ISSN 2464-4617 (print)
ISSN 2464-4625 (online)
Computer Science Research Notes - CSRN 3401
http://www.wscg.eu
WSCG 2024 Proceedings

PLOD: Point cloud level of detail for polygon mesh

Eric Konks
Phystech School of Applied Mathematics and Computer Science
Moscow Institute of Physics and Technology
Institutskiy Pereulok, 9
Dolgoprudny, Moscow Oblast, 141701, Russia
eric.konks@gmail.com

Alexandr Shcherbakov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, 119991, Russia
alex.shcherbakov.as@gmail.com

Abstract

Rendering high-polygonal models from distant perspectives has certain performance issues related to high density of subpixel triangles, which can be solved by levels of detail, a classical optimization method. Since a mesh occupies a small area on the screen, an alternative representation of the geometry in the form of a point cloud can replace the original geometry with little or no change to the image, and allowing for significant performance improvements due to the smaller number of primitives rendered. This paper presents a method for automatically generating a point cloud from a polygonal mesh with nested levels of detail. It also considers a method for rendering cloud with dynamically varying cloud density in real time based on view frustum and distance.

Keywords

point cloud, level of detail, impostor, point based rendering, optimization

1 INTRODUCTION

Modern 3D models in real-time computer graphics applications can be extremely detailed, with millions of polygons and high-resolution textures. However, models only occupy a certain area on the screen, which may not correspond to excessive geometry detail, thereby causing performance degradation without any improvement in image quality.

Various rendering technologies may be used to present a complex and heavy 3D scenes [DGY07]. The main methods are primitive rasterization and ray tracing, which have proven to be simple, robust and widely used. Geometry simplification and visibility culling are also common techniques to achieve interactive frame rates for a rendering application.

One of the most common optimization methods are levels of detail [Kru+97]. The basic idea is to create simplified versions of the original model that can replace each other, while preserving or slightly changing the original image. Simplification can occur in a number of ways. First of all by reducing the number of rendered primitives, which naturally affects performance and the amount of memory required. Replacing subpixel triangles with larger triangles increases the coverage of the fragment shader due to a greater ratio between active and helper invocations. Simplification can also affect the materials and lighting shaders used, speeding up the rendering process. It is clear that if a model is simplified significantly, visual quality tends to deteriorate. Therefore, in order for the degradation to have little effect on the image, the simplified model should replace the original model at a certain moment, when the difference between the details is not noticeable on the corresponding area of the screen. In other words, some dynamic real-time switching between levels of detail is required depending on the change of the distance to the camera and consequently the projected screen space size.

Manual adjustment of LOD (level of detail) is hampered by the need to balance performance and quality. Therefore, various optimization techniques and generation methods are designed to speed up the process of creating new assets and models, allowing the author to focus on direct visual representation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2 RELATED WORKS

The most obvious way to create a level of detail is to simplify the geometry by reducing the number of poly-

gons, what can be done automatically using various algorithms.

There are various ways to algorithmically generate simplified polygonal models [GGK02]. For example, clustering-based algorithms that use different criteria depending on the required topology accuracy. Simplification is performed by partitioning vertices into clusters followed by iterative edge collapsing.

Another method is remeshing [Kha+22], where instead of using existing polygons, new vertices are sampled on the mesh surface and a completely different mesh is created by triangulation. This approach produces a fairly uniform distribution of triangles of approximately the same size, but topology details may be compromised due to the discreteness of the sampling. Therefore, it is necessary to adjust the parameters and use sufficient depth and resolution.

LOD generation by mesh resampling can be combined with vertex data compression and streaming to reduce memory consumption [NFS22]. After resampling the mesh, new vertices are merged into patches and the data is quantized. Later, during rendering, the data is decompressed on GPU at the hardware tessellation stage. For mesh parts close the camera, patches have higher tessellation factor and it's decompression version produces more triangles. Thus, the method reduces both memory consumption and rendering time.

