CHAPTER 11

Automatic Handling of Materials in Nested Volumes

Carsten Wächter and Matthias Raab
NVIDIA

ABSTRACT

We present a novel and simple algorithm to automatically handle nested volumes and transitions between volumes, enabling push-button rendering functionality. The only requirements are the use of closed, watertight volumes (along with a ray tracing implementation such as NVIDIA RTX that guarantees watertight traversal and intersection) and that neighboring volumes are not intended to intersect each other, except for a small overlap that actually will model the boundary between the volumes.

11.1 MODELING VOLUMES

Brute-force path tracing has become a core technique to simulate light transport and is used to render realistic images in many of the large production rendering systems [1, 2]. For any renderer based on ray tracing, it is necessary to handle the relationship of materials and geometric objects in a scene:

> To correctly simulate reflection and refraction, the indices of refraction on the front- and backface of a surface need to be known. Note that this is not only needed for materials featuring refraction, but also in cases where the intensity of a reflection is driven by Fresnel equations.

> The volume coefficients (scattering and absorption) may need to be determined, e.g., to apply volume attenuation when a ray leaves an absorbing volume.

Thus, handling volumetric data, including nested media, is an essential requirement to render production scenes and must be tightly integrated into the rendering core. Ray tracing, i.e., its underlying hierarchy traversal and geometry intersection, is always affected by the limits of the floating-point precision implementation. Even the geometrical data representation, e.g., instancing of meshes using floating-point transformations, introduces further precision issues. Therefore, handling volume transitions robustly at least requires careful modeling of the volumes and their surrounding hull geometry. In the following we will distinguish three cases to model neighboring volumes. See Figure 11-1.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_11

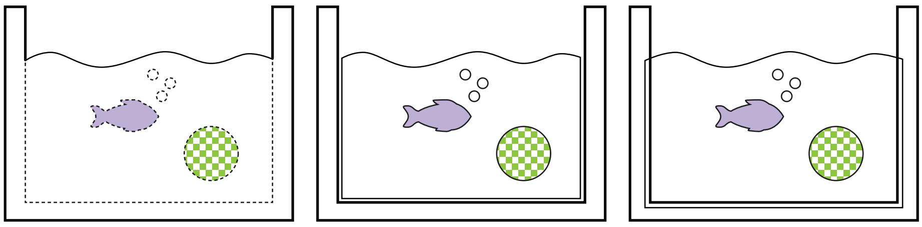

Figure 11-1. *Left: explicit boundary crossing of volumes marked with dashed lines. Center: air gap to avoid numerical problems. Right: overlapping volumes.*

11.1.1 UNIQUE BORDERS

The seemingly straightforward way shares unique surfaces between neighboring volumes to clearly describe the interface between two (or more) media. That is, anywhere two transparent objects meet, such as glass and water, a single surface mesh replaces the original two meshes and is given a special type. Artists will typically not be able to model volumes this way, as it requires manual splitting of single objects into many subregions, depending on which subregion touches which neighboring volume. Given the common example of a glass filled with soda, including gas bubbles inside and touching the borders, it is not feasible to subdivide the meshes manually, especially if the scene is animated. Another major complication of the scheme is that each surface needs to provide separate materials for front- and backface, where sidedness needs to be clearly defined.

Note that the unique surfaces can also be duplicated, to provide a separate closed hull for each volume. As a result, all tedious subdivision work is avoided. In practice, however, it is rather difficult to force the surfaces to exactly match up. Artists, or implicitly the modeling/animation/simulation software itself, may pick different subdivision levels, or the instancing transforms for the neighboring hulls may differ slightly due to floating-point precision mathematics. Therefore, the ray tracing implementation in combination with the rendering core would need to be carefully designed to be able to handle such "matching" surfaces. The ray tracing core, which includes the acceleration hierarchy builder, must guarantee that it always reports all "closest" intersections. The rendering core must then also be able to sort the list of intersections into the correct order.

