Real Time Multiscale Rendering of Dense Dynamic Stackings

Elie Michel, Tamy Boubekeur

Real Time Multiscale Rendering of Dense Dynamic Stackings

Élie Michel[1] and Tamy Boubekeur[2,1]

[1]LTCI, Télécom Paris, Institut Polytechnique de Paris
[2]Adobe

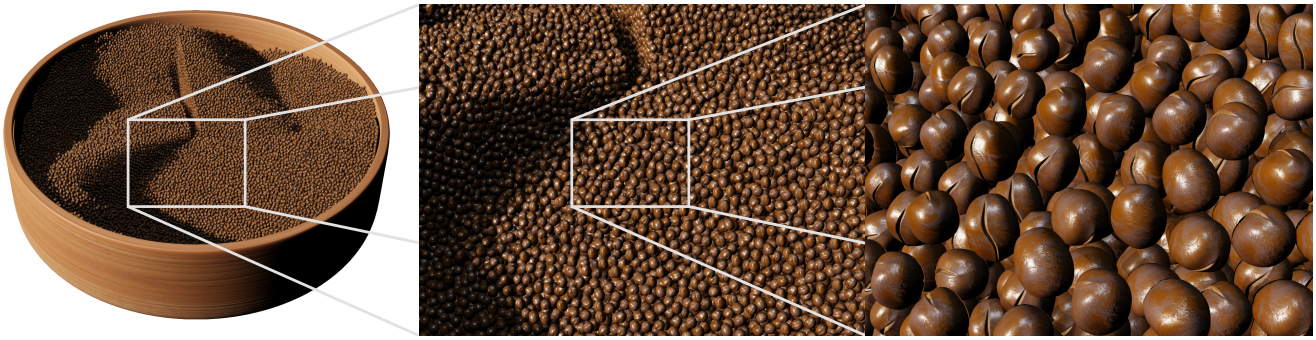

Figure 1: *Our level-of-detail method exploits quasi-spherical impostors to render, in real time, fully dynamic stackings made of millions of similar objects, with variable materials and orientations, while seamlessly integrating into deferred shading.*

Abstract

Dense dynamic aggregates of similar elements are frequent in natural phenomena and challenging to render under full real time constraints. The optimal representation to render them changes drastically depending on the distance at which they are observed, ranging from sets of detailed textured meshes for near views to point clouds for distant ones. Our multiscale representation use impostors to achieve the mid-range transition from mesh-based to point-based scales. To ensure a visual continuum, the impostor model should match as closely as possible the mesh on one side, and reduce to a single pixel response that equals point rendering on the other. In this paper, we propose a model based on rich spherical impostors, able to combine precomputed as well as dynamic procedural data, and offering seamless transitions from close instanced meshes to distant points. Our approach is architectured around an on-the-fly discrimination mechanism and intensively exploits the rough spherical geometry of the impostor proxy. In particular, we propose a new sampling mechanism to reconstruct novel views from the precomputed ones, together with a new conservative occlusion culling method, coupled with a two-pass rendering pipeline leveraging early-Z rejection. As a result, our system scales well and is even able to render sand, while supporting completely dynamic stackings.

CCS Concepts

• ***Computing methodologies*** → ***Rasterization; Visibility;***

1. Introduction

Fruits in a market-place, coffee beans in a roaster or bolts at the hardware store are typical examples of stackings found in 3D scenes. They challenge level-of-details (LoD) mechanisms to achieve both high speed rendering and detail preservation. At each extremity of the LoD chain, existing methods are well covered by the literature. Closer views are better handled using a mesh-based model that can be progressively simplified [**Hoppe96**] while further views leverage point-based rendering [**Gross07**]. But none of these models fits well the transition phase, when stacked elements – which we call *grains* in the reminder of this paper – cover tens to hundreds of pixels. Under this regime, mesh-based simplification makes whole elements vanish when pushed too far, while point-based rendering lacks high frequency details that should still be clearly visible.

Indeed, the self-similarity of the stacking naturally leads to per-grain *impostors* for this transition scale. When the number of visible grains is very large compared to the number of possible view angles, it becomes worth precomputing a few views and then, at runtime, picking for each grain the closest one. There are two ways

to describe an impostor, either as a projection of the geometry of a grain or as a rich splat. The former relates to mesh-based models while the latter relates it to point-based graphics, which suggests this is a good candidate as a transition model.

Impostors come with their own limitations e.g., memory consumption, hardware support or overdraw, that we propose to overcome making three assumptions that stem from the typical properties of stacked grains:

- **quasi-spherical shape**: the grain's surface is bounded between two co-centered spheres, namely an inner sphere of radius r and an outer sphere of radius R; the closer these radii, the more efficient our approach is,
- **moderate shape diversity**: all grains share the same (or only a few) silhouette, which prevents memory consumption and improves caching,
- **density**: occlusion culling becomes more impactful as density increases, even if approximate, as long as grains don't intersect each others.

Fortunately, the loss of generality induced by these hypotheses is in practice largely mitigated as noticed by [**Moon07**] and [**Meng15**]. The diversity of shape can be masked by arbitrary scale/rotation of each grain together with procedural variations of its material attribute maps. Our approach is oblivious to the grain material model: in practice, we exemplify our method on standard microfacet models rendered using deferred shading. Moreover, relaxing the quasi-spherical or density hypothesis only leads to progressively degraded performances, but not to any gap in visual appearance. Based on these assumptions, we make the following contributions:

- a real time splitting process of the input grain set into per-scale/drawing model buffers, leveraging an analysis of when to split (Section 6) and how to do it (Section 7),
- a sampling scheme for the impostors suited to our quasi-spherical proxy (Section 5) and improving their visual appearance w.r.t. ground truth,
- a novel occlusion culling mechanism tailored for dense stackings of quasi-spherical objects (Section 8), that helps alleviating rendering prior to determining the exact grain shape,
- an efficient rendering pipeline for a cloud of many impostors based on early-Z rejection (Section 9) – a hardware mechanism not natively suitable for impostors, whose actual shape remains unknown up to the sampling of their maps.

