Depth Images

Sprites, Layers and Trees

Before we begin...

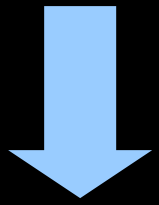
A quick plug for Image-Based Rendering:

“A traditional Z-buffer algorithm ... will have to take the time to render every polygon of every object in every drawer of every desk in a building even if the whole building cannot be seen”

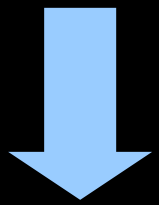
– Greene, Kass and Miller, SIGGRAPH '93

Outline

- Depth Sprites [Shade et al. '98]



- Layered Depth Images [Shade et al. '98]
  - The k-Buffer [Callahan '05]



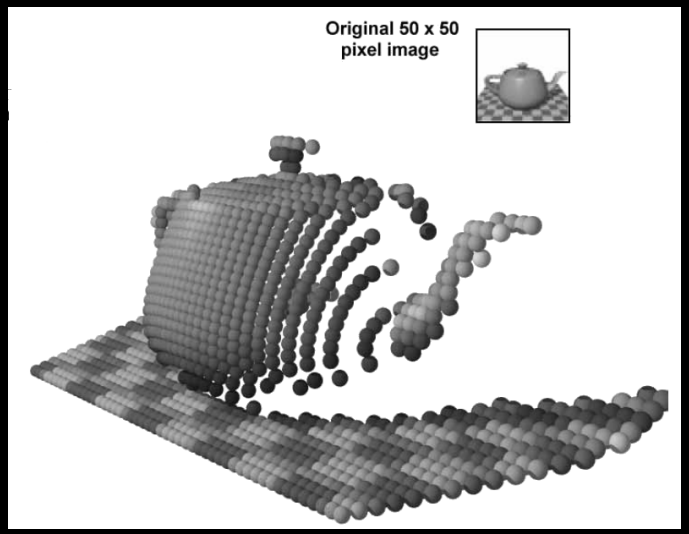
- LDI Trees [Chang et al. '99]

Just for reference...

- [Shade et al. '98] Jonathan Shade, Steven Gortler, Li-wei He, Richard Szeliski, "*Layered Depth Images*", SIGGRAPH 1998.
- [Callahan '05] Steven P. Callahan, "*The k-Buffer and Its Applications to Volume Rendering*", M. S. Thesis, 2005.
  - Extensions: [Bavoil et al. '07] Louis Bavoil et al., "*Multi-Fragment Effects on the GPU using the k-Buffer*", I3D 2007.
- [Chang et al. '99] Chun Chang, Gary Bishop, Anselmo Lastra, "*LDI Tree: A Hierarchical Representation for Image-Based Rendering*", SIGGRAPH 1999.

Terminology

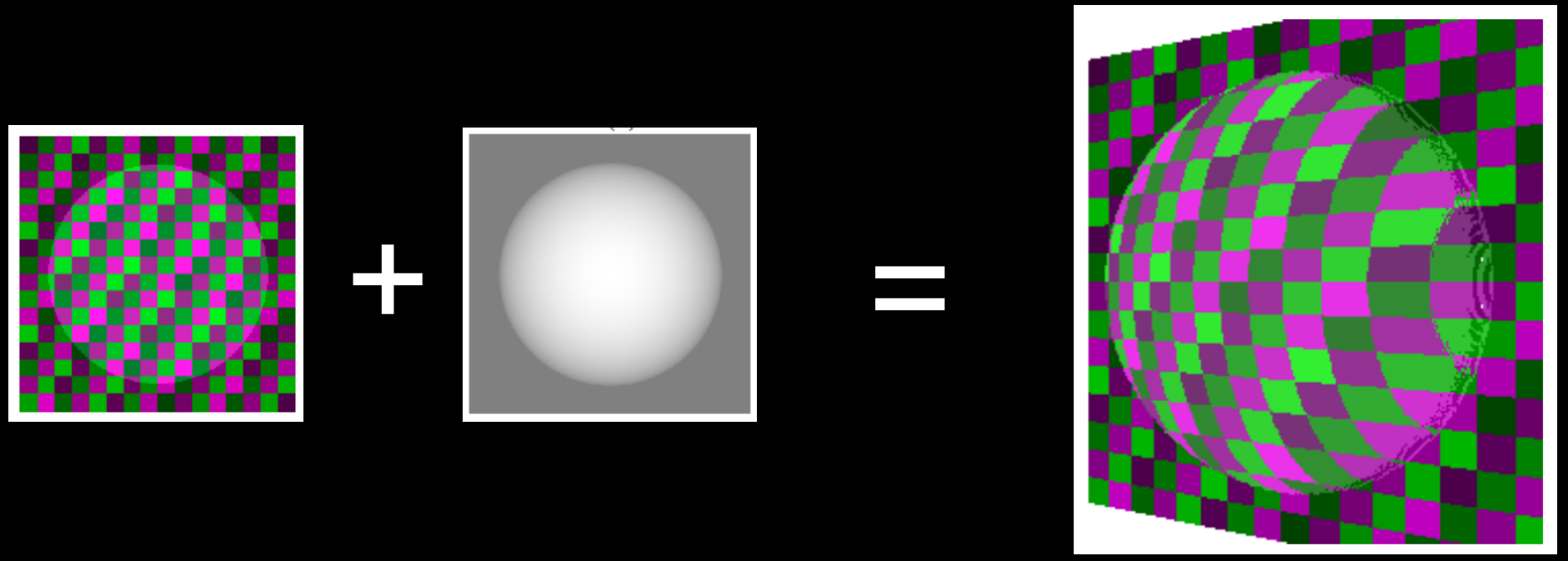
- Surfel: Surface Element
  (like pixel = picture element)



- Splatting: Drawing a surfel on the screen
  - Simplest: draw a 1 pixel point
  - More refined: Draw a quad/circle, with size attenuated by distance to account for perspective projection

Depth Sprites

Impostor with depth channel (for correct parallax)

