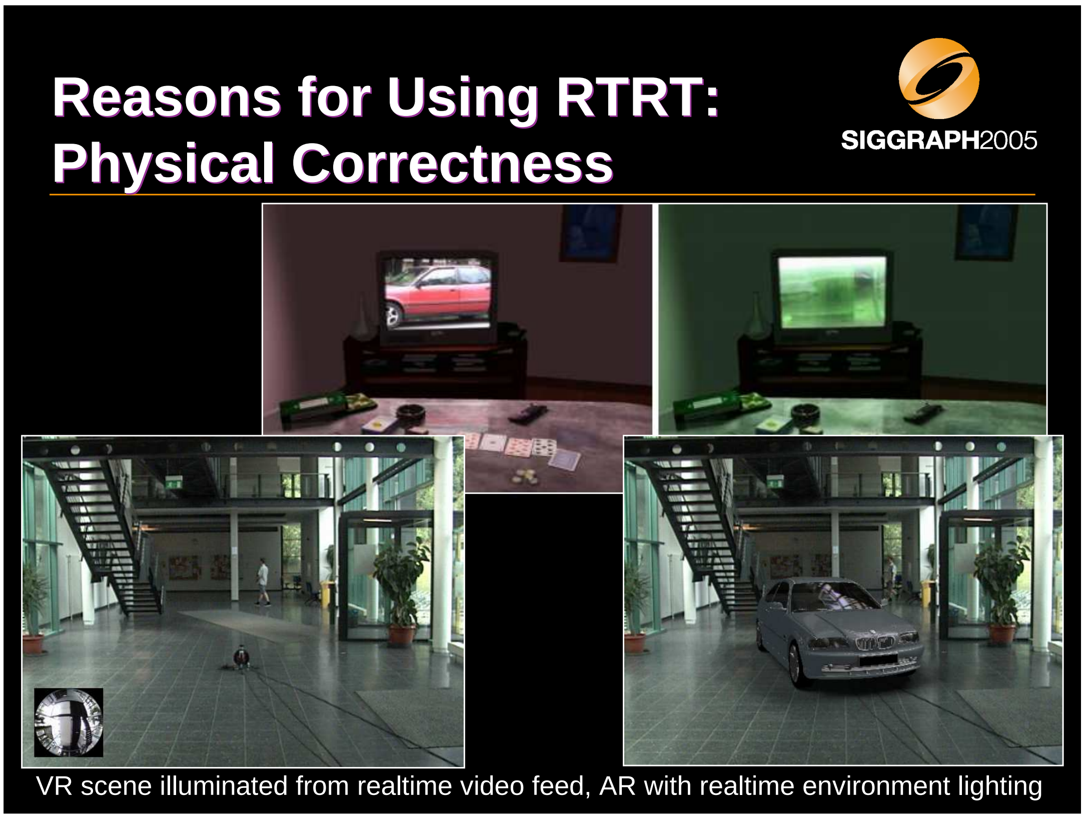
Textured Phong for comparison



BTF Data Courtesy R. Klein, Uni Bonn

This image shows the difference between using texture mapping (lower right image) and using measured BTF data, which captures the fine detail of illumination on surfaces with micro structure like leather.

The data sets have been provided by Prof. Klein, Bonn University and are directly rendered on trimmed NURBS surfaces.

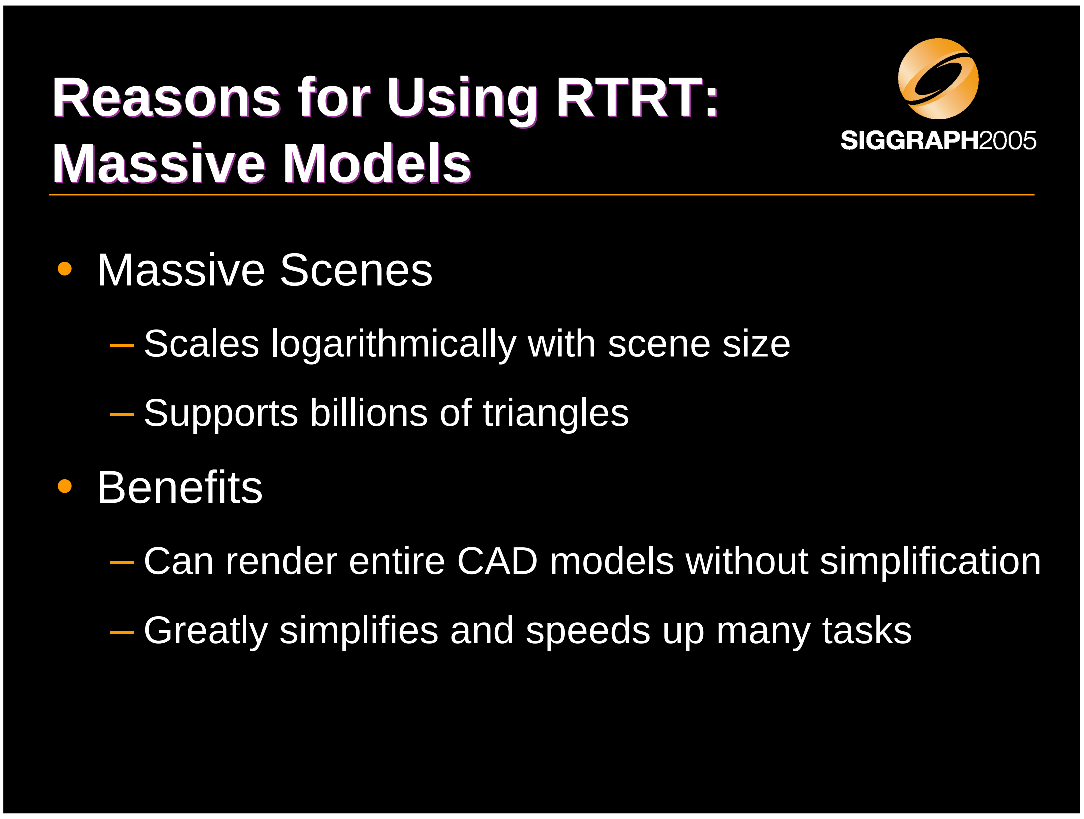


Reasons for Using RTRT: Physical Correctness

SIGGRAPH2005

VR scene illuminated from realtime video feed, AR with realtime environment lighting

Ray tracing can also be used in a VR or mixed reality context. Instead of compositing multiple 2D images, here the compositing is performed in the surface shaders of the models. In addition environment lighting is integrated from the TV screen and a 180 degree light probe, which also provides the reflections on the car.



Reasons for Using RTRT: Massive Models

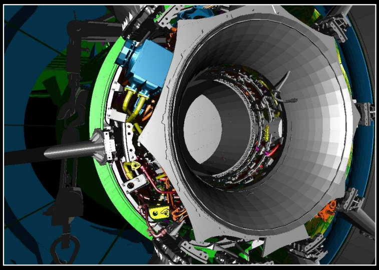
SIGGRAPH2005

- Massive Scenes
  - Scales logarithmically with scene size
  - Supports billions of triangles
- Benefits
  - Can render entire CAD models without simplification
  - Greatly simplifies and speeds up many tasks

See before.

