Alternative sampling method in Mesh2Splat

DH2323 Computer Graphics and Interaction

Oskar Hokkanen Eriksson (oskhe@kth.se)
Jingwen Zhuang (zhuan@kth.se)

KTH Royal Institute of Technology

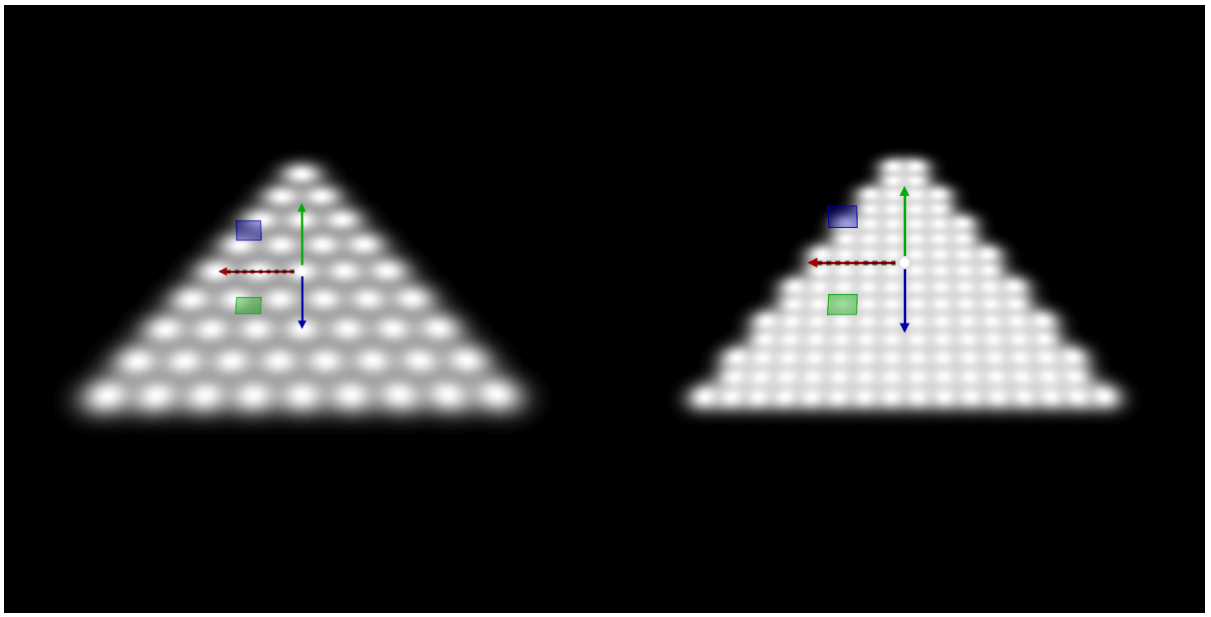

Figure 1: Our implementation (left side) compared to the original method (right side), comparing jagged edges of a triangle.

Abstract

When sampling a surface in computer graphics there are many approaches. Mesh2Splat is a surface splatting approach that is used to convert 3D meshes into 3D Gaussian Splatting. In this paper we use barycentric coordinates to sample triangles in the Mesh2Splat solution in order to solve problems such as jagged edges and distortion. Although our solution is far from perfect, it shows promising results in solving the problems.

1 Introduction

Mesh2Splat is a fast surface splatting approach used to convert 3D meshes into 3DGS (3D Gaussian Splatting) models by exploiting the rasterizer's interpolator. (Scolari 2024; Scolari 2025)

One of the most important steps in Mesh2Splat is to sample the points where the Gaussians will be placed for the input mesh. Currently Mesh2Splat is exploiting the OpenGL's built-in rasterizer, where each pixel is one sampled point for spawning gaussians. There are also other

ways to sample points for a triangle in a mesh, such as Poisson disk sampling.

While the method is clever and fast it can pose some problems, for example, jagged edges when rendering a set of Gaussians representing a triangle caused by square grid rasterization and distorted sampled points caused by orthogonal projection.