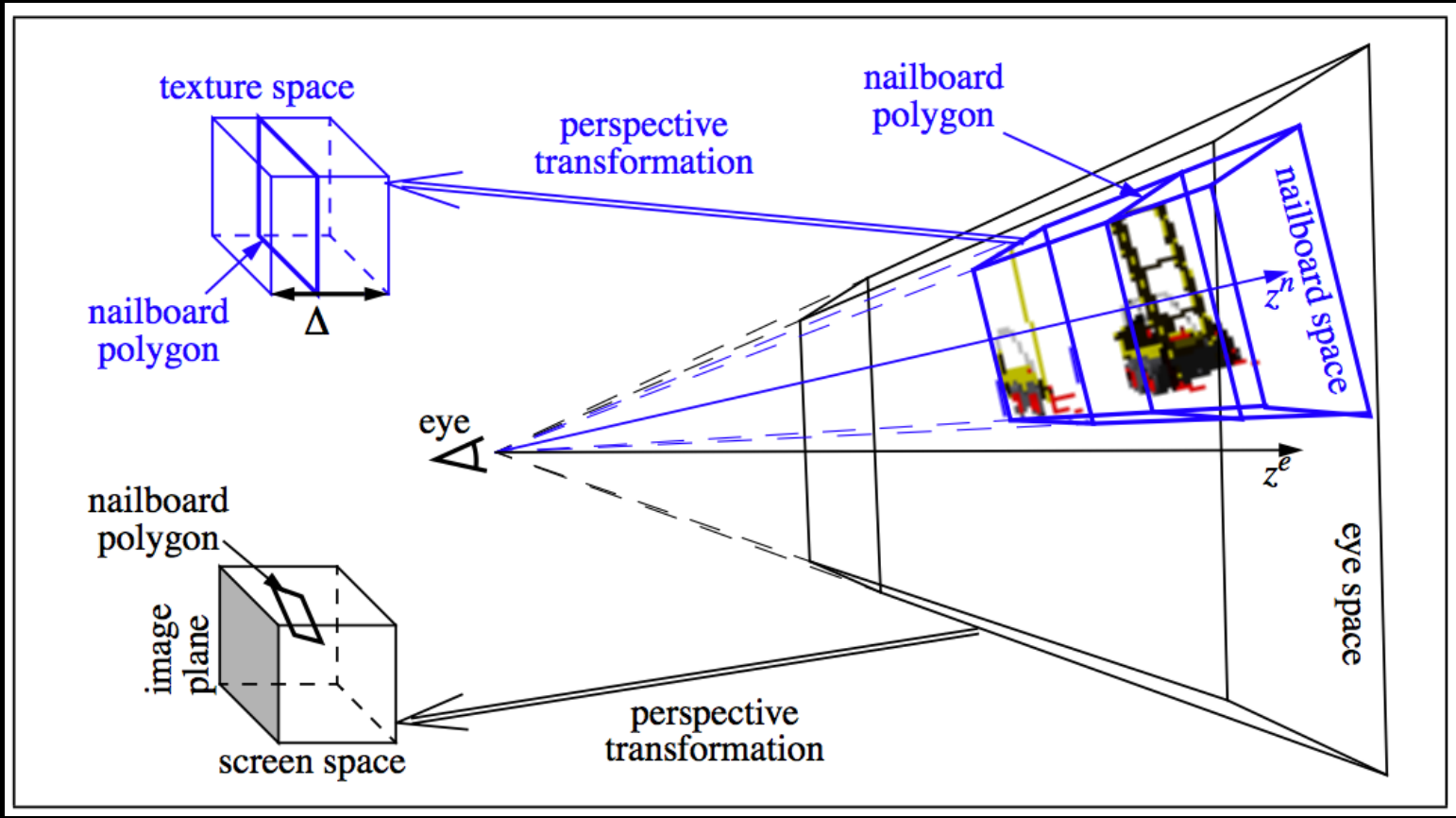


Question of the Day...

- Depth images (a.k.a. RGBZ images) have been known for a long time (e.g. Greene and Kass '93, Chen and Williams '93)...
- ... so what makes depth sprites different?

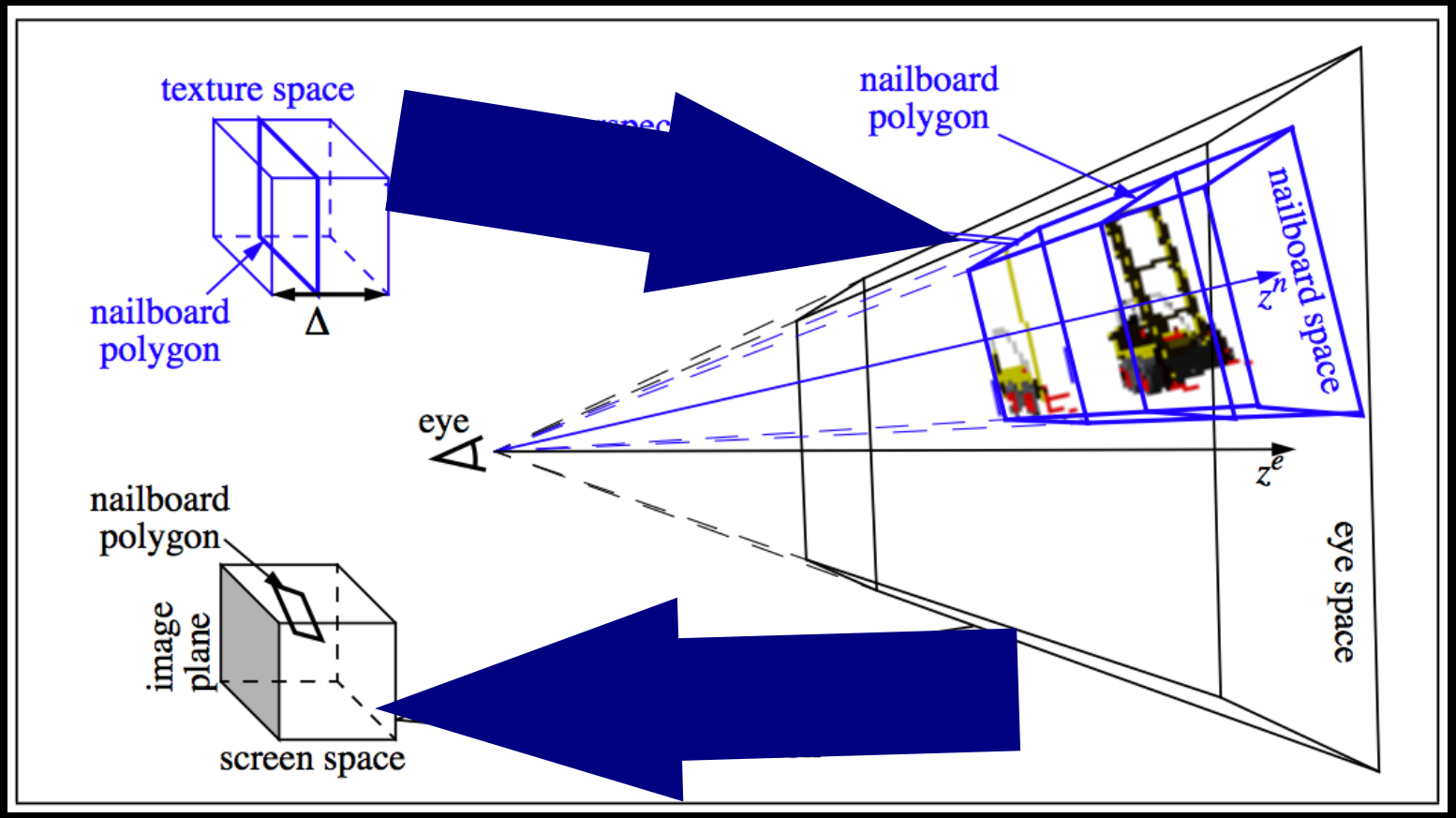
- A. Depth sprites represent single objects (like traditional sprites) and assume a smooth, continuous surface. This allows a particular hole-filling paradigm to be applied.

Depth Sprites



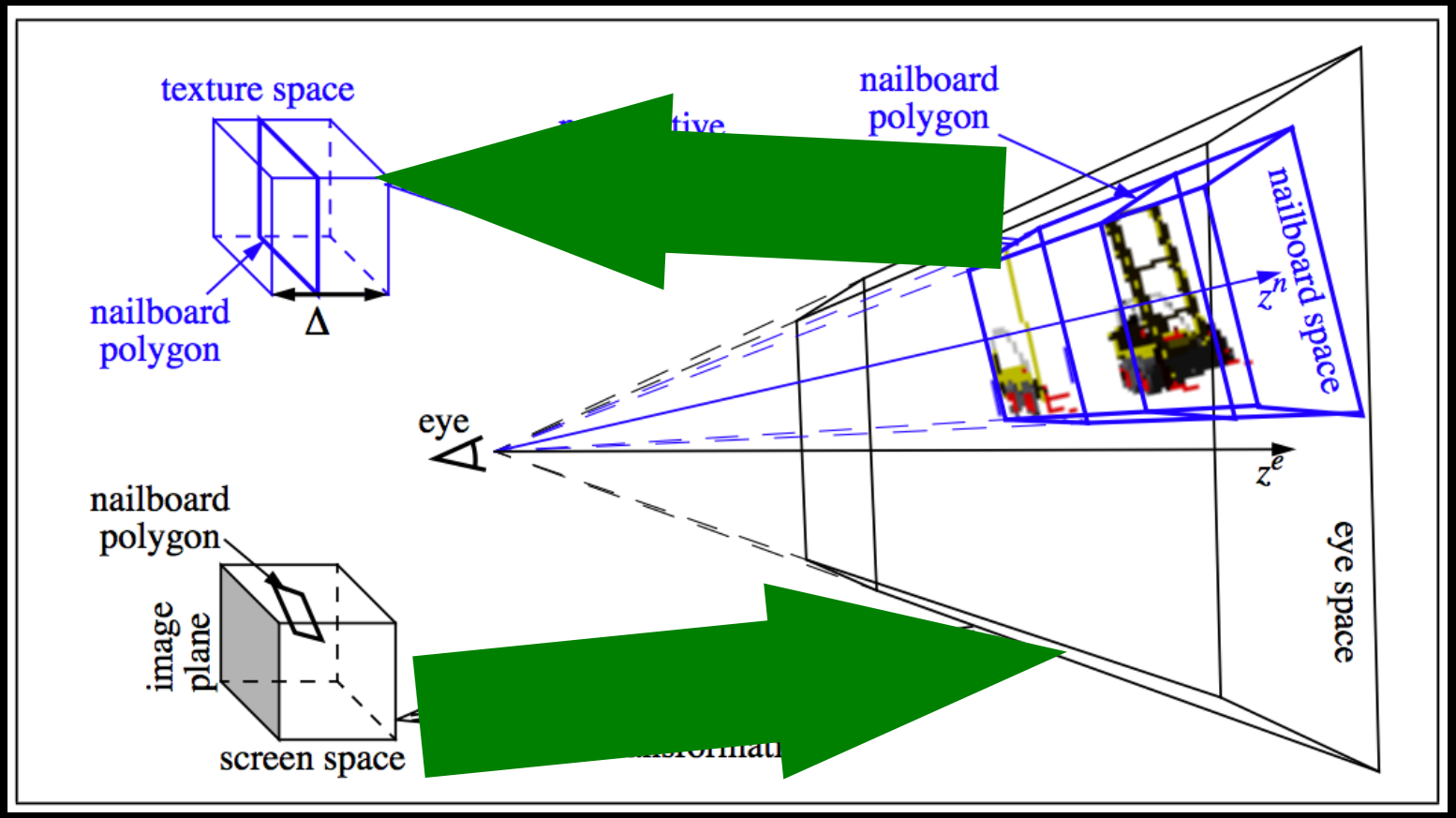
[Schaufler '97] Gernot Schaufler, "*Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes*", Eurographics Rendering Workshop 1997.