Completely different way is to use some alternative, image-based representation, an imposter, instead of the original polygonal representation. The term imposter refers to some object that replaces the original geometry and is rendered faster with little noticeable visual difference [Déc+03]. The method uses very simple geometric objects that can be rendered with a pre-generated image facing the camera. The disadvantages of this approach include the dependence of the image on the angle at which the object is observed. To address this problem, atlases are used in which the impostor consists of multiple images, each corresponding to a specific viewing angle [MFL21]. However, if there is insufficient variation, intermediate angles can provide a visually poor substitute for the original geometry. In addition, given that modern lighting models require material data, the memory consumption of impostors increases significantly.

One drawback of the approaches described above is that there is no unambiguous criterion by which the transition to the simplified model should be made. It is unclear at what point this transition can be made imperceptibly and when subpixel detail does not affect the image. To deal with this problem, a triangular mesh can be broken down into small clusters of vertices – meshlets [Bad+20]. Each meshlet is a set of connected vertices that represents some part of the geometry. Using mesh shaders, we can perform occlusion and LOD selection based on these individual meshlets, thereby maintaining a constant number of polygons on the screen. For example, Nanite technology in the Unreal Engine 5 heavily relies on the cluster mesh representation [Kar22]. This technology consists of automatic generation of a cluster representation of the mesh with levels of detail, a cluster streaming system, software rasterization and dynamic switching. In addition, it uses impostor atlas for the most distant LOD.

Triangular polygons are not the only way to represent geometry. For example, points can also be used as a drawing primitive [LW00]. They define the position of some point in the space in which the object exists, but, unlike polygons, they do not uniquely define a surface. Therefore, points must have sufficient density to visually simulate the continuity of a surface. In addition to position, a point can have other characteristics such as normal, tangent, color, metalness and other material properties.

The idea of using points as impostors was first introduced by Wimmer et al. [WWS01]. Since the screen resolution is limited, with enough point primitives it is possible to cover the required number of pixels on the screen without visible holes. This paper discusses a method for image-based generating of point cloud using multiple virtual cameras that perform ray tracing to produce object samples. Each point is used to sample a specially filled texture with the point radiance contribution for the current view. This point cloud is then rendered using the hardware drawing capabilities of point primitives.

Point primitives in graphics pipeline may be used to render raw point cloud [Wim15]. The method uses glPointSize for high-quality splatting to get good visual quality and appropriate surface representation. However some graphics API do not fully support point size, leaving us to use only 1 pixel points. Also screen space filters may be used [PGA11] to remove artifacts due to low cloud density. But these additional computations reduce the performance, which is very important for our usage case.

Our work further develops point cloud representation of polygon mesh using modern generation method and rendering techniques.

3 PROPOSED SOLUTION

Cloud generation

To create an alternative representation of geometry in the form of a point cloud, it is necessary to sample points on the surface of models in a certain way. The sampling should provide visual correspondence of the geometry to the original polygonal mesh from any possible angles, as well as have a number of properties that allow to maximize the quality of surface sampling. For

this task, the distribution based on blue noise is suitable, the frequency parameters of which allow to reduce the number of 'holes' or even totally eliminate them. Uniform distribution is extremely important because it allows to create the illusion of surface continuity without using real-time smoothing or restoring filters. Uniform distribution also has a positive impact on performance, as fewer points fall within a single pixel, reducing overdraw. To obtain such a distribution, we modernize the sampling algorithm using the Poisson sampling algorithm presented in [CCS12].

The cloud generation scheme consists of generating a pool of points and Poisson disk filtering. The first step sets the initial cloud surface density D_0, on which the minimum sampling distance will depend, as well as a density reduction step to specify nested levels of detail. This results in some set of densities D_i and $\forall i < j \Rightarrow D_i > D_j$.

Then we are generating a pool of points by sampling points on the surface of polygons with the largest density. The *pool* may be represented by some spatial hash container. The algorithm listing 1.

Listing 1: Point sampling

```
vector<float> densities = {D0, ... };
for (const auto &p : polygons)
{
  float area = getPolygonArea(p);
  int count = (size_t)area * D0;
  count = max(count, 1);
  while (count--)
  {
    auto sample = polygonSample(p);
    pool.add(sample);
  }
}
return pool;
```

For each density, starting from the lowest density, a Poisson disk is sampled from the *pool* of pre-generated points on the surface of the mesh. Moreover, the set of points S_i corresponding to the smaller density is a subset of the larger density $S_i \subset S_{i-1}$. The algorithm listing 2.