In this paper we will explore sampling techniques to prevent the mentioned problems above, and try to apply those techniques to Scolari's (2024) Mesh2Splat method.

2 Related work

In the following section, we will introduce basic rendering concepts, Mesh2Splat, its sampling method, as well as Blue noise sampling.

Rasterization is the process of converting an image in vector graphics format to raster graphics format (pixels), making it viewable on a monitor. Triangles are usually the main primitives that models in 3D space are built upon, partly because of their simplistic and flexible shape that makes it easy to build almost any model from it. In recent years, Gaussian splatting has been used for rendering 3D scenes, providing high-quality, fast, real-time graphics.

2.1 Mesh2Splat implementation

The steps Mesh2Splat employs in order to render 3D Gaussians are: The system loads a mesh, passes it through the vertex shader, and then computes Gaussians' properties for individual triangles in the geometry shader stage. The system generates sampling points from the rasterization stage, and then stores data in a buffer in the fragment shader stage. A more detailed explanation of each step can be found in the original paper (Scolari, 2024).

2.1.1 Mesh2Splat sampling

In the rasterization step of the Mesh2Splat method, 3D Gaussians are placed at every pixel that makes up the triangle in the triplanner orthogonal plane. Utilizing GPU shaders, the process of mapping each Gaussian to each pixel is efficient and can be done in parallel. The graphics pipeline can therefore be used to sample each triangle, generating the 3D-position, normal, and UV coordinates for each point where a 3D Gaussian will be placed. (Scolari, 2024)

The orientation and scale are computed in the Geometry Shader and are then sent with the rest of the properties to the rasterization stage, where a Gaussian is placed for each pixel with all the data interpolated, exploiting the rasterizer. All the generated Gaussians are then stored in a shared SSBO buffer in the fragment shader stage. (Scolari, 2024; Scolari, 2025)

2.1.2 Current shortcomings

As mentioned in the introduction, one of the problems that might arise from the current method is jagged edges and distortion.

Scolari mentions many directions for future work on his solution. One direction is to create a technique that performs the primitive conversion in 3D space, which would remove the need for the triplanner projection.

We did a pilot test to demonstrate that using orthogonal projection during the rasterization step would cause loss of sampled points for more tilted triangles. The results shows that if we are sampling points directly in 3D space using either barycentric coordinates or a custom software rasterizer, it would preserve the sample point count no matter how the triangle is being tilted in the space (Figure 2).

```
PS C:\Users\lambo\Developer\mesh2sp
lat_poc_gpu\build\Debug> ."C:/Users
/lambo/Developer/mesh2splat_poc_gpu
/build/Debug/mesh2splat_poc.exe"
Flat triangle coordinates:
(0, 0, 0)
(1, 0, 0)
(0, 0.5, 0)
Tilted triangle coordinates:
(0, 0, 0)
(1, 0, 0)
(0, 0.353553, 0.353553)

--- Counts ---
CPU Bary flat  : 561
CPU Bary tilted: 561
CPU Raster flat: 625
CPU Raster tilt: 625
GPU flat       : 576
GPU tilted     : 407
```

Figure 2. Results from the pilot test.

The problem with jagged edges for each triangle in the mesh after conversion is because the sampling of Gaussian spawn points are from the rasterizer, which means the sampled points are placed in a square grid. For triangles' edges that are not perfectly aligned with the grid will have jagged sample points (Figure 3).

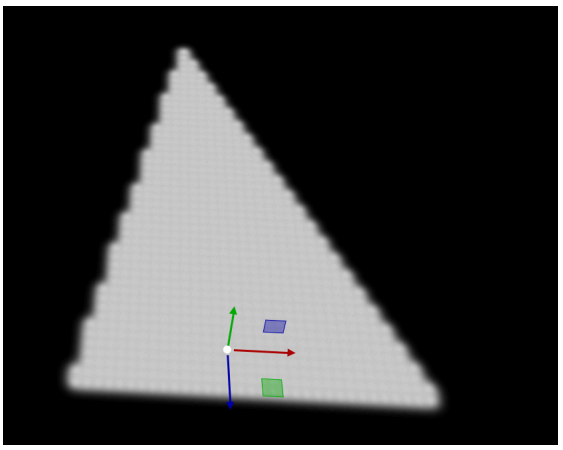

Figure 3. Jagged edges of a triangle using the Mesh2Splat sampling method.