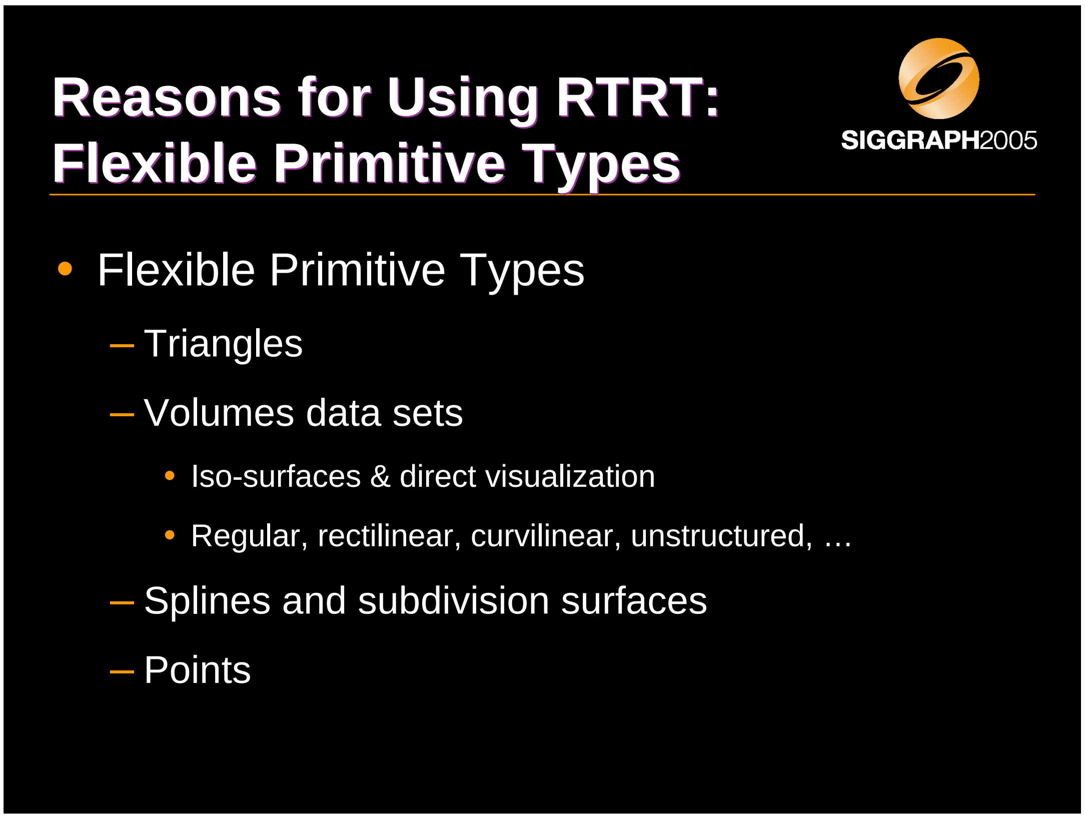
Reasons for Using RTRT: Massive Models

SIGGRAPH2005



Ray tracing has been the first and (to our knowledge) only technology to interactively render the entire Boeing 777 data set. It consists of 350 million polygons and takes up to 30 GB of data on disk. Every detail is models including tiny screws, cables, pipes, values, and many more. With ray tracing this model can be rendered interactively even on a dual-processor PC with 2-3 fps at video resolution.

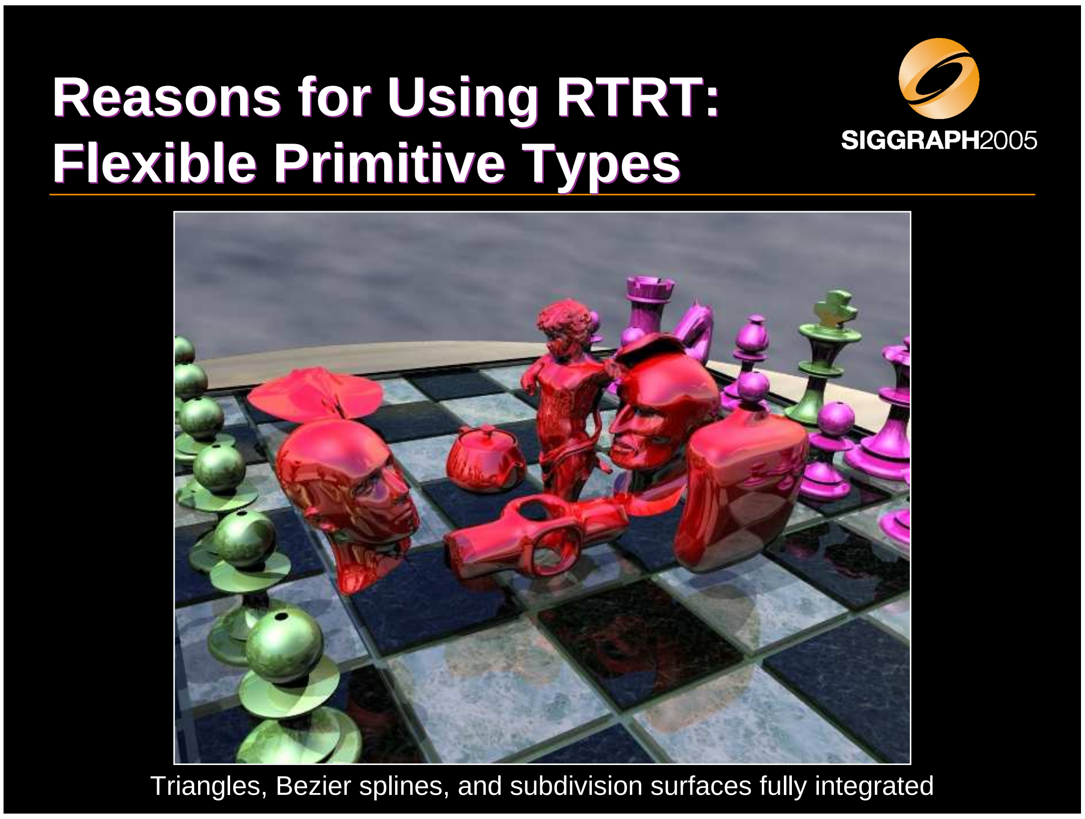
The right image contains 365 000 plants with a total of roughly 1.5 billion polygons. All leafs use alpha-mapped textures leading to an extremely high depth complexity. Still the scene can be rendered with interactive performance on a decent PC cluster. Even smooth lighting from the sky dome can be integrated. An good approximation is then shown during interaction but the image converges to a high quality solution with a few seconds.


Reasons for Using RTRT: Flexible Primitive Types

SIGGRAPH2005

- Flexible Primitive Types
  - Triangles
  - Volumes data sets
    - Iso-surfaces & direct visualization
    - Regular, rectilinear, curvilinear, unstructured, …
  - Splines and subdivision surfaces
  - Points

See before.