As a result, our approach is versatile enough to model various use cases and scales well up to extreme amounts of grains such as in sand rendering.

2. Related works and background

Level-of-Detail LoD methods intend to generate simplified versions of a complex object that are visually equivalent at a given distance while computationally lighter. Surfacic mesh simplification methods, either based on repeated contractions of edges guided by some cost function [**Hoppe93**; **Hoppe96**; **Garland97**] or by spatial clustering [**Rossignac93**], have become standard LoD methods and can even be applied to very large meshes [**Lindstrom00**]. However, as pointed out by Cook et al. [**Cook07**], such methods fail when the geometry is an aggregation of already simple elements, which vanish if simplified further – such as our grains. We can simplify the geometry of the grain itself, but not beyond.

Volumetric models also have their LoD mechanisms. On voxel-based models, the SGGX distribution [**Heitz15**] and follow ups [**Zhao16**; **Loubet18**] have enabled techniques for downsampling a volume without altering its visual appearance. Hierarchical structures can be used to organize data in a tree whose traversal is dynamically adapted to the view point, either with voxels [**Crassin09**; **Kampe13**] or with points [**Rusinkiewicz00**; **Gobbetti05**]. Most of these techniques assume static geometry though. All these LoD methods are designed for a single class of model. Sequential point trees [**Dachsbacher03**] are an interesting evolution of QSplat [**Rusinkiewicz00**] using an hybrid model, but it makes the same assumption that the point cloud is static. The inter-model transition was recently successfully addressed in the surface-to-volumetric context by Loubet et al. [**Loubet17**]. Their setting is more general than ours but designed for off-line rendering and not tacking advantage – because not assuming – of self-similarity.

In this paper, we focus on dynamic element positions. This prevents us from using techniques that precompute clusters of geometry to merge, like *Occluder Fusion* [**Wonka00**] or *CellVIEW* [**LeMuzic15**]. The latter is a case of molecular visualization, which generally involves LoD of dense aggregates of spheres that motivated dedicated research, as surveyed by Miao et al. [**Miao19**]. Although such visualization techniques deal with static perfect spheres, usually uniformly colored, setting them aside from many issues we intend to tackle here, they need to handle very large amounts of atoms for which they develop inspiring advanced drawing strategies.

Impostors One of the most extreme simplification consists in using *billboards*. A billboard, or planar impostor, is made of one single plane, and its whole appearance is encapsulated in (the maps of) its material, with its perceived shape being expressed by its silhouette, reproduced using transparency. The extreme simplicity of a billboard's geometry allows to invest more resources in shading, with its associated material containing information about the normal field of the original geometry, and even the depth component leveraged by *relief mapping* techniques [**Policarpo05**].

The limits of a single billboard are quickly reached, usually because of the limited range of directions for which it is valid, but they are at the root of many lightweight approximation models in computer graphics. Aggregated billboards are often called *multi-view impostors* since they address the view dependency of planar billboards. Maciel and Shirley [**Maciel95**] build a LoD hierarchy in which billboards are precomputed for some key directions. Billboard clouds [**Decoret02**] extract several billboards using a Hough transform to approximate a high definition mesh model. A typical use case of billboards, and hence multi-view impostors, is tree rendering, like Meyer et al. [**Meyer00**] and more recently Bruneton et al [**Bruneton12**] whose method is related to ours, though their model remains surfacic at all scales and they use impostor for the different goal of accounting for foliage's semi-transparency.

Todt et al. [**Todt07**] provide a good overview of the possi-

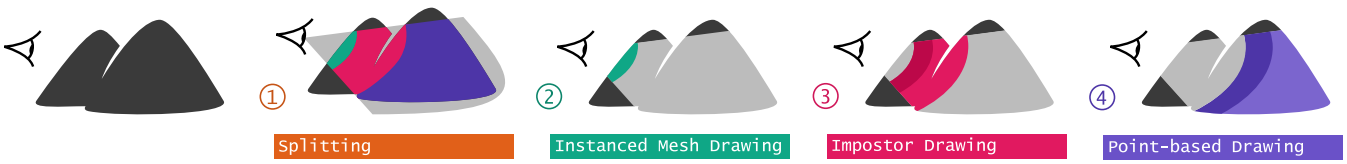

Figure 2: *Anatomy of a dense stacking rendering sequence. A first step splits the stacking into several element arrays used to feed subsequent draw calls (Section 7), also applying culling to early discard hidden points (Section 8). Impostors are drawn in two passes (Section 9). Point-based drawing typically discard all points beyond a limit distance. Our contributions concern steps ① and ③.*

ble parametrizations of a spherical impostor, however they focus on a different use case where a single complex model is rendered, leading them to different design choices. In particular, their selection of precomputed directions, and advanced compression, projections and intersection refinement schemes, while saving memory, quickly becomes too prohibitive to apply for each grain in our scenario. Some of these limitations are addressed by Brucks [**shaderbits18**] who, similarly to our approach, also make the impostors dynamically relightable by storing maps that represent the attribute field (like the G-Buffer) rather than a static grain light field. However, Brucks renders order of magnitude less impostors than in our use case, so they can still afford storing depth maps and computing relief mapping. Since they use it for trees, they also deal with significantly smaller inter-impostor occlusion.

Filtering Filtering attribute-encoding images, as mandatory with mipmapping, is not trivial for attribute with non-linear response, such as normal and roughness maps. This issue has been addressed by Tan et al. [**Tan08**], *LEAN mapping* [**Olano10**] and then *LEADR mapping* [**Dupuy13**], which are compatible with our method. More recent works even try to adapt the concept of mip-maps to the BSDF itself rather than to its attribute maps [**Xu17**].

Granular materials For off-line rendering, representing granular media has been the subject of specialized models that rely on similar hypotheses about the rendered scene. Moon et al. [**Moon07**] propose a method called *Shell Tracing*, using light paths whose precision depends on their depth within the sand volume. Meng et al. [**Meng15**] developed a multiscale volumetric path tracing method designed for production pipelines, later extended by Muller et al. [**Muller16**] to dynamic scenes. Unfortunately, these methods are deeply grounded in the offline rendering world. For instance Muller's GSDF are precomputed for a random orientation of the grain, building upon core ideas of Monte Carlo path tracing which we precisely cannot leverage in forward real time rendering. To the best of our knowledge, little work has been carried out on the specific task of rendering dense dynamic stackings e.g., sand, in real time.