Listing 2: Cloud generation

```
int N = densities.size() - 1;
for (int lvl = N; lvl >= 0; --lvl)
{
  float density = densities[lvl];
  float r = poissonRadius(density);
  // Remove samples from next lods
  // as part of the current lod
  for (int i = lvl + 1; i <= N; ++i)
    for (sample : samples[i])
      pool.removeSamples(sample, r);
  // Then sample what is left
  while (!pool.isEmpty())
  {
    auto sample = pool.pop();
    samples[lvl].push_back(sample);
    pool.removeSamples(sample, r);
  }
  pool.restore();
}
return samples;
```

After sampling, we have a list of sets *samples*, where $samples_i$ corresponds to $S_i \setminus S_{i-1}$. We can combine all sets into a single list, where first n_i points of set S_0 correspond to a cloud of set S_i with surface density D_i.

Since the points are sampled on the surface of polygons, all attributes of the corresponding vertices are interpolated. Each point in the cloud has a position, normal and texture coordinates. So by rendering this cloud using S_0 as a vertex buffer, we can sample all the necessary textures and bake the material into points, which will save us from having to use textures later. However, this approach has a couple of difficulties. First, it requires no randomization of texture coordinates, which can occur in non-trivial shaders. Second, a point itself does not provide any derivatives used to compute the mip level of the material texture. This problem can be solved by knowing in advance at what distance the cloud will be further rendered or by calculation based on Poisson radius. The mip level calculation has not been considered in detail in this paper. Since the cloud is designed to be rendered at long distances, the use of some constant mip level is appropriate in this case.

Dynamic cloud density

As a result of generation, we have a point cloud with all necessary data for drawing on the screen (Fig 1).

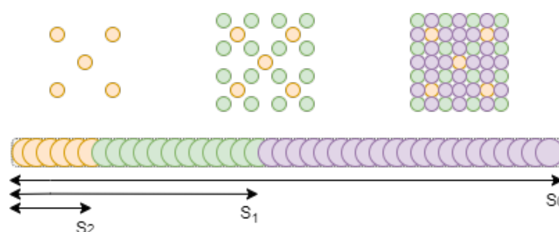

Figure 1: Point cloud list with nested densities

The point cloud with the structure of nested densities allows us to dynamically vary the number of drawn primitives. To do this, we need to find such a density of the cloud that, when projected on the screen, the density is sufficient to obtain a complete image without hole artifacts. Knowing the camera parameters and object position in view space we can estimate this value individually for each instance of the object.

Let L be the distance to the cloud, z_{near} be the near plane of the camera frustum, $(target_x, target_y)$ be the render resolution of target texture. Then we can find the surface density D_{target} that the cloud must have at distance L and making an angle α with the camera forward vector to completely cover the given area on the screen without noticeable holes.

$$D_{target} \approx \frac{target_x target_y}{zNearArea} \frac{z_{near}^2}{L^2 cos^2 \alpha} \quad (1)$$

As a result, it is sufficient to draw only the first n_i points corresponding to the smallest upper bound of the possible density D_i, where $D_i >= D_{target}$ and $D_{i-1} < D_{target}$. The calculation of the number of primitives for vertex input can be performed on the GPU using compute or mesh shaders, which may give even more performance benefits to the application.

Rendering

Points can be rendered in two ways: using the compute or graphics pipeline. The compute method based on 64 bit atomic operation is well described in [SKW22] and achieves better performance in comparison to the hardware method. However the method has special render architecture and relies on 64 bit unsigned textures to perform depth testing and visibility calculation. Since our approach is a hybrid approach in which polygonal meshes are used equally with point clouds, we choose the graphics pipeline as it simplifies the implementation of the method in classical rendering.

We will look at several optimizations that noticeably improve performance when rendering point primitives.

Normal culling. Since unlike polygons points have no face, we need somehow determine orientation to cull out points, that can't be visible. In this case, point normal can be used for culling calculation. So culling can be done by the hardware in a a view port clipping by setting the position of a point in the vertex shader outside the boundary of the clip space. The condition is that the point-to-camera vector and the normal vector must point in different direction. This reduces the number of fragment shader invocations and allows us for significant performance improvements in case of not too dense clouds.