2.2 Sampling alternatives

There are many different sampling methods, one of which is random Blue noise sampling.

Random sampling randomly adds values to an area to be sampled. Blue noise sampling aims to create a more uniform sample distribution than other methods while still being random. This method can help reduce anti-aliasing.

3 Implementation and results

One way of sampling uniform points in a triangle is using barycentric coordinates. Given a triangle defined by its vertices $v0, v1, v2 \in R^2$ $(or\ R^3)$, any point p inside the triangle can be expressed as:

$$p = \lambda_0 v_0 + \lambda_1 v_1 + \lambda_2 v_2$$

subject to:

$$\lambda_0 + \lambda_1 + \lambda_2 = 1,\ \lambda_i \geq 0\ for\ i = 0, 1, 2$$

We define an integer parameter $N \geq 1N$, representing the number of subdivisions per edge. The set of all integer barycentric

coordinates satisfying $i + j + k = N$ where $i, j, k \geq 0$, corresponds to a discrete sampling of the triangle. Each such triplet (i, j, k) can be mapped to a point within the triangle via:

$$\lambda_0 = \frac{i}{N},\ \lambda_1 = \frac{j}{N},\ \lambda_2 = \frac{k}{N}$$

The corresponding point p_{ijk} is computed as:

$$p_{ijk} = \lambda_0 v_0 + \lambda_1 v_1 + \lambda_2 v_2$$

The set of valid indices (i, j, k) for this sampling is:

$$\left\{(i, j, k) \in Z^3_{\geq 0} \mid i + j + k = N\right\}$$

This generates $\frac{(N+1)(N+2)}{2}$ uniformly spaced points across the triangle, forming a regular triangular grid.

3.1 Integrate into Mesh2Splat

Our CPU sampling system converts 3D mesh triangles into 3D Gaussian splats by performing dense sampling across each triangle's surface and computing material properties at each sample point.

3.1.1 High-Level Process Flow

The conversion is orchestrated by `Renderer::convertMeshToGaussiansCPU`, which:

1. Iterates through all loaded meshes in renderContext.dataMeshAndGlMesh
2. For each triangle face, calls sampleTriangleCPU_Internal to generate Gaussians
3. Accumulates all generated Gaussians into renderContext.readGaussians
4. Updates the GPU buffer via updateGaussianBuffer

3.1.2 Triangle Sampling Algorithm

3.1.2.1 Geometric Setup

For each triangle defined by vertices p0, p1, p2:

```
glm::vec3 e1 = p1 - p0;  // First edge
glm::vec3 e2 = p2 - p0;  // Second edge
glm::vec3 n = glm::normalize(glm::cross(e1, e2));  // Face normal
```

3.1.2.2 Orthonormal Basis Construction

Creates a local coordinate system for the triangle:

- X-axis: Normalized first edge (e1)
- Y-axis: Computed via cross product to be perpendicular to both X and normal
- Z-axis: Triangle normal

This basis is converted to a quaternion for Gaussian orientation. All Gaussians will face the same direction as the triangle.

3.1.2.3 Scale Computation

Gaussian scales are derived from triangle geometry, and the scale will be calculated based on the average length of all edges of the triangle:

```
float triangleArea = 0.5f * glm::length(glm::cross(e1, e2));
float avgEdgeLength = (glm::length(e1) + glm::length(e2)) * 0.5f;
float isotropicScale = (avgEdgeLength / float(m)) * scaleFactor;
isotropicScale = std::max(isotropicScale, 1e-7f);
glm::vec3 S(isotropicScale, isotropicScale, 1e-7f);
```

3.1.2.4 Barycentric Sampling

Uses a triangular sampling pattern based on barycentric coordinates:

```
for (int u = 0; u <= m; ++u) {
    for (int v = 0; v <= m - u; ++v) {
        float fu = float(u) / m;
        float fv = float(v) / m;
        float fw = 1.0f - fu - fv;  //
Third barycentric coordinate
```

This ensures uniform distribution across the triangle surface with (m+1)(m+2)/2 total samples.

