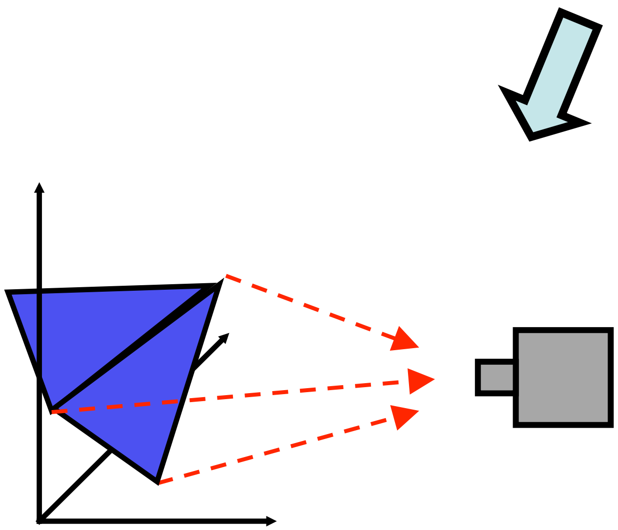
Real-Time Ray Tracing

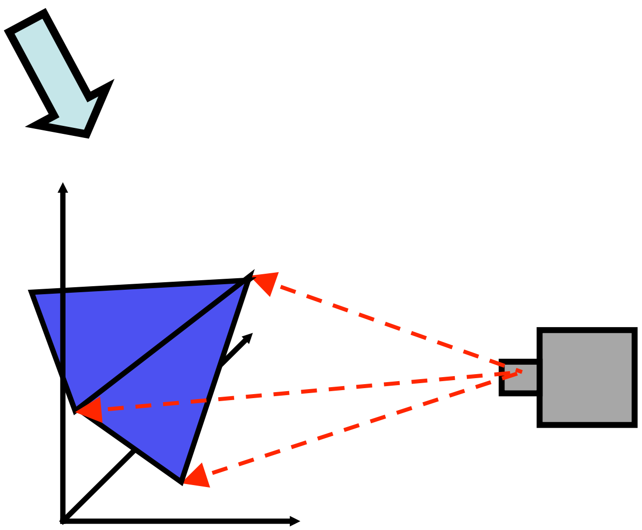
Introduction to Realtime Ray Tracing

- 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

Rendering in Computer Graphics



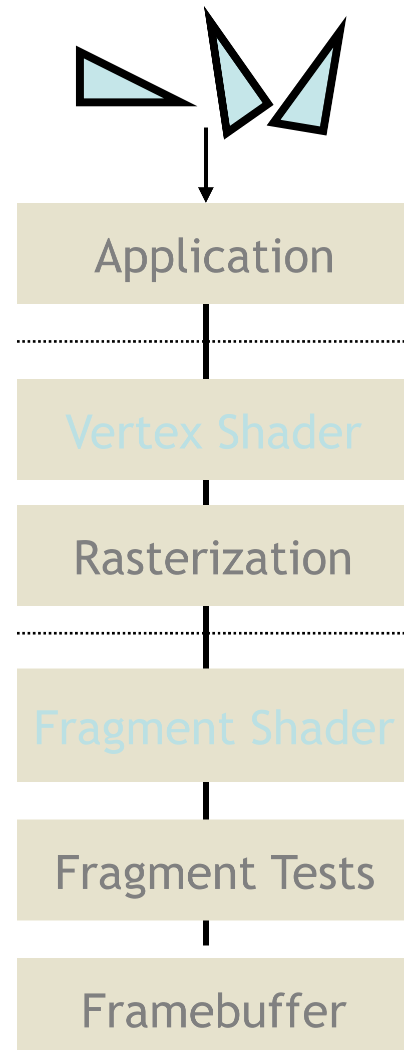


Rasterization:
Projection geometry forward

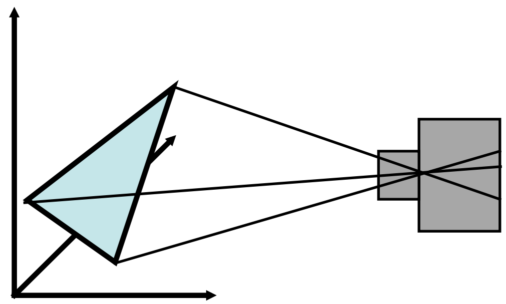
Ray Tracing:
Project image samples backwards

Current Technology: Rasterization

- **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

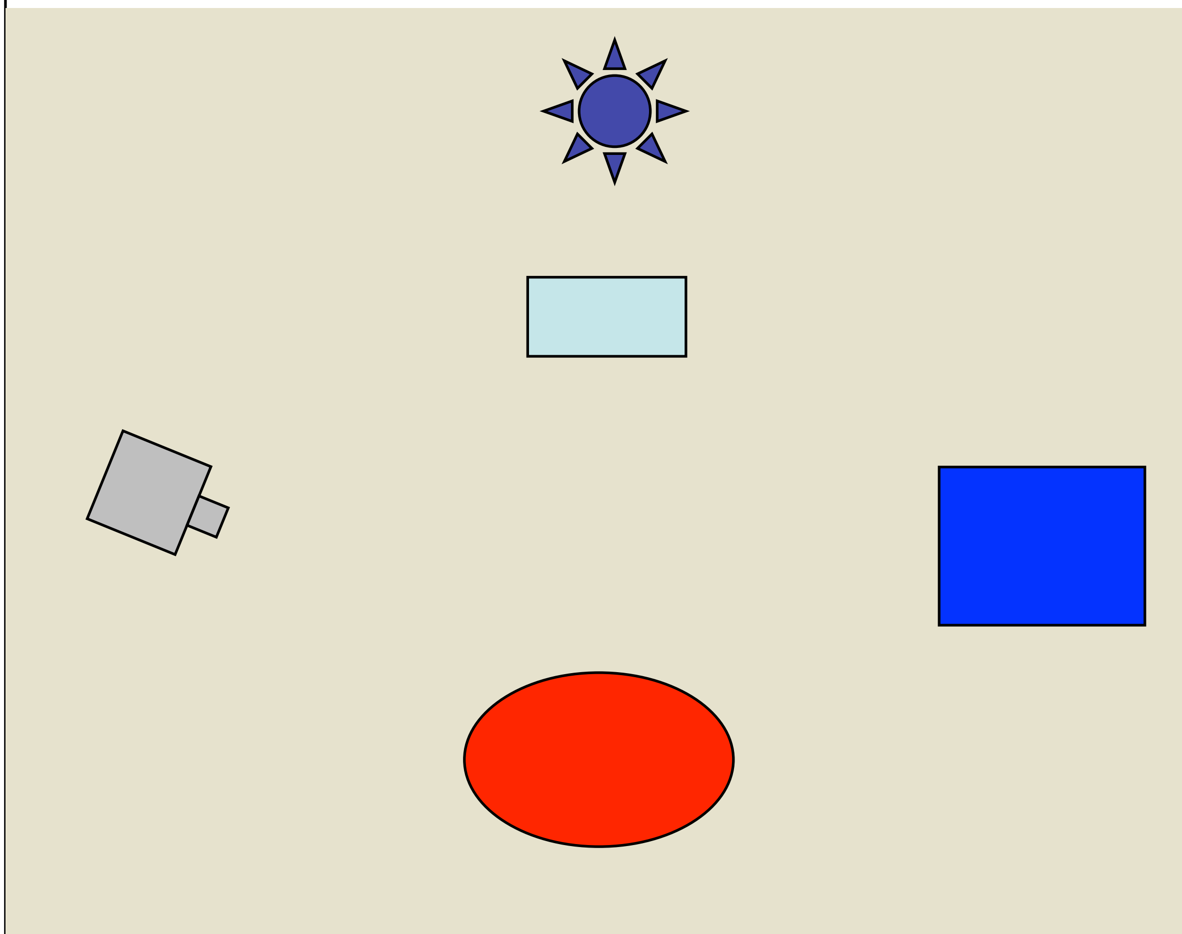


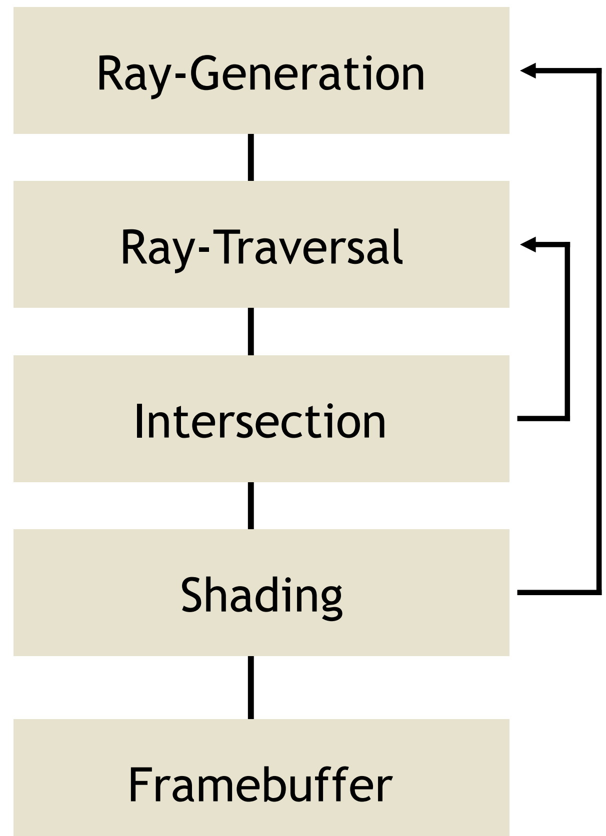
Current Technology: Rasterization



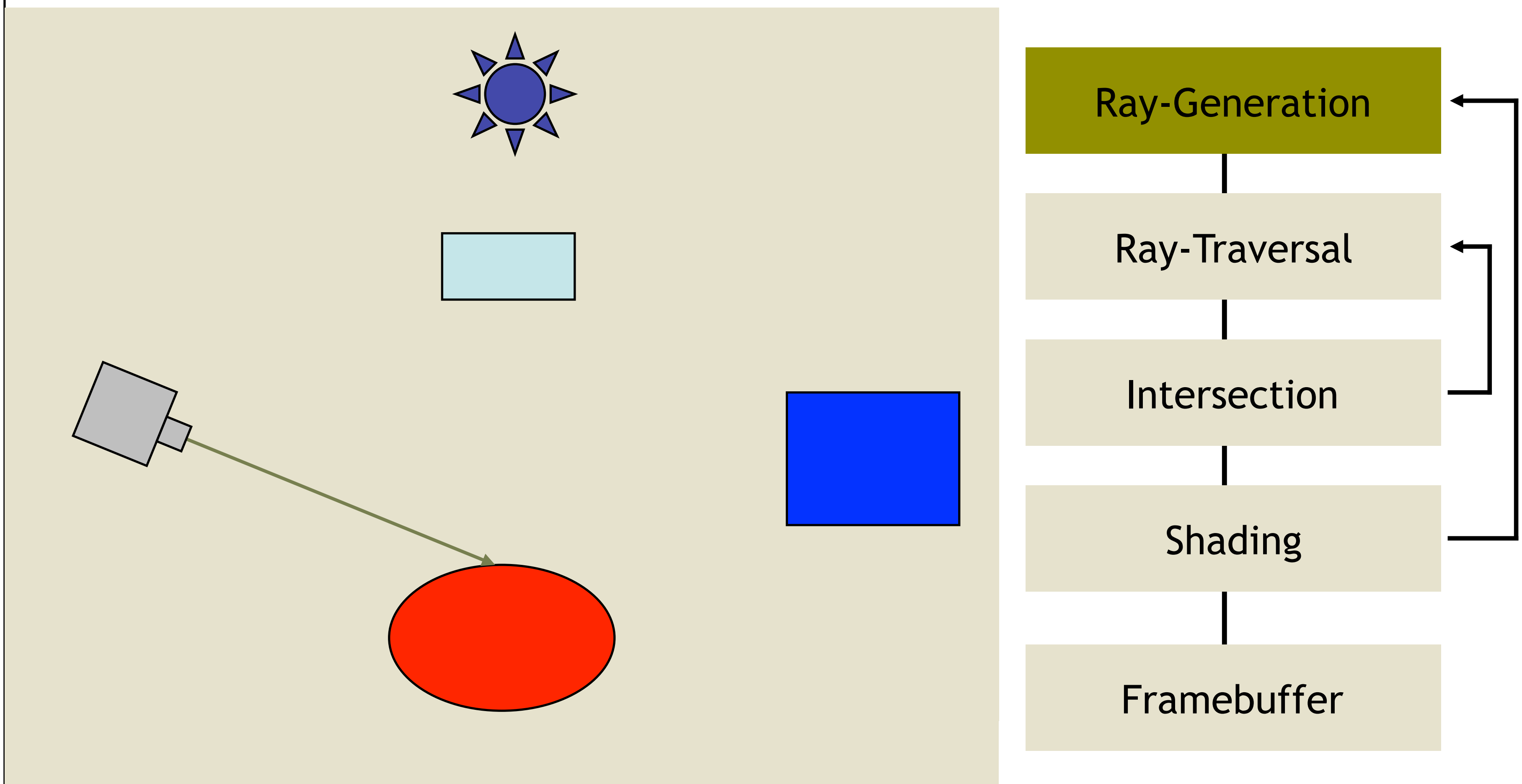
- **Primitive operation of all interactive graphics !!**
  - Scan converts a single triangle at a time
- **Sequentially processes *every* triangle *individually***
  - Cannot access more than one triangle at a time
  - ➔ But most effects need access to the entire scene: Shadows, reflection, global illumination

What is Ray Tracing?