3. Pipeline overview

Figure 2 describes the sequence of draw events involved in rendering our aggregate of grains. The splitting step ① orchestrates rendering by routing each grain toward one model or another depending on its location. It discards some of them based on an occlusion map which is also further reused at step ③ to speed up drawing. Additionally, it early rejects grains out of the view frustum. Models

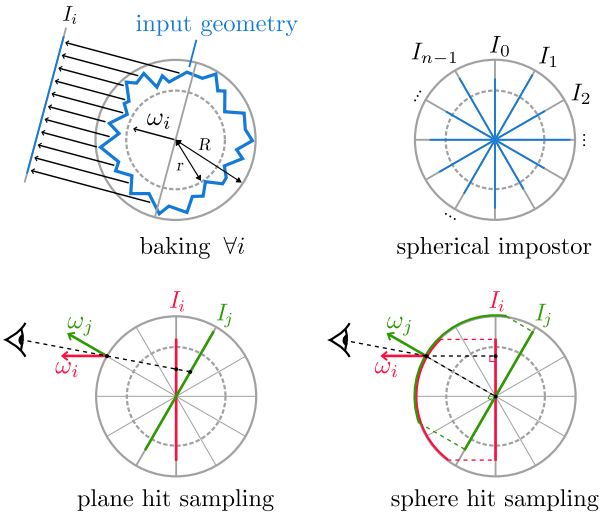

Figure 3: *A spherical impostor is a set of concentric planar impostors I_i precomputed for different directions ω_i, as well as an inner radius r and outer radius R. At render time, we use only the most relevant ones. They can be sampled as planes (bottom left) or as hemispheres (bottom right), or using our mixed sampling scheme (Section 5).*

for close ②, mid ③ and far ④ grains are then rendered individually using the element buffers resulting from the splitting. We recall that, as a general rule of thumb, modern hardware-accelerated rasterization pipelines require closer elements to be rendered first to limit unnecessary fragment processing. Steps ② and ④ are the two models that we intend to bridge, respectively mesh-based and point-based, so we focus on the splitting process ①, which involves *when* (Section 6) and *how* (Section 7) to split the input point cloud, as well as on the impostor rendering ③.

4. Impostors for dense stackings

Our mid-scale representation of the stacking is a cloud of impostors. The concept of spherical impostor is not new per se, but it can come with many flavors so we discuss which one is the best suited for real-time rendering of dense aggregates.

4.1. General rendering pipeline

We base our work on a spherical impostor model made of co-centered planar impostors facing different directions. Prior to ren-

dering, the impostors are precomputed and then at render time, the only planes to be sampled are those whose normal vector is close enough to the view direction (Figure 3).

Precomputation The impostor depends on the set of N view directions $(\omega_i)_{i=1...N}$ for which the grain's response is precomputed. For each view index i, a $p \times p$ sprite of the grain is rendered from a view point in direction ω_i, storing for each pixel the material attributes (albedo, roughness, normal, etc) through an atlas of maps $(I_i)_{i=1...N}$ where I_i is the response at different positions of the sphere in direction ω_i.

Runtime In order to reduce as much as possible the geometric footprint of the impostors, we use simple sprites, e.g. OpenGL's GL_POINTS, as our drawing primitive. The sprite size is computed in the vertex shader to ensure that it covers the whole outer sphere of the grain. When drawing the impostor, we fetch the object's appearance attributes at a given point in a given direction. The main steps of impostor sampling are **(i)** to seek for the indices of the appropriate precomputed views (planar impostors) given the orientation and position of the grain in camera space, **(ii)** to sample the right texel from the impostor maps and **(iii)** to interpolate the responses of different planes. The interpolation weights ensure the visual continuity by progressively fading out the contribution of a plane when the view point changes. In the design of such a sampling, two choices can have a major impact : the *parametrization*, and the *definition*, i.e. the density of the sampling. The latter is discussed in Section 6 when analyzing the bias introduced by the impostors.

4.2. Parametrization

In order to still benefit from hardware texture filtering (mipmaps), especially to reduce aliasing when grains become very small on screen, we use the parametrization that Todt et al. [**Todt07**] calls Sphere-Plane. When sampling the atlas of precomputed views, the view index represents a direction and the texel coordinate a position offset, not the other way around. In practice, this means that precomputed views are rendered using orthographic cameras.

There remains to decide on the directions to precompute. The list (ω_i) of such directions must verify:

- **coverage**: there must always be a billboard close enough to the viewing direction among the precomputed atlas,
- **compactness**: each precomputed view has a video memory footprint which must be small especially if there are many different types of grain,
- **speed**: to sample a given viewing direction, we need to efficiently determine the index of the closest view in (ω_i).

Coverage and compactness suggest an as regular as possible sampling of the unit sphere such as the distribution of Fibonnacci points [**Keinert15**] or a statically optimized mesh [**Todt07**]. But speed is crucial in our scenario so we opt for a distribution based on a subdivided octahedron (see Figure 4) as in [**shaderbits18**]. Not only finding the index i of the closest view in this distribution is done in $\mathcal{O}(1)$ with a little constant, but moreover it is easy to get the four closest views with their coefficients, which is important for interpolation.

