Assignment 3 — 3DGS Rendering

Version: 1.0

Deadline: 2025-05-25T23:59

In this exercise, you will have to implement the render kernel of 3D Gaussian Splatting (3DGS). 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. This sheet will contain details of 3DGS, which you will need again in the last assignment, where you implement the complete pipeline. Please refer to the original papers [1, 2], if you are curious about further details.

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 (See section 7). You can find the framework in the upstream Git repository at `https://assignments.icg.tugraz.at/gpuprogramming/3dgs-rendering-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) was published in 2023 by Kerbl et al. [1]. It provides a novel way of doing inverse rendering, meaning to learn a 3D scene from a set of 2D images and corresponding camera parameters. We will not train our own model in the assignment, but rather use an already trained model.

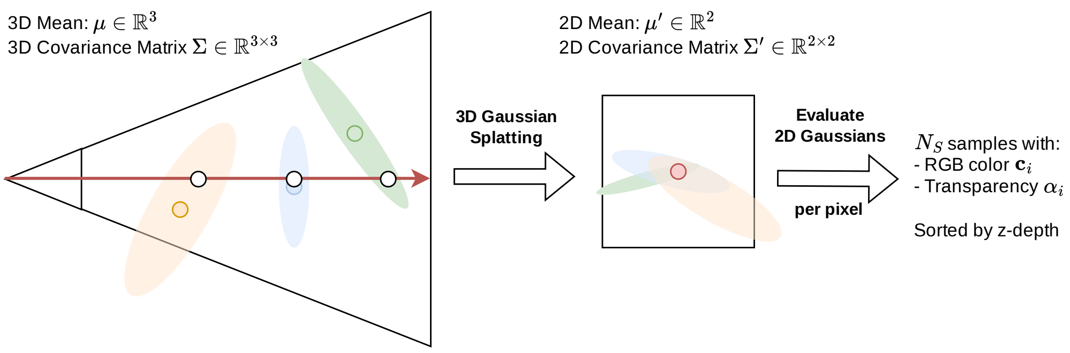
3D Mean: $\mu \in \mathbb{R}^3$
3D Covariance Matrix $\Sigma \in \mathbb{R}^{3\times 3}$
3D Gaussian Splatting
2D Mean: $\mu' \in \mathbb{R}^2$
2D Covariance Matrix $\Sigma' \in \mathbb{R}^{2\times 2}$
Evaluate 2D Gaussians
per pixel
N_S samples with:
- RGB color $\mathbf{c}_i$
- Transparency α_i
Sorted by z-depth

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 [2]. 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.

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 $\Sigma \in \mathbb{R}^{3\times 3}$, which defines the extent of the Gaussian in 3D. It is composed of a Rotation matrix R, which is represented by a quaternion, and a diagnonal scaling matrix S: $\Sigma = RSS^TR^T$
- 3D mean $\mu \in \mathbb{R}^3$ (center)
- Spherical harmonics (SH) coefficients, which allow the Gaussians to have view-dependent appearance (color)
- An opacity value $\sigma \in \mathbb{R}$

Each Gaussian is defined in 3D as $G(\mathbf{x}) = e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)}$

2.2 Splatting

These 3D Gaussian point clouds can be efficiently rendered via elliptical weighted average (EWA) splatting [2], where points are "thrown onto" the screen (splatted), in order of their distance to the camera (see Figure 1). Under the perspective transform, these 3D Gaussians would not form a valid 2D Gaussian on the image plane. However, they can still be approximated as a 2D Gaussian by applying a local affine approximation.

The 2D Gaussian $\Sigma_2 \in \mathbb{R}^{2\times 2}$ is therefore given 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, and J is the Jacobian of the affine approximation of the projective transformation (we will discuss this in more detail in the last assignment). 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.

2.3 Rendering

After splatting, we know the position $\boldsymbol{\mu_2}$ and shape Σ_2 of the Gaussian splat on the 2D image plane. For rendering, will need the inverse 2D covariance matrix Σ_2^{-1}. 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

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

This gives us our $\alpha = \sigma \cdot G_2(\mathbf{x}')$ value at this pixel, which can then be used for blending. Like 3DGS, we will use a treshold $\epsilon_\alpha = \frac{1}{255}$ and discard all Gaussians for this pixel with $\alpha < \epsilon_\alpha$.

We already discussed, that blending has to be performed front-to-back. This is done in practice by sorting the Gaussians by the distance of their 3D mean in clip space (after applying the projection matrix) at every frame.

During splatting, we also already evaluated the SH coefficients for our current viewing direction, to obtain view-dependent color $\mathbf{c} \in \mathbb{R}^3$. The simplest way to render our N sorted Gaussians after splatting would be now 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 $T_i \in [0,1]$.