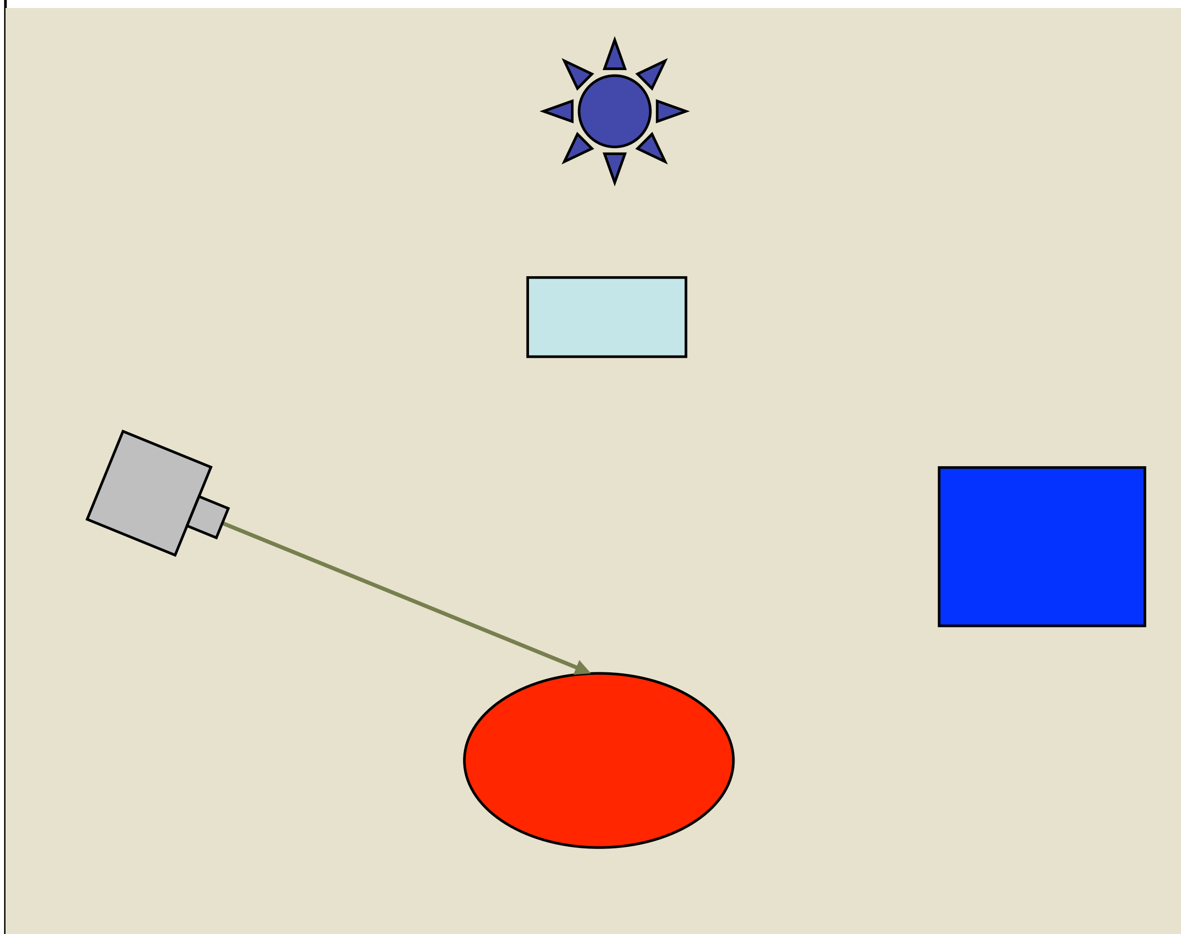


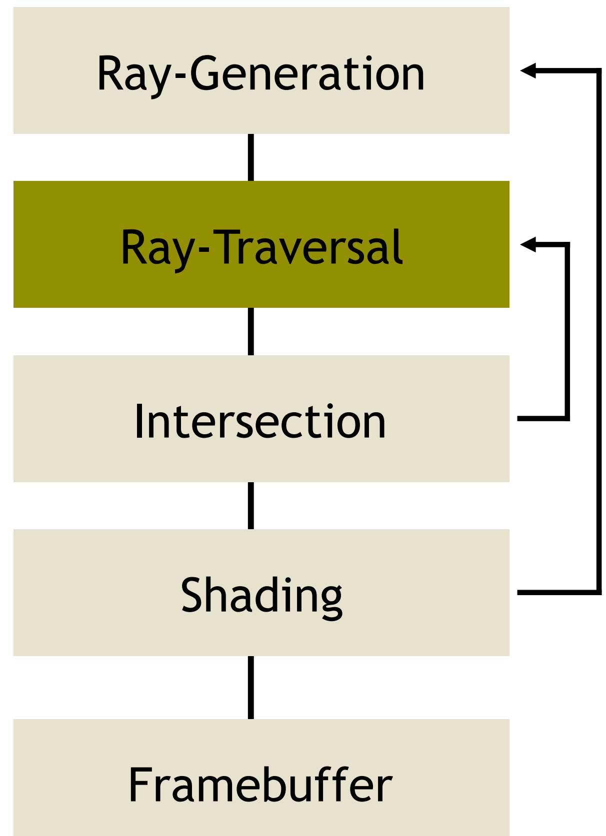


What is Ray Tracing?

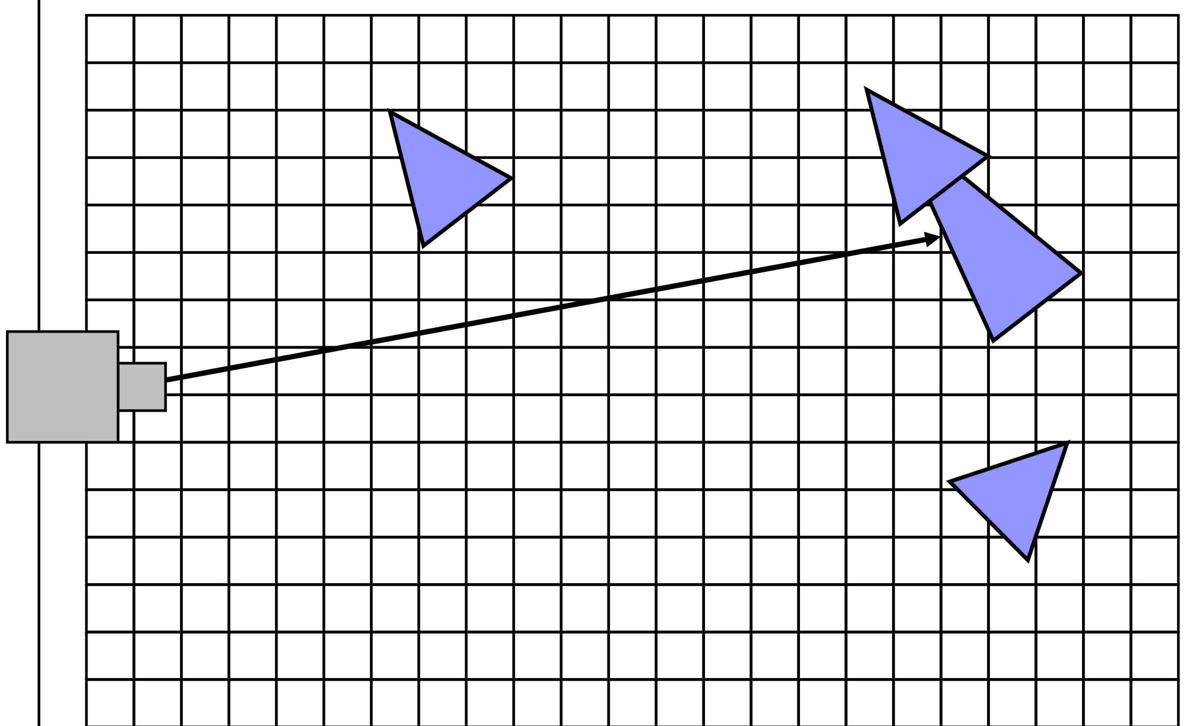


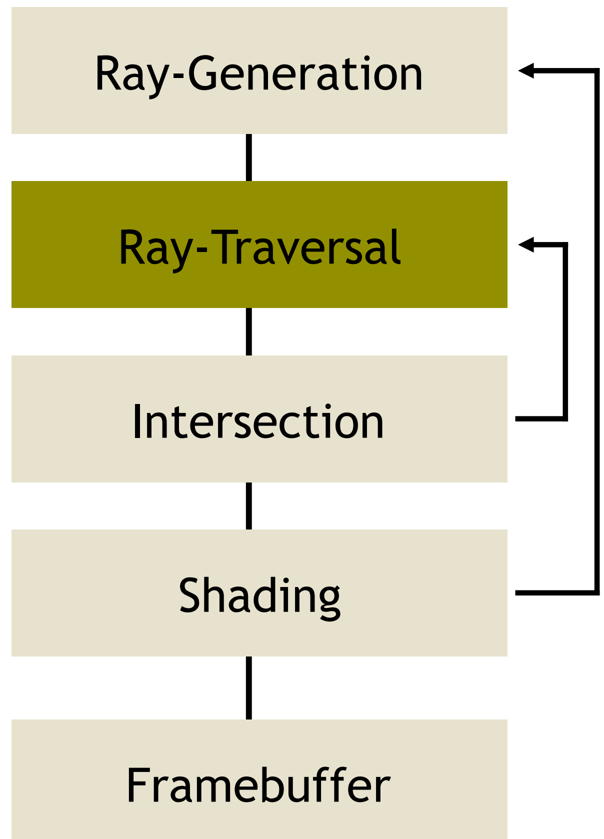
What is Ray Tracing?



