Saliency-Aware Pruning for 3D Gaussian Splatting

Problem Define & Motivation

3D Gaussian Splatting (3DGS) achieves state-of-the-art (SOTA) quality and speed in 3D reconstruction through Gaussian representation of point clouds and efficient tiling rendering algorithms. It has been applied to multiple areas, such as VR/AR and gaming, and 3D mapping (DJI Terra). We regard this project as a great chance to learn and examine the 3DGS methodology.
However, the massive number of Gaussians consumes high VRAM. After running a training demo, we recognized there were too many unwanted artifacts in the trained Gaussians. Therefore, we decided to explore feasible ways to reduce the total number of Gaussians by pruning those in less important areas (such as the sky or walls), while retaining the main object's Gaussians. This motivated us to propose a geometric heuristic clustering-based pruning method. We evaluate its effectiveness through three cluster-level metrics: Spatial Variance, Depth Variance, and Multi-view Visibility Ratio.

1. Reproducibility

1.1 Environment Setup

Prerequisite: RTX 40-series GPU (RTX 50 series are complex to setup because of its new Blackwell Architecture, though we are using the 50 series to run this project)

Brief Setup Steps: 1. Git clone from github 2. Create a new conda environment and activate it; 3. Install PyTorch that corresponds to your CUDA version; 4. Install gsplat at local folder `./gsplat_source`. 5. Install more dependencies (`./gsplat_source/examples/requirements.txt` + below)
***Note**: For detailed setup steps, please refer to **README → Setting up the environment**

Dependencies table

Component	Role
PyTorch + CUDA	Training, rendering, tensors
gsplat	Rasterization, `splats` checkpoint format
pycolmap	COLMAP read the camera parameters at `sparse/0` cameras
segment-anything	SAM masks on rendered images
scipy	cKDTree for clustering
numpy, tqdm, imageio	Arrays, progress, I/O

1.2 Dataset

All experiments use scenes from the **Mip-NeRF 360** dataset (`360_v2`). Each scene directory contains: 1. RGB training views at `images/`; 2. Camera parameters at folder `sparse/0/`; 3. Initial sparse point cloud (`poses_bounds.npy`). Images and camera intrinsics are downscaled by a factor of 4 during both training and pruning.

1.3 Training

We trained 3DGS models for 30,000 steps using `gsplat`'s `simple_trainer.py` with the `DefaultStrategy`. Checkpoints are saved as `.pt` files containing all Gaussian parameters (`means`, `quats`, `scales`, `opacities`, `sh0`, `shN`).

```
python ./gsplat_source/examples/simple_trainer.py default
       --data_dir ./data/360_v2/<scene_name>  --result_dir ./results/<scene_name>_demo  \
       --data_factor 4
```

1.4 Pruning Pipeline

Step A - Preprocess (`pruning.preprocess`): We generated the viewing images from trained model to ensure consistency. We rendered them with `gsplat.rasterization` using the trained checkpoint, then runs SAM (ViT-H) on each rendered image. Region maps of each view are saved as `view_XXXX.npy` files.

```
python -m pruning.preprocess --ckpt results/<scene_name>_demo/ckpts/ckpt_29999_rank0.pt \
       --data_dir data/360_v2/<scene_name> --sam_ckpt sam_vit_h_4b8939.pth \
       --mask_dir sam_masks --data_factor 4
```

Step B — Project & Cluster (`pruning.cluster`): Projects Gaussians onto SAM masks through the pinhole camera projection function, excluding the occluded Gaussians by comparing their depth for each pixel. Then groups Gaussians into clusters using constrained union-find (geometric distance + semantic consistency).

```
python -m pruning.cluster --ckpt results/<scene_name>_demo/ckpts/ckpt_29999_rank0.pt \
       --data_dir data/360_v2/<scene_name> --mask_dir sam_masks
       --cluster_output cluster_result.pt --data_factor 4
```

Step C — Score (`pruning.score`): Computes the G* heuristic value for each cluster. We normalized three components: spatial variance, depth variance, and multi-view visibility ratio, then combined with weights `alpha`, `beta`, `gamma`.

```
python -m pruning.score --ckpt results/<scene_name>_demo/ckpts/ckpt_29999_rank0.pt \
       --cluster_input cluster_result.pt --scores_output cluster_scores.pt
```

Step D — Prune (`pruning.prune`): Removes all clusters with G* score below a threshold by applying a boolean keep-mask to every tensor in the pre-trained checkpoint.

```
python -m pruning.prune --ckpt results/<scene_name>_demo/ckpts/ckpt_29999_rank0.pt \
       --scores cluster_scores.pt --output pruned_model.pt --score_threshold 1.0
```

1.5 Hyperparameters Usage

Parameter	Default	Description
`data_factor`	4	Image downscale factor
`normalize`	True	Boolean value indicating to apply camera normalization
`delta_dist`	0.01	KD-tree radius for geometric neighbor pairs
`semantic_tol`	0.1	Max conflict ratio across co-visible views to permit merging
`max_views`	50	Max views count for projection & clustering
`alpha`	0.1	Spatial variance weight
`beta`	0.1	Depth variance weight
`gamma`	1.0	Multi-view visibility weight
`mvr_mean_frac_weight`	0.3	Coefficient between `view_coverage` and `mean_visible_frac` in MVR
`score_threshold`	Dynamic	G* score threshold
`prune_ratio`	0.9	Percentile-based threshold if `score_threshold` is not given

1.6 Visualization

Viewed pruned models interactively:

```
python ./gsplat_source/examples/simple_viewer.py --ckpt pruned_model.pt --port 8080
```

1.7 Results

Storage Reduction Result

Scene	Original	Pruned	Reduction
bicycle	1.50 GB	402 MB	73.8%
bonsai	275 MB	46.4 MB	83.1%
counter	243 MB	119 MB	51.0%
garden	1.10 GB	94.1 MB	91.6%
kitchen	392 MB	234 MB	40.3%
room	338 MB	119 MB	64.8%
stump	1.05 GB	278 MB	73.5%

Visualization Result All pruned models visually retain the foreground subject at the scene center while removing background Gaussians.
***Note**: For detailed Image Comparing, please refer to the **Appendix**.