Depth Sprites



Forward Mapping Surfels
(can create holes)

Depth Sprites



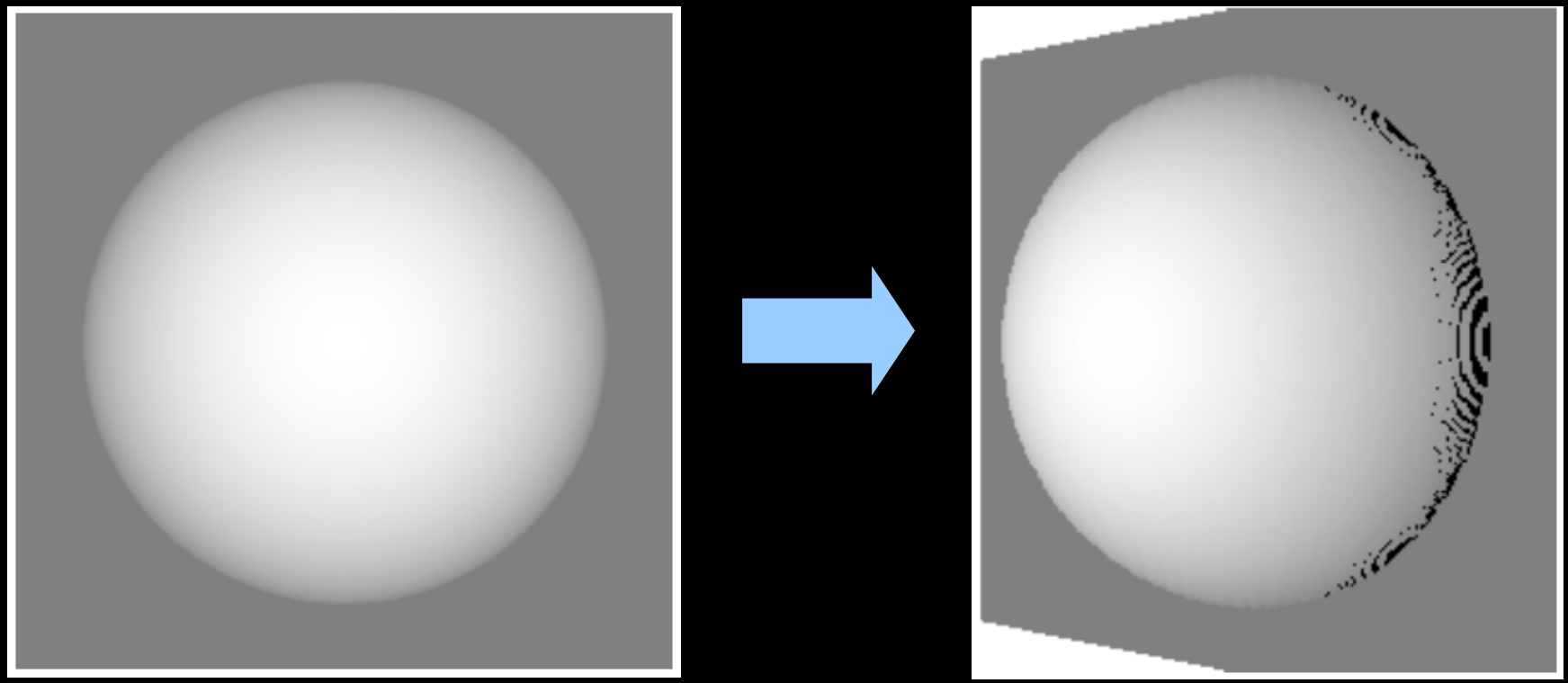
Backward Mapping Surfels (standard texture map)
Observe: Backward mapping creates no holes

Depth Sprites

- Naïve Rendering:
  - Forward map each surfel to output image plane
  - Problem:
    - Lots of disocclusion, undersampling artifacts, difficult to fill holes after reprojection
- Two-Step Rendering:
  - Forward map only the depth component to an “intermediate space” without final camera projection
  - Fill holes
  - Backward map the color and depth channels from the quad in the output image plane