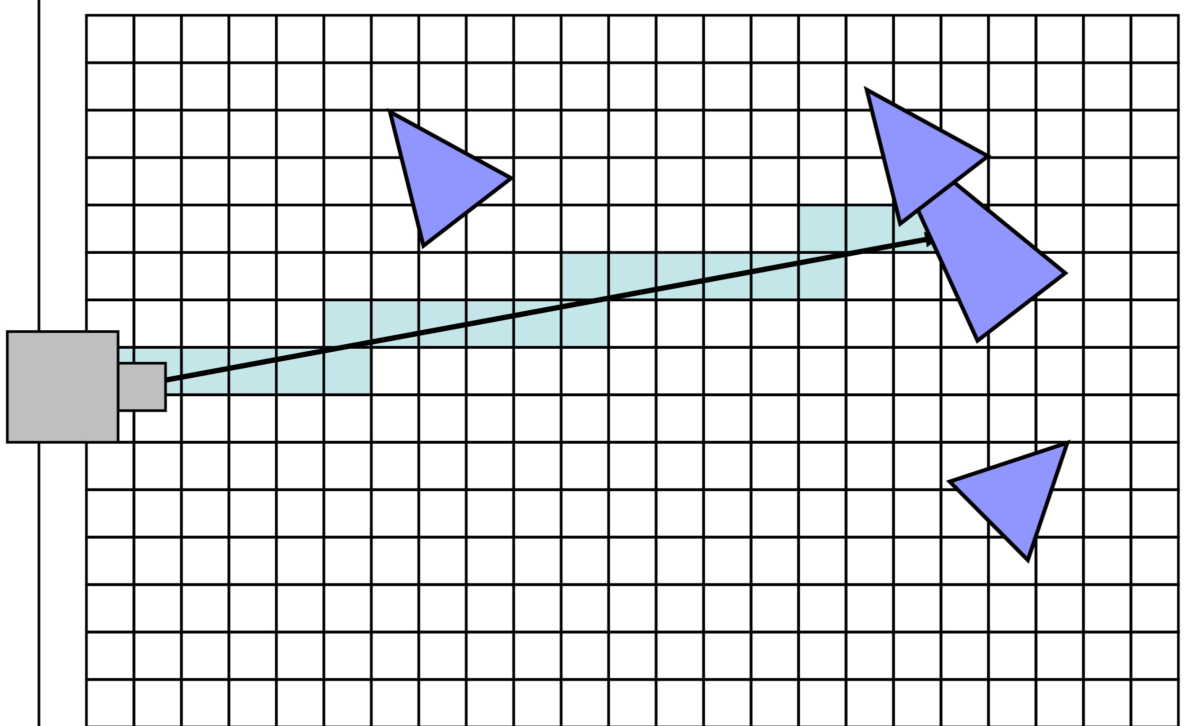


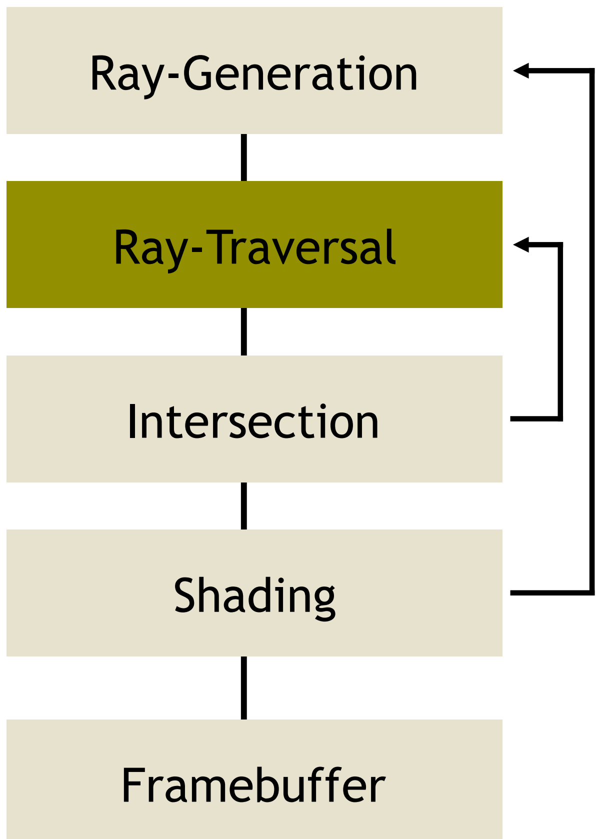
What is Ray Tracing? Traversal



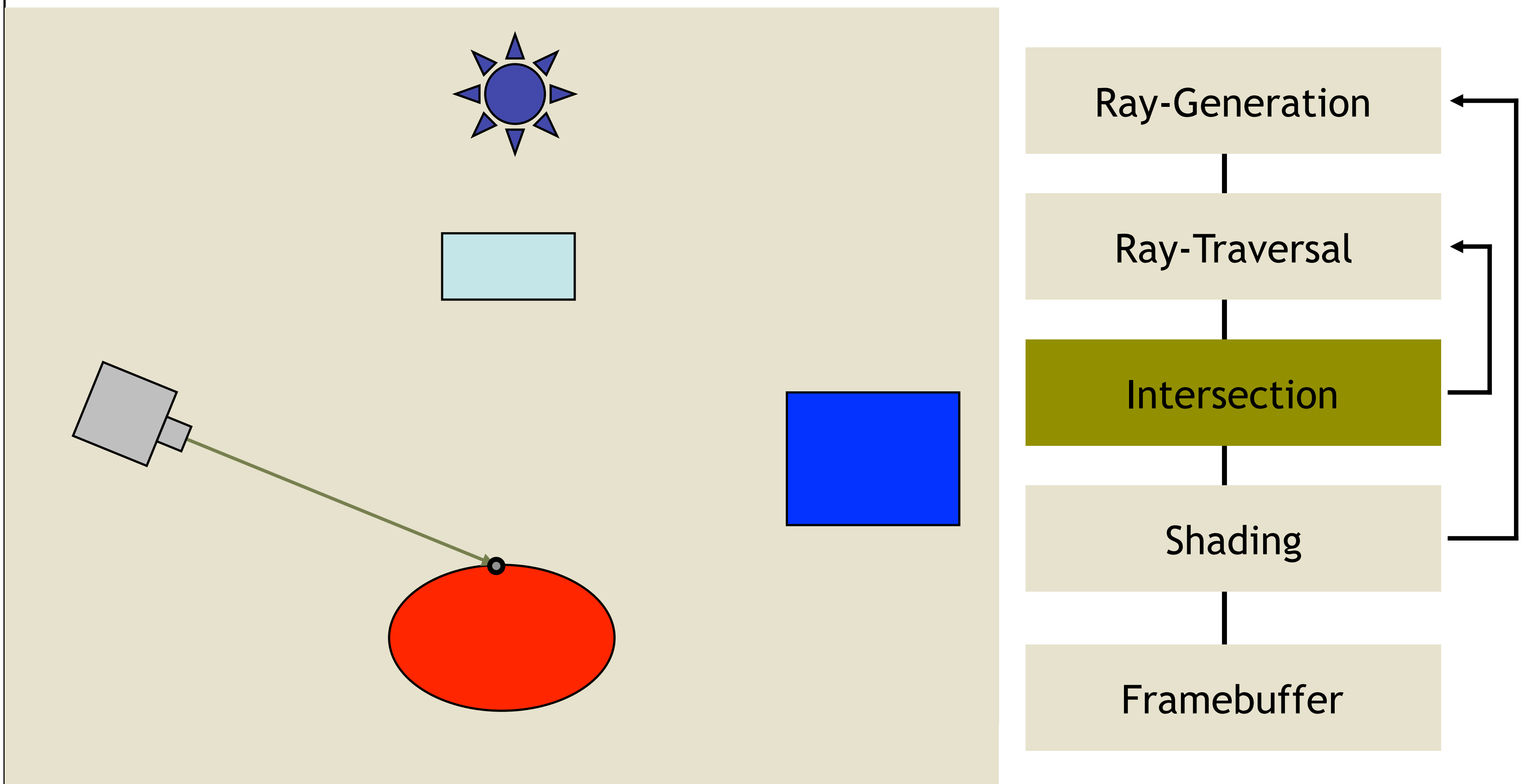


What is Ray Tracing? Traversal

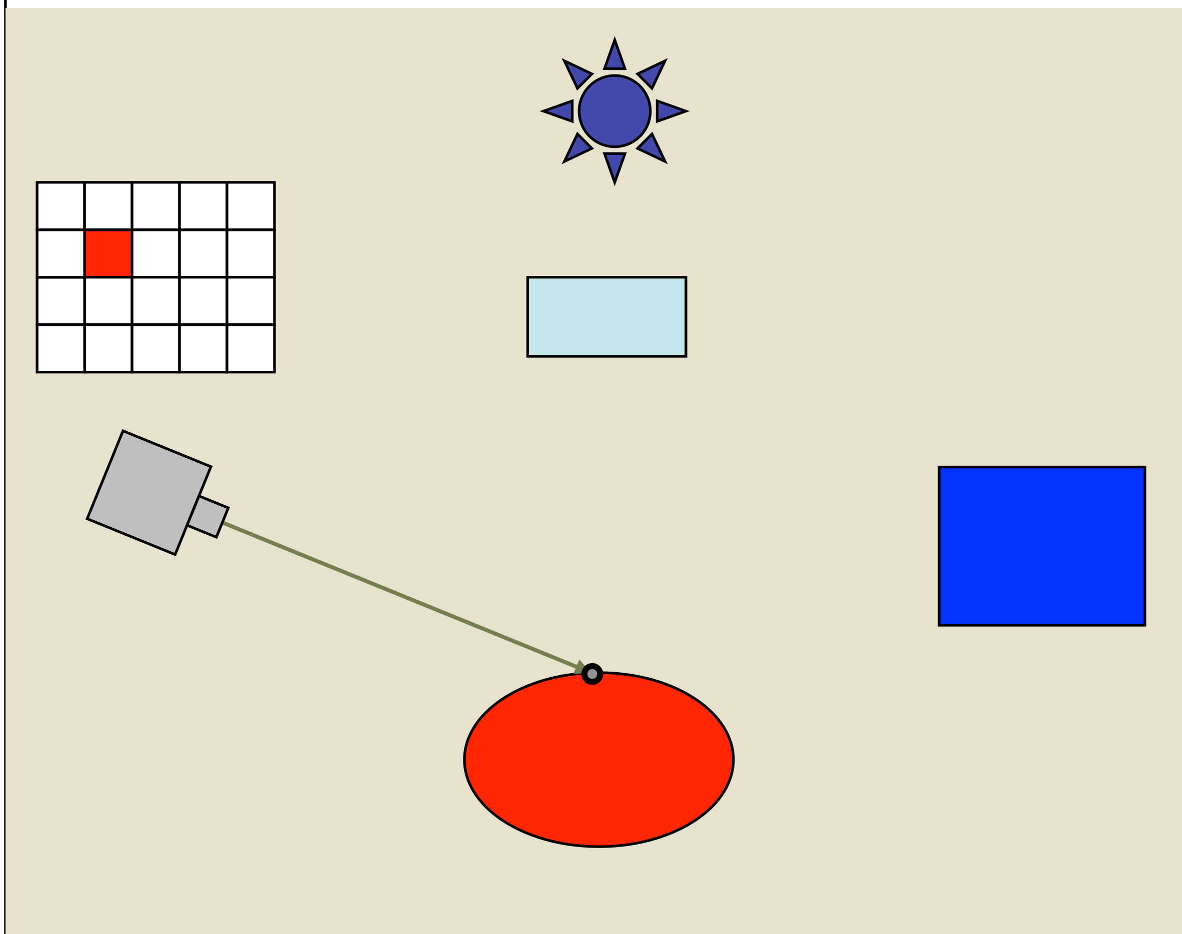


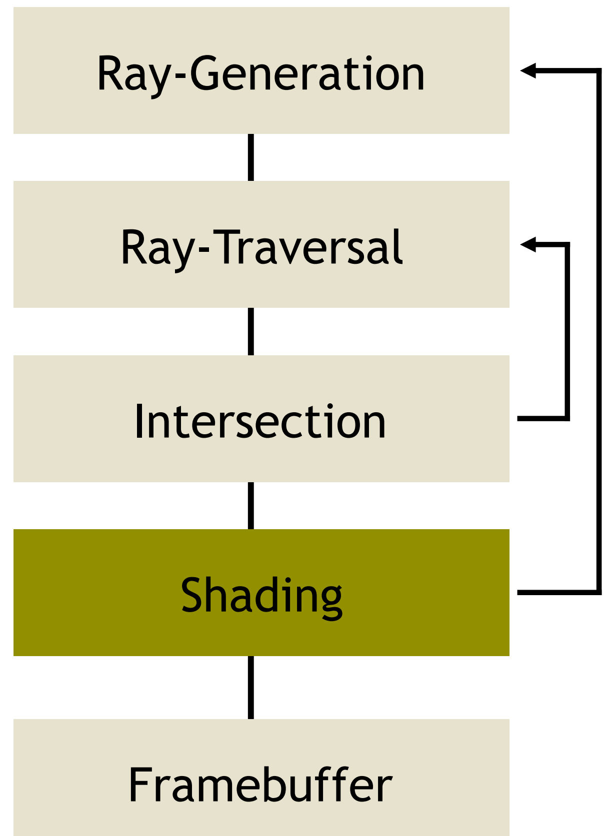


What is Ray Tracing?