3.1.3 Material Property Sampling

3.1.3.1 Attribute Interpolation

For each sample point, vertex attributes are interpolated using barycentric weights:

- Position: P = fw * p0 + fu * p1 + fv * p2
- Normal: Interpolated vertex normals
- UV coordinates: For texture sampling
- Tangent vectors: For normal map transformation

3.1.3.2 Texture Sampling

The `sampleTextureAtUV` function performs CPU texture lookups:

```
// Convert UV to pixel coordinates
int x = static_cast<int>(uv.x *
textureInfo.width) % textureInfo.width;
int y = static_cast<int>(uv.y *
textureInfo.height) % textureInfo.height;
```

Supports different texture types:

- Base Color: RGBA albedo
- Metallic-Roughness: Metallic (blue channel) + Roughness (green channel)
- Normal Maps: Tangent-space normals
- Emissive: Self-illumination

3.1.3.3 Material Property Computation

`computeMaterialPropertiesAtUV` combines texture samples with material factors:

```
// Base color with factor modulation
glm::vec4 baseColor =
sampleTextureAtUV(material.baseColorTexture
, uv);
outColor = baseColor *
material.baseColorFactor;
// PBR properties
float metallic = metallicRoughnessTexel.b *
material.metallicFactor;
float roughness = metallicRoughnessTexel.g
* material.roughnessFactor;
```

3.1.4 Normal Map Processing

The system performs a proper tangent-space to world-space normal transformation:

```
if (glm::length(normalMapNormal) > 0.1f) {
    glm::vec3 tangent =
glm::vec3(interpolatedTangent);
    glm::vec3 bitangent =
glm::cross(interpolatedVertexNormal,
tangent) * interpolatedTangent.w;
    glm::mat3 TBN = glm::mat3(tangent,
bitangent, interpolatedVertexNormal);
    finalNormal = glm::normalize(TBN *
normalMapNormal);
}
```

3.1.5 Gaussian Data Structure Population

Each sample generates a `utils::GaussianDataSSBO` with:

- Position: g.position = glm::vec4(P, 1.0f)
- Scale: g.scale = glm::vec4(S, 0.0f) (isotropic scaling)
- Rotation: g.rotation = glm::vec4(Q.w, Q.x, Q.y, Q.z) (quaternion)
- Color: g.color = glm::vec4(sh0, opacity) (RGB + alpha)

- PBR Properties: g.pbr = glm::vec4(metallic, roughness, 0, 0)
- Normal: g.normal = glm::vec4(finalNormal, 0.0f)

3.2 Our results and compared to the original Mesh2Splat

In this section we will provide comparisons with the original method and discuss how one might evaluate the results.

3.2.1 Performance

3.2.1.1 Algorithmic complexity

Triangle Processing Complexity

```
// Per triangle: O(m²) where m is sampling density
for (int u = 0; u <= m; ++u) {
    for (int v = 0; v <= m - u; ++v) {
        // Work per sample point
    }
}
```

Total samples per triangle: (m + 1) × (m + 2) / 2
Scaling implications:

- Quadratic growth: Doubling m produces ~4× more Gaussians
- Memory explosion: Large meshes with high sampling can exhaust RAM quickly
- Processing time: Each triangle's conversion time scales as O(m²)

Overall System Complexity

```
Total Gaussians = Σ(triangles) × (m + 1) × (m + 2) / 2
Total Processing Time = O(N × m²)
Where N = number of triangles in the mesh.
```

3.2.1.2 CPU processing bottlenecks

Our system uses the CPU to execute functions and compute all the values required to generate Gaussians for each triangle in the mesh. It is a single-core operation, and it can only process one triangle at a time.