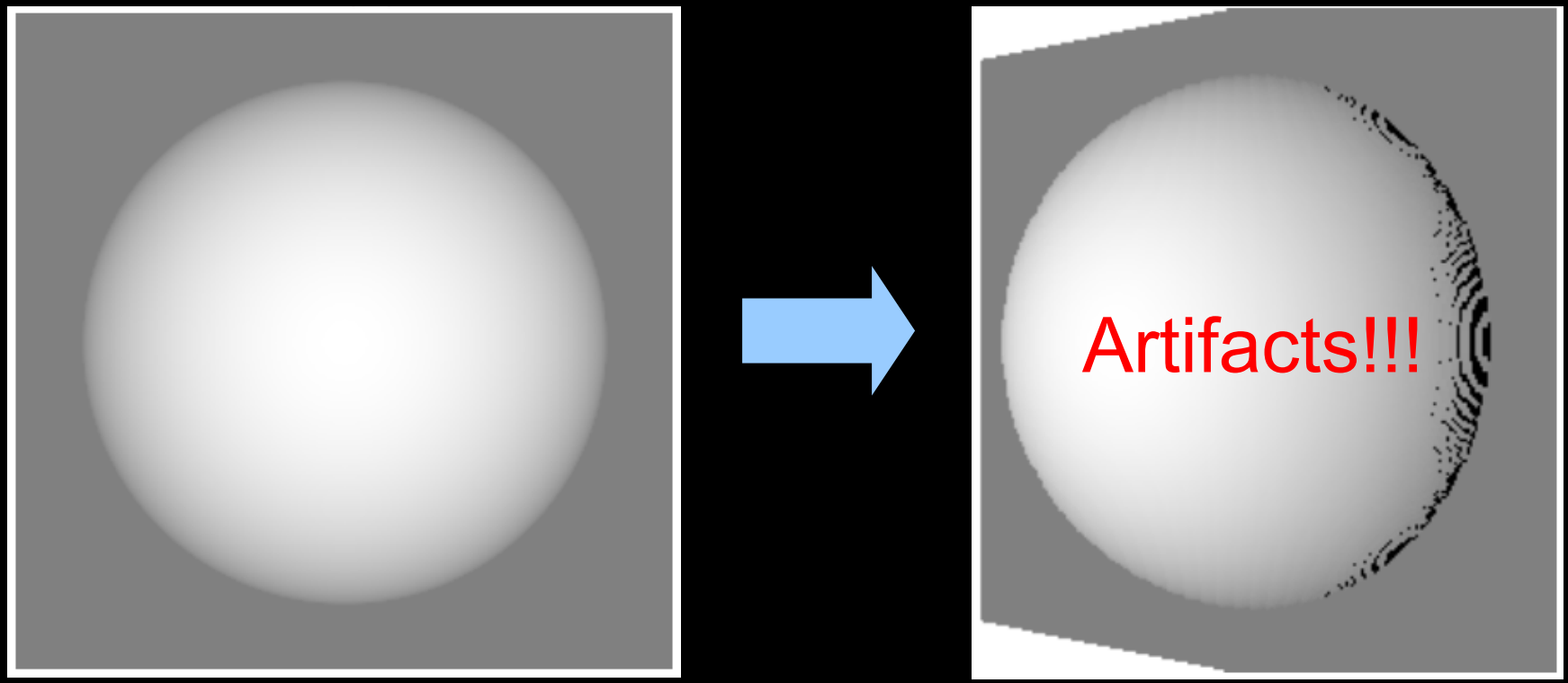
Depth Sprites

Result of forward warping depth map



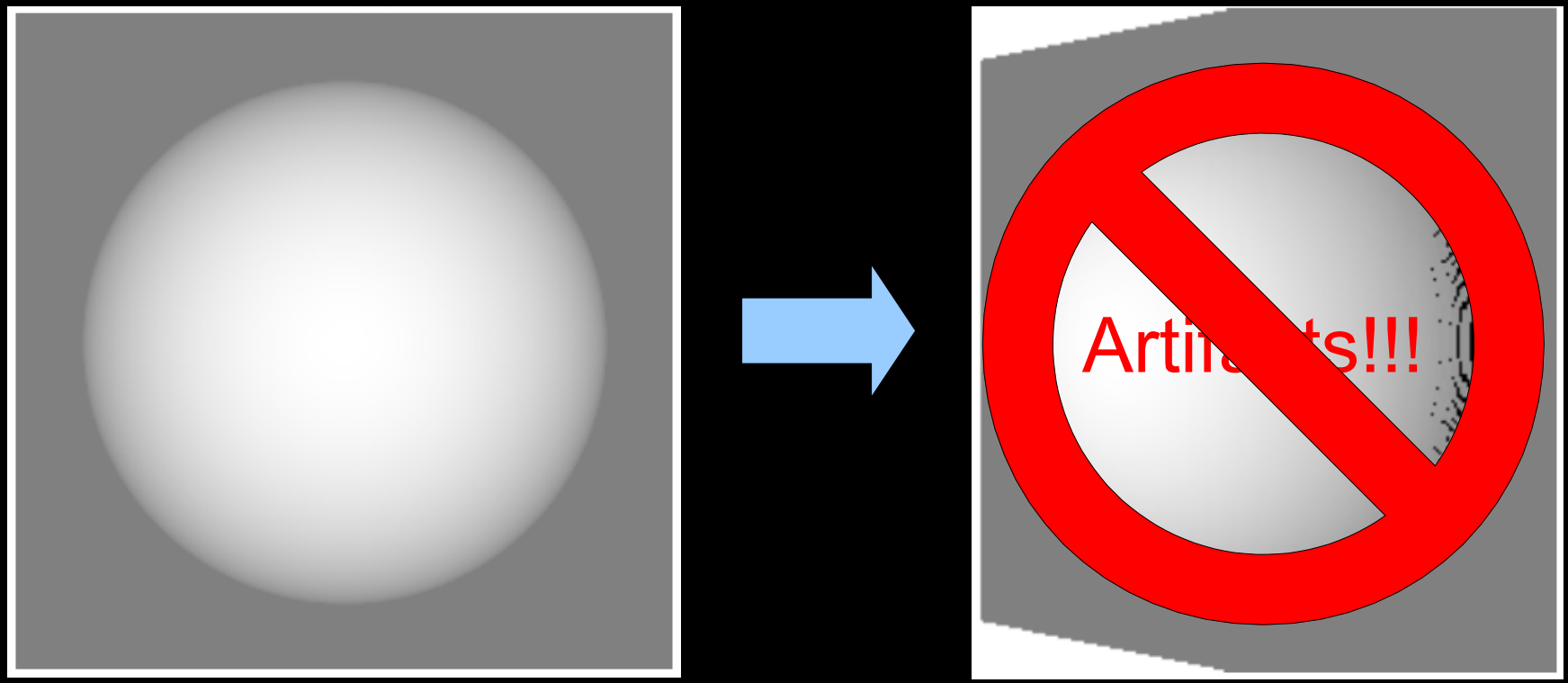
Depth Sprites

Result of forward warping depth map



Depth Sprites

Result of forward warping depth map



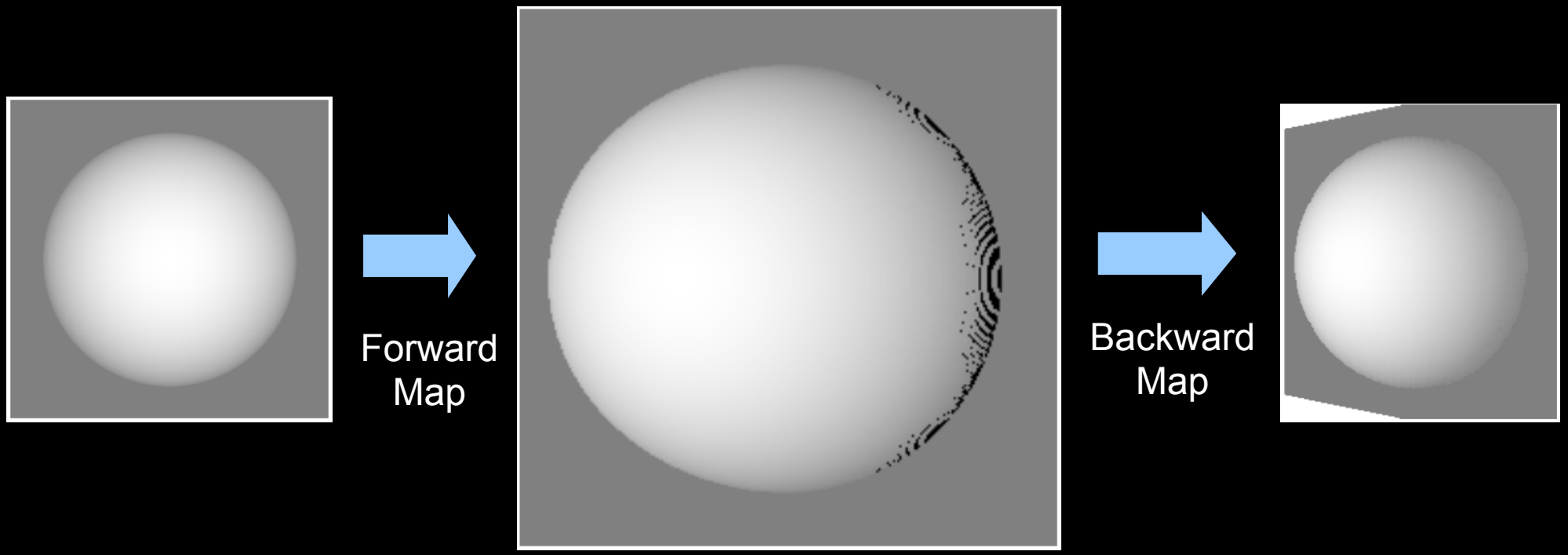
Depth Sprites

Intermediate Step

Depth Sprites

Intermediate Step

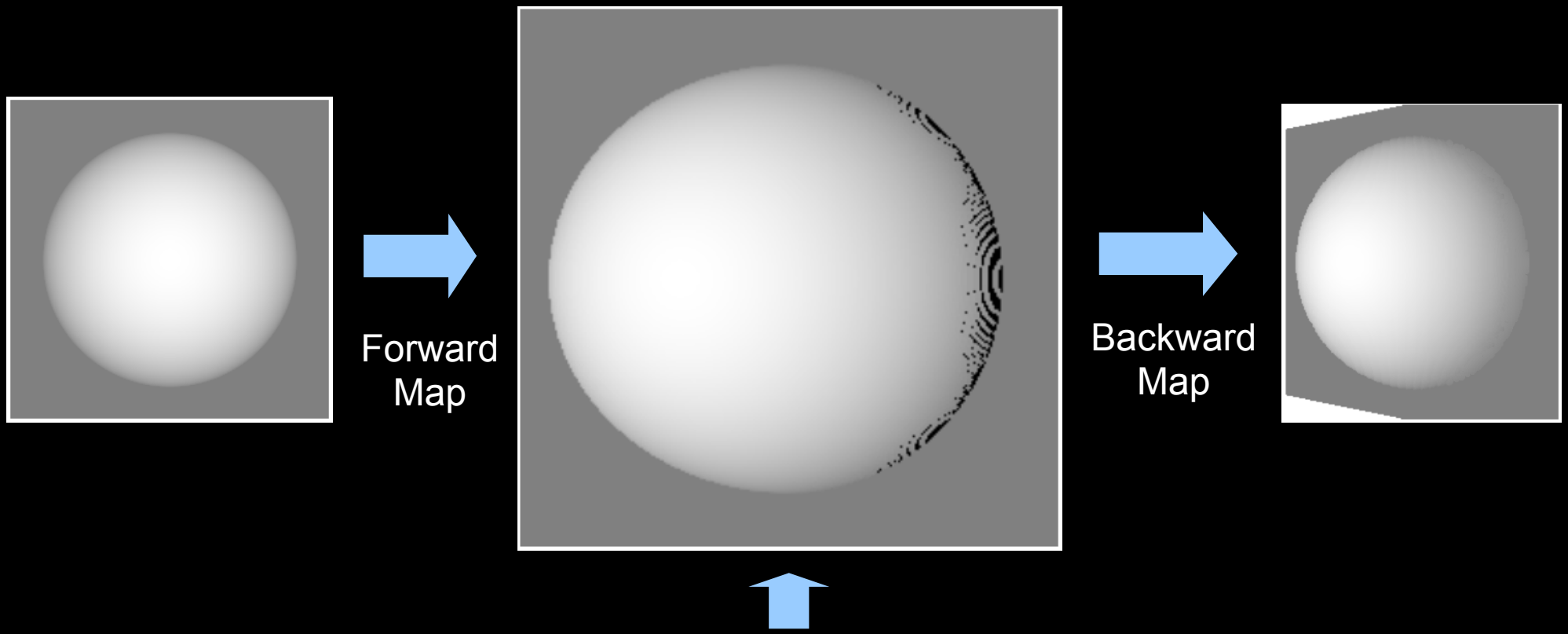
Forward map depth channel by “local parallax” only



Depth Sprites

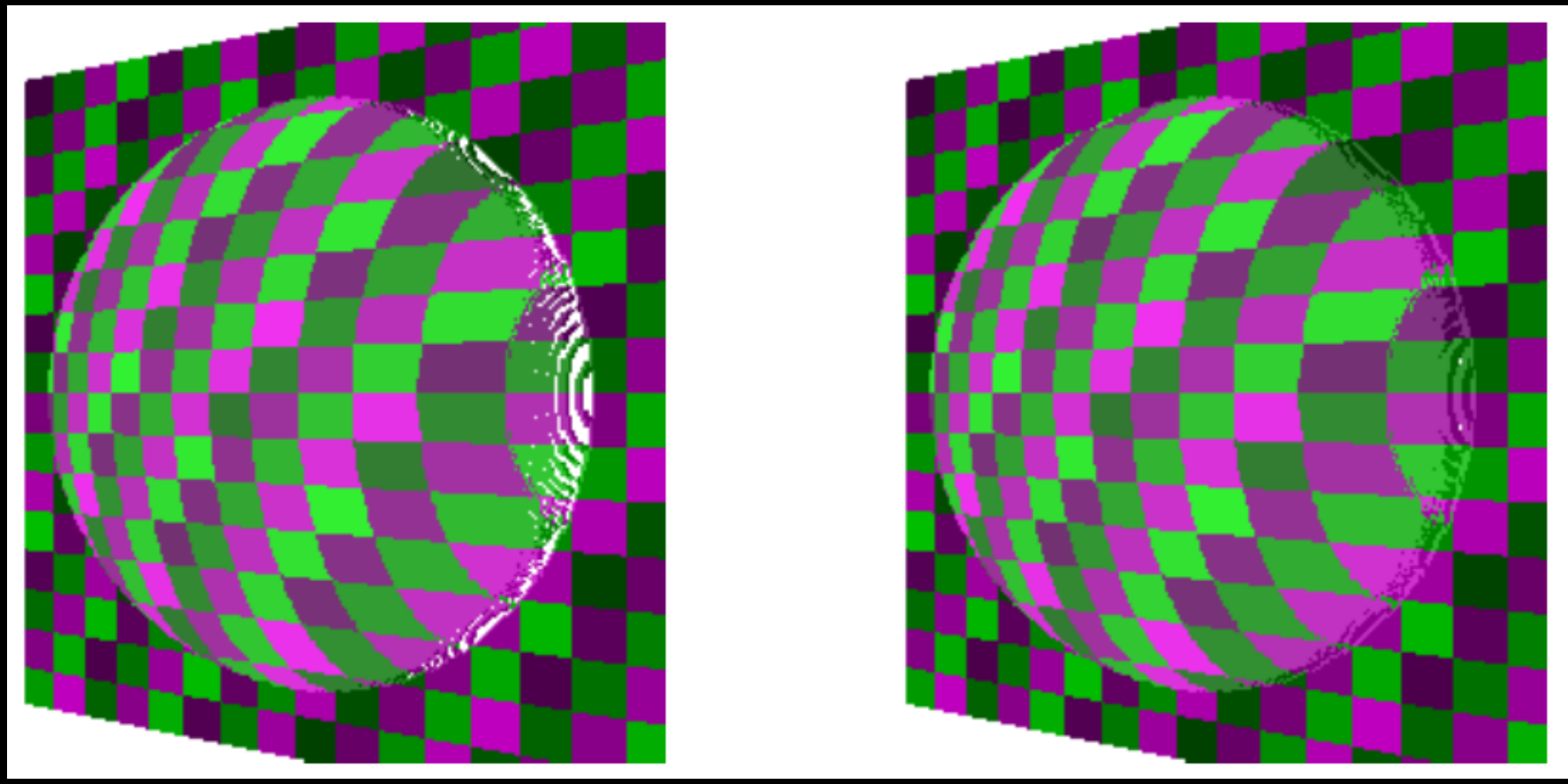
Intermediate Step

Forward map depth channel by “local parallax” only



Fill holes here, great for large zoom-ins etc

Depth Sprites



Forward Mapping only

Two-step + Hole-filling

Depth Sprites

- Limitation: The silhouette of the object represented by the sprite must fit into the silhouette of the final quad (required for backward mapping)
  - Workarounds:
    - Guess the silhouette size of the object by transforming its bounding box, and adjust the destination image and quad sizes accordingly (problem: wastes pixels)
    - Use multiple depth sprites, say along the six axis-aligned directions – boundary overflows in one are compensated by another (problem: more stuff to render)

Depth Sprites

- Limitation: Not good for objects with high-frequency, discontinuous detail
  - Hole-filling is no longer justified, so naïve forward mapping works just as well

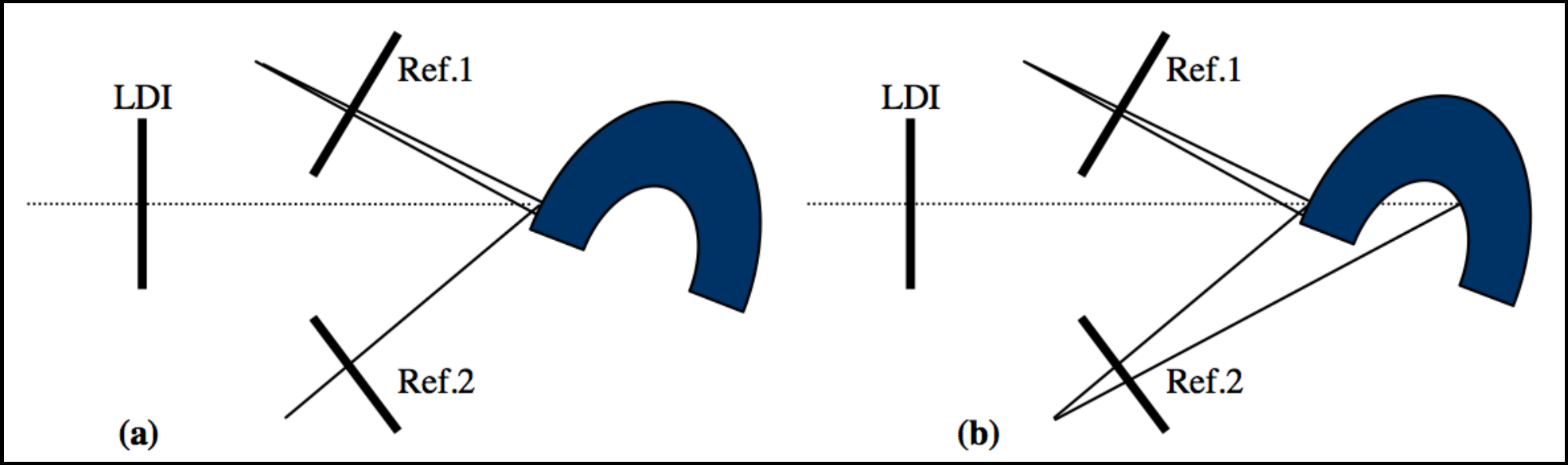
Layered Depth Images

- Big problem with depth images of general collections of objects (hole-filling very non-trivial): Disocclusion Artifacts
  - When viewpoint changes slightly, holes appear as hidden surfaces not sampled by the original image are exposed
  - Obvious solution: Somehow store samples from these surfaces as well
  - Q. How???
  - A. Use layers of depth, i.e. multiple surfels at each image pixel