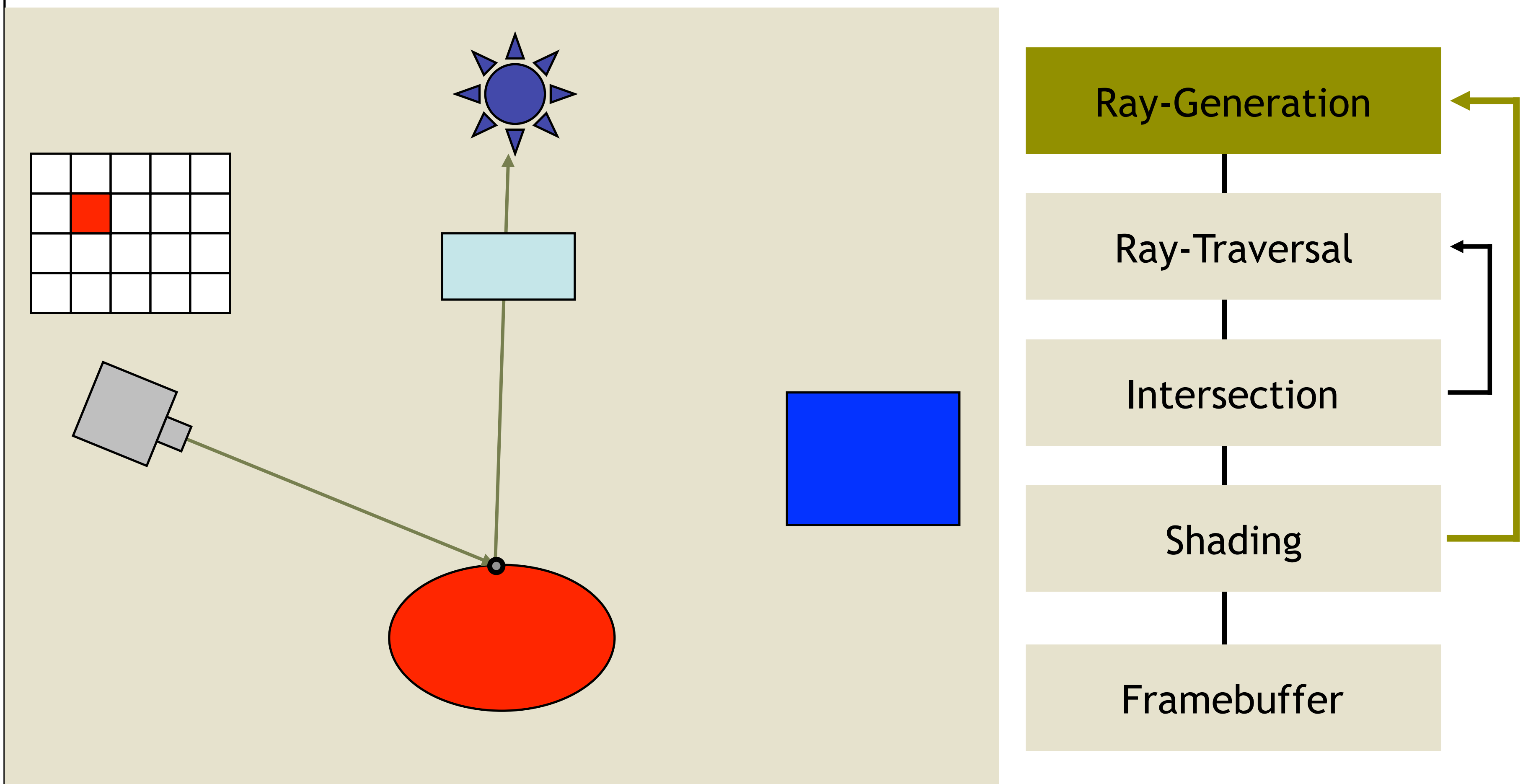


What is Ray Tracing?

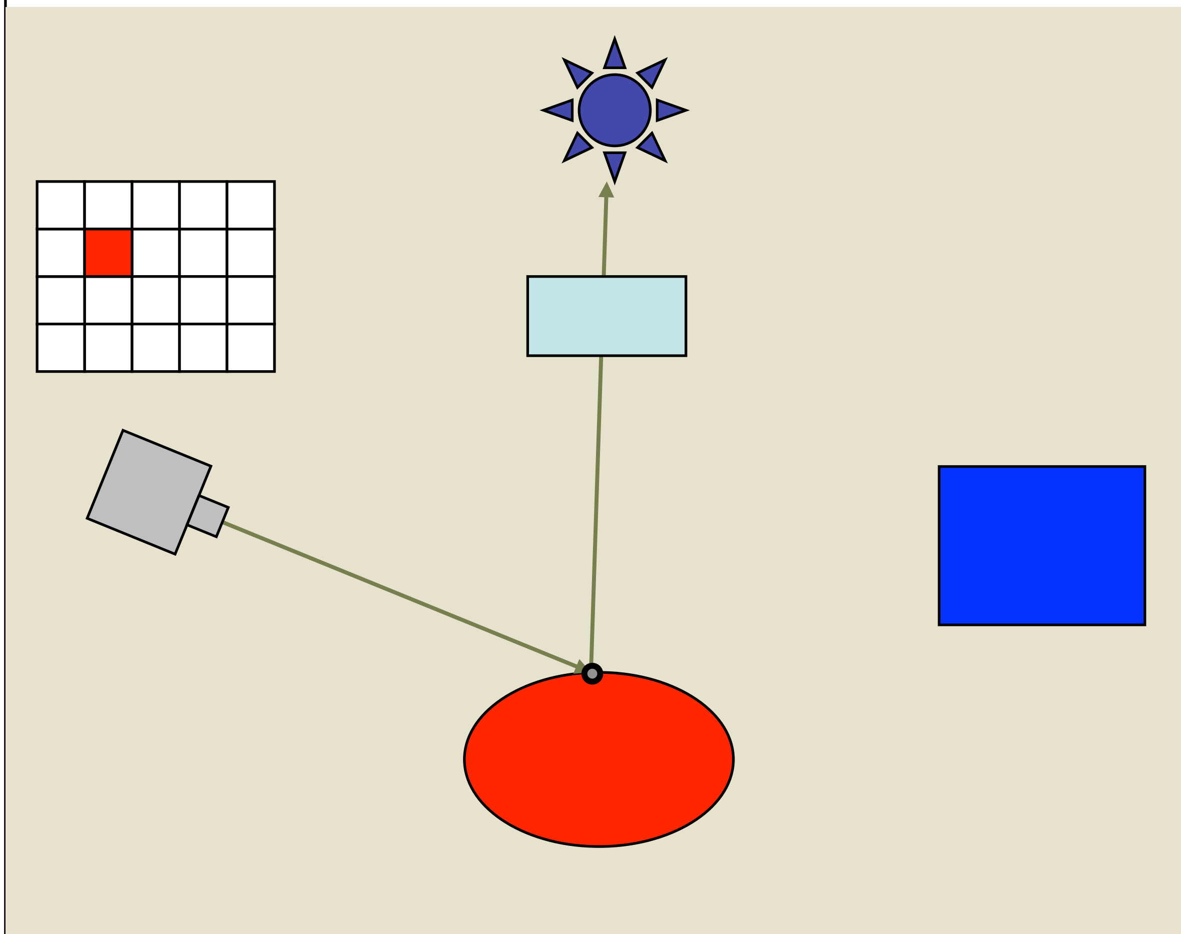




What is Ray Tracing?



What is Ray Tracing?