Software vertex fetching. To perform transformation and normal culling, it is not necessary to have information about the material, which is always fetched by the input assembly. To get only the necessary data, we obtain the data by sampling vertex buffer directly with an empty vertex shader input. This optimization saves us memory bandwidth and reduces the total number of read operations, since the vertex shader gets only the position and normals. Subsequently, only for points that have not been culled, the fragment shader fetch information about the shading material.

Conservative rasterization. Some graphics API allow conservative rasterization for point primitives, which is very useful for our method, since it increases point screen coverage. In this mode, a single primitive can produce up to 4 fragments. This means that visually the surface density increases by the same number of times. That is, we can render significantly fewer primitives while relatively preserving the same image.

4 EXPERIMENTAL RESULTS

As a demonstration of the method, a cloud was generated for a high-polygonal geometrically small mesh. Performance measurements was performed on the RTX 3060 at 1920x1080 screen resolution and using the Vulkan API. Additionally, the renderer uses temporal anti-aliasing, which works great for point primitives.

The density steps for the point cloud are degrees of two. Along with performance comparison, we compared images using mean square error (MSE) and structural similarity index measure (SSIM).

A highly polygonal mesh consists of 60k triangles, while the cloud at maximum density contains ten times fewer primitives, which is already sufficient for mid-range display with comparable visual quality (Fig. 2). Point cloud has significantly better performance compared to the high-polygonal mesh, and render time tends to decrease gradually (Fig. 3). The tendency has clear steps where cloud density switches to a lower level of detail (60m, 80m, 120m, 160m).

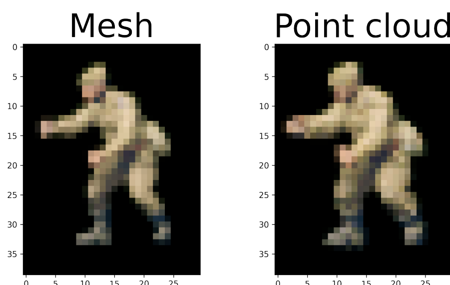

Figure 2: High polygonal mesh. Image comparison at 50 meters

MSE and SSIM metrics has pretty low variation and confirm the similarity of the rendered images (Fig. 4 Fig. 5).

A low-polygon mesh contains 1k polygons, and the cloud for it contains 41k points (Fig. 6). Since the triangles are quite large, the point cloud loses at short and medium ranges in terms of performance (Fig. 7), and only at large distances (250m) does the cloud gain an advantage. This result is not surprising since the cloud has much more compute pressure on the vertex processing stage. However, hardware conservative rasterization can significantly reduce the number of vertex shader calls, allowing the cloud to have fewer rendered primitives, but at the cost of losing visual detail.