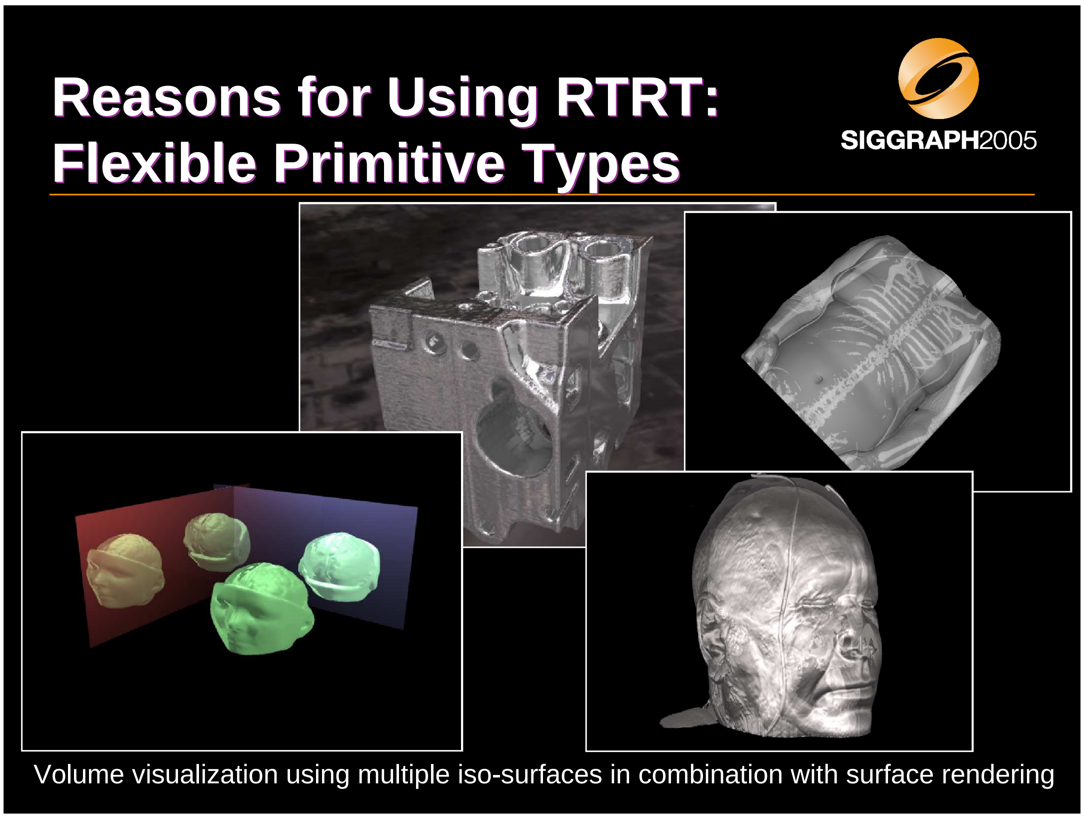


Reasons for Using RTRT: Flexible Primitive Types

SIGGRAPH2005

Triangles, Bezier splines, and subdivision surfaces fully integrated

The image shows a mixture of triangles, splines, and subdivision surfaces rendered directly using ray tracing at interactive performance.

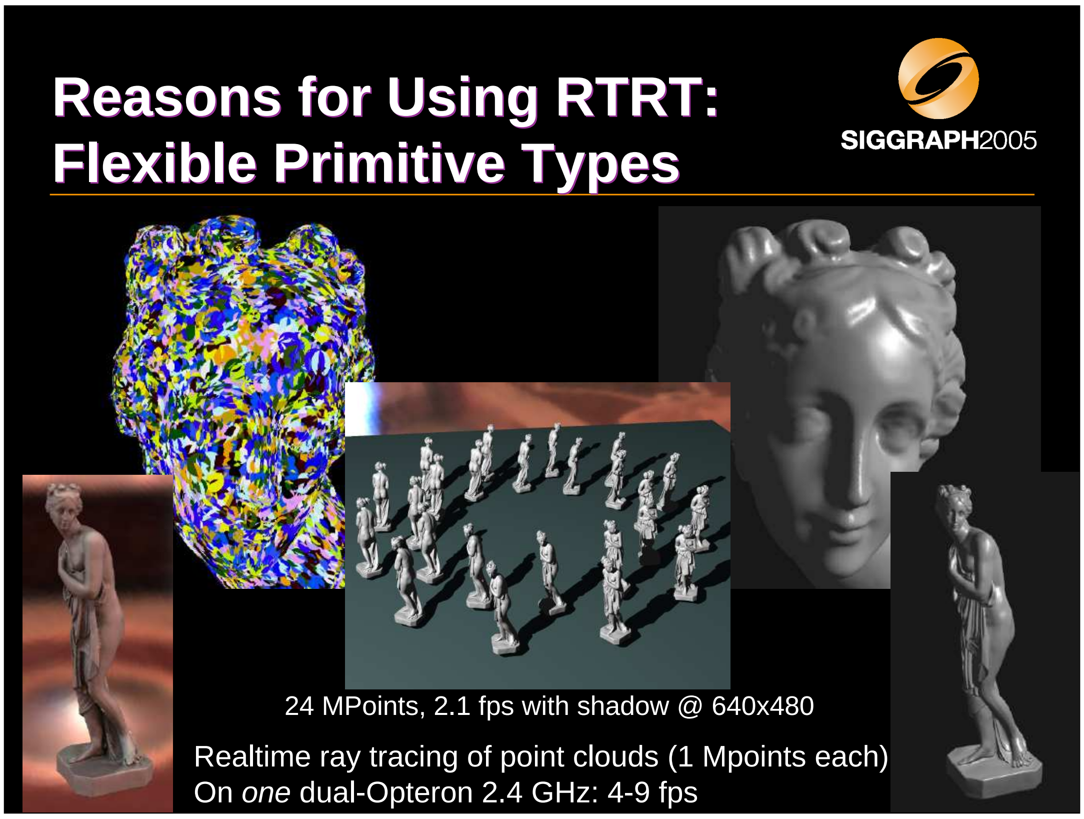


Reasons for Using RTRT: Flexible Primitive Types

SIGGRAPH2005

Volume visualization using multiple iso-surfaces in combination with surface rendering

Volume data sets can also be visualized interactively with ray tracing. These images show iso-surfaces (combined with surfaces in the lower left). IN recent work we also integrated direct volume rendering as well as volume rendering semi- and unstructured data sets at interactive performance.