Compared to the original GPU version, our method is significantly slower, but our system is a proof-of-concept. It has the potential to utilize the GPU pipeline using compute shader since the operation is triangle-independent.

Another consideration is that even if we did it with compute shaders, it is hard to say if the performance will beat the built-in rasterizer of OpenGL, which is highly optimized for the GPU.

3.2.2 Jagged edges

Compared to the original method (Figure 4), our method (Figure 5) yields better results in terms of reducing jagged edges for converted triangles.

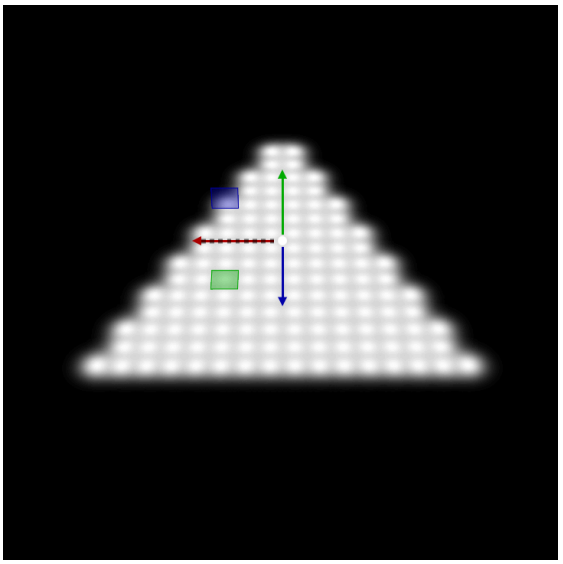

Figure 4. Triangle rendered with the original sampling method.

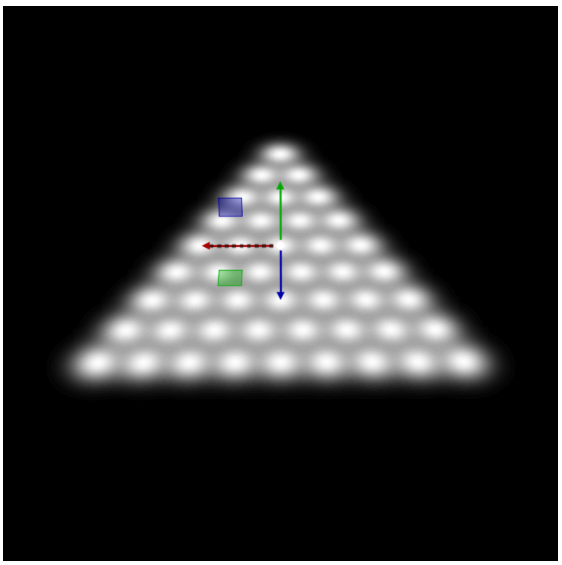

Figure 5. Triangle rendered using our sampling method.

3.2.3 Assessment of visual fidelity

Similar methods to Scolari (2024) could be a suitable method to assess the visual fidelity of the solution. Three cases could be measured against each other: Our solution, the Mesh2Splat solution, and a ground truth.

The measurements to be assessed are: Structural Similarity Index Measure (SSIM), Peak Signal-to-Noise Ratio (PSNR), and Learned Perceptual Image Patch Similarity (LPIPS). These metrics measure image quality based on a base image, the loss of quality of an image or and the perceptual similarity of images.

Scolari (2024) uses the Halcyon Engine to perform the tests. This would maybe be an option for us, however, there are other possibilities that could be picked if an evaluation were to be done. Scolari (2024) also mentions that the PBR material properties were fixed during comparison, which is something we would also consider.

Scolari (2024) uses the Scifi Helmet 3D model by Pavlovic (2024), this helmet would be one of the models compared when assessing the visual accuracy compared to the ground truth and the Mesh2Splat method. The model is used in this work when comparing the sampling density and can be found in Figure 6.

