Assignment 4 — 3DGS Full Pipeline

Version: 1.0

Deadline: 2025-06-15T23:59

In this exercise, you will have to implement the entire 3D Gaussian Splatting (3DGS) rendering pipeline, using tile-based rendering. We encourage you to experiment with different parallelization strategies, shared memory, try out warp communication, synchronization and more. You will likely also run profiling, evaluate thread divergence, and derive different strategies for different sub problems.

1 Getting started

You are provided with a personal Git repository that you can use for working on this task. This repository is also used for submission to the automated testing system (See section 7). The framework will be automatically cloned when you create your own repository. You can find the framework in the upstream Git repository at `https://assignments.icg.tugraz.at/gpuprogramming/3dgs-full-framework`. You can also manually pull the template into a personal Git repository. In any case, we might require you to merge bug fixes or changes required for automated testing at a later time from the template repository, thus you should keep our history to ease potential merging.

2 Background: 3D Gaussian Splatting (3DGS)

3D Gaussian Splatting (3DGS), published in 2023 by Kerbl et al. [1], learns a 3D scene from a set of 2D images and corresponding camera parameters (inverse rendering). In constrast to other similar methods, which mostly rely on volume rendering techniques, it can be rendered efficiently in real-time. Our assignments only cover the rendering part, not training.

Disclaimer: The following sections contain mathematical details about 3DGS. We provide the implementation of all mathematical expressions in our reference implementation, as the assignment is not intended to be a math exercise. However, since 3DGS is currently a hot topic in computer graphics, you might benefit from trying to understand the underlying concepts. Additionally, if you want to improve performance on a conceptual level, you will

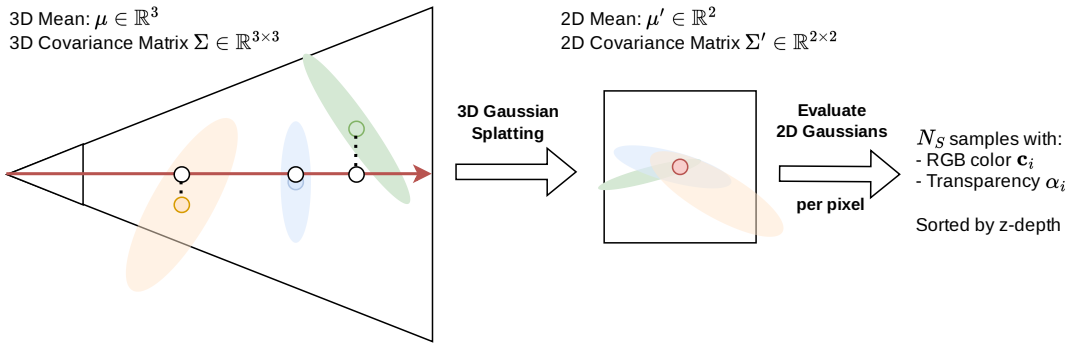


Figure 1: 3D Gaussians are defined by a 3D mean and a 3×3 covariance matrix. They can be transformed to 2D Gaussians on an image plane by applying a local affine approximation of the projective transform [3]. Each pixel calculates the opacity of all (relevant) 2D Gaussians and performs alpha blending of the resulting RGB-color and α (transparency) values. The blending order is defined by the z-depth of each 3D Gaussian's mean in view-space.

also require some background knowledge. Please refer to the original papers [1, 3] if you are curious about further details.

2.1 Scene Representation

In 3DGS, the scene is represented by a large number of 3D Gaussians, which is essentially a colored point cloud with non-spherical points. Each 3D Gaussian is represented by

- 3D covariance matrix $\Sigma \in \mathbb{R}^{3\times3}$, which defines the extent of the Gaussian in 3D. It is composed of a Rotation matrix R (stored as a quaternion), and a diagonal scaling matrix S: $\Sigma = RSS^TR^T$. This ensures that Σ is positive-definite.
- 3D mean $\mu \in \mathbb{R}^3$ (center).
- Spherical harmonics (SH) coefficients per RGB-color channel, that represent the Gaussian's view-dependent color.
- An opacity value $\sigma \in \mathbb{R}$.

Each Gaussian is defined in 3D as $G(\mathbf{x}) = e^{-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)}$. Evaluation in 3D would require costly ray-tracing/ray-marching, so we instead rasterize/splat them onto the image plane.