11.1.2 ADDITIONAL AIR GAP

The second approach allows for a slight air gap between neighboring volumes to relax most of the mentioned modeling issues. Unfortunately, this leads to new floating-point mathematics issues caused by common ray tracing implementations: An ϵ offset is needed for each ray origin when generating new segments of the path, in order to avoid self-intersection [4]. Thus, when intersecting neighboring volume hulls, one (or more) volume transitions may be completely skipped, so it is important that the air gap is modeled larger than this ϵ offset. Another major downside of inserting small air gaps is even more dramatic though, as air gaps will drastically change the appearance of the rendering because there are more volume transitions/refractions happening than intended. See Figure 11-2.

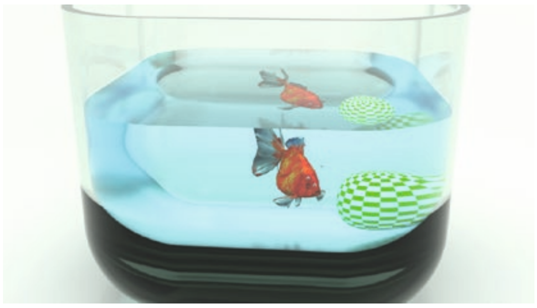

Figure 11-2. *Modeling the aquarium with a slight air gap.*

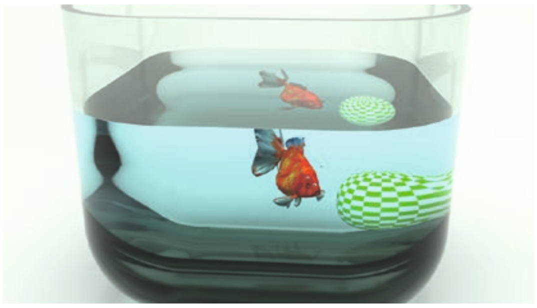

Figure 11-3. *Slightly overlapping the water volume with the glass bowl.*

11.1.3 OVERLAPPING HULLS

To avoid the downsides of the previous two schemes, we can force the neighboring volumes to overlap slightly. See Figure 11-3. Unfortunately, this third approach introduces a new problem: the ordering and the number of the path/volume intersections will not be correct anymore. Schmidt et al. [3] impose a valid configuration by assigning priorities to each volume. This requires explicit artist interaction that can be tedious for complex setups, especially when doing animations.

Note that, in addition to the three schemes mentioned, there is yet another, noteworthy special case: fully nested/enclosed volumes that are contained completely within another volume. See the colored objects in Figure 11-1. These are usually expected to cut out the surrounding volume. Some rendering implementations may also allow mixtures of overlapping or nested volumes.

Obviously this does not help to reduce the complexity of the implementation at all, as previously mentioned issues still exist when entering or leaving neighboring volumes. These transitions are even trickier to detect and to handle correctly as a path is allowed to travel through "multiple" volumes at once. Thus, our contribution is targeted at renderers that only handle a single volume at a time.

In the following, we describe a new algorithm to restore the correct ordering of the path/volume intersections when using the overlapping hull approach, without manual priority assignments. It has been successfully used in production for more than a decade as part of the Iray rendering system [1].

11.2 ALGORITHM

Our algorithm manages a stack of all currently active (nested) materials. Each time a ray hits a surface, we push the surface's material onto the stack and determine the materials on the incident and the backface of the boundary. The basic idea is that a volume is entered if the material is referenced on the stack an odd number of times, and exited if it is referenced an even number of times. Since we assume overlap, the stack handling further needs to make sure that only one of the two surfaces along a path is actually reported as a volume boundary. We achieve this by filtering out the second boundary by checking if we have entered another material after entering the current one. For efficiency, we store two flags per stack element: one indicating whether the stack element is the topmost entry of the referenced material, and the other if it is an odd or even reference. Once shading is complete and the path is continued, we need to distinguish three cases:

1. For reflection, we pop the topmost element off the stack and update the topmost flag of the last previous instance of the same material.
2. For transmission (e.g., refraction) that has been determined to leave the newly pushed material, we not only need to pop the topmost element but also need to remove the previous reference to that material.
3. For same material boundaries (that are to be skipped) and for transmission that has been determined to enter the new material, we leave the stack unchanged.