Layered Depth Images

- Q. Why not store multiple views from a set of nearby locations?
- A. Because LDIs scale with depth complexity, which may be lower than the number of views chosen
  - Redundant repetitions of the same surfel across views are collapsed to a single entry
  - Advantages:
    - Low memory requirements
    - Less overdraw

Layered Depth Images



Construction of a single layered depth pixel from two reference images

Layered Depth Images

Construction:

1. Depth-peeling: Store first k intersections of ray through pixel
   - Fails on scenes with high depth complexity – first k intersected surfaces may not be the important ones visible in a target view
2. Multiple views: Reproject depth images taken from different angles to the same view
   - How do we choose these angles?
   - Reprojection may be even less efficient than depth peeling in hardware
3. "From-Region Raytracing": A little of everything

Layered Depth Images

- Depth peeling
  - Raytracing
  - Multipass (each pass peels back one layer)
  - Single pass with a k-buffer [Callahan '05]

- Multiple Views
  - Use synthetic or real-world data for different views
  - Splat views to a single multi-layer depth image using multi-pass depth-peeling (inefficient) or k-buffer

The k-Buffer

- What is it?
  - A multi-fragment buffer, implementable in hardware
  - Variations on the theme:
    - Multiple depth values @ each pixel
    - Multiple RGBZ values @ each pixel
    - Multiple possible states for Schrödinger's Cat @ each pixel
    - ...
- We specifically use it for...
  - ... multiple RGBZ values at each pixel

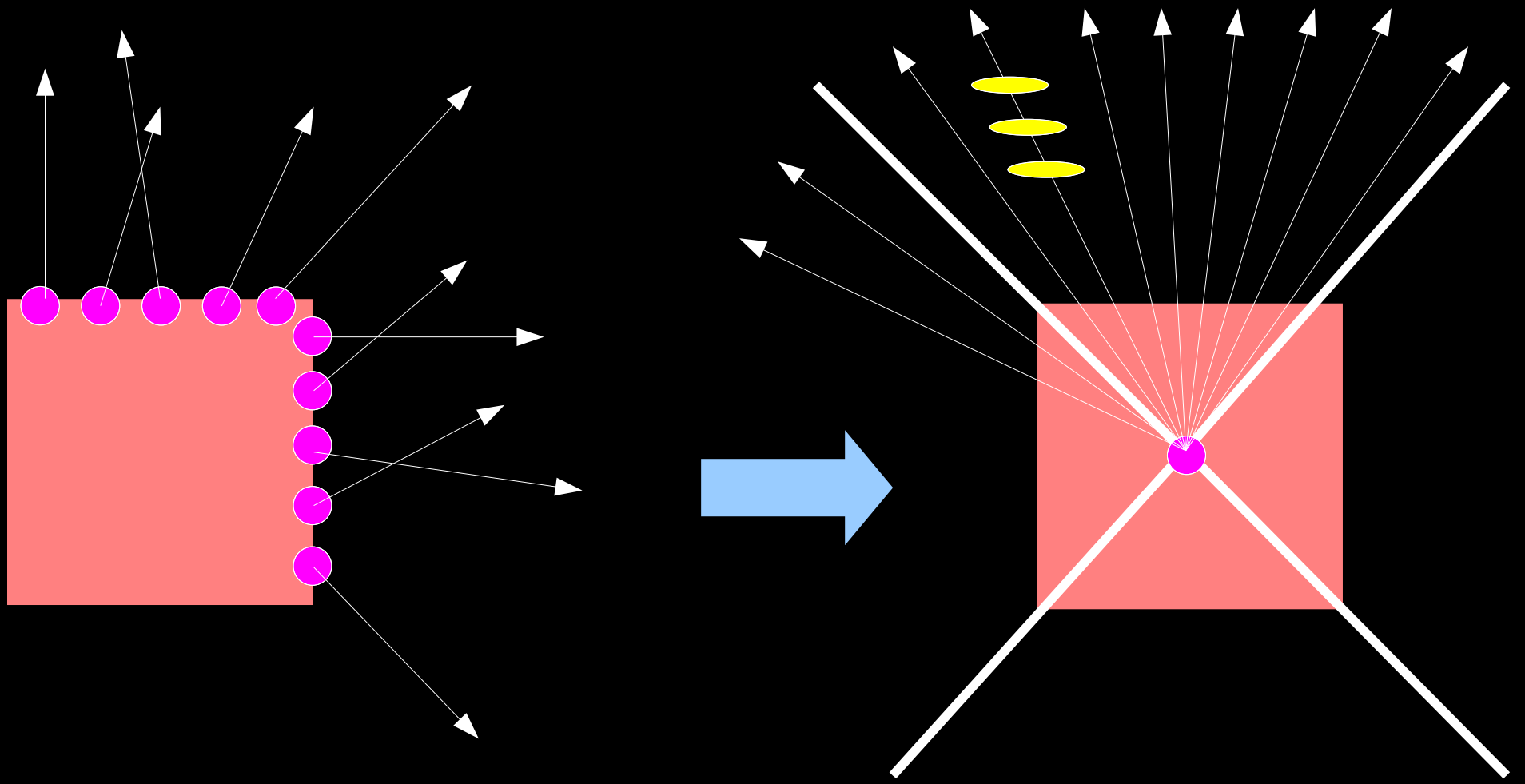
The k-Buffer

- Limitations on current hardware:
  - Requires Read-Modify-Write step that is atomic w.r.t. fragments that get written to the same pixel location
  - Hence sample implementations by authors are:
    - Software (Mesa)
    - Hardware (GeForce 7900 GTX): Unsynchronized, leads to occasional artifacts