2.2 Splatting

3D Gaussian point clouds can be efficiently rendered via elliptical weighted average (EWA) splatting [3], where points are "thrown onto" the screen (splatted) in front-to-back order (see Figure 1).

Under the perspective transform, 3D Gaussians do not form a valid 2D Gaussian on the image plane. However, they can still be approximated with a 2D covariance matrix $\Sigma_2 \in \mathbb{R}^{2\times2}$ by taking the first two rows and columns of matrix Σ':

$$\Sigma' = JW\Sigma W^T J, \tag{1}$$

where W is the view transformation. J is the Jacobian matrix, obtained by performing a first-order taylor expansion of the projective transformation at the 3D view-space Gaussian mean. Essentially, we approximate the entire Gaussian under the local orthographic projection, applied to its mean (see [3] for details). The introduced approximation error is usually insignificant.

The 2D mean $\mu_2 \in \mathbb{R}^2$ is simply the projection of the 3D mean μ onto the image plane, using the view-projection matrix.

At this stage, we also evaluated the spherical harmonics (SH) coefficients to obtain view-dependent RGB color $\mathbf{c} \in \mathbb{R}^3$, using the vector from the camera's position to the 3D mean as view-direction. We will use SH of order 3 with 16 coefficients per color channel.

2.3 Rendering

After splatting, we know the position μ_2 and shape Σ_2 of the Gaussian splat on the 2D image plane. Each 2D Gaussian can be evaluated at a pixel $\mathbf{x}' \in \mathbb{R}^2$, by multiplying the per-Gaussian opacity value σ with the 2D Gaussian contribution G_2

$$G_2(\mathbf{x}') = e^{-\frac{1}{2}(\mathbf{x}'-\mu_2)^T\Sigma_2^{-1}(\mathbf{x}'-\mu_2)}, \tag{2}$$

where Σ_2^{-1} is the inverse 2D covariance matrix. We can then perform front-to-back alpha blending with $\alpha = \sigma \cdot G_2(\mathbf{x}')$. Like 3DGS, we will use a treshold $\epsilon_\alpha = \frac{1}{255}$ and discard all Gaussians for this pixel with $\alpha < \epsilon_\alpha$. The blending sort-order is determined by the z-coordinate of each Gaussian's 3D mean in view-space.

Note: This global sort can lead to "popping" artifacts, as the sort-order can suddenly change along rays during camera rotation. Removing these artifacts would require a computationally intensive per-pixel sort, a topic explored in other recent research [2].

The simplest way to render the N sorted Gaussians after splatting is to iterate over all Gaussians for every pixel $\mathbf{x}'$ and calculate the final pixel color C as

$$\mathbf{C} = \sum_{i=1}^{N} \mathbf{c}_i \alpha_i \cdot T_i, \tag{3}$$

$$T_{i+1} = T_i \cdot (1-\alpha_{i+1}) = \prod_{j=1}^{i-1}(1-\alpha_j), \text{ with } T_0 = 1, \tag{4}$$

with RGB color $\mathbf{c}_i \in \mathbb{R}^3$, transparency $\alpha_i \in [0,1]$, and transmittance $T_i \in [0,1]$.

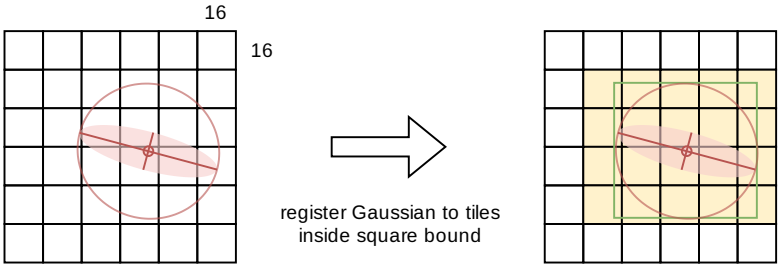


Figure 2: The 2D Gaussian can be bounded with a circle (red) with radius 3× the largest eigenvalue of the 2D covariance matrix, which is guaranteed to contain 99.7% of its contribution. By bounding the circle again with a square (green), we can identify a number of tiles (yellow) which might be affected by this Gaussian.