Note that in the case where the path trajectory is being split, such as tracing multiple glossy reflection rays, there needs to be an individual stack per spawned ray.

In the case of the camera itself being inside of a volume, an initial stack needs to be built that reflects the nesting status of that volume. To fill the stack, a ray can be traced from outside the scene's bounding box toward the camera position recursively.

11.2.1 IMPLEMENTATION

In the following we present code snippets that provide an implementation of our volume stack algorithm. One important implementation detail is that the stack may never be empty and should initially contain an artificial "vacuum" material (flagged as odd and topmost) or an initial stack copied from a preprocessing phase, if the camera is contained in a volume.

As shown in Listing 11-1, the data structure for the volume stack needs to hold the material index and flags that store the parity and topmost attribute of the stack element. Further, we need to be able to access materials in the scene and assume that they can be compared. Depending on the implementation, a comparison of material indices may actually be sufficient.

Listing 11-1. *The material index, flags, and scene materials.*

```
struct volume_stack_element
{
  bool topmost : 1, odd_parity : 1;
  int material_idx : 30;
};

scene_material *material;
```

When a ray hits the surface of an object, we push the material index onto the stack and determine the actual incident and outgoing materials indices. In the case that the indices are the same, the ray tracing code should skip the boundary. The variable `leaving_material` indicates that crossing the boundary will leave the material, which needs need to be respected in `Pop()`. See Listing 11-2.

Listing 11-2. *The push and load operations.*

```
void Push_and_Load(
  // Results
  int &incident_material_idx, int &outgoing_material_idx,
  bool &leaving_material,
  // Material assigned to intersected geometry
  const int material_idx,
  // Stack state
  volume_stack_element stack[STACK_SIZE], int &stack_pos)
{
  bool odd_parity = true;
  int prev_same;
  // Go down the stack and search a previous instance of the new
  // material (to check for parity and unset its topmost flag).
  for (prev_same = stack_pos; prev_same >= 0; --prev_same)
    if (material[material_idx] == material[prev_same]) {
      // Note: must have been topmost before.
      stack[prev_same].topmost = false;
      odd_parity = !stack[prev_same].odd_parity;
      break;
    }

  // Find the topmost previously entered material (occurs an odd number
  // of times, is marked topmost, and is not the new material).
  int idx;
  // idx will always be >= 0 due to camera volume.
  for (idx = stack_pos; idx >= 0; --idx)
    if ((material[stack[idx].material_idx] != material[material_idx])&&
        (stack[idx].odd_parity && stack[idx].topmost))
      break;

  // Now push the new material idx onto the stack.
  // If too many nested volumes, do not crash.
  if (stack_pos < STACK_SIZE - 1)
    ++stack_pos;
  stack[stack_pos].material_idx = material_idx;
  stack[stack_pos].odd_parity = odd_parity;
  stack[stack_pos].topmost = true;

  if (odd_parity) { // Assume that we are entering the pushed material.
    incident_material_idx = stack[idx].material_idx;
    outgoing_material_idx = material_idx;
  } else { // Assume that we are exiting the pushed material.
    outgoing_material_idx = stack[idx].material_idx;
    if (idx < prev_same)
      // Not leaving an overlap,
      // since we have not entered another material yet.
      incident_material_idx = material_idx;
    else
      // Leaving the overlap,
      // indicate that this boundary should be skipped.
      incident_material_idx = outgoing_material_idx;
  }

  leaving_material = !odd_parity;
}
```

When the rendering code continues ray tracing, we need to pop the material from the stack, as shown in Listing 11-3. For transmission events, this will only be called if `leaving_material` is set, and in that case two elements are removed from the stack.