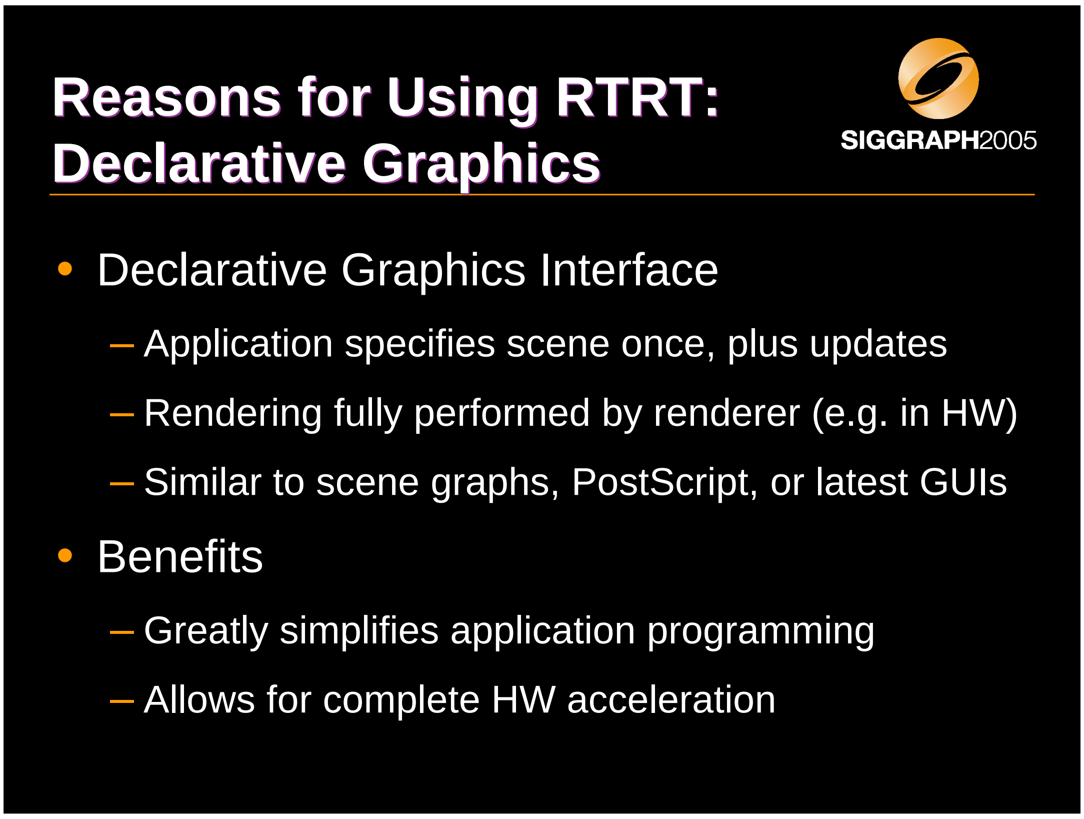
Reasons for Using RTRT: Flexible Primitive Types

SIGGRAPH2005

24 MPoints, 2.1 fps with shadow @ 640x480

Realtime ray tracing of point clouds (1 Mpoints each)
On *one* dual-Opteron 2.4 GHz: 4-9 fps

These images show large point clouds being interactively ray traced. Even huge scenes with 24 million points can be rendered interactively with shadows.



Reasons for Using RTRT: Declarative Graphics

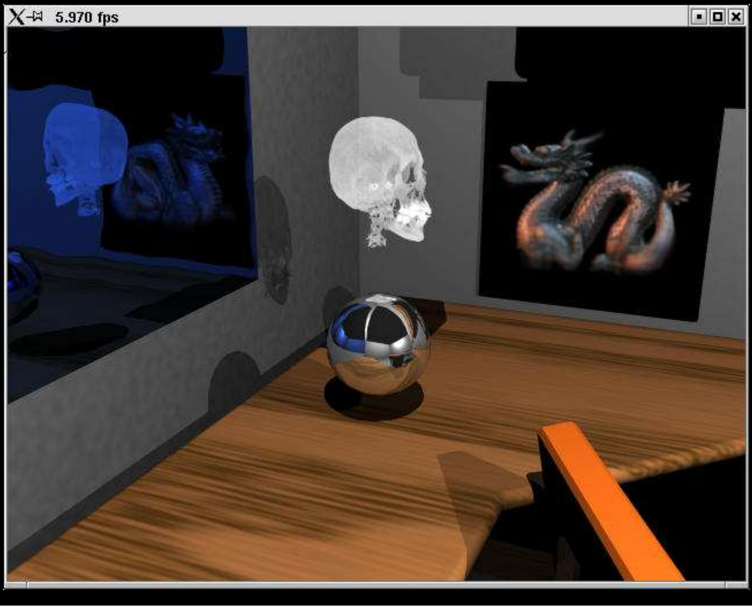
SIGGRAPH2005

- Declarative Graphics Interface
  - Application specifies scene once, plus updates
  - Rendering fully performed by renderer (e.g. in HW)
  - Similar to scene graphs, PostScript, or latest GUIs
- Benefits
  - Greatly simplifies application programming
  - Allows for complete HW acceleration

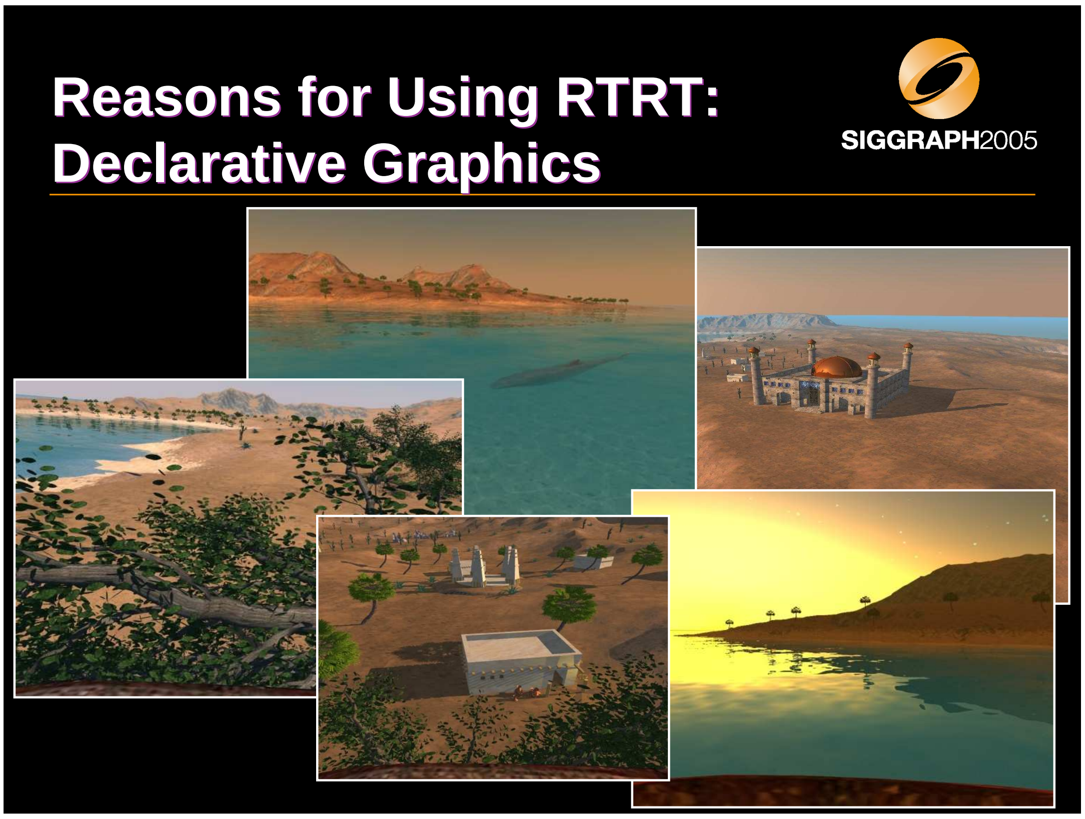
See before.