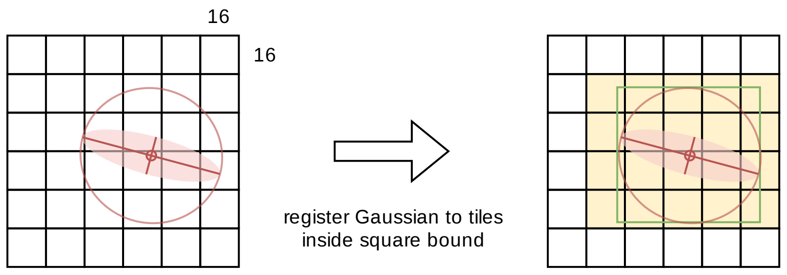

Figure 2: The 2D Gaussian can be bounded with a circle (red), by considering the eigenvalues of the 2D covariance matrix. By bounding the circle again with a rectangle (green), we can identify a number of tiles (yellow), which could be affected by this Gaussian.

2.4 Tiling

Since a 3DGS model can sometimes have millions of Gaussian, we want to limit the number of Gaussians we have to consider for each pixel. In practice, this is done by splitting the image into tiles, in our case of size 16×16 pixels. We can then bound each 2D Gaussian with a square by considering its eigenvalues, and identify a number of tiles which could be affected by this Gaussian (see Figure 2). Each Gaussian is then registered to every tile it affects. We refer to this stage as *Duplication*.

After *Duplication*, each tile has to sort its Gaussians again by distance. This could be done separately by each tile (partitioned sort). Sadly, this requires us to insert the values such that all entries of a tile lie consecutively in memory. Another option is to sort all ⟨tile, Gaussian⟩ combinations globally by a combined key of (tile index, depth). This has the nice effect that it groups all entries by their tile index, and sorts them by their depth at the same time. Afterwards, we simply have to find the range (from, to) of each tile in this global list.

Each pixel now only has to consider the Gaussians that fall onto its tile, instead of all of them. Note that bounding a Gaussian with a square is pretty conservative, and many tiles will not be affected afterall. Additionally, also not all pixels of a tile might be affected by a Gaussian (meaning $\alpha < \epsilon_\alpha$).

3 Assignment 2 (20 points)

For this assignment, your task is to take the sorted list of ⟨tile, Gaussian⟩ combinations, together with the computed Gaussian splats, and perform the final rendering. The data is already loaded into buffers in the `Dataset` and uploaded to the GPU in `DatasetGPU`.

`tile_ranges` contains the offsets (from, to) for each tile and points into the global list of `tile_splat_entries`. This list contains only indices, which are needed to retrieve the actual Gaussian's values for evaluation and blending.

You are given the Gaussian splats, with separate arrays for each of the different components:

- Inverse 2D Covariance matrix + opacity value: These are packed together into a single `float4` value. Note that the covariance matrix is symmetric, and can therefore be represented with only 3 values, instead of 4.
- 2D Mean: In pixel coordinates and stored as `float2`.
- RGB Color: Already evaluated from SH coefficients and stored as `float3` values.

Your task is to compute the contribution for each pixel by iterating over all Gaussians in the list for the corresponding tile front-to-back, evaluating their α values (see Eqn. 2) and perform the blending (see Eqn. 3).

4 Implementation

Your task is to implement the `run` method in `Render_GPU` and output the correctly blended image.

We added a very simple (and very slow) CPU implementation in `Render_CPU` that iterates over all pixels sequentially and retrieves the Gaussians for the corresponding tile. The CPU implementation also performs early stopping after transmittance reaches a certain threshold $\epsilon_T < 10^{-4}$.

The submission system is only timing the `run` method. If you need any additional device buffers (which should not be needed to succeed in this assignment), you can allocate them in the `setup` method. All evaluations have to happen on the GPU. You are not allowed to compute any parts of the equations 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.

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

5 Points

The points of the assignment are split as follows

- 10 pt Implement a correct parallel implementation
- 5 pt match the average performance of *basic*
- 5 pt match the average performance of *advanced*
- +5 pt bonus points for optimizations that push beyond *advanced*

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

- 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 (see CPU implementation)
- Avoid unnecessary synchronization operations

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-render <input_dir>`»

Supported toolchains:

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

Launch parameters:

- `<input_dir>` : Path to the input directory, which contains all necessary input files (`frameinfo.json` and the corresponding five `.dat` files)

You also need to install the CUDA toolkit[3] on your machine. In order 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

[1]`http://www.cmake.org/`
[2]`http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html`
[3]`https://developer.nvidia.com/cuda-toolkit`
[4]`https://developer.nvidia.com/nsight-compute`
[5]`mailto:michael.steiner@tugraz.at?subject=[Ass03]%20Bug%20Report`

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 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.

9 FAQ

Are we allowed to use `CUB` for this assignment? Yes, using CUB is allowed.

Is it allowed to add new variables to the header file? No, we overwrite all files except the on the ‘render.cu‘ file on the test system.

References

[1] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. “3D Gaussian Splatting for Real-Time Radiance Field Rendering”. In: 42.4 (2023).

[2] 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.