2.4 Tile-Based Rendering

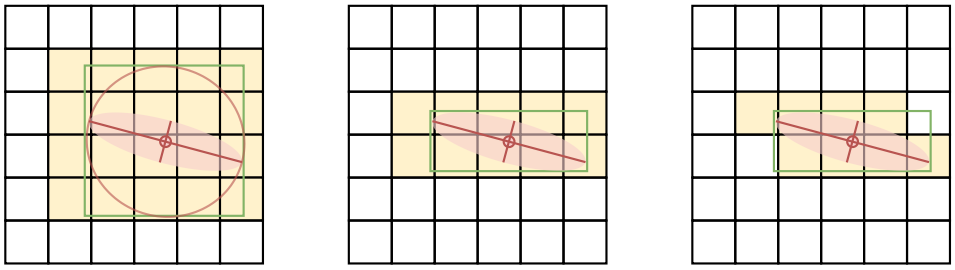
In this section, we discuss the individual **Stages** of the tile-based rendering as performed by the original 3DGS paper [1]. This should act as a starting point for your implementation of an efficient 3DGS renderer, however, feel free to diverge from it if you feel that you can get better performance.

Evaluating each Gaussian for every pixel (as done in our provided reference implementation) is very costly, since 3DGS models often consists of millions of Gaussians. Therefore, 3DGS splits the image into tiles, e.g. of size 16×16 pixels, and each Gaussian is only registered to the tiles where it contributes (see Figure 2). To find the Gaussian's approximate area of influence (extent), we can bound it with a circle by considering the eigenvalues of its 2D covariance matrix. A circle with a radius of 3× the largest eigenvalue contains at least 99.7% of a Gaussian's contribution. Further, by bounding this circle with a square, we can efficiently calculate the approximate number of affected tiles for each Gaussian during the **Preprocess** stage. Summing up those values gives us the total number of ⟨tile, Gaussian⟩ combinations, and the prefix sum gives each Gaussian's offset.

In the **Duplication** stage, each Gaussian registers itself to every tile it affects. For correct front-to-back blending by tile, we need to sort the Gaussians of each tile again by z-depth in view-space. Sorting these entries separately per tile requires all entries of a tile to lie consecutively in memory (partitioned sort), which in turn requires some form of synchronization during duplication. Although this can be done efficiently, you might not get better performance from it.

Instead, 3DGS instead performs a simpler global **Sort** of all ⟨tile, Gaussian⟩ combinations by generating combined keys of ⟨tile index, depth⟩. Conveniently, this groups all sort entries by tile index and sorts the entries of each tile by depth.

Afterwards, we simply have to **Identify** each tile's range of entries (from, to) in this global list, e.g. via a kernel with one thread per global sort entry, comparing its tile index with the previous/next entry.



a) Circle/Square Bounding b) Rectangle Bounding c) Rectangle Bounding + Tile-based culling

Figure 3: Comparison of different bounding strategies. Bounding a 2D Gaussian with a circle (a) gives a very conservative bound, leading to many tiles where it does not actually contribute. It can also be bounded with a rectangle (b) by considering the variance of the 2D Gaussian in x/y-direction. Finally, there might still be tiles where the contribution is below the threshold ϵ_α, which can be determined with tile-based culling (c).

Finally, in the **Render** stage, each pixel now only has to consider the Gaussians that fall onto its tile. You already implemented this last stage in Assignment 3.

2.5 Optimizations

If you follow the previous section, you will arrive at the exact implementation of 3DGS, as proposed by Kerbl et al. [1]. The following optimizations go beyond that, and are considered more "advanced". We advice you to only consider them once your implemention of the tile-based renderer works.

Tighter Bounding Bounding a Gaussian with a square is pretty conservative, registering it to many tiles without any pixel where $\alpha > \epsilon_\alpha$. This leads to many unnecessary computations, as these entries have to be created during duplication, sorted, and finally loaded during rendering. We show different bounding strategies in Figure 3. Firstly, we can fit a tighter rectangular bound by using the $3\times$ standard deviation of the 2D covariance matrix in x/y-direction, instead of the largest eigenvalue:

$$\text{std}(\Sigma_2)_x = \sqrt{\Sigma_2[0][0]}, \qquad \text{std}(\Sigma_2)_y = \sqrt{\Sigma_2[1][1]}. \tag{5}$$

This still produces unnecessary tiles, which can be identified with a tile-based culling approach. Additionally, these discussed strategies do not consider the Gaussian's opacity value σ, essentially treating it as having $\sigma = 1$. For details on these topics, confer to sections B.1 and B.2 in the Appendix of [2].