Reasons for Using RTRT: Declarative Graphics

SIGGRAPH2005



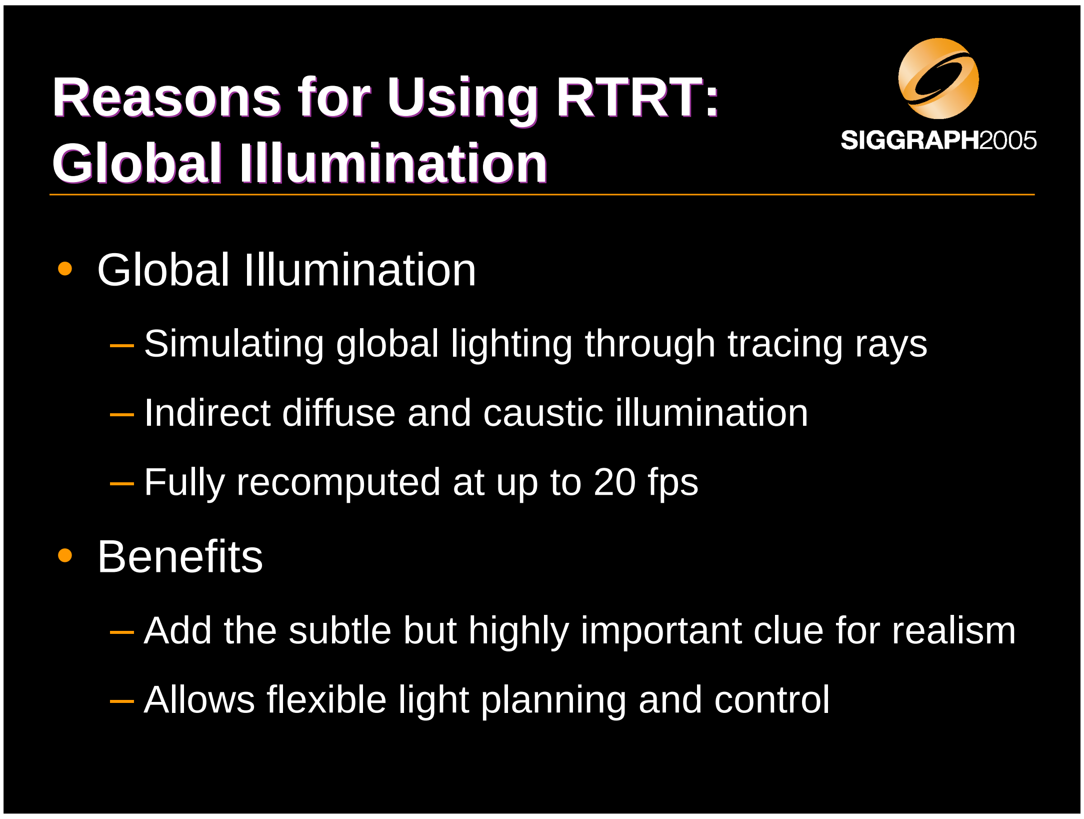
This simple image the orthogonality of geometry, appearance, and lighting that is a prerequisite for declarative scene descriptions. The scene contains simple surfaces, a volume and a hologram (light field) together with a procedural wood shader, a bump mapped mirror, and direct volume rendering effects. Every object and appearance has been separately modeled, but the ray tracer combines all the combinations of effects fully automatically and always physically-correct (at least as long each object is correctly modeled).



Reasons for Using RTRT: Declarative Graphics

SIGGRAPH2005

These images are from a prototype computer game running in realtime on the ray tracer. It consists of more than 40 million polygons and all optical effects are fully simulated at rendering time. All trees are fully models and no LOD is being used.

