Assignment 2 — Prefix Sum Blending

Version: 1.0

Deadline: 2025-05-04T23:59

The objective of this exercise is to learn and use basic algorithms for parallel prefix sum and reduction and applying them to alpha blending. 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. The exercise should also introduce you to the basics of 3D Gaussian Splatting (3DGS), which we will explore in more detail in the last two assignments.

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-blending-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. In 3DGS, the scene is represented by a large number of 3D Gaussians, which is essentially a colored point cloud where the point's extent is defined by a 3D covariance matrix instead of just being spherical.

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

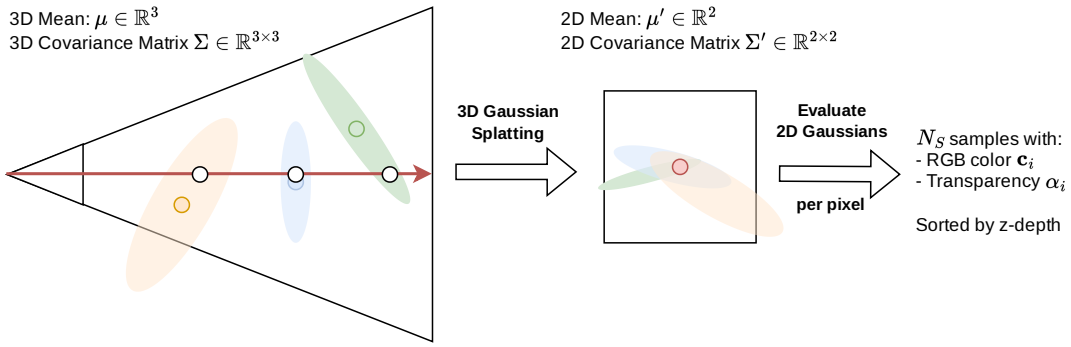

Figure 1: 3D Gaussians are defined by a 3D mean and a 3×3 covariance matrix. They can be transformed to 2D Gaussians on an image plane by applying a local affine approximation of the projective transform [2]. Each pixel calculates the opacity of all (relevant) 2D Gaussians and performs alpha blending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

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. This allows for very efficient rendering either using the standard rasterization pipeline or with custom compute kernels (e.g., in CUDA).

3 Assignment 2 (20 points)

For this assignment, you are provided with the RGB color and α values for all samples of each pixel as input. These samples are taken from an actual 3DGS render. The data is already loaded into buffers in the `Dataset` and uploaded to the GPU in `DatasetGPU`.

The goal of assignment 2 is to perform alpha blending for a pre-defined number of N_S samples per pixel:

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

$$T_{i+1} = T_i \cdot (1 - \alpha_{i+1}) = \prod_{j=1}^{i-1} (1 - \alpha_j), \text{ with } T_0 = 1 \tag{2}$$

with RGB color $c_i \in \mathbb{R}^3$, transparency $\alpha_i \in [0,1]$, and transmittance $T_i \in [0,1]$ at sample i. Transmittance essentially tells you how visible sample i is. The final color $\mathbf{C}$ can be computed by summing up the contributions of all N_S samples. In our case, N_S is always a power of two and is constant across all pixels.

3.1 Task 1: Parallel Prefix Sum

Your first task is to implement equation 2 and compute the transmittance values T_i for all N_S samples per pixel with a parallel prefix sum, also called exclusive/inclusive scan. What will be your scan operation? To get any points for this task, you will have to use multiple threads per pixel that work cooperatively. You can see the lecture slides for inspiration on how to do this efficiently on GPU.

Note: You are also allowed to already compute $T_i \cdot \alpha_i$ at this stage.

3.2 Task 2: Parallel Reduction

After computing all transmittance values T_i (or $T_i \cdot \alpha_i$), perform a parallel reduction to compute a color for each pixel (see equation 1). Your task is to compute a parallel reduction across all the weighted color values. Again, you will only get points if you use multiple threads per pixel.

Note: Even though some of the rays will not need all N_S samples, you are still required to iterate over all of them. In future assignments, we will allow an adaptive number of samples (Gaussians) per pixel and perform early stopping if transmittance T_i becomes too small.

4 Implementation

The main steps for the assignment are as follows:

- Implement the `prefixSumWeights` method of the `PrefixSumBlending_GPU` class, which takes α as input and computes transmittance T (or alternatively already $T \cdot \alpha$)
- Implement the `reduceColor` method of the `PrefixSumBlending_GPU` class, which takes α, color $\mathbf{c}$, and transmittance T per sample as input and outputs pixel color $\mathbf{C}$

We added a very simple (and very slow) CPU implementation in `PrefixSumBlending_CPU` that iterates over all pixels and samples sequentially and performs the blending operation with a prefix sum and reduction.

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. The computation of the prefix sum and reduction must 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 (utilize mutliple threads per ray)
- 5 pt match the average performance of *basic* - which is a simple (slow) implementation
- 5 pt match the average performance of *advanced* - which is shared memory based implementation
- +10 pt bonus points for optimizations that push beyond *advanced*

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

- If you use shared memory, avoid bank conflicts
- Use warp-intrinsics (shuffles) to communicate between cooperating threads
- Efficient load and store operations
- Could everything be done with a single prefix sum?

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: «`./prefixsum <input_dir>`»

Supported toolchains:

- gcc 8.x, 9.x, 10.x depending on your Linux distribution[2]

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

- 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 three `.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 lecturer[5] and we will find a solution for you. At first, you can try `Google Collab`. 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 «`master`» 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.

[3] `https://developer.nvidia.com/cuda-toolkit`
[4] `https://developer.nvidia.com/nsight-compute`
[5] `mailto:michael.steiner@tugraz.at?subject=[Ass02]%20Bug%20Report`

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, should this change.

9 FAQ

Which values for N_S should my code support? Your solution should work for $N_S \in [64, 128, 256, 512]$. We encourage you to use `template` parameters, instead of copying the same kernel multiple times.

Are we allowed to use `CUB` for this assignment? CUB is explicitly allowed (e.g., using its block-wide primitives). We still encourage you to experiment with a custom prefix sum and reduction implementation, as this teaches you important concepts in GPU programming.

What if I need some additional GPU memory for my optimized solution? You are allowed to allocate additional buffers, if you need them. Note: You will have to declare them as global variables in the ‘prefixsum.cu‘ source file (the test system replaces all the files except ‘prefixsum.cu‘ in your submission).

The assignment contains the hint "Could everything be done with a single prefix sum?". Is it allowed to not perform any calculations in ‘prefixSumWeights‘ and instead doing everything in ‘reduceColor‘? Yes, you are allowed to combine the kernels for the performance challenge, but you will still have to do all the computations in parallel and you may not skip empty values. Hint: If you choose to use a single prefix sum, think about the properties of your reduction operation. Which reduction operation property would you violate in this case (Wikipedia: Reduction Operator), and how do you accommodate for that?

Do I need to add optimization notes if I dont want the bonus points? No, if you do not want to get bonus points or take part in the performance optimization challenge, you dont have to add it.

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.