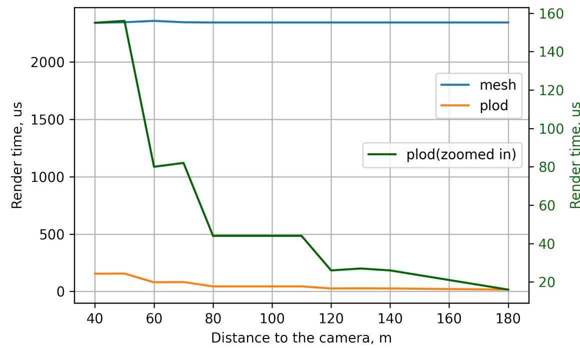

Figure 3: High polygonal mesh. Performance comparison

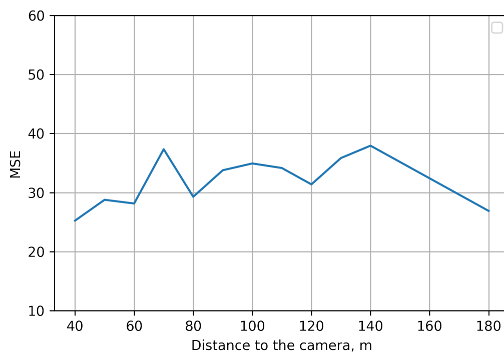

Figure 4: High polygonal mesh. MSE

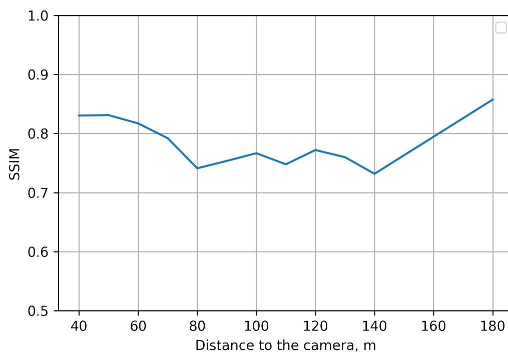

Figure 5: High polygonal mesh. SSIM

MSE and SSIM metrics for point cloud rendered with conservative rasterization shows less image similarity, but the overall scores trend is identical (Fig. 8 Fig. 9).

In the presence of multiple rendering optimizations, the bottleneck of the current rendering method is the rasterization stage, which limits the SM occupancy of the vertex stage. Comparing the images using MSE and SSIM scores, it can be seen that, the point cloud image tends to degrade due to the rasterization of points at the edge of the cloud, which slightly expands the silhouette of the model. Full distance range image comparisons for high-polygonal mesh (Fig. 10a) and for a low-polygonal large mesh (Fig. 10b).

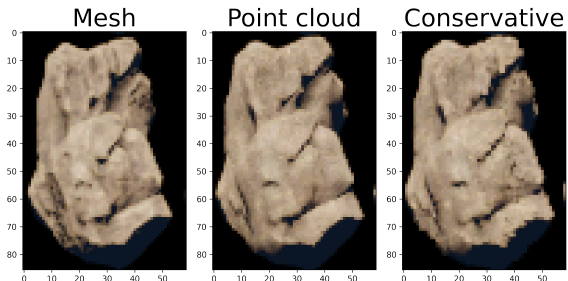

Figure 6: Low polygonal mesh. Image comparison at 50 meters of mesh, point cloud and conservative point rasterization

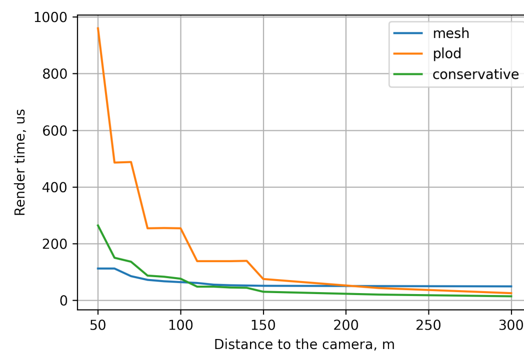

Figure 7: Low polygonal mesh. Performance comparison

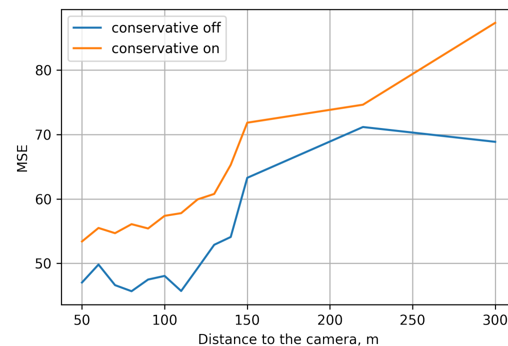

Figure 8: Low polygonal mesh. MSE

5 CONCLUSION

We presented a method for rendering and generating a special type of level of detail that significantly optimizes the long-range rendering of high-polygonal models with non-critical image changes. Due to dynamic density, each mesh has a distance at which point cloud may replace the image with performance gains. The point cloud level of detail shows significant performance advantages for highly polygonal models even at medium distances. In this method, low-polygon meshes can be replaced at the large distances at which imposters are typically used. Therefore, the point cloud can be considered as 3d impostors that can incorporate mesh detailing and impostor performance.