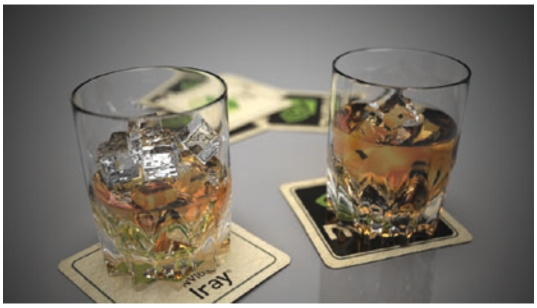

Figure 11-4. *Modeling the whiskey glass with a slight air gap.*

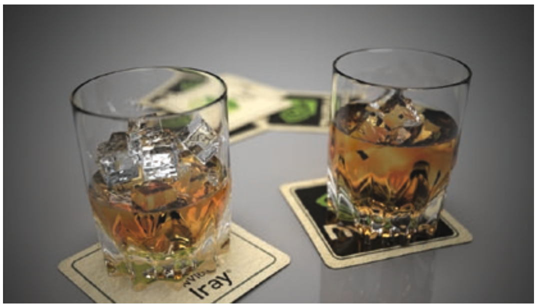

Figure 11-5. *Slightly overlapping the whiskey volume with the glass.*

Listing 11-3. *The pop operation.*

```
void Pop(
  // The "leaving material" as determined by Push_and_Load()
  const bool leaving_material,
  // Stack state
  volume_stack_element stack[STACK_SIZE], int &stack_pos)
{
  // Pop last entry.
  const scene_material &top = material[stack[stack_pos].material_idx];
  --stack_pos;

  // Do we need to pop two entries from the stack?
  if (leaving_material) {
    // Search for the last entry with the same material.
    int idx;
    for (idx = stack_pos; idx >= 0; --idx)
      if (material[stack[idx].material_idx] == top)
        break;

    // Protect against a broken stack
    // (from stack overflow handling in Push_and_Load()).
    if (idx >= 0)
      // Delete the entry from the list by filling the gap.
      for (int i = idx+1; i <= stack_pos; ++i)
        stack[i-1] = stack[i];
    --stack_pos;
  }

  // Update the topmost flag of the previous instance of this material.
  for (int i = stack_pos; i >= 0; --i)
    if (material[stack[i].material_idx] == top) {
      // Note: must not have been topmost before.
      stack[i].topmost = true;
      break;
    }
}
```

11.3 LIMITATIONS

Our algorithm will always discard the second boundary of an overlap that it encounters. Thus, the actual geometry intersected depends on the ray trajectory and will vary depending on origin. In particular, it is not possible to trace the same path from light to camera as from camera to light, which makes the method slightly inconsistent for bidirectional light transport algorithms such as bidirectional path tracing. In general, the lack of an explicit order for which boundary to remove may lead to removing the "wrong" part of the overlap. For example, the water will carve out the overlap region from the glass bowl in Figure 11-3 for rays that enter the glass first. If the overlap is sufficiently small, as intended, this is not a problem that causes visible artifacts. If, however, a scene features large overlap, as, e.g., the partially submerged ice cubes floating in Figures 11-4 and 11-5, the resulting error can be large (although one can argue about the visible impact in that scene). Thus, intended intersecting volumes should be avoided, but will not break the algorithm or harm correctness of later volume interactions along the path.

Imposing an explicit order on the volume by assigning priorities [3] will resolve this ambiguity at the price of losing push-button rendering functionality. This solution has its limits, as ease of use is essential to many users, such as those that rely on a ready-to-use library of assets, light setups, and materials, without knowing any of the technical details.

Managing a stack per path increases state size, so highly parallel rendering systems may carefully need to limit volume stack size. While the provided implementation catches overflows, it does nothing beyond avoiding crashes.

ACKNOWLEDGMENTS

An early version of this algorithm was conceived in collaboration with Leonhard Grünschloß. One part of the aquarium scene was provided by Turbosquid and the other by Autodesk.

REFERENCES

[1] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The Iray Light Transport Simulation and Rendering System. arXiv, https://arxiv.org/abs/1705.01263, 2017.

[2] Pharr, M. Special Issue On Production Rendering and Regular Papers. *ACM Transactions on Graphics 37*, 3 (2018).

[3] Schmidt, C. M., and Budge, B. Simple Nested Dielectrics in Ray Traced Images. *Journal of Graphics Tools 7*, 2 (2002), 1–8.

[4] Woo, A., Pearce, A., and Ouellette, M. It's Really Not a Rendering Bug, You See... *IEEE Computer Graphics and Applications 16*, 5 (Sept 1996), 21–25.