Ray-Generation

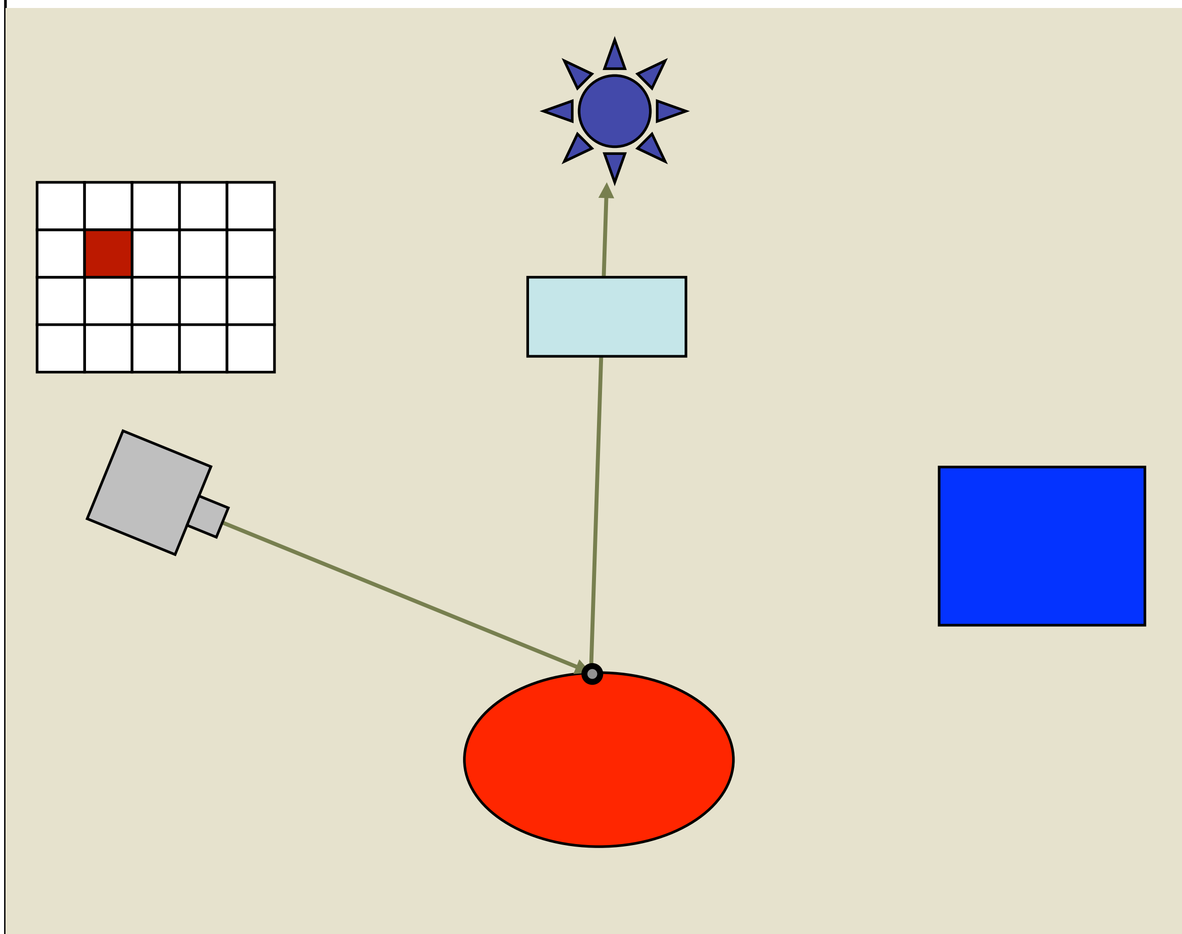
Ray-Traversal

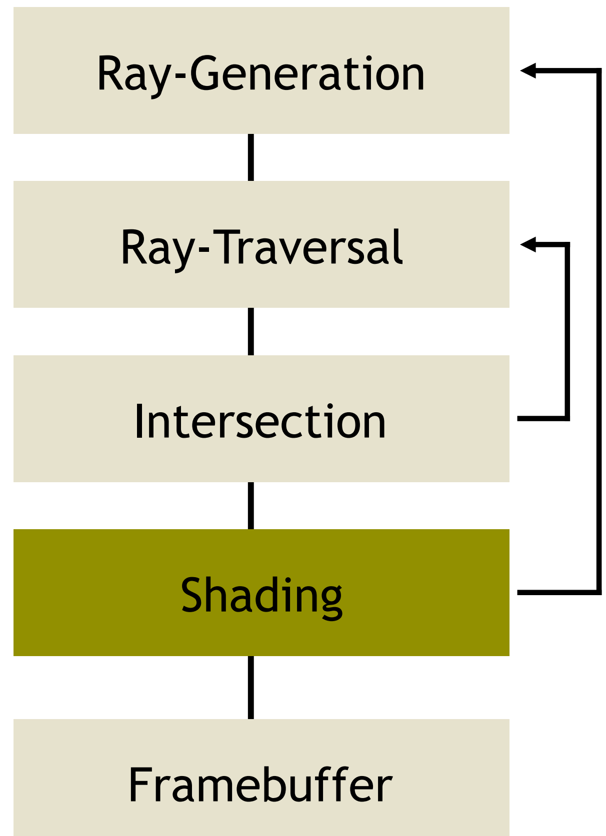
Intersection

Shading

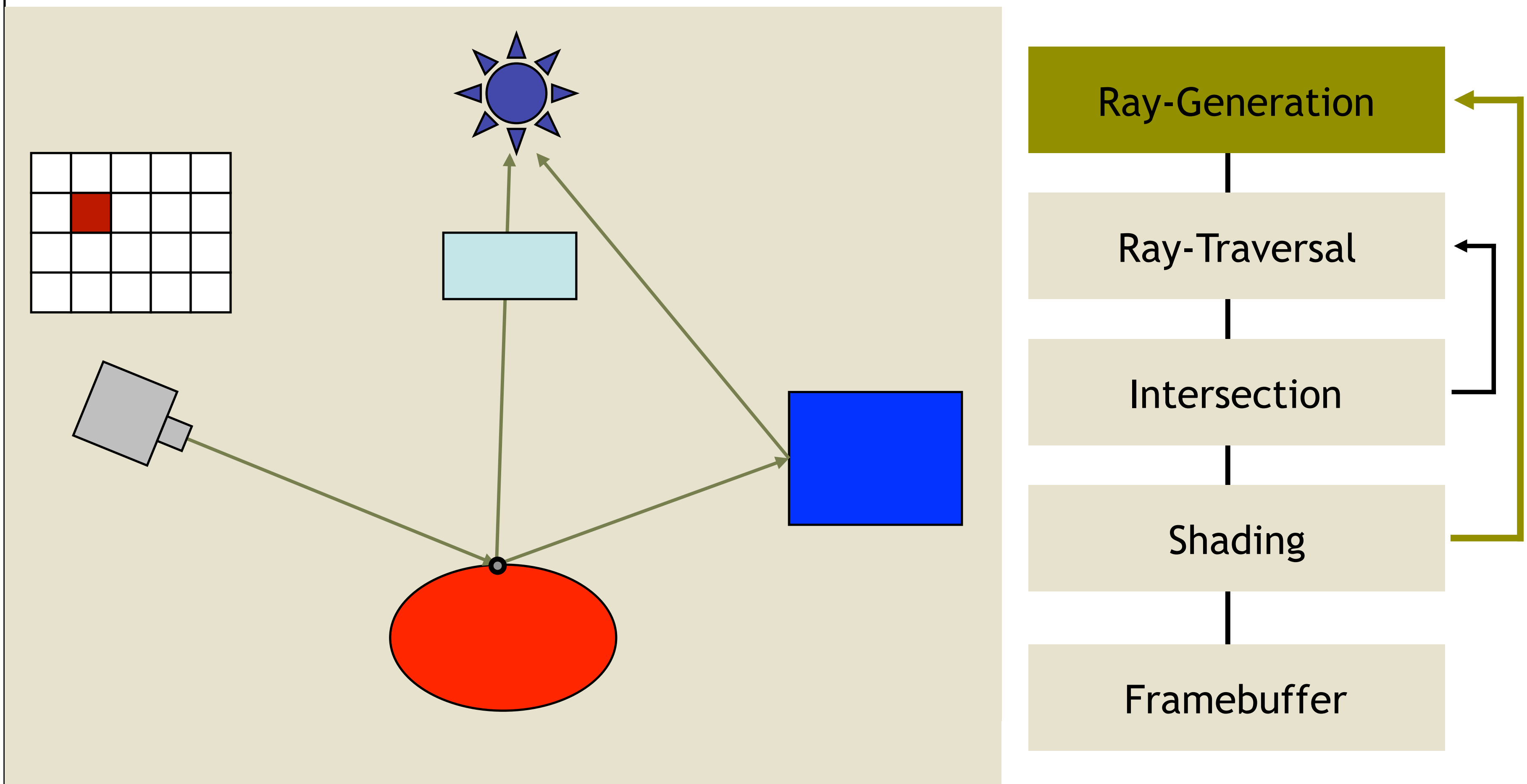
Framebuffer

What is Ray Tracing?

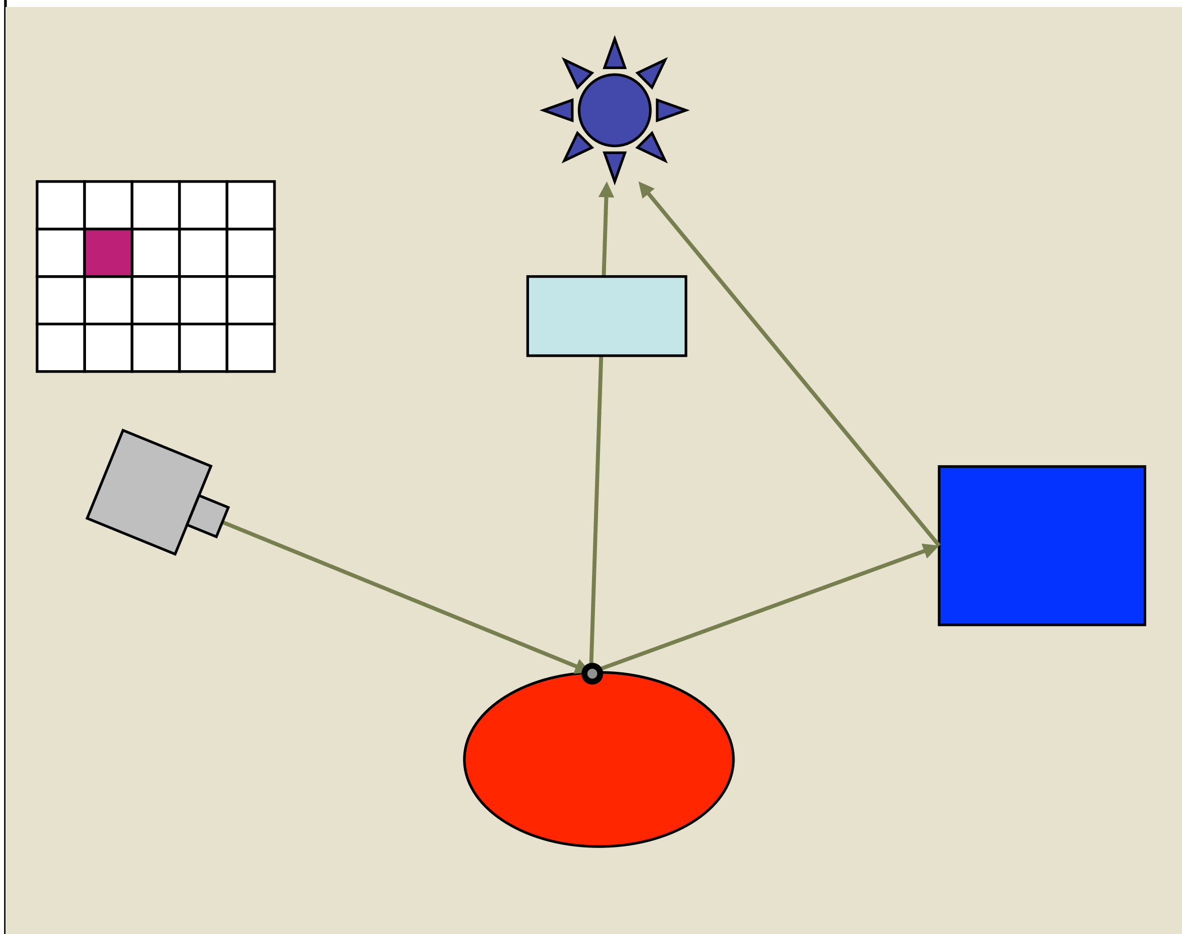


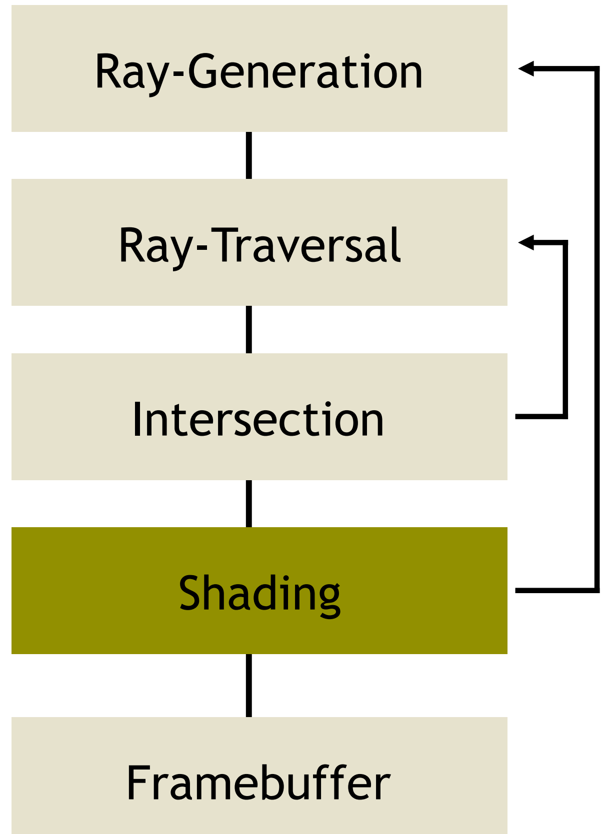


What is Ray Tracing?

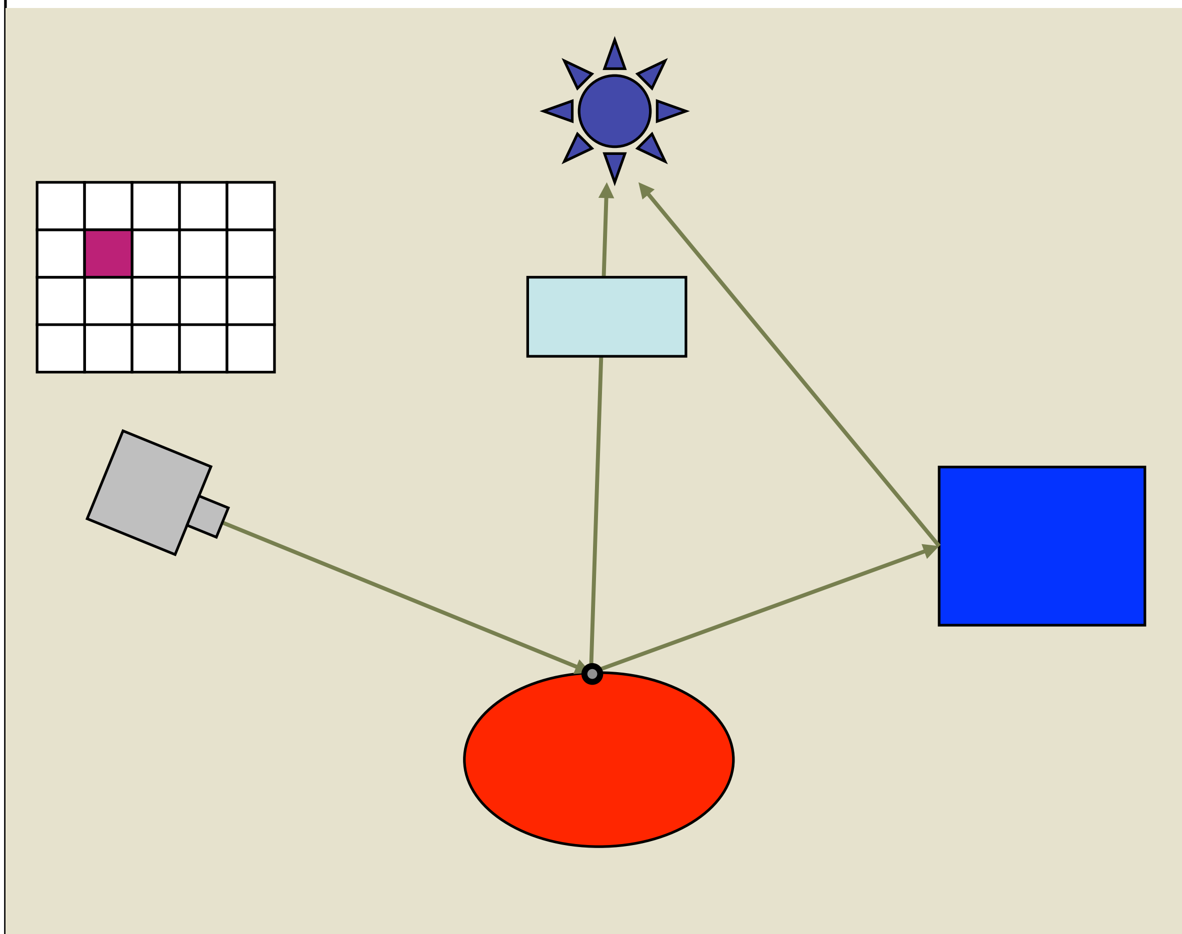


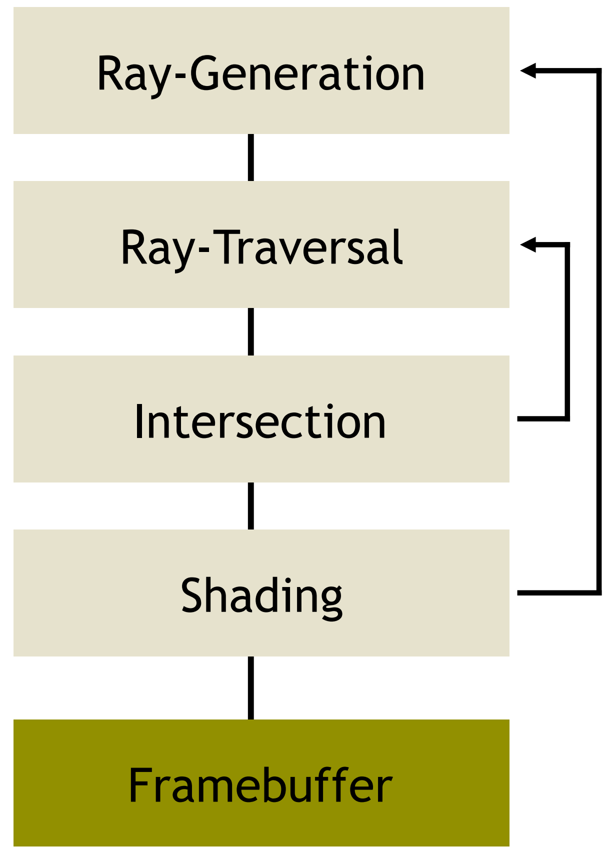
What is Ray Tracing?