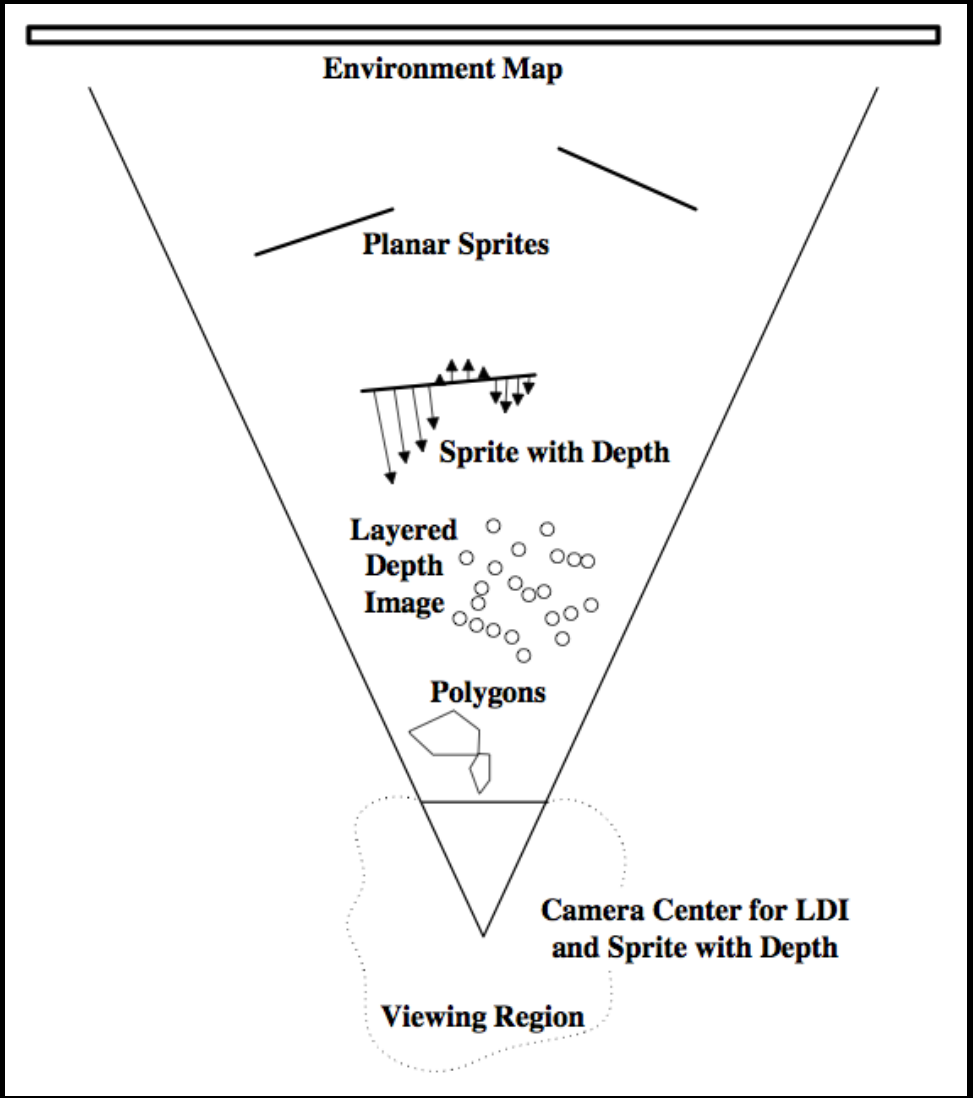
Layered Depth Images

- "From-Region Raytracing"
  - Similar to lightfield sampling
  - Sample the space of rays in a cube confining the user locations for which the LDI will be used
    - Use standard techniques for uniform sampling
  - Store the c nearest fragments (with depth) along each ray
  - Reproject and splat all fragments to a single layered depth environment map from the centre of the cube
    - ... possibly using a k-buffer

Layered Depth Images



Levels of Detail



LDI Trees

- Q. What happens when we use per-object LDIs, all at the same spatial resolution, for massive scenes?
- A. Massive oversampling in display space

- Q. Why?
- A. Because objects far away are sampled at the same rate as ones nearby, although a perspective rasterization samples distant objects more sparsely than near ones

LDI Trees

- Another way of looking at the problem:
  - A single LDI does not preserve the rates at which different reference images (some from close by, some from far away) sample an object
  - (This is indeed the same problem)

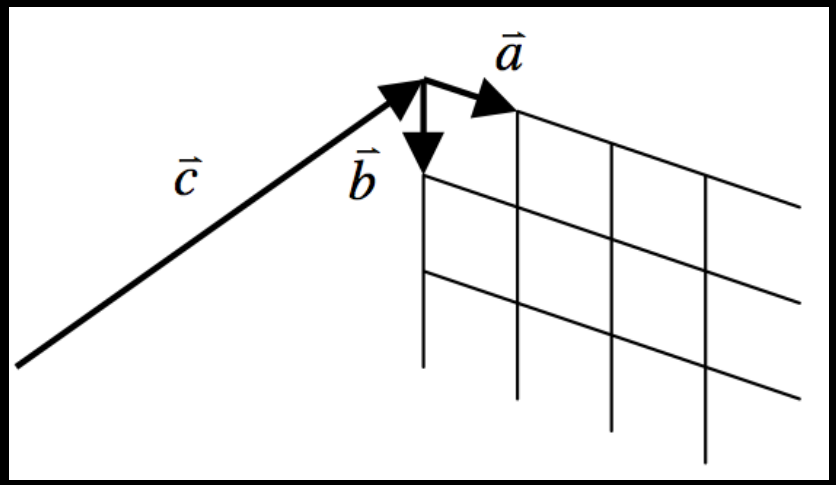
LDI Trees

- Rationale: An object far away is rendered at small absolute resolution…
  - (Another way to see this is: we maintain constant near plane resolution)
- … so can use a lower resolution LDI
- But the object still needs high-res when seen close-up!
- … so create a tree of LDIs
  - Essentially, an output-resolution, hierarchical, LOD structure for the scene

LDI Trees

Construction:

- Impose an octree on the scene, associating an LDI for the contents of each octree cell on each of its faces
- For each input view:
  - Associate a “stamp size” with its pixels



LDI Trees

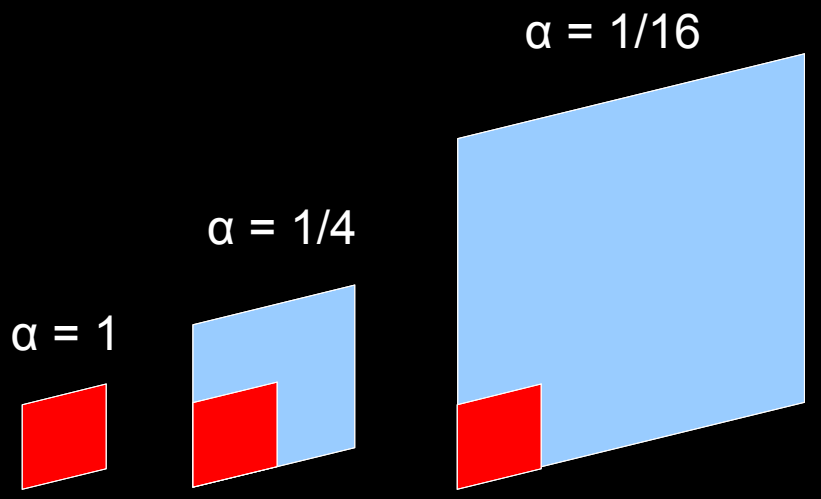
Construction (contd.):

- For each input view:
  - Associate a “stamp size” with its pixels
  - For each pixel:
    - Find its 3D position by reverse projection
    - Find its octree cell (octree level is given by stamp size)
    - Splat it into the LDI of the cell
      - Update 4 nearest neighbours, with alpha values of each new entry determined by the size of the overlap of the pixel's stamp with the neighbour
    - These are called “unfiltered pixels”

LDI Trees

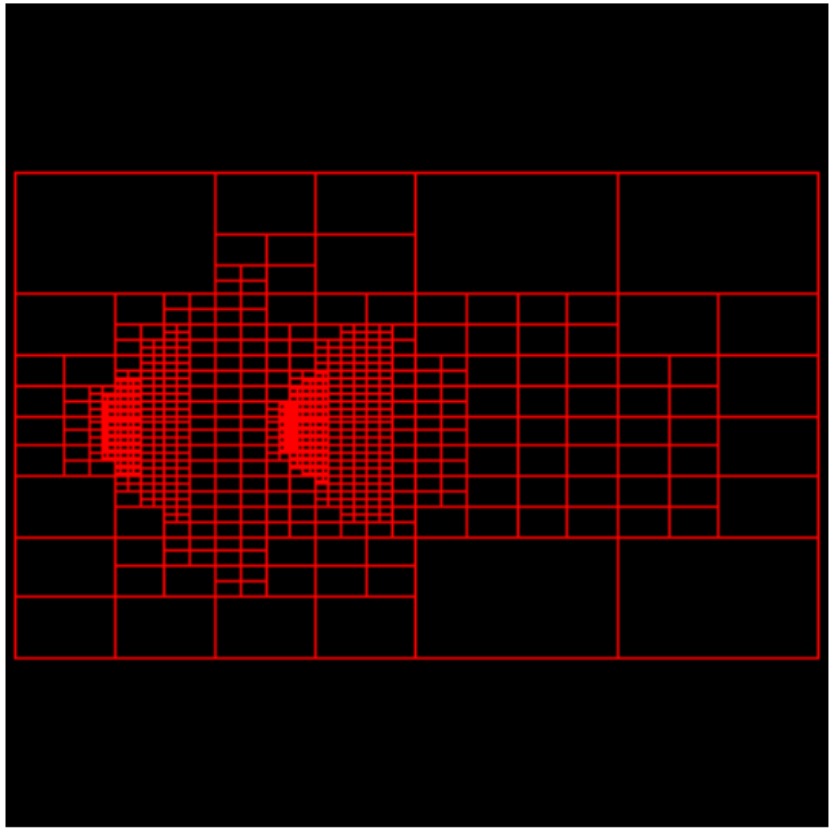
Construction (contd.):