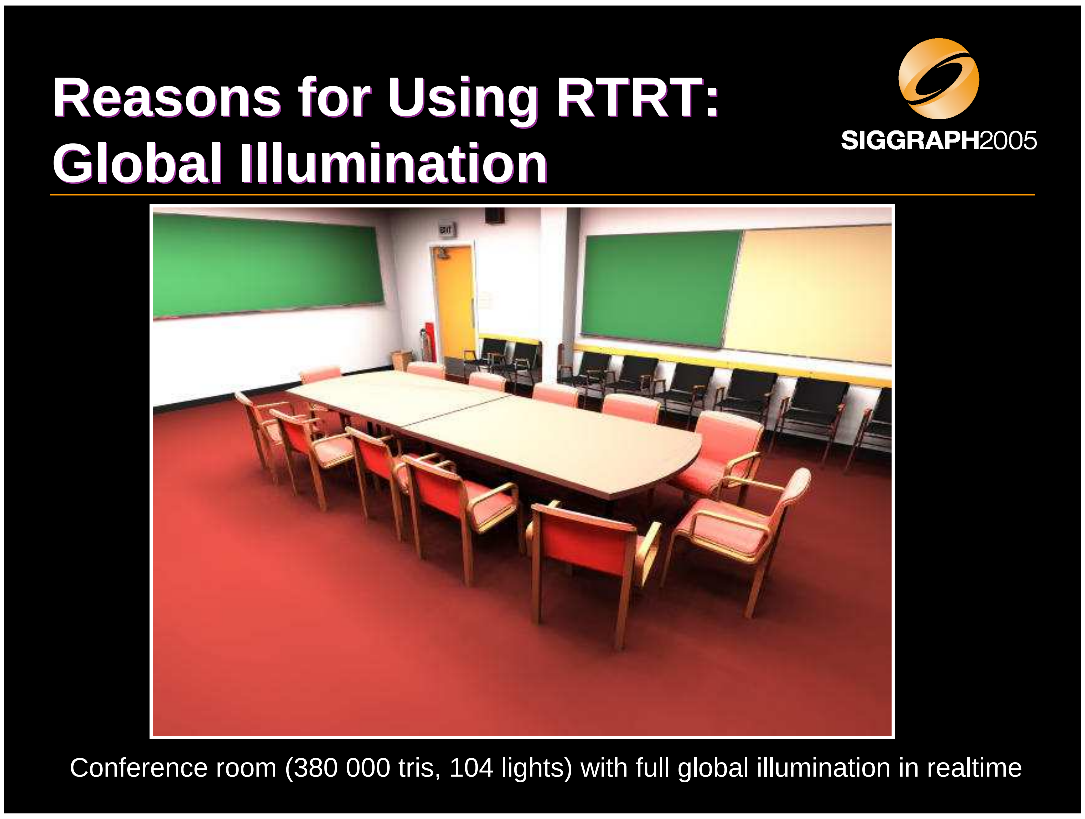


Reasons for Using RTRT: Global Illumination

SIGGRAPH2005

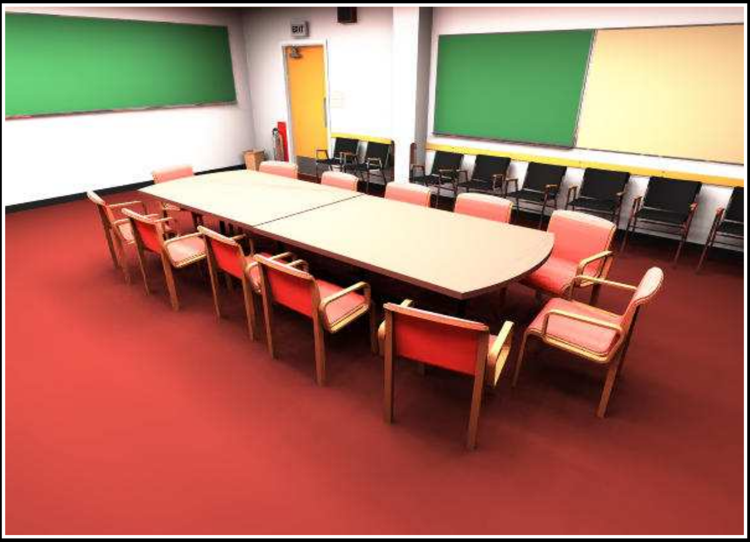
- Global Illumination
  - Simulating global lighting through tracing rays
  - Indirect diffuse and caustic illumination
  - Fully recomputed at up to 20 fps
- Benefits
  - Add the subtle but highly important clue for realism
  - Allows flexible light planning and control

See before.



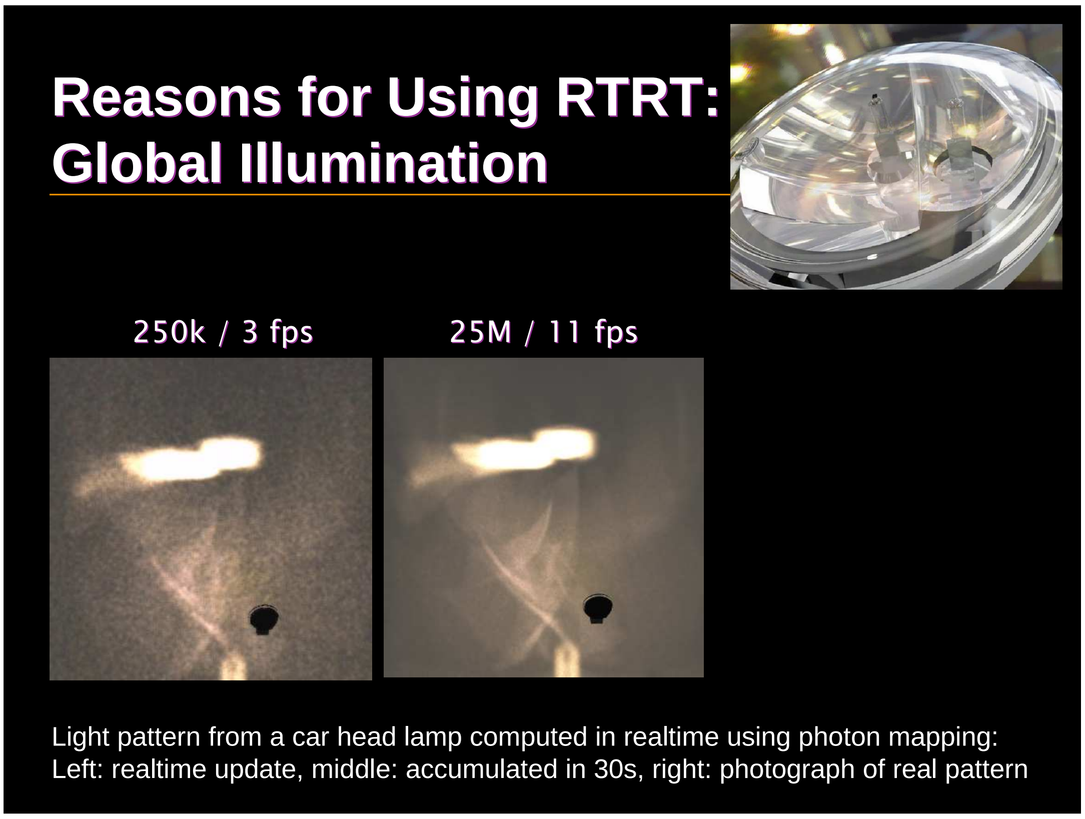
Reasons for Using RTRT: Global Illumination

SIGGRAPH2005

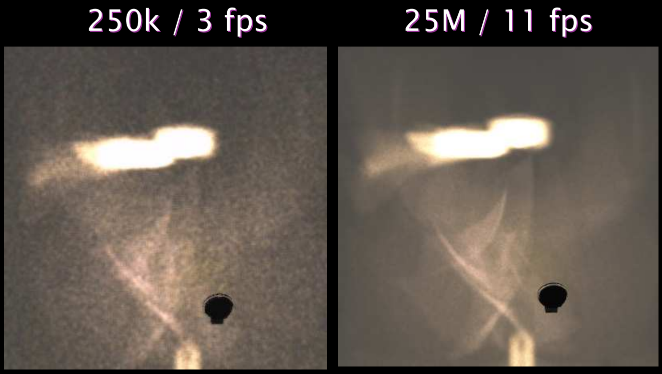


Conference room (380 000 tris, 104 lights) with full global illumination in realtime

Conference room rendered with global illumination (converged view). Due to the shape of the light sources that are long and thin oriented along the table, the differences in the shadow boundaries are clearly visible being almost sharp in one direction and very smooth in the other. The scene also contains many specular materials such as metal frames of chairs and metal frames of the boards.



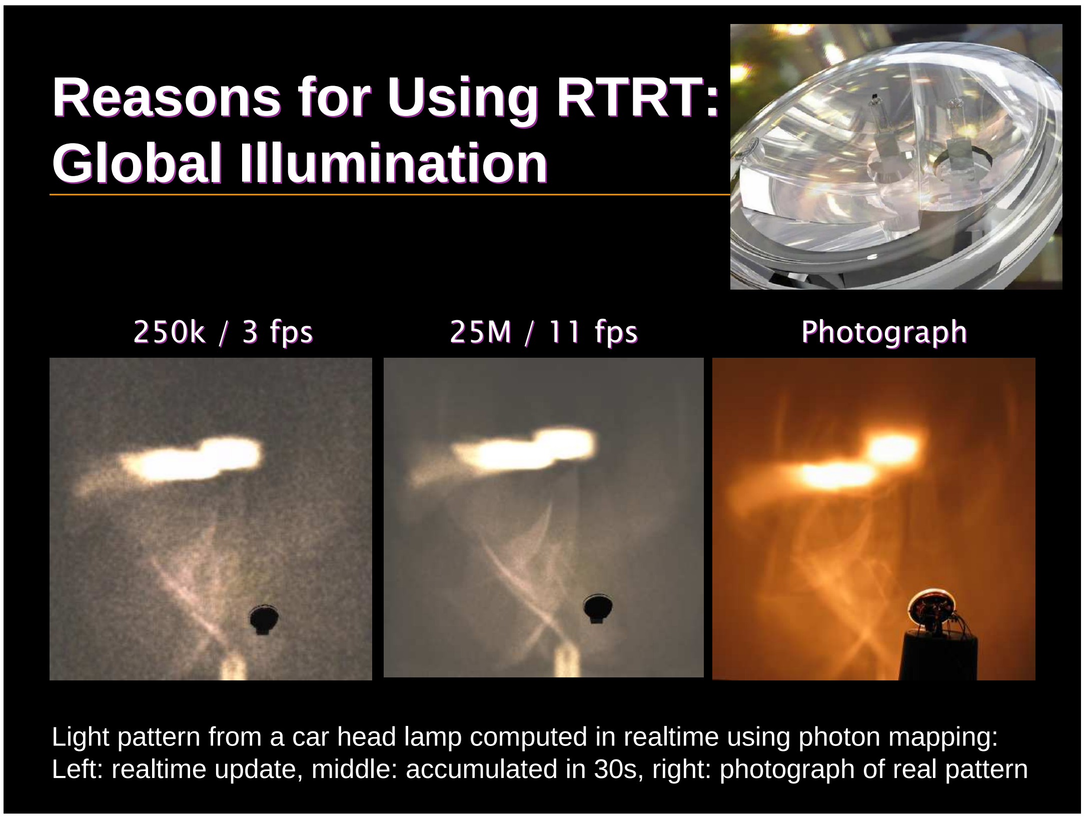
Reasons for Using RTRT: Global Illumination