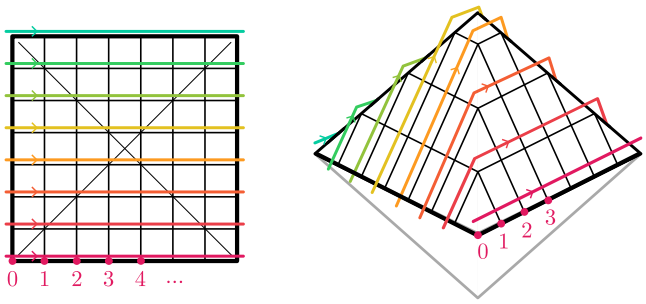

Figure 4: *We precompute view directions using vertices of a subdivided octahedron (the L_1 sphere) since mapping them to integer indices is computationally efficient; here with $n = 8$ vertices by edge boils down to $N = 128$ views.*

At runtime, we compute the sampled directions at the extent of a whole grain, and not once for each fragment i.e., we assume that camera rays are almost parallel for each fragments covered by a grain. Though this is inexact in general, it does not introduce strong distortion for grains whose extent on screen is limited to 100 pixels – our use case – and provides a significant speed-up.

```
void DirectionToViewIndices(
    vec3 d, uint n, out uvec4 i, out vec2 alpha
) {
    d = d / dot(vec3(1,1,1), abs(d));
    vec2 uv =
        (vec2(1, -1) * d.y + d.x + 1) * (n - 1) ↩
            / 2;
    uvec2 fuv = uvec2(floor(uv)) * uvec2(n, 1);
    uvec2 cuv = uvec2(ceil(uv)) * uvec2(n, 1);
    i.x = fuv.x + fuv.y;
    i.y = cuv.x + fuv.y;
    i.z = fuv.x + cuv.y;
    i.w = cuv.x + cuv.y;
    if (d.z > 0) {
        i += n * n;
    }
    alpha = fract(uv);
}
```

Listing 1: *Return in i the indices of the four closest precomputed views to the sampling direction d, assuming that the number of precomputed views is $N = 2n^2$, and in α the coefficients for interpolating between respectively $(i_0, i_2) \leftrightarrow (i_1, i_3)$ and $(i_0, i_1) \leftrightarrow (i_2, i_3)$. Note that instead of using the total number of views N, our procedure handles the number n of subdivisions along the edge of the octahedron.*

In practice, one can refer to the Listing 1 for a GLSL implementation (more listings can be found in supplemental material). The input direction d is normalized using the L_1 norm $L_1(d) = |d_x| + |d_y| + |d_z|$ and then converted to integer indices.

Note that in the atlas of a rich impostor, an index stores multiple maps, for the multiple attributes of the G-buffer. But they all conceptually share the same alpha transparency. Special care must be

taken to account for alpha premultiplication when computing the mipmaps.

5. Sampling quasi-spherical impostors

Once a precomputed map I_i has been selected for sampling, different strategies may be adopted to decide which texel to read. We propose a sampling scheme that improves the visual appearance of under-defined impostors while remaining lightweight.

The most common choice is to assume that the the view direction is perfectly aligned with precomputed direction ω_i and compute the offset using intersection of the camera ray with the precomputed view plane. We call this *planar sampling* (Figure 3, bottom left) and note P the texel it selects. In practice the camera and precomputed directions are not always well aligned, because we can store only a limited number of views. This results in ghosting artifacts, which particularly impact sharp visual features (Figure 12) and stems from the distance between the plane and the actual geometry of the grain.

A second strategy consists in computing the intersection of the camera ray with an spherical proxy and then project this point along the precomputed view direction onto the precomputed plane (Figure 3, bottom right). This *spherical sampling* gives another texel S. When using the average of r and R as radius of the sphere proxy, this greatly reduces ghosting, but cuts out parts of the object.

Therefore, we introduce a *mixed sampling* for quasi-spherical proxies. More precisely, we combine P and S depending on the relative distance d of the grain center to the camera ray normalized by the sphere's radius:

$$M = \frac{d}{R}P + (1 - \frac{d}{R})S$$

This sampling succeeds at combining the benefits of both planar and spherical samplings, namely preserving silhouettes and sharp visual features. For a fixed memory budget and visual loss, this translates into more grains rendered as impostors and less as meshes, improving the overall performances.

6. Impostors' validity range

We are not simply looking for a model that works at a given mid-scale, we also need to be able to smoothly transition from one model to another. In order to identify view conditions under which both mesh-based and impostor-based rending match, enabling us to substitute them, we must be able to quantify the range of validity of the impostor. This range of validity depends on the $p \times p$ amounts of spatial samples per view and the number $N = 2n^2$ of views precomputed for a mapping based on an octahedron with n subdivisions.

At the limit mesh-impostor distance L, the apparent grain diameter in pixels must match the size of the precomputed view, which gives us p proportional to R/L. The proportionality factor depends on the camera field of view and the screen resolution (see supplemental material for details). So from now on we assume that p is known and seek for N.

To do so, we need to know the maximum angle θ between a planar impostor's normal and the view direction for the impostor to return the right value. With the notations of Figure 5, we look for the limit view angle beyond which A'C' exceeds the world space size $t = R/p$ of a texel, i.e. beyond which our model will sample the wrong texel. This constraint writes as follows:

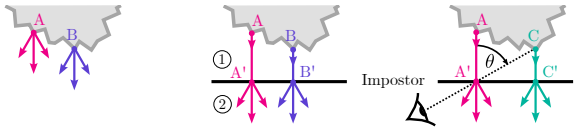

Figure 5: *A simple planar impostor replaces original geometry (left) with a plane (middle). At baking time ① attributes are projected, then used at render time ② . This is valid up to a limit value of θ (right).*

$$A'C' \leq t \tag{1}$$

On another hand, the maximum distance between a point on the true surface of the grain and its projection onto the impostor is the outer radius R:

$$CC' \leq R \tag{2}$$

This can be written using the angle θ between the impostor's normal and the view direction:

$$A'C' \leq R\sqrt{\frac{1}{\cos^2\theta} - 1} \tag{3}$$

To verify inequality (1), we can therefore look for:

$$R\sqrt{\frac{1}{\cos^2\theta} - 1} \leq R/p \implies |\theta| \leq \arccos\sqrt{\frac{1}{1+1/p^2}} \tag{4}$$

So each precomputed view is valid in a cone of angle $2\arccos\sqrt{\frac{1}{1+1/p^2}}$. This value has to be compared with the maximum angle between two neighbor points of the octahedron. Figure 6 shows the evolution of this angle depending on the angular