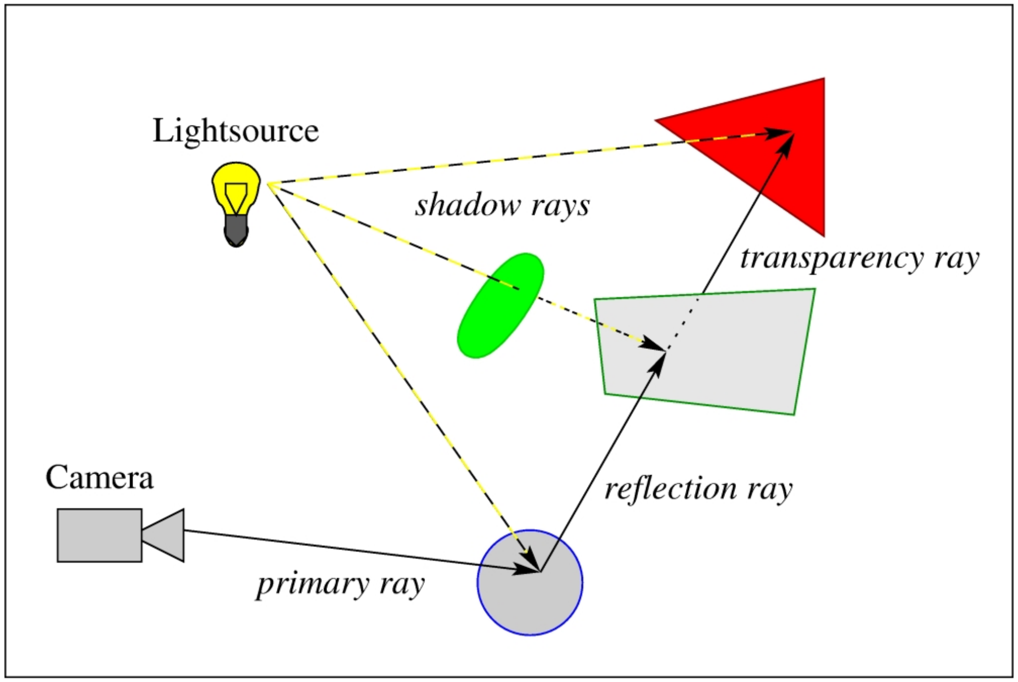


What is Ray Tracing?





What is Ray Tracing?



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

➔ Fundamental Technology for Next Generation Graphics

Rasterization vs. Ray Tracing

- **Rasterization**
  - For each triangle:
    - Find the pixels it covers
    - For each pixel: compare to closest triangle so far

Requires Z-buffer: track distance per pixel

- **Ray tracing**
  - For each pixel:
    - Find the triangles that might be closest
    - For each triangle: compute distance to pixel

Requires spatial index: a spatially sorted arrangement of triangles

Rasterization vs. Ray Tracing

- **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

Rasterization vs. Ray Tracing

- **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

Rasterization vs. Ray Tracing

- **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, …

Rasterization vs. Ray Tracing

- **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

Rasterization vs. Ray Tracing

- **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

Rasterization vs. Ray Tracing

- **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**

Real-time Ray Tracing

- **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

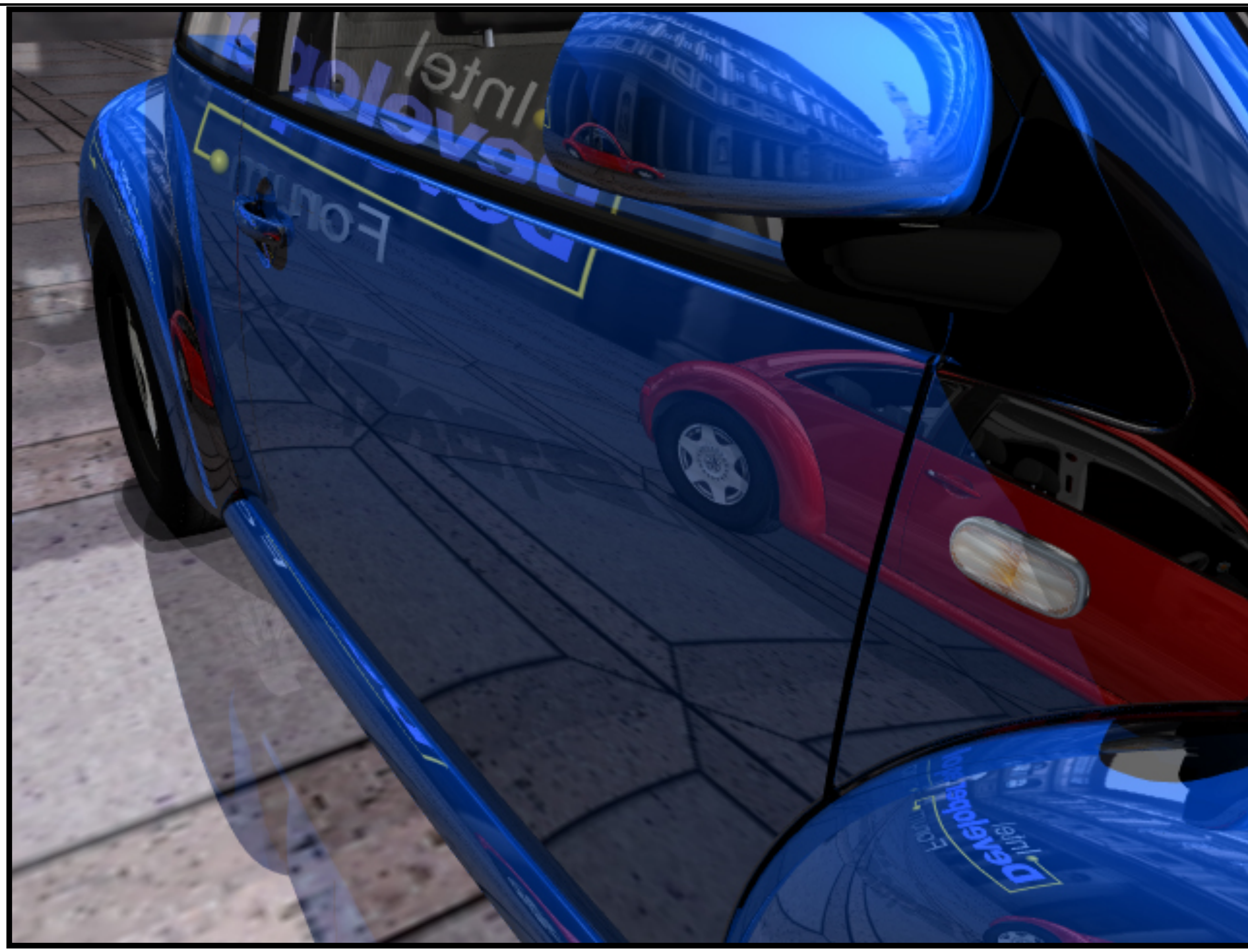
Reasons for Using RTRT

- **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
  - Realtime global illumination

Highly Realistic Images

- **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

Highly Realistic Images



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

Reasons for Using Ray Tracing

- **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

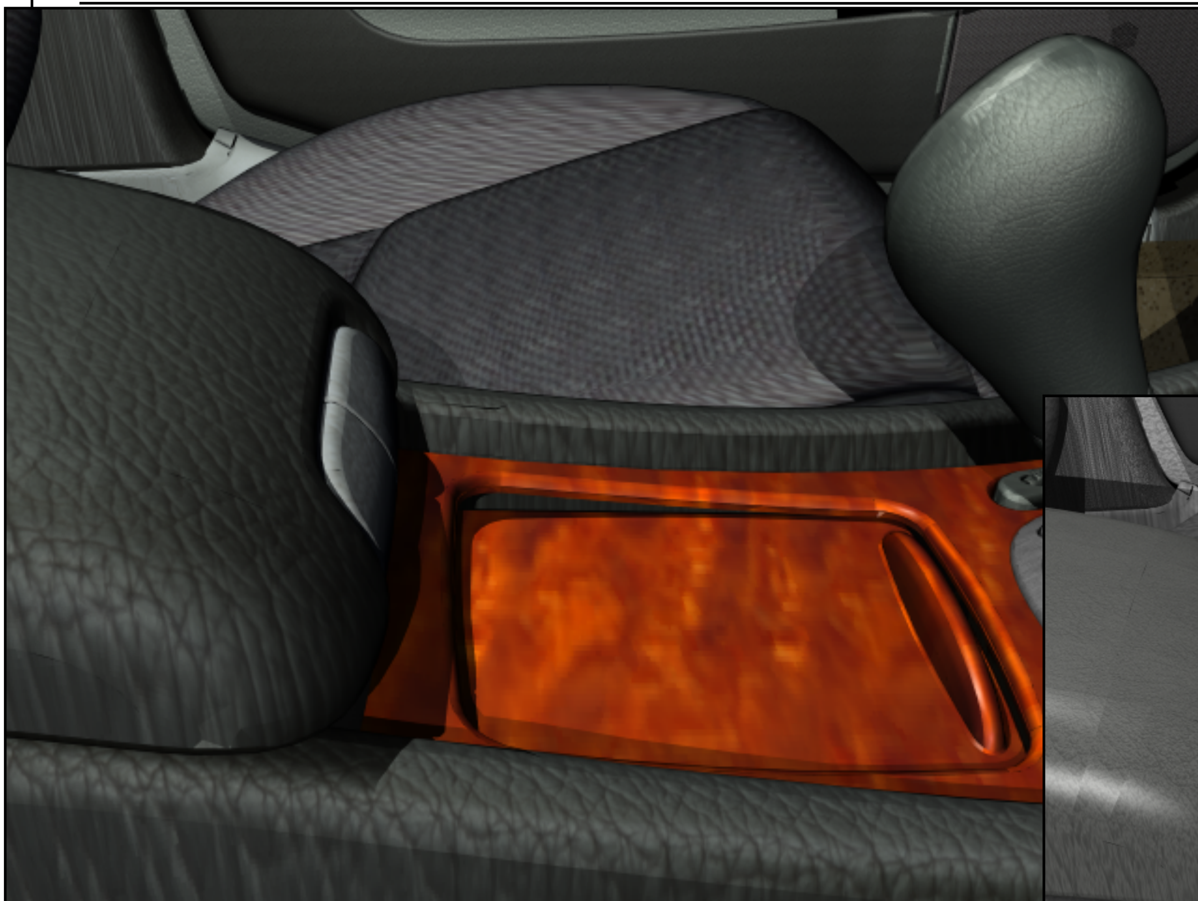
Reasons for Using RTRT: Physical Correctness