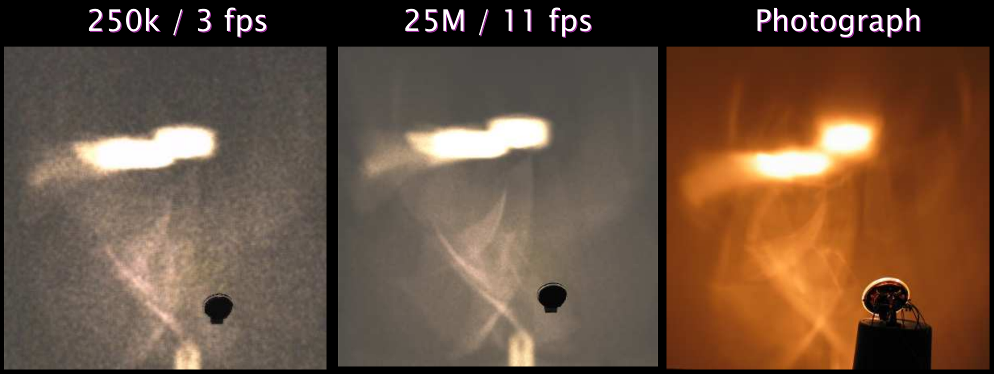
Light pattern from a car head lamp computed in realtime using photon mapping:
Left: realtime update, middle: accumulated in 30s, right: photograph of real pattern

This image shows the results of interactive photon mapping for simulating the emission properties of this complex car headlight consisting of 800,000 polygons (top image). The left image shows the quality of simulating the illumination of the headlight on a grey wall with 250,000 photons. We can fully simulate the illumination at 3 fps on a cluster of 25 PCs. Once the accumulate 30 seconds worth of photons (25 million) and then only visualize them, we even reach 11 fps.

The quality of the simulation results is extremely good …

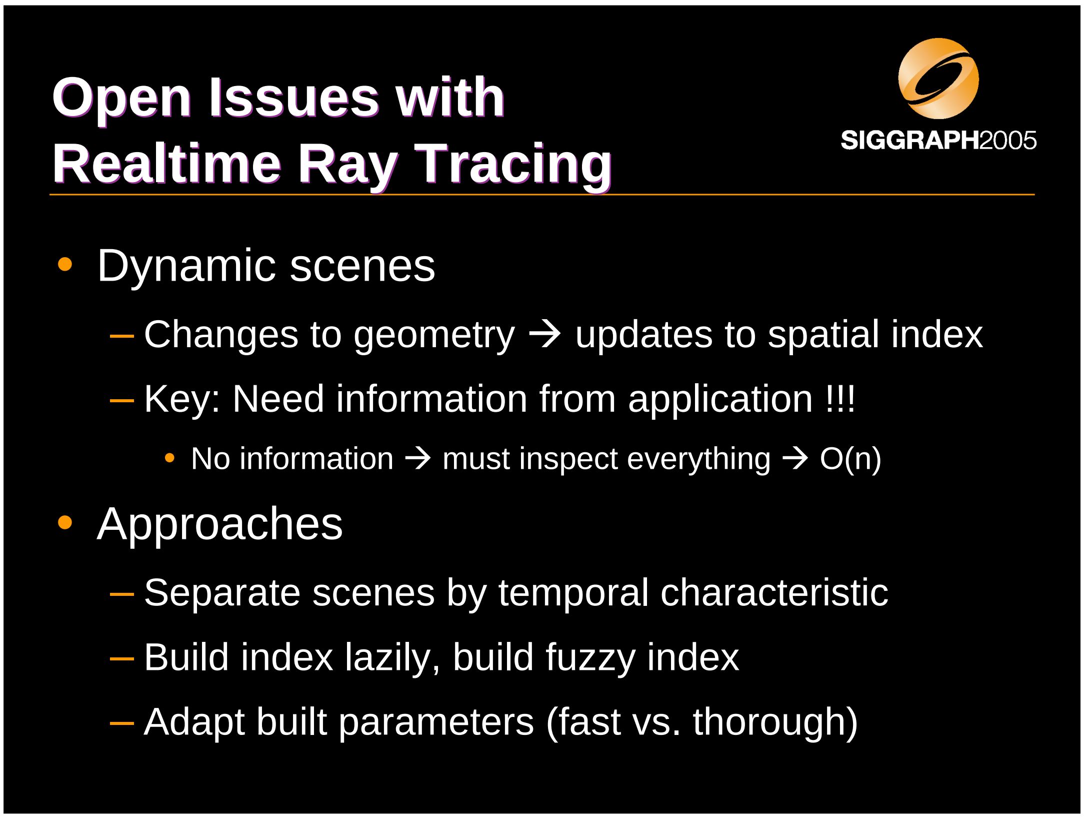


Reasons for Using RTRT: Global Illumination



Light pattern from a car head lamp computed in realtime using photon mapping:
Left: realtime update, middle: accumulated in 30s, right: photograph of real pattern

As the image on the far right shows. Nearly all features are accurately represented. Commercial applications take several hours for this type of simulation.