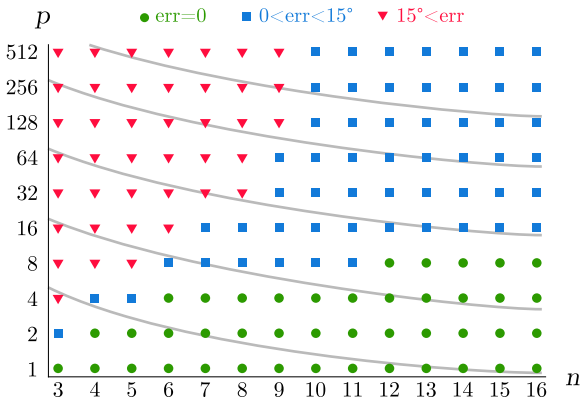

Figure 6: *Mean angle error for different trade-offs of the two parameters n (subdivisions of the octahedron) and p (pixels per side) of a spherical impostor. Grey lines are iso-weight i.e., two dots on the same line correspond to impostors occupying the same amount of video memory. See supplemental material for exhaustive data.*

definition n and spatial definition p. More detailed tables can be found in supplemental material. In practice, we use less views than the theoretical threshold since our mixed sampling scheme (Section 5) largely helps reducing artifacts for under-resolved impostors.

7. Model discrimination

Now that we know the range of validity of the impostor, we can dynamically discriminate the grain cloud into three subparts, namely the grains rendered using mesh instances, those rendered as impostors and the further ones rendered as points. We assume that all models are compatible with indexed rendering, which means that rather than when rendering K points, an *element buffer* of K indices can be provided to tell which grains to render.

The splitting process consists in building those elements buffers from the array of all stacked elements. The number of elements being by hypothesis very large, it is not possible to pay for the round trip to the CPU, hence we perform this splitting entirely on the GPU, in compute shaders. It is also not even possible to sort the whole buffer by the distance of the grains to the view point.

Even if the splitting apparently distinguishes between only three models, it is more convenient to see it as operating on an arbitrary m number of models because next sections will add an extra state for culled points (Section 8) and then split the impostors into two arrays for more efficient rendering (Section 9).

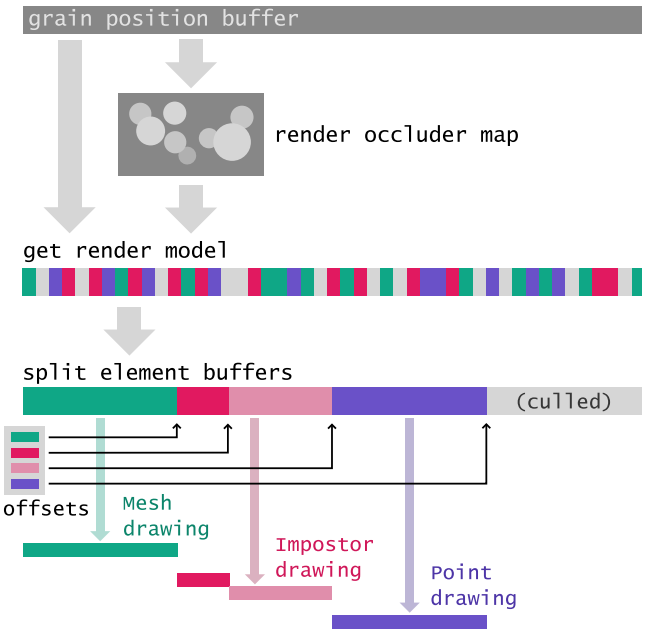

Figure 7: ***Grain splitting.*** *Given the position of the grains, we first render a map of the most likely occluder grains and then distinguishes which model to use for each grain, building contiguous element buffers for each subsequent draw call. Impostor rendering requires two element buffers to render occluder candidates first (Section 9).*

We assume that we have a function `uint getRenderModel(uint element)` that returns for an element index the index from 0 to $m-1$ of the model that must be used to draw it. This basically fetches the position buffer to check the distance to the grain against the thresholds, and will later on include culling. The output element arrays are written next to each other in a buffer allocated with the same size as the input element array. Besides this output, the methods returns a list of m offsets within the buffer to tell at which index each element array starts (Figure 7).

Global atomic splitting We adopt a simple and effective method made of two steps. First, we atomically count the number of elements per model, in order to determine the output offsets. In a second step, we insert element indices in the output using for each model a second counter besides the offset to keep track of where is the next available index. This counter is atomically incremented each time an element is written. This process requires calling `getRenderModel` twice for each grain. Although this function gets more complex when culling is added, caching its output between the first and the second steps saves only a few tenths of millisecond on a stacking of 1.6M elements which is not worth the overhead of allocating a cache buffer. Once the element buffers are ready, the offsets can be used to build a command buffer adapted to each model in a simple compute buffer.

Scalability At this point, we have a pipeline able to render stackings of which grains can smoothly turn from meshes to points. But, as we intend to draw a large number of grains, we need to improve the scalability of the pipeline. Indeed, the use of impostors make the fragment processing even heavier than it usually tends to be in modern engines, so in the next sections we make use of the relative density of the stacking to **(i)** reduce the number of points emitting fragments (Section 8) and **(ii)** reduce the number of emitted fragments that reach the fragment shader (Section 9).

8. Occlusion Culling

In a dense stacking, a large proportion of the elements is totally invisible. We propose in this section a novel occlusion culling that is conservative i.e. it does not cull visible objects, and based on the quasi-spherical proxy assumption. The occluder map it computes is further reused to improve per-fragment occlusion culling (Section 9).

Occlusion culling operates before the actual shape of grains is known, but can use the quasi-spherical proxy to early detect occlusions. If the inner sphere of a grain totally hides the outer sphere of another one, then no matter their actual shapes the second one will never be visible and can hence be safely culled out. As illustrated in Figure 8, the inner sphere of a grain close to the view point creates a cone of occluded positions (dashed lines). This cone is eroded with the outer sphere to give a set of grain centers that can be culled (occlusion cone). Yet, it is far too expensive to test every pair of grains for occlusion. And while in theory this could be executed using hardware occlusion queries [**Sekulic04**], with the inner sphere being the occluder and the outer sphere the proxy, it is not practical as it would require to render grains sequentially.