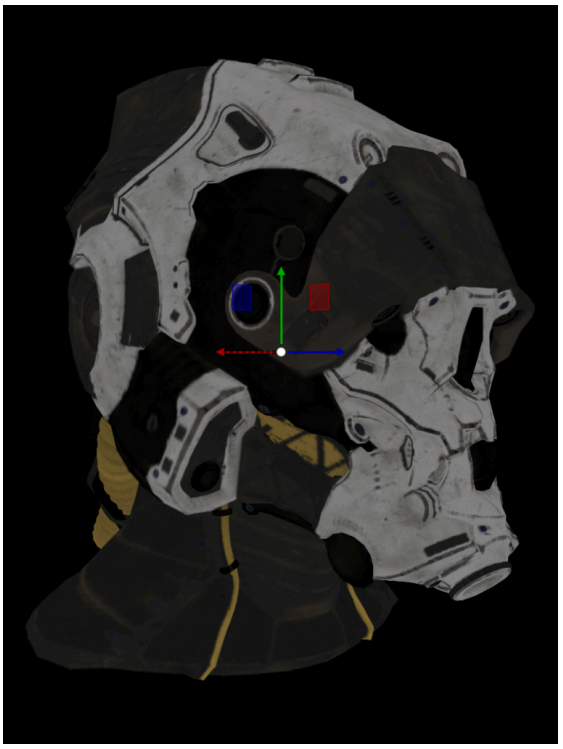

Figure 6. Our method with a sampling density of 16 per triangle edge, Gaussian count: ~3.5M

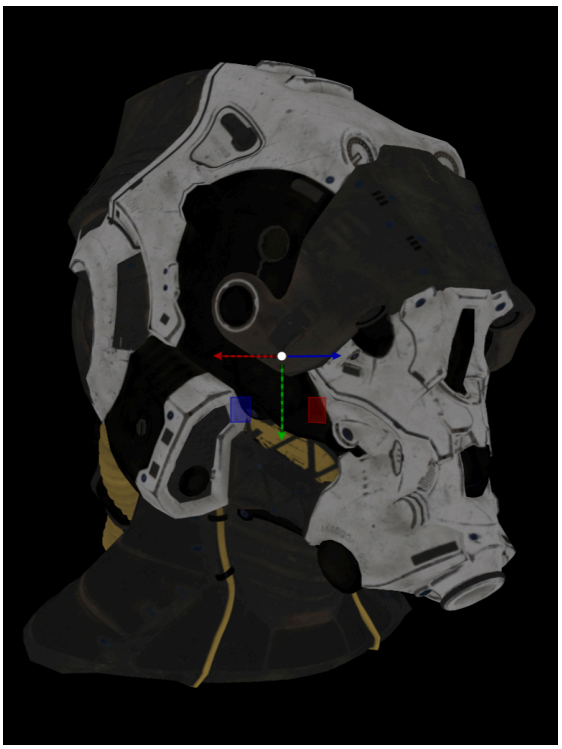

Figure 7. Original Mesh2Splat with sampling density 1 in 1024x1024 resolution, Gaussian count: ~2.9M

4 Future work

4.1 Move to GPU

Since our implementation was implemented with the use of CPU and the original method was implemented with GPU, it is safe to say that our implementation was not faster. Our method is a proof-of-concept where we could improve the performance later for the same calculation.

Currently, our method is only using one CPU core to compute the Gaussians for each triangle, one at a time. But our method has great potential for utilizing the GPU power. Our calculation is done per-triangle, it is possible for us to use compute shaders where the GPU cores can process each triangle in parallel.

4.2 Merge overlapping Gaussians

Currently, our method will sample the spawning points for Gaussians for each triangle within its face and on its edge to ensure a consistent pattern. This will, however, generate overlapping Gaussians on all edges of each triangle in the mesh, which will cause an overhead in performance. This can be further optimized by merging all Gaussians in the same position with their properties.