- For each input view:
  - Associate a “stamp size” with its pixels
  - For each pixel
    - Splat it into lowest LDI possible
    - Splat it into LDIs of all ancestor cells, reducing alpha values at each step
      - Main Idea: Overlap fraction keeps (roughly) halving as the stamp size of the destination LDI grows – decreasing alpha models this
    - These are called “filtered pixels”



LDI Trees

Reference Image



Octree after two reference images

LDI Trees

- Rendering:
  - Traverse the tree
    - For each cell
      - If the stamp of the cell LDI has projected size ≤ 1 pixel
        - Splat all filtered and unfiltered pixels in the LDI
      - Else
        - If the cell has children
          - Recurse into them
          - Splat all unfiltered pixels
        - Else
          - Splat all filtered and unfiltered pixels in the LDI

LDI Trees

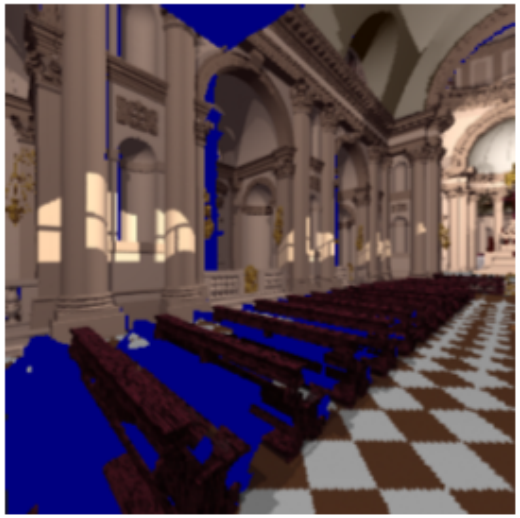


Figure 8: A new view generated from four reference images (at the same position).

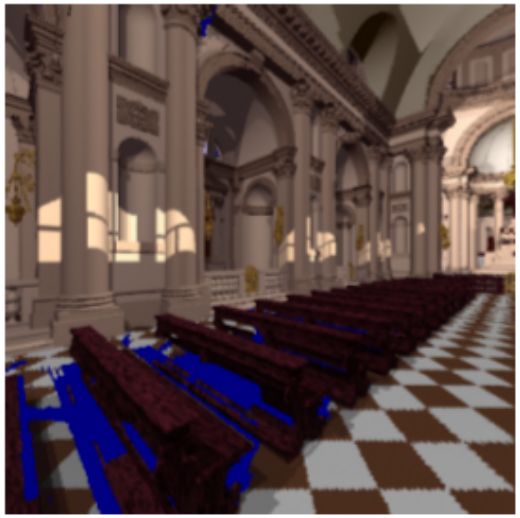


Figure 9: A new view generated from 12 reference images (at three different positions).

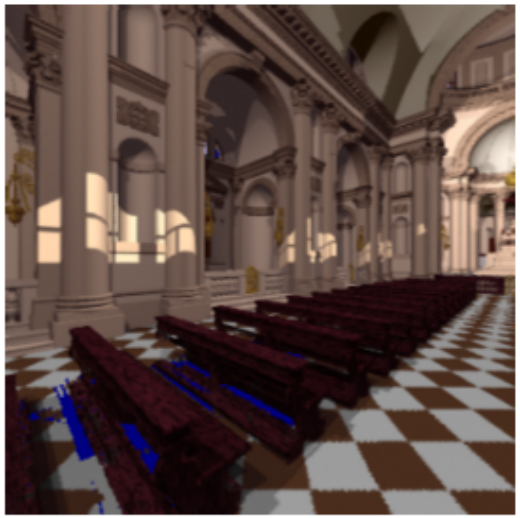


Figure 10: A new view generated from 36 reference images (at 9 different positions).

Figure 11: A new view generated from 36 reference images. Gap filling is enabled.

LDI Trees

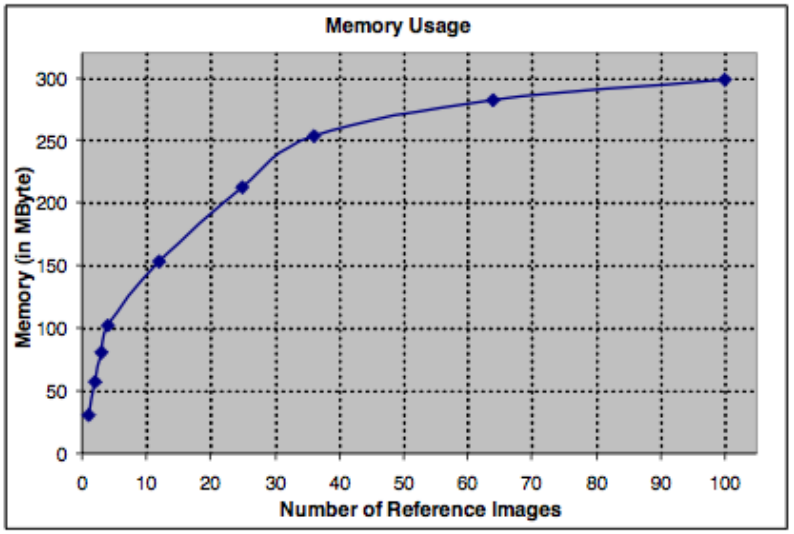


Chart 1: The memory usage of LDI trees.

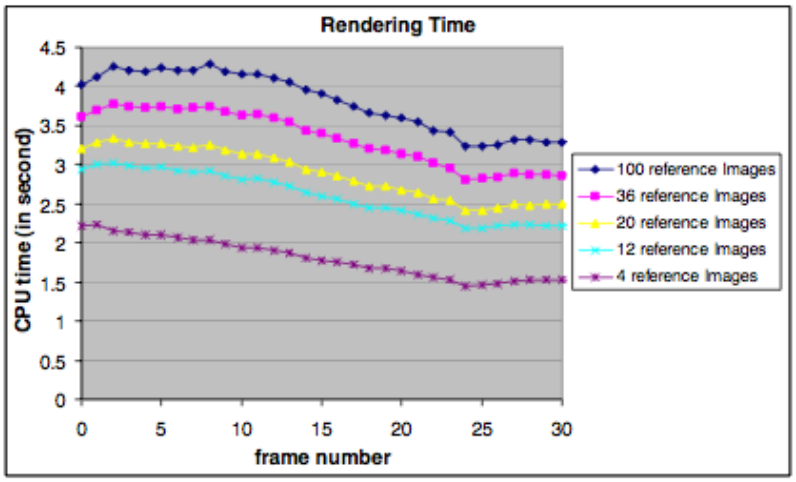


Chart 2: The rendering time.

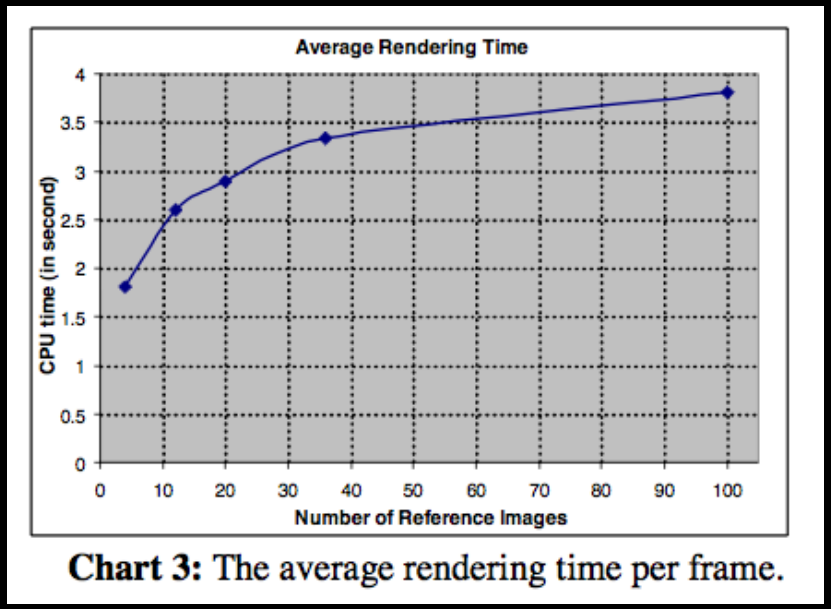


Chart 3: The average rendering time per frame.