Occluder map Instead, we test each grain – the *occludee* – against exactly one other grain that we chose carefully – the *occluder can-*

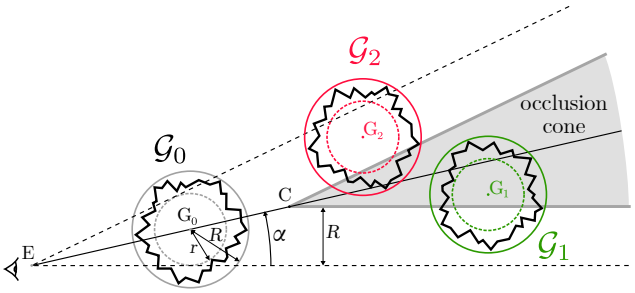

Figure 8: *Since a grain is bounded by two spheres of radius r and R, when the center of a grain $\mathcal{G}_1$ lies in the occlusion cone, it is fully occluded by $\mathcal{G}_0$. On the contrary, the grain $\mathcal{G}_2$ might not be totally hidden and cannot culled out.*

didate. A grain can hide another one if it is at the same time closer to the view point, and projects around the same pixel on screen. So, prior to the occlusion test, we render the whole point cloud a first time. The z-test ensures that we keep the closest grain for each pixel i.e. the most likely to hide other grains. At this stage, no attribute fetching or computation is executed, instead the framebuffer is filled with the occluder candidate parameters – one occluder per pixel. These parameters are the position and radius of the occluder's inner sphere, fitting in a standard four-component color attachment.

To fill this *occluder map*, one must compute the point sizes: drawing points of exactly one pixel each would mean that the occluder candidate of a point is always the grain that projects on the same pixel but is closer to the camera. This has perfect chances of picking the right occluder candidate when it finds one, but will most of the time not find any other occluder candidate than the grain itself. On the other side, drawing the points using their inner radius is not the best choice either, because it will too often suggest an occluder candidate that is actually not occluding the point. Our trade-off is to render points large enough for all pixels to be covered by a few fragments while remaining as small as possible. In practice, for as dense as possible stackings viewed at distance for which impostors are used, we found experimentally that optimal values are located between 0.15 to 0.20 times the inner radius *r*.

Splitting Once this occluder map has been generated, the discrimination function `getRenderModel` in the splitting shader computes the screen pixel onto which a grain's center gets projected, and samples the occluder map at this coordinate. This gives the parameters of an occluder to test the current point against using the procedure detailed in Listing 2. If the point is inside the occlusion cone, the function returns an index corresponding to no model.

```
bool IsOccluded(vec3 g1, mat4 proj, sampler2D ↩
    occMap){
   vec4 clip = proj * vec4(g1, 1.0);
   vec4 occ = texture(occMap,clip.xy/clip.w↩
       *.5+.5);
   if (occ == NONE) return false;
   vec3 g0 = occ.xyz;
   float r = occ.a;
   float cosBeta
       = dot(normalize(g0), normalize(g1 - g0 * R↩
           / r));
   if (cosBeta < 0) return false;
   float sinAlpha = r / length(g0);
   float sin2Beta = 1. - cosBeta * cosBeta;
   float sin2Alpha = sinAlpha * sinAlpha;
   return sin2Beta < sin2Alpha;
}
```

Listing 2: *Returns true if the grain at position `g1` is occluded, given an occluder map rendered using the same projection matrix as the current view. This map contains the position `g0` and inner radius `r` of another grain or a mock value `NONE` (used to clear the buffer before rendering the map). Coordinates are in camera space.*

9. Efficient drawing

Sampling an impostor's maps is a costly operation, both in terms of memory bandwidth and computing power. There are two ways to reject a fragment before it reaches the fragment shader. One rather drastic is to discard the whole point, this was the goal of Section 8. But this is not enough. For many grains beyond the first layer, only a couple of fragments are visible out of the tens or hundreds that it may cover. We still end up with hundreds of millions of fragments to shade, so we leverage another mechanism: *Early-Z Rejection*.

Visibility These hundreds of millions of fragments outnumber by several orders of magnitude the pixel count of a typical HD render (2M pixels) or even a 4K render (8M pixels), so there is mechanically a large proportion of *wasted fragments*, i.e. fragments that reach the fragment shader but are ultimately not visible on screen. This phenomenon is commonly referred as *overdraw* .

Over-shading is not specific to impostors, it is actually the prior motivation of deferred shading. But the core difficulty that impostor rendering introduces is the impossibility to determine the visibility of a fragment before sampling the maps. This deferred shape evaluation prevents us from using strategies such as visibility buffering [**Burns13**].

Early-Z Rejection The early-Z rejection is automatically performed by modern GPU's rendering pipelines [**Sekulic04**]. If a fragment lies behind the one already stored in the output buffer, then it can be rejected without being processed, provided that the shader does not override fragment's depth. Thus the benefits of early-Z rejection depend on the order in which points are rendered, and we have too many points to sort them front to back. Nevertheless, what early-Z rejection tells us is that the visibility does not need to be perfectly solved in order to gain in efficiency. We can split the grains into the *likely visible* ones and the *likely hidden* ones, and render the former first. This first draw call fills almost all pixels with their final value, so the second one sees most of its fragments early rejected. This is referred to below as the *double draw* scheme.