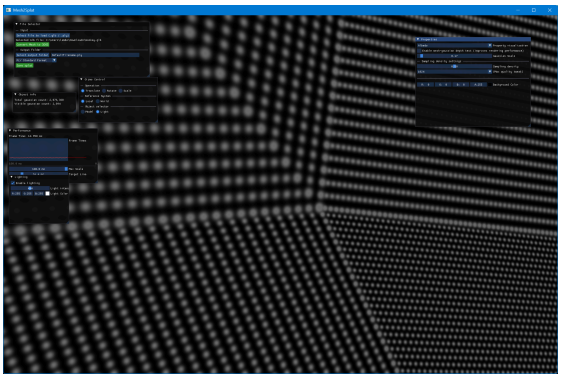

Figure 8. Overlapping Gaussians on the edges.

4.3 Sampling limit for smaller triangles

Our current method may have performance overhead and redundant Gaussians generated for a mesh that has different-sized triangles. As we can see in Figure 9, when we reduce the scale of rendered gaussians, we can see that areas with more flat surfaces and bigger triangles will have less gaussian coverage, while areas with more detailed, smaller triangles will have more gaussian coverage. It is expected in our method, but it can be further optimized to set a limit on how many Gaussians can be generated for smaller triangles to reduce performance overhead.

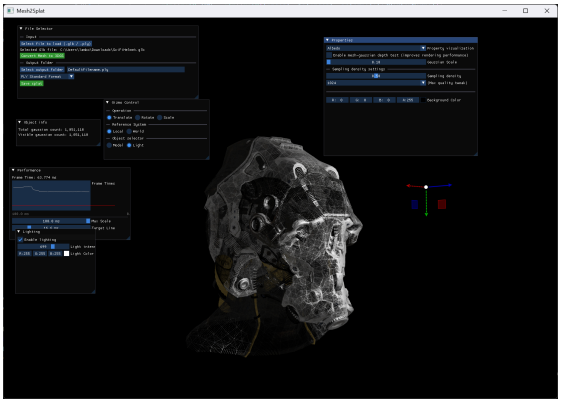

Figure 9. Scaled-down Gaussians on Sci-Fi helmet model.

4.4 Reduce sampling for slender triangles

Our method performs great when the triangle tends toward an equilateral shape, and it will yield the best result if the triangle is an equilateral triangle. However, some meshes that consist of acute-angled triangles with one angle close to zero, will affect the distribution of the sampling points (Figure 10).

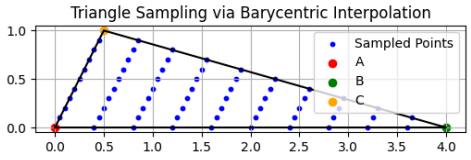

Figure 10. A slender triangle with our sampling solution

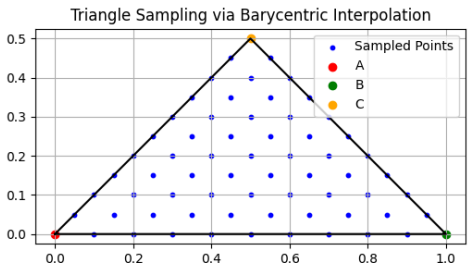

Figure 11. Triangle closer to an equilateral triangle, resulting in better distributed points.

The slender triangles may benefit from reduced sample points. One possible solution is to further divide the triangle into smaller triangles, and then sample within the small triangles.

AI tools used

When trying to understand the codebase of Mesh2Splat, VScode Copilot was used. Since there were a lot of new methods and computations we had not used before we found this useful to quickly get a grasp of concepts that we then could read up on more.

References

Pavlovic, M. (2024). “Sci-fi helmet model,” provided by Quixel. License: CC Attribution Share Alike 3.0. https://creativecommons.org/licenses/by-sa/3.0/. [Online]. Available: https://quixel.se/usermanual/quixelsuite/doku.php?id=ddo_samples [Pages xi, 40, and 47.]

Scolari, S. (2025). Mesh2Splat (Version 1.0.0) [Computer software]. https://github.com/electronicarts/Mesh2Splat

Scolari, S. (2024). Mesh2Splat : Gaussian Splatting from 3D Geometry and Materials (Dissertation, KTH Royal Institute of Technology). Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-359582