Load balancing Some provided scenes might contain a few very large Gaussians, contributing to every tile on the screen. This might lead to a large overhead during duplication, when this Gaussian's thread needs to write all those combinations to memory. This problem is amplified by increasing the number of tiles, e.g. when using higher resolutions or reducing the tile size. One possible solution is to perform load balancing, i.e. performing computations for large Gaussians with multiple threads.

3 Assignment 4 (20 points)

Your task is to implement the full 3DGS rendering pipeline, using a tile-based rendering approach, as discussed in Section 2.4. You are given the actual learned 3D Gaussian point cloud as a dataset, which is loaded from a `.ply` file and already uploaded to the GPU in `DatasetGPU`. We will render this dataset from different camera views, defined in the provided camera transforms `.json` file.

We will use the `ship` and `lego` model from Assignment 3, as well as actual real-world models, which you can download from the Teachcenter.

We already provide a very slow GPU reference implementation, which you can take as a starting point. The `prepare` kernel performs the splatting steps, and calculates the square bounds of each Gaussian in pixel coordinates (`rect_min, rect_max`).

You will have to split the screen into tiles, and use the pixel bounds to calculate tile bounds, i.e. which tiles could be affected by this Gaussian. Afterwards, perform per-tile sorting by depth, and use your tile-based rendering from Assignment 3 to generate the final image. We advice you to follow the stages as proposed in Section 2.4, but feel free to diverge after your initial implementation works.

4 Implementation

Your task is to implement the `run` method in the `Renderer` class, and output the rendered image for a given camera view. For this assignment, you are given complete freedom over this `Renderer` class (.h and .cu file). Feel free to add buffers, private methods and whatever you deem important to improve performance. Only the inherited public method signatures must stay unchanged.

We added a simple GPU implementation in `Renderer_Reference` that performs the splatting step in `prepare`, sorts all splats globally by z-depth using CUB, and then launches a `render` kernel with one thread per pixel that iterates over all Gaussian splats.

The submission system is only timing the `run` method. You can allocate additional device buffers in the `setup` method. You are also provided with the full dataset during setup, allowing you to create a new representation if you deem the provided dataset structure

inefficient. However, you are **not** allowed to change the content inside the original dataset structure, as it is also used by the reference implementation.

You will probably also have to allocate memory in the `run` method, as you do not know the number of global sort entries beforehand. We will perform a `warmup` iteration where you can allocate the memory, which will not be timed. We would advise you to allocate more memory than you need, and only resize those buffers if you require more memory than you already allocated (an example is provided). Also, make sure to free all memory before resizing an existing buffer. We also provide an example of how the construct the combined 64-bit key (tile index + depth) in the `constructKey` method.

All evaluations have to happen on the GPU. You are not allowed to compute any significant parts of the pipeline on CPU. To properly calculate runtime, we call the `run` method multiple times. Make sure to perform all the computations in the run method at every iteration and that you do not share any data between runs (or from warmup).

The framework also determines whether your implementation is correct, comparing the output image to the GPU reference generated image.

5 Points

The points of the assignment are split as follows

- 10 pt Implement a correct tile-based implementation
- 10 pt match the average performance of *3dgs-base*
- +5 pt bonus points for optimizations that push beyond *3dgs-base*

3dgs-base is an exact implementation of the pipeline, as discussed in Section 2.4 (without further optimizations). Disclaimer: You can implement your renderer however you want (does not need to be tile-based), as long as you match the performance of *3dgs-base*.

Consider the following hints if you want to get better performance:

- The most compute-intensive part will probably be the rendering stage. Consider the hints from Assignment 3:
  - Load data in batches from global memory for the whole tile, instead of separately for each pixel
  - If you use shared memory, try to avoid bank conflicts
  - Perform early stopping if full transmittance is reached
  - Avoid unnecessary synchronization operations
  - Discard Gaussians early if they do not contribute to any pixel in a tile

- Reduce the number of tile/Gaussian combinations with efficient bounding or culling
- Perform load balancing to increase performance for large Gaussians
- Use an efficient global sort algorithm. If you use CUB Radix Sort, then take a look at the `end_bit` parameter.
- Bring the dataset into a different form that allows for faster querying
- Optional: Build your own super efficient renderer that is not tile-based. Feel free to come up with your own ideas, as long as it is fast and produces the correct result!