Implementation Fortunately, the question of determining likely visible grains has already been answered: those are the occluder candidates of the occlusion culling step. They represent a thin shell

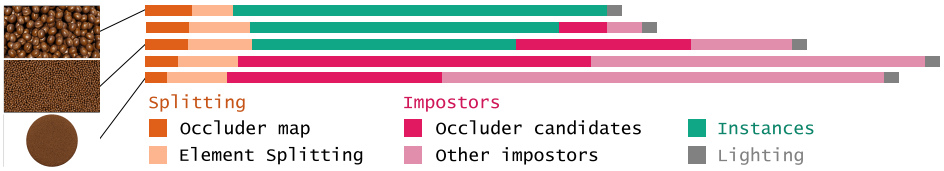

Figure 9: *Breakdown of several frames' draw sequence during a reference shot from tight to large view over a stack of 1.6M coffee beans. On the left-hand side are (top-down): first frame, middle frame and last frame. These results focus on the transition from meshes to impostors.*

of closer grains for which most of the fragments are visible. In practice, we make the splitter distinguish separate elements buffers for occluder candidates and remaining points. When rendering impostor, the same draw call is repeated twice with these different element arrays. This simple change brings a significant speed-up to the overall impostor rendering.

10. Results

Figure 10: *Impostor clouds built from diverse grain models. Impostors use 128 precomputed views ($n = 8$) of 128×128 pixels each.*

The performance of our C++/OpenGL implementation has been measured on an Nvidia GeFroce GTX 1070 graphics chip with 8GB of VRAM, on frames of 1920×1080 pixels. We focus the performance tests on the transition from impostors to meshes, where it is the most critical. We compare our impostor cloud at different angular resolutions to instances of the original grain mesh or a simplified mesh.

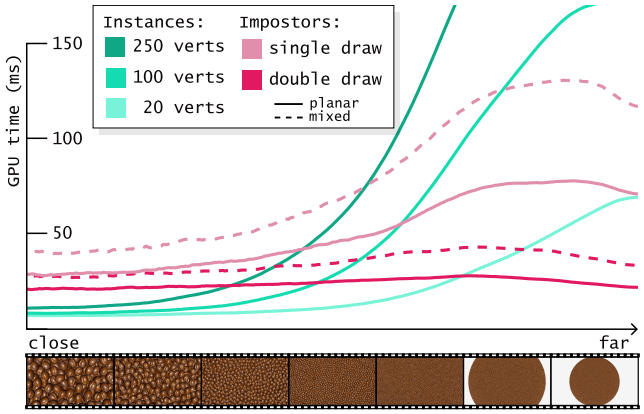

Figure 11: *Render time on a scene made of 1.6M grains. Thumbnails of the test sequence can be found bellow the horizontal axis. Impostors use 128 views of 128 pixels. Both instances and impostors use our occlusion culling method.*

Breakdown The overall render time of a frame is subject to various factors. First, it varies significantly with the view point. In order to grasp the benefits of our method on real case scenarios, we evaluated performances during a backward dolly shot, from tight to large. Left-hand side of Figure 9 shows first, middle and last frames of this test shot and breakdowns of these key frames. This qualitative evaluation already highlight a few points. First, although it is not negligible, the splitting process is not the bottleneck. Second, occluder map render time increases as point size grows on screen. Third, the Z-prepass does not have a significant impact as this draw call involves almost no fragment processing. Fourth, the core element splitting is rather constant, until most grains get frustum culled in closer views. Last, despite being unbalanced in number of points, the first and second draw calls of impostor rendering takes similar times. This is satisfactory as it suggests that we found a reasonable trade-off between rendering a few costly points first and then more points but which are less visible.

Performances This high variability of draw mixture within a single frame makes it hard to draw proper conclusions, so in Figure 11 we compare scenarios without splitting, where only one of meshes or impostor models is used. The timings for impostor rendering do not depend on the original complexity of the grain, so we compare them to several meshes. A first thing to notice is that we indeed

need a hybrid model since when the number of grains within the view frustum is low (close viewpoint) instanced meshes are more efficient than impostors while as it increases impostors eventually outperform instances.

The shape of instance and impostor curves are different because the former is more affected by the number of vertices to draw (vertex bounded) while the latter is related to the number of pixels (pixel bounded). In case of a perfectly pixel bounded rendering, our test shot should take a constant render time. The results of Figure 11 show that it is not the case when naively drawing all the impostors at once (*single draw*). This is because of the large number of overdrawn fragments. Our double draw scheme on the other hand succeeds at reducing overdraw, as shown by its more constant render time. It thus makes our mixed sampling competitive despite its overhead.

As discussed in Section 6, visual accuracy sets a minimal distance at which transitioning from meshes to impostors. These results show that when the grains have shapes requiring a low amount of vertices, pushing this threshold distance further can increase performances. For more complex grains, the threshold is already beyond the cross point between green and red lines so there is no interest in increasing it. Even when combining our approach with usual mesh LoD, the vertex count does not reduce beyond a few tens, so an eventual switch to impostors is beneficial.

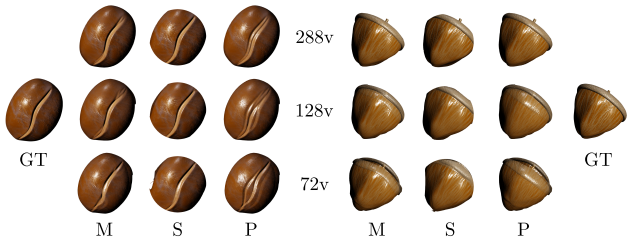

Figure 12: *Impostors rendered using mixed (M), spherical (S) or planar (P) samplings with various number of precomputed views, along with ground truth (GT).*

Visual loss Figures 10 shows impostor clouds of twenty thousand points at different scales and in different scenarios, illustrating the variety of possible grain shapes. Figure 1 is a more extreme example featuring two million grains.

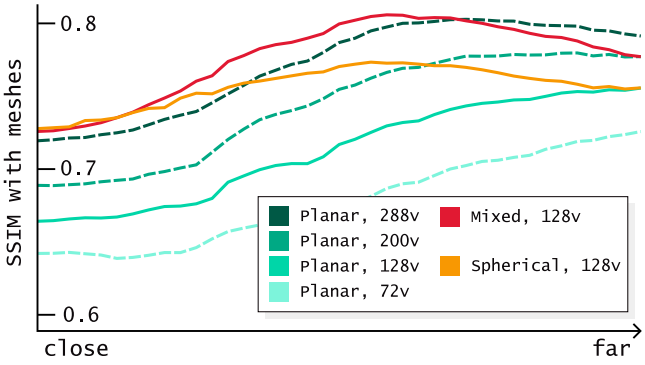

Figure 13: *SSIM measures using different sampling schemes on the coffee bean : planar, spherical and mixed (ours).*

To evaluate the visual loss of our model, we measured the structural similarity (SSIM) between animations rendered using different impostors on one hand and a reference render using meshes on another hand. Figure 13 compares variations of the choice of sampling scheme at fixed memory use with variations of the number of stored precomputed views. The stacked grain used for this example is the coffee bean of Figure 12, left. We see that for equivalent memory requirements, our mixed sampling gives better visual accuracy.