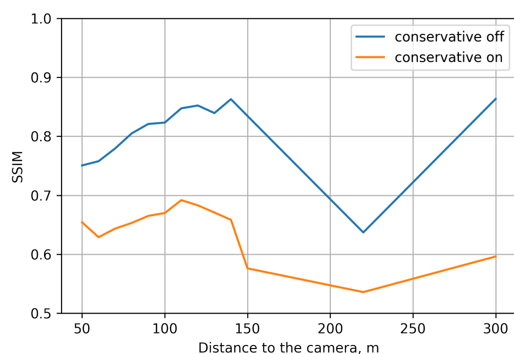

Figure 9: Low polygonal mesh. SSIM

The disadvantages of point clouds are the small image changes that can be noticeable, especially with conservative rasterization, due to the difference between point and triangle rasterization and the difference in mip levels between mesh rendering and point sampling.

Another drawback is the dependence on screen resolution, so we need to draw more points to match the screen space density, losing performance due to redundant vertex invocations. However compute rasterization and point size in some API are able to mitigate this limitation, by increasing the screen primitive size.

Moreover, by increasing the size of primitives and decreasing their number, we can adjust the ratio between image quality and performance. Finding the optimal ratio between quantity and size is a task for future research.

REFERENCES

[Bad+20] Abhishek Badki et al. "Meshlet Priors for 3D Mesh Reconstruction". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 2846–2855.

[CCS12] Massimiliano Corsini, Paolo Cignoni, and Roberto Scopigno. "Efficient and Flexible Sampling with Blue Noise Properties of Triangular Meshes". In: *IEEE transactions on visualization and computer graphics* 18 (Jan. 2012), pp. 914–24.

[Déc+03] Xavier Décoret et al. "Billboard clouds for extreme model simplification". In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 689–696.

[DGY07] Andreas Dietrich, Enrico Gobbetti, and Sung-Eui Yoon. "Massive-Model Rendering Techniques: A Tutorial". In: *IEEE Computer Graphics and Applications* 27.6 (2007), pp. 20–34.

[GGK02] Craig Gotsman, Stefan Gumhold, and Leif Kobbelt. "Simplification and Compression of 3D Meshes". In: *Tutorials on Multiresolution in Geometric Modelling* (Feb. 2002).

[Kar22] Brian Karis. "Journey to Nanite". en. In: *High Performance Graphics* (2022).

[Kha+22] Dawar Khan et al. "Surface Remeshing: A Systematic Literature Review of Methods and Research Directions". In: *IEEE Transactions on Visualization and Computer Graphics* 28.3 (2022), pp. 1680–1713.

[Kru+97] Mike Krus et al. "Levels of detail & polygonal simplification". In: *XRDS* 3 (1997), pp. 13–19.

[LW00] Marc Levoy and Turner Whitted. "The Use of Points as a Display Primitive". In: 2000.

[MFL21] Martin Misiak, Arnulph Fuhrmann, and Marc Erich Latoschik. "Impostor-based Rendering Acceleration for Virtual, Augmented, and Mixed Reality". In: *Proceedings of the 27th ACM Symposium on Virtual Reality Software and Technology*. VRST '21. Osaka, Japan: Association for Computing Machinery, 2021.

[NFS22] Anton Nikolaev, Vladimir Frolov, and Alexandr Shcherbakov. "Mesh compression method with on-the-fly decompression during rasterization and streaming support". en. In: 2022, pp. 209–216.

[PGA11] Ruggero Pintus, Enrico Gobbetti, and Marco Agus. "Real-time rendering of massive unstructured raw point clouds using screen-space operators". In: *Proceedings of the 12th International Conference on Virtual Reality, Archaeology and Cultural Heritage*. VAST'11. Prato, Italy: Eurographics Association, 2011, pp. 105–112.

[SKW22] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. "Software Rasterization of 2 Billion Points in Real Time". In: *Proc. ACM Comput. Graph. Interact. Tech.* 5.3 (July 2022).

[Wim15] Michael Wimmer. "Rendering Large Point Clouds in Web Browsers". In: 2015.

[WWS01] Michael Wimmer, Peter Wonka, and François Sillion. "Point-Based Impostors for Real-Time Visualization". In: Jan. 2001, pp. 163–176.

(a) Image comparison. Left is a high polygonal mesh. Right is a point cloud

(b) Image comparison. Left is a low polygonal mesh. Middle is a point cloud. Right is conservative rasterization for point cloud