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

Physical Correctness

Rendered directly from trimmed NURBS surfaces, with smooth environment lighting

Reasons for Using RTRT: Physical Correctness

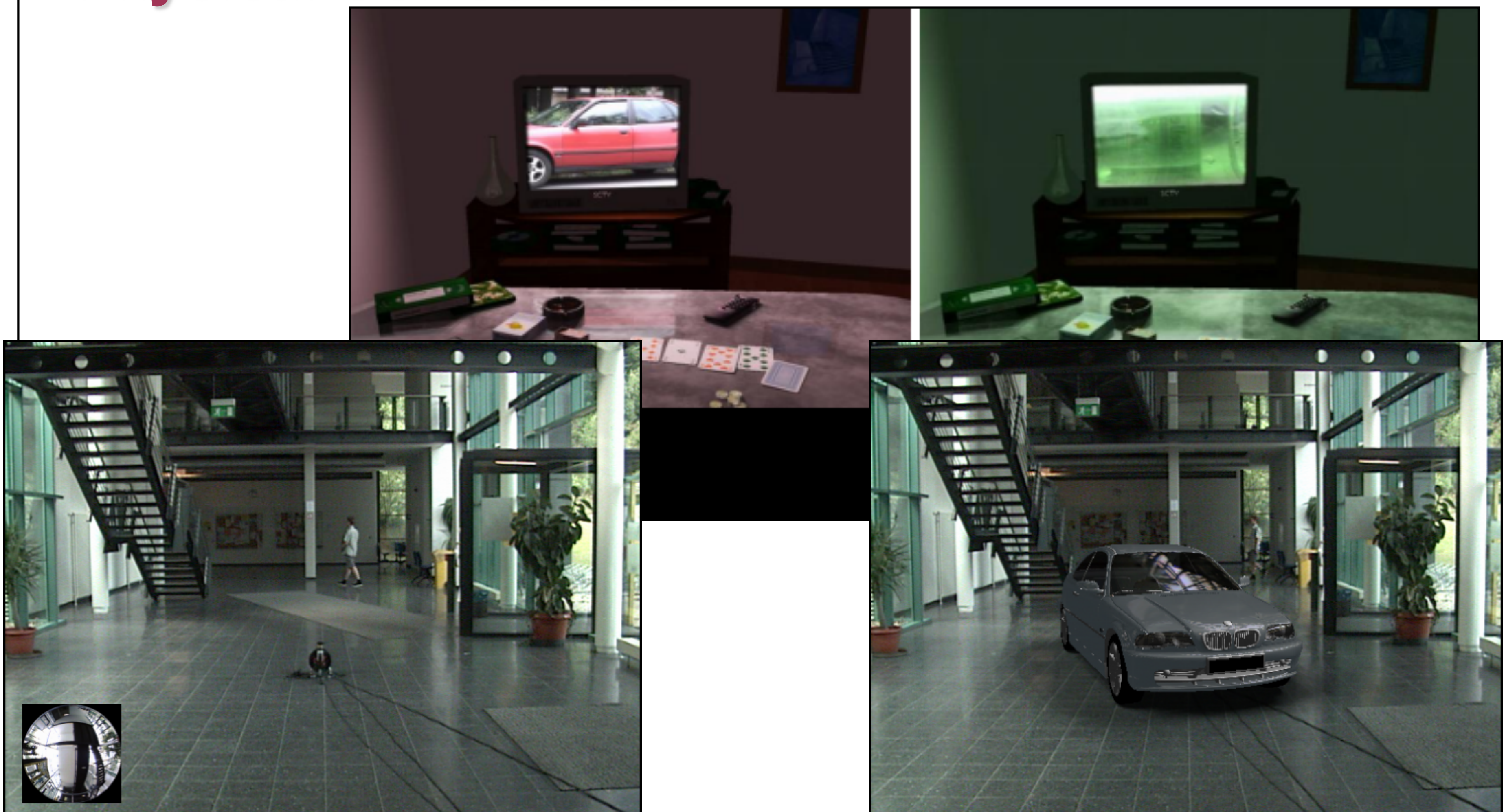


Textured Phong for comparison

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

BTF Data Courtesy R. Klein, Uni Bonn

Reasons for Using RTRT: Physical Correctness

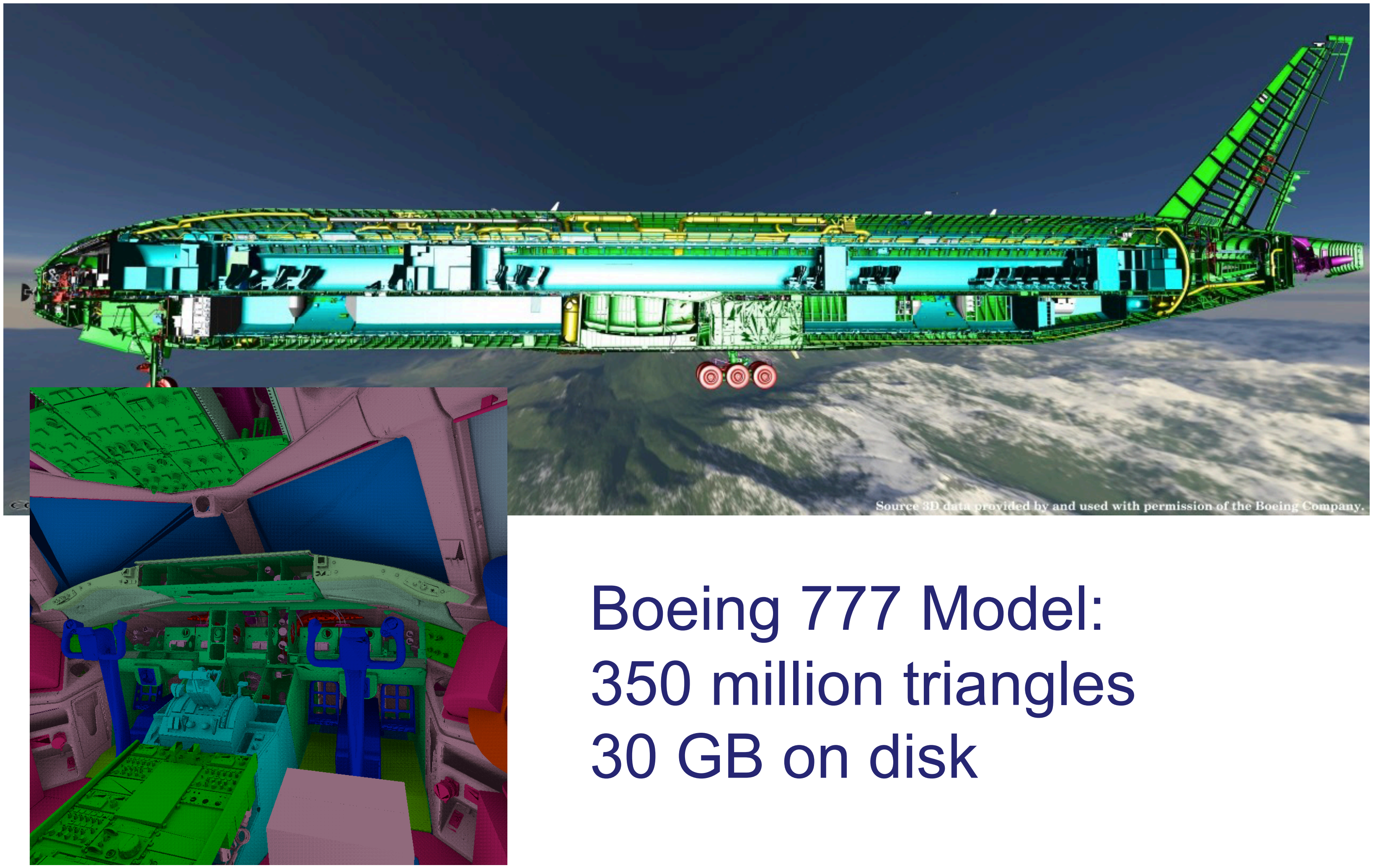


VR scene illuminated from realtime video feed, AR with realtime environment lighting

Reasons for Using RTRT: Massive Models

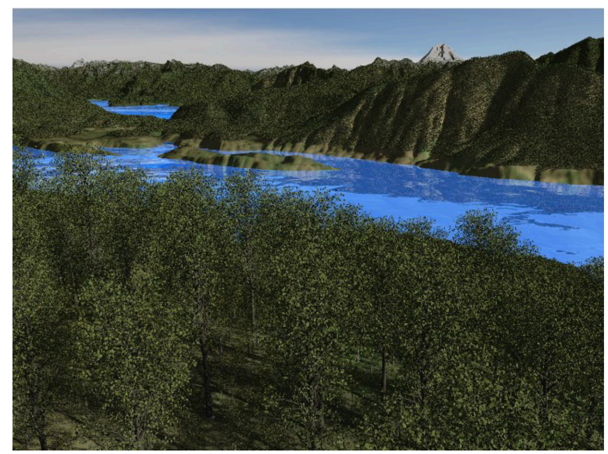
- **Massive Scenes**
  - Scales logarithmically with scene size
  - Supports billions of triangles
- **Benefits**
  - Can render entire CAD models without simplification
  - Greatly simplifies and speeds up many tasks