Figure 14: *View frustum with (left) and without (right) our grain occlusion culling. Some of the remaining points may actually be hidden, but it is ensured that no visible point is removed.*

Occlusion culling Figure 14 illustrates the effect of our occlusion culling on a dense volume of grains. As shown by the graph of Figure 15, the ability of our method to cull grains decreases progressively as we relax the hypothesis of a non null inner radius r. The effect of the culling varies with the frame. The green curve was measured with a narrower field of view. The camera rays are in that case more parallel, hence there is less occlusion detected with our method. The blue curve was measured on a more favorable scenario, where grains in the foreground hide significant parts of the whole set.

11. Discussion

Properties Our methods can render stackings of millions of dynamic objects in real time, leveraging the similarity of dense quasi-spherical grains using impostors to design a transition LoD which provides an efficient trade off – both in terms of accuracy and speed – when individual grains only cover a few pixels on the screen. This is tracktable thanks to our new sampling scheme that reduces

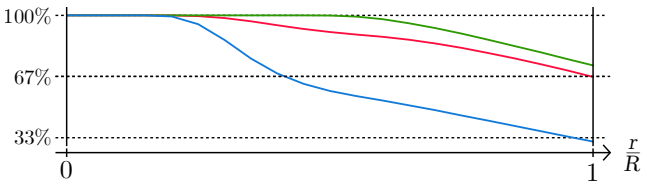

Figure 15: *Proportion of grains still rendered after the occlusion culling step, depending on the inner over outer radius ratio on three different shots. Variations among shots are due for instance to a larger foreground for the blue curve.*

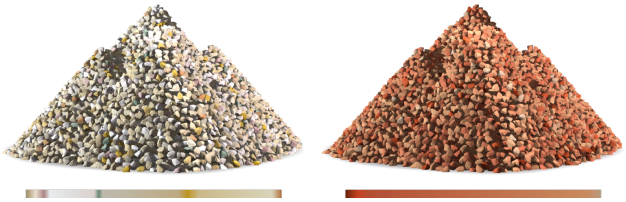

Figure 16: *Impostor cloud where precomputed attributes such as normal vectors are used along with dynamic procedural attributes such as albedo values, which is drawn from the color ramp underneath each image.*

memory usage for a given visual loss, together with a coupled per-grain/fragment occlusion mechanism. Contrary to instancing, the complexity of the impostors is independent on the original model. Thanks to our occluder map and splitting scheme, it is mostly dependent on the output resolution. Our method is compatible with arbitrary animation of the grain positions, as shown in the supplemental video, and scales to large stackings (Figure 17).

Being designed to feed the G-pass of a deferred shading engine, dynamic procedural variations of the grain can be coupled with the precomputed data at render time (Figure 16), reducing further potential repetition effects while expanding visual diversity. Moreover, our method degrades gracefully regarding all of its hypotheses (quasi-spherical grains, density, moderate shape diversity), either in accuracy or in efficiency depending on the application context.

Limitations For non quasi-spherical enough grains, visual loss must be balanced with more precomputed views. At some point, the hypothesis of quasi-spherical shape made by our mixed sampling scheme becomes as invalid as using planar sampling. An extreme example that breaks our hypothesis is a tubular element, e.g. a threaded nut. Also, grains must not intersect each other. We do not change the fragment depth when rendering them, so they are sorted by the depth of their center, provoking popping artifacts in case of intersection. Writing a precomputed depth in the Z-buffer when rendering the impostor is possible, but at the expense of important performance reduction because this would turn off early-Z rejection. Another consequence of this per-grain depth is that the standard shadow maps cannot render grains' self-shadows. The rendering model that we used to render far grains beyond the validity range of our impostor is subject to aliasing. To improve the transition from impostors to pure point based rendering, more advanced existing point-based models could be used. Note that we did not chose to switch to a surface-based representation, such as Bruneton et al. [**Bruneton12**] do, because we did not want to give up on the ability to animate grains. A grain being bounded by a sphere, the corners of precomputed views are always left unused. Todt et al. [**Todt07**] use a distorted mapping to address this issue, but this prevents us from using standard mipmaps.

Future work Our method can be further developed along several diretions. First, we do not use any form of texture compression and use heavy 32-bit color attachments. We could also defer the sampling of attributes in a separate pass to save memory bandwidth.

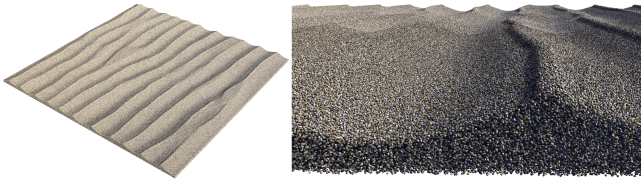

Figure 17: *A large dynamic scene made of 20M sand grains and rendered with our method in 56ms on a GeForce 1070 GPU.*

Impostor rendering would query only the alpha channel, to build a visibility buffer [**Burns13**]. Second, the attribute field captured by our rich impostors have a lower dimensionality than the light field captured by radiance impostors. Hence, there is more redundancy in our representation, that could be compressed better, storing only the mapping from position on the bounding sphere and ray orientation to UV space, which yields interesting filtering issues to address. Last, we could accumulate fragments during the first of the two draw calls. This would enable cross grain alpha blending and hence reduce aliasing when grains come very close to being points. Accumulation disables the benefits of early-Z rejection but it is mostly the second pass that benefits from it.