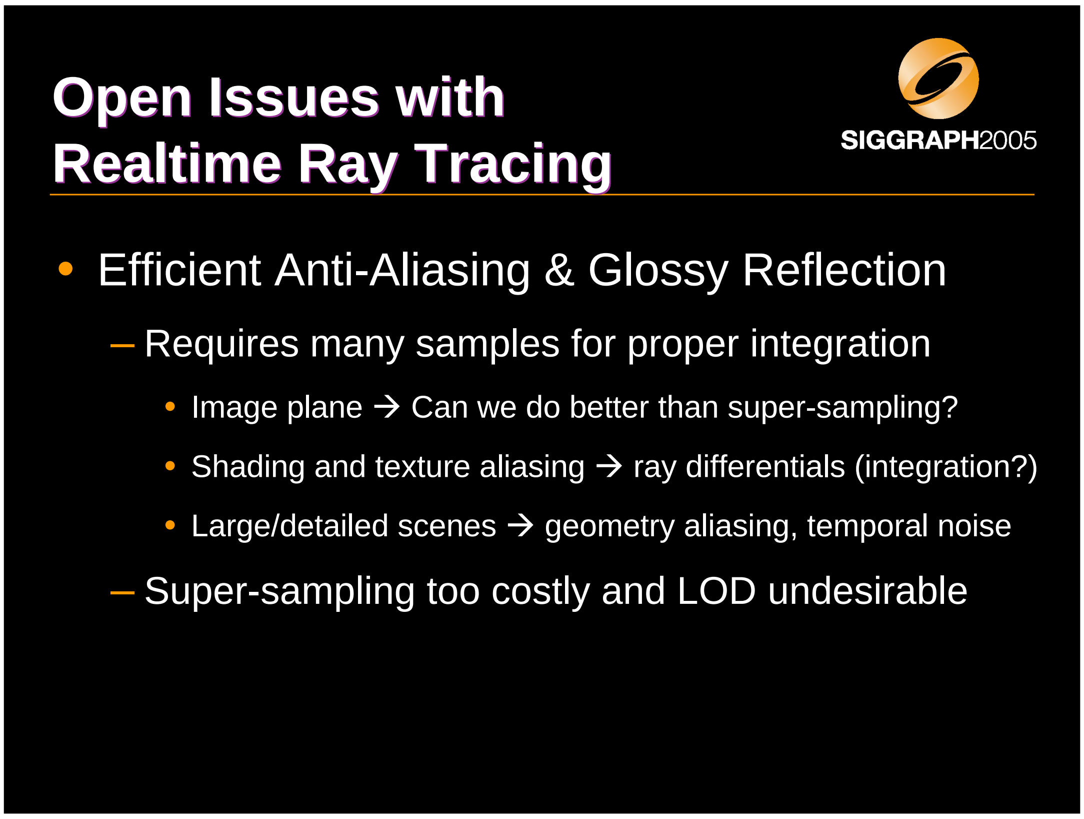
Open Issues with Realtime Ray Tracing

SIGGRAPH2005

- Dynamic scenes
  - Changes to geometry → updates to spatial index
  - Key: Need information from application !!!
    - No information → must inspect everything → O(n)
- Approaches
  - Separate scenes by temporal characteristic
  - Build index lazily, build fuzzy index
  - Adapt built parameters (fast vs. thorough)

Many dynamic scenes work reasonable well already with ray tracing but a lot of improvements can still be done. In particular fully dynamic scenes are still problematic.

This problem can only be solved efficiently when the application can provide enough information about the movement of surfaces. If now information is known, any rendering algorithms must resolve to sequentially render every single surface. However, with the proper information, significant speed-up can be reached. The remaining question is which information can be made available, how is it represented best, and how can the renderer make best use of it.

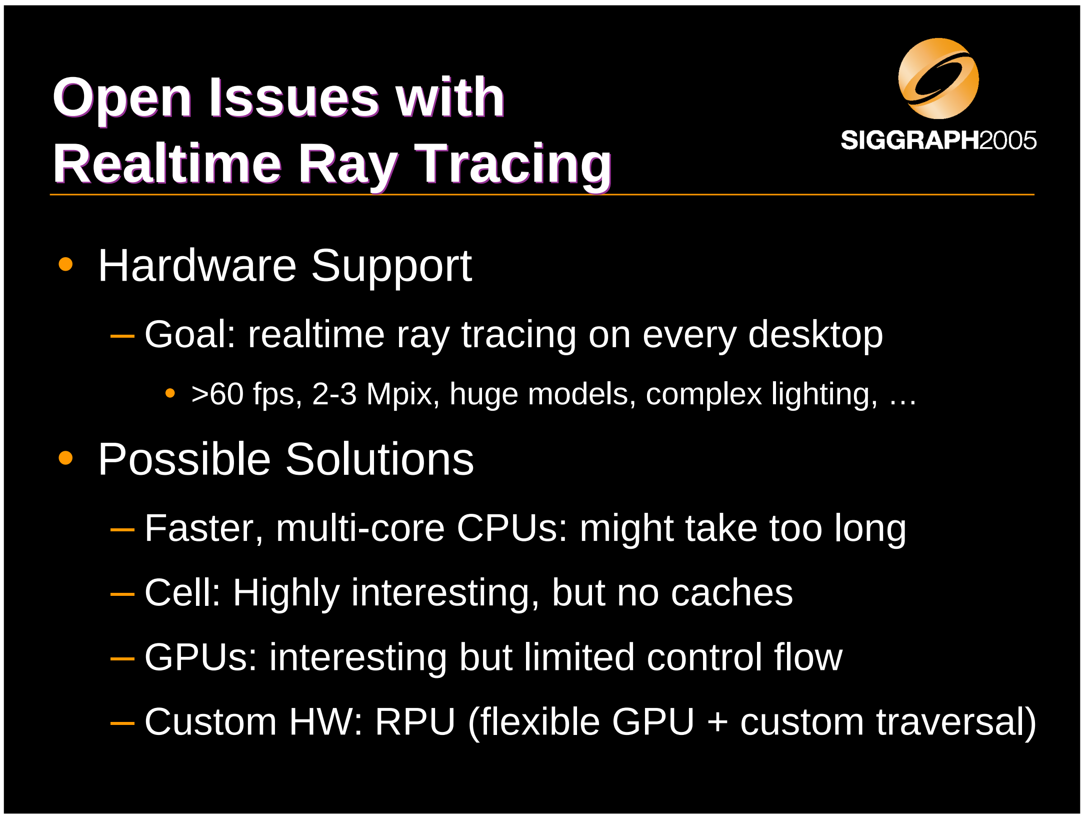


Open Issues with Realtime Ray Tracing

SIGGRAPH2005

- Efficient Anti-Aliasing & Glossy Reflection
  - Requires many samples for proper integration
    - Image plane → Can we do better than super-sampling?
    - Shading and texture aliasing → ray differentials (integration?)
    - Large/detailed scenes → geometry aliasing, temporal noise
  - Super-sampling too costly and LOD undesirable

Anti-aliasing and glossy reflections are just one example of the difficulty to integrate over large domains efficiently with a point sampling approach. This is still a large open question.



Open Issues with Realtime Ray Tracing

SIGGRAPH2005

- Hardware Support
  - Goal: realtime ray tracing on every desktop
    - >60 fps, 2-3 Mpix, huge models, complex lighting, …
- Possible Solutions
  - Faster, multi-core CPUs: might take too long
  - Cell: Highly interesting, but no caches
  - GPUs: interesting but limited control flow
  - Custom HW: RPU (flexible GPU + custom traversal)

There has been a lot of work recently on improved hardware support for realtime ray tracing. While PC clusters are a reasonable solution for larger industries, they cannot work in the mass market. Here a small-form factor (PC) solution needs to be found.

A number of options are available ranging from highly-parallel multi-core designs of general purpose CPUs, over the new Cell architecture to GPUs and custom ray tracing hardware. Exactly what will be the best solution for what market is still an open question.

SIGGRAPH2005