Reasons for Using RTRT: Massive Models



Boeing 777 Model:
350 million triangles
30 GB on disk

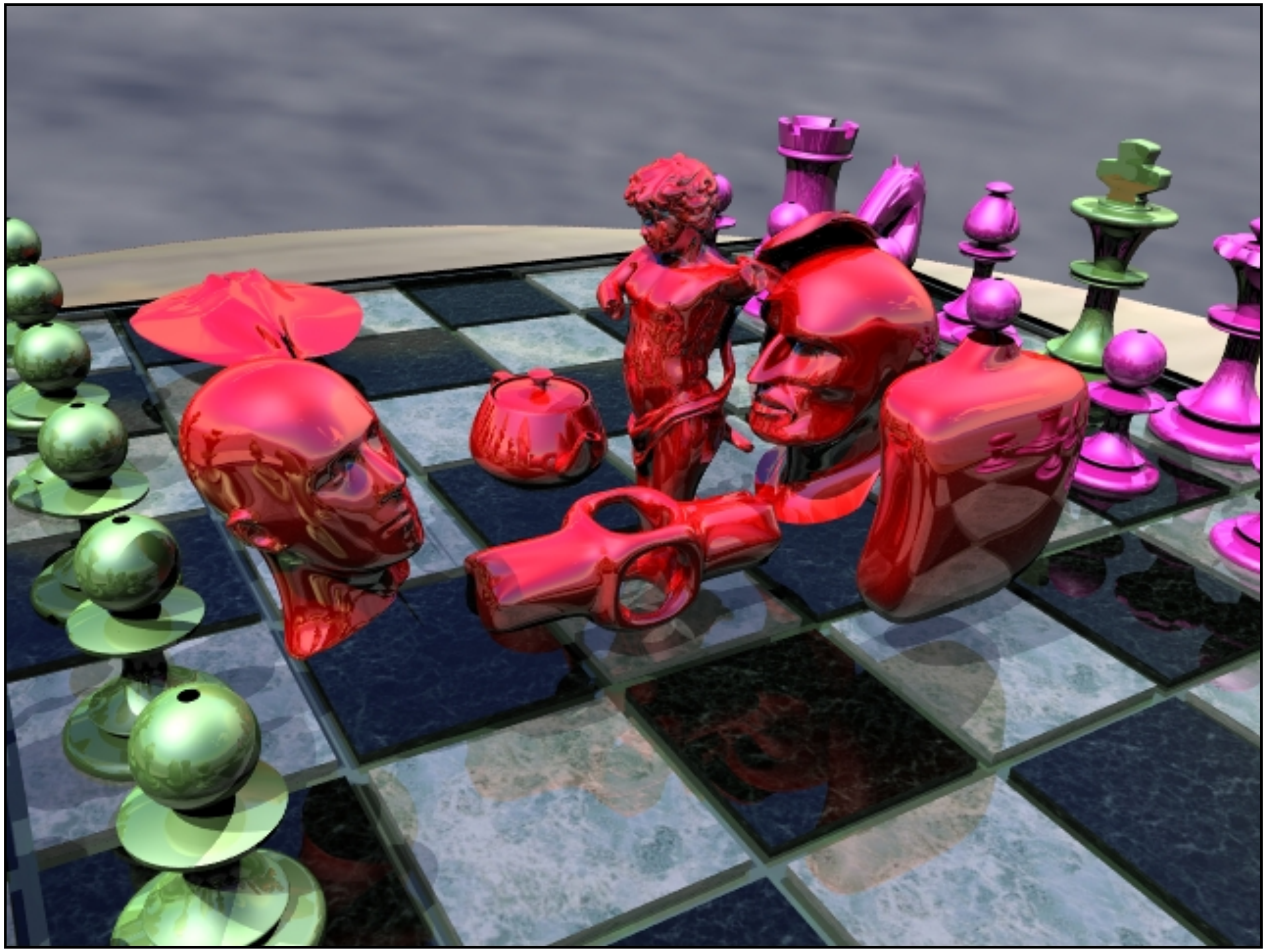
Reasons for Using RTRT: Massive Models



Using RTRT: Flexible Primitive Types

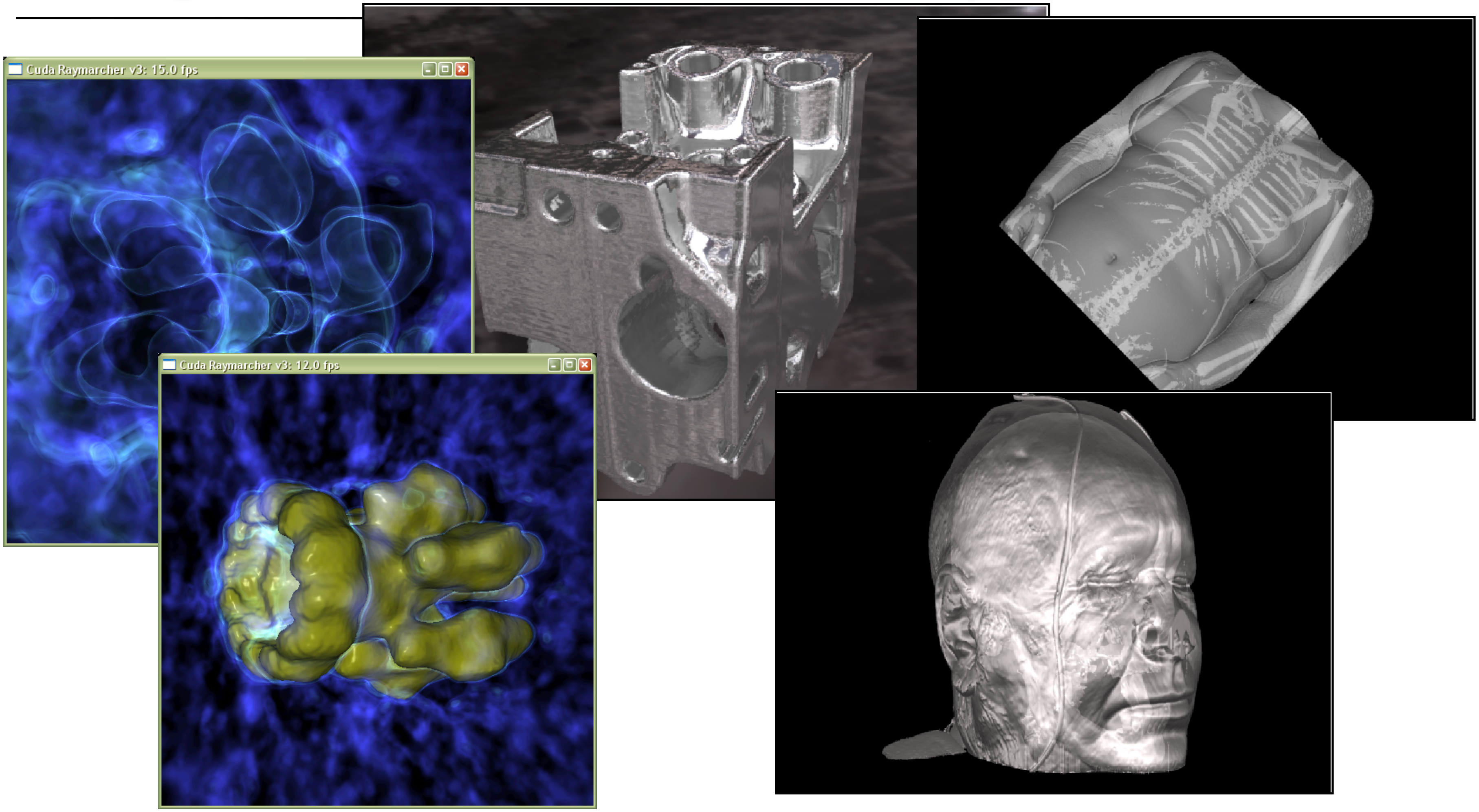
- **Flexible Primitive Types**
  - Triangles
  - Volumes data sets
    - Iso-surfaces & direct visualization
    - Regular, rectilinear, curvilinear, unstructured, …
  - Splines and subdivision surfaces
  - Points

Using RTRT: Flexible Primitive Types



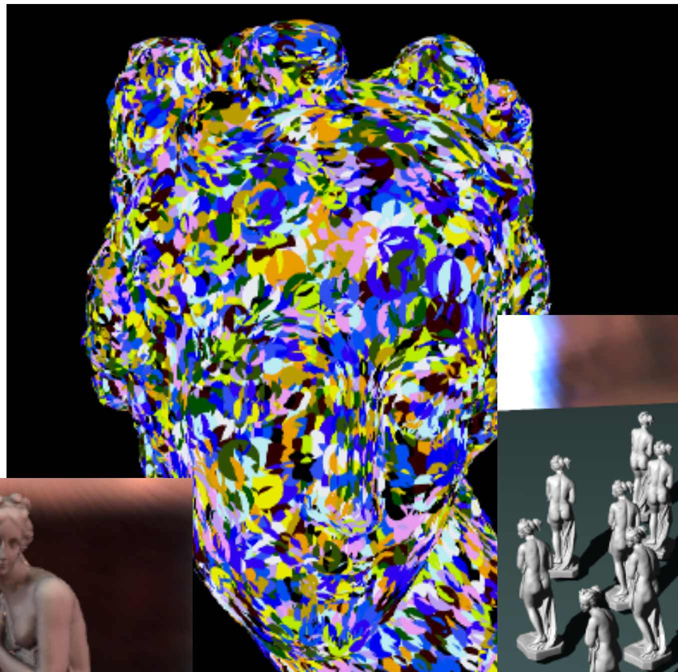
Triangles, Bezier splines, and subdivision surfaces fully integrated

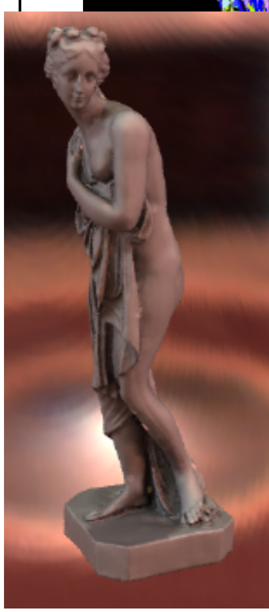
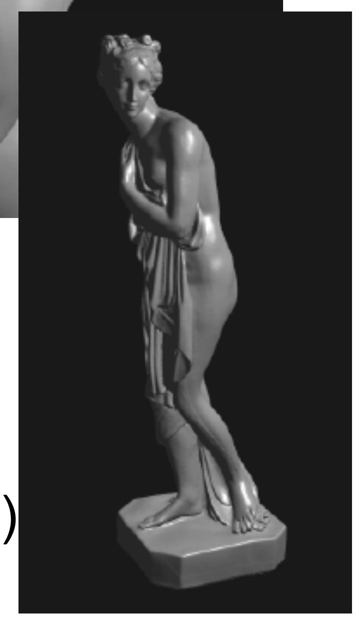
Using RTRT: Flexible Primitive Types



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

Using RTRT: Flexible Primitive Types



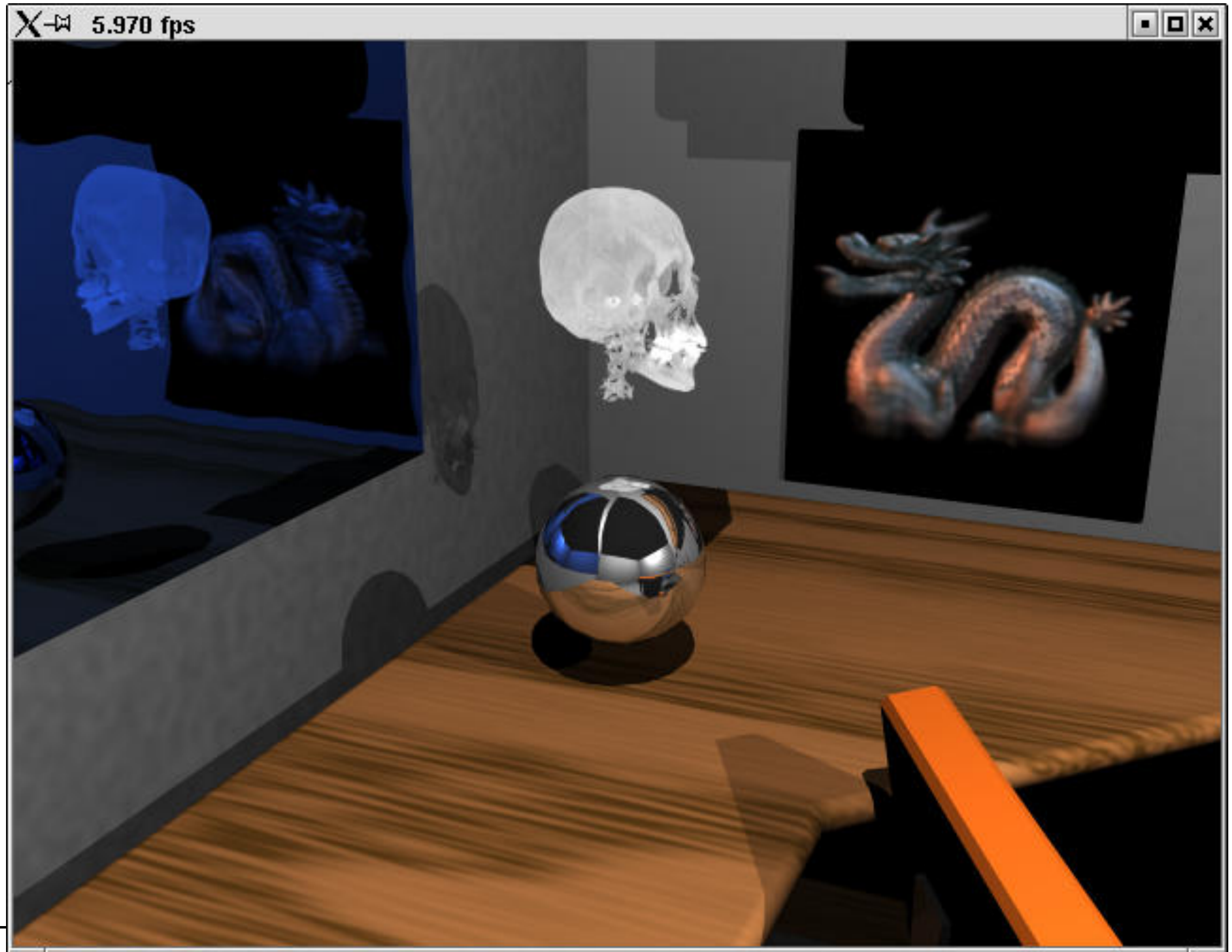
24 MPoints,

Realtime ray tracing of point clouds (1 Mpoints each)

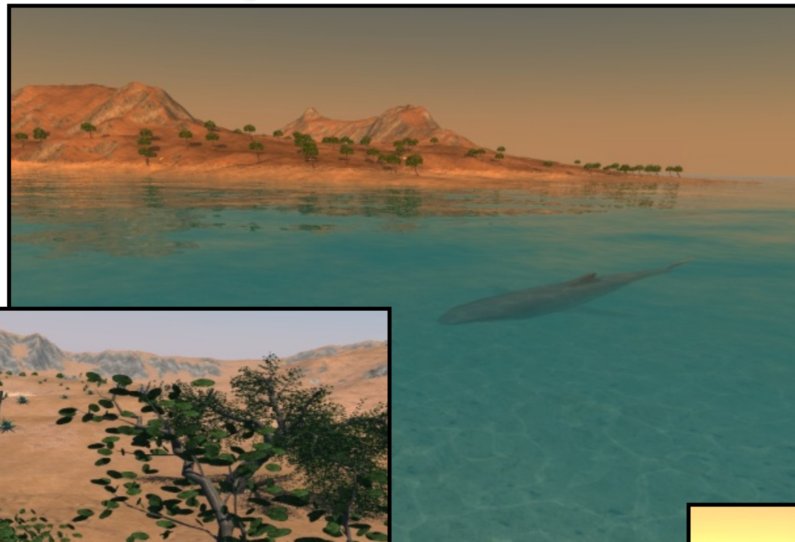
Reasons for Using RTRT: Declarative Graphics

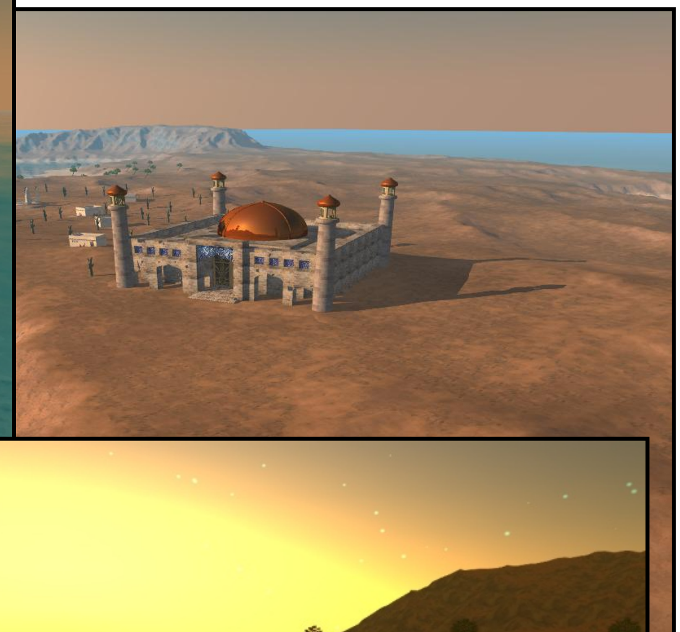
- **Declarative Graphics Interface**
  - Application specifies scene once, plus updates
  - Rendering fully performed by renderer (e.g. in HW)
  - Similar to scene graphs, PostScript, or latest GUIs
- **Benefits**
  - Greatly simplifies application programming
  - Allows for complete HW acceleration

Reasons for Using RTRT: Declarative Graphics



Reasons for Using RTRT: Declarative Graphics





Reasons for Using RTRT: Global Illumination

- **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

Reasons for Using RTRT: Global Illumination

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

Open Issues with Real-time Ray Tracing

- **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)

Open Issues with Real-time Ray Tracing

- **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

Open Issues with Real-time Ray Tracing

- **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: very promising with Fermi
  - Custom HW: RPU (flexible GPU + custom traversal)