Make sure to document various optimizations as noted in 8 to receive the bonus points.

6 Framework

In the root directory you can find a CMake[1] (version 3.16 or newer) script to generate a build environment. A modern C++ toolchain is required to compile the framework.

Setup:

1. Clone repository to directory of your choice
2. Create build folder «`build`», change to build folder and call «`cmake .. -DCC=*`» (replace * with the compute capability of your GPU, alternatively use GUI to setup)
3. Build (Visual Studio or «`make`»)
4. Run: «`./3dgs-full <points-file> <camera-file> <output-dir>`»

Supported toolchains:

- gcc 8.x, 9.x, 10.x depending on your Linux distribution[2]
- Visual Studio 2017+ on Windows

Launch parameters:

- `points-file`: Path to the gaussian point cloud `.ply` file
- `camera-file`: Path to the camera transforms `.json` file
- `output-dir`: Path to the output directory

[1] `http://www.cmake.org/`
[2] `http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html`

There are several other optional launch parameters. They can be viewed with `--help` or in the `main.cpp` source file.

You also need to install the CUDA toolkit[3]. To profile and debug GPU code, we recommend NVIDIA NSight Compute[4]. The exercise requires an NVIDIA GPU. If you don't have a graphics card fulfilling these requirements, contact the teaching assistant[5] and we will find a solution for you. At first, you can try `Google Colab`. See the tutorial in teach center.

If you are experiencing build problems, make sure the compiler you are using matches one of the versions recommended above, you have installed the NVIDIA CUDA toolkit version 11.x and CMake is able to find the corresponding CUDA libraries and headers.

You can change the build to different compute capabilities by toggling the CMake options `CCxx` using the CMake GUI, the CMakeCache file, or command line arguments to CMake.

Bugs: If you encounter any bugs in the framework please do share them with us, such that we can adapt the framework.

7 Submission format

To hand in your assignment, you first have to create a repository. To create the repository, login to `https://courseware.icg.tugraz.at/` and go to «`Submissions`» and create a repository for the course using the «`create repo`» button. You can then find the repository at `https://assignments.icg.tugraz.at/` (and you should additionally be notified via email). Please keep the directory structure in your repository the same as in the framework to enable automated testing of your submission.

You can develop in the «`main`» branch, pushes there will not trigger the `CI` to run, hence you can also freely commit work in progress. To test your submission on our system and/or submit, push to the branch «`submission`». You can also push here as often as you want, if you want to check your performance. The last push (before the deadline) to this branch will count as your submission, so make sure that your final version is pushed to this branch and builds/runs without issue (check the output of the `CI`). The `CI` will report to you if everything worked and should also show you the current performance charts.

8 Performance Optimization (Challenge)

In GPU programming we are always interested in making kernels as fast as possible. In addition to the points you can earn for handing in a correct exercise, one of you will be awarded the honors of being the most efficient GPU programmer. We let all submissions

[3] `https://developer.nvidia.com/cuda-toolkit`
[4] `https://developer.nvidia.com/nsight-compute`
[5] `mailto:michael.steiner@tugraz.at?subject=[Ass04]%20Bug%20Report`

compete against each other and seek the fastest code. To receive the points for the optimization task and to take part in the challenge, include a file «`challenge.xx`» in your submission, which documents the steps you have taken to speed up your initial implementation. The file format does not matter, but «`.md`» (Markdown) or «`.pdf`» would be preferred.

Note: To receive the points for the optimization sub-task, this description is required!

We will test your submission on a GPU with compute capability 7.5 (GeForce RTX 2080Ti). You will be notified if this changes.

References

[1] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". In: *ACM Transaction on Graphics* 42.4 (2023).

[2] Lukas Radl, Michael Steiner, Mathias Parger, Alexander Weinrauch, Bernhard Kerbl, and Markus Steinberger. "StopThePop: Sorted Gaussian Splatting for View-Consistent Real-time Rendering". In: *arXiv preprint arXiv:2402.00525* (2024).

[3] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. "EWA Splatting". In: *IEEE Transactions on Visualization and Computer Graphics* 8.3 (2002), pp. 223–238.