Result Discussion We get a roughly great pruning outcome, but observed that the outcome is very dependent on the SAM output, specifically the display of each component in the original images. For example, for the room scenes, there is no obvious main object in the original images, so the final result can only retain most of the objects and exclude the background Gaussian clusters. We test the manual sample method for Garden & Bicycle, filtering for specific images (cam_poses) that target the wanted object. This produces a great memory reduction (91.6%, 73.8%) and visual outcome.
From this observation, an important step that ensures the reduction performance is how the objects are captured. More filter solutions could be explored further, or restricting the way that users shoot the images.

2. System Architecture

2.1 Pipeline Overview

The pruning pipeline is a **five-stage post-training pipeline** that operates on a fully trained 3DGS checkpoint. It does not modify the training loop and requires no gradient computation.

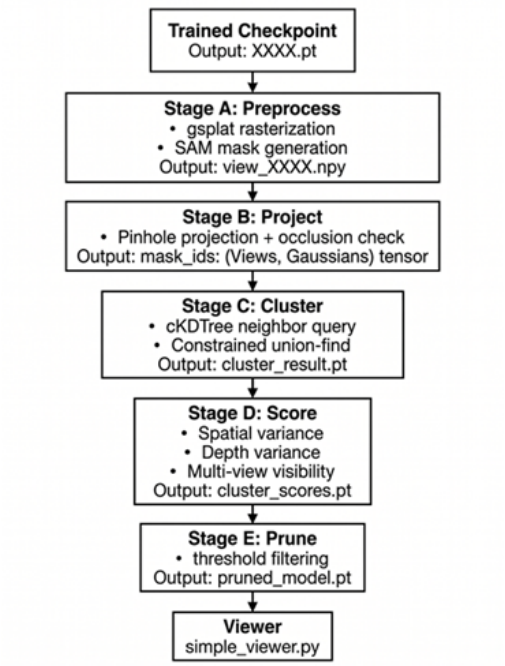


Figure 1: Pipeline Overview

2.2 GPU/CPU Split Work

GPU handles all the parallel workloads: gsplat rasterization, SAM inference, Gaussian projection (Batched matrix multiplication over all views), G* score computation.
CPU handles operations that are sequential or lack GPU-native implementations: scipy cKDTree construction and querying,union-find cluster merge, boolean mask pruning.

3. Implementation Details

3.1 Camera Loading (`./pruning/camera.py`)

We use `SceneManager` to load the camera parameters from `./data/360_v2/<scene_name>/sparse/0/`. In the function `load_cameras`, we extract world-to-camera matrices `[R | t]` for each image, constructs intrinsic matrices `K` (scaled by `data_factor`), and computes image dimensions. We apply `normalize=True` throughout the process to ensure the camera views consistent with the trained Gaussian results.

3.2 Rendering & SAM Mask Generation (`./pruning/preprocess.py`)

`render_all_views` renders each camera view by inverting the camera-to-world matrix to obtain the world-to-camera views matrix `viewmat`, then calling `gsplat.rasterization` with the Gaussian parameters.

`load_splats` decodes raw checkpoint parameters. Remind of the 3D covariance matrix $\Sigma = R \cdot S \cdot S^T \cdot R^T$, `quats` will transform to R, `scales` will transform to the S later. Together they define the 3D covariance matrix.`opacities` (range: [0,1]) defines the opacity of the Gaussians. Spherical harmonics coefficients (`sh0` + `shN`) defines the color of each Gaussian. The SH degree (inferred as `sqrt(num_coeffs) - 1`) defines how the color change along with the viewing angles.

`generate_sam_masks` runs SAM ViT-H on each rendered image that generated from `render_all_views`. `masks` is a list, and each mask element is a dictionary that corresponds to a unique id, including `"segmentation"` (boolean [H,W], true if it belongs to this mask), `"area"` (int) etc. Then sort the resulting masks by area, so that in the case where a pixel have several valid mask, smaller area will overrite larger area, making the SAM segmentation get finer. Then write them to a region map where each pixel fills with

the index of the mask that covers it. We save the region maps to `./sam_masks/view_XXXX.npy` for each view.

We also save rendered images for each view to `./sam_masks/renders/` for visual inspection.

3.3 Gaussian Projection (`pruning/project.py`)

`project_gaussians` projects Gaussian to each SAM mask, return mask_ids (Views, Guassians) tensor and the value represent the corresponding mask id. We loaded the camera parameters (camera intrinsics and extrinsics) and then do the batched operation. We apply the matrix operation to convert the Gaussians from world coordination to camera coordination `p_cam`, and then get its depth `z` and compute its `u` and `v` that represent their coordinates on the image. Finally, we do an occlusion checking. We initialized a `depth_buffer` to compute and store the minimum possible depth of the gaussian that map to the pixel, then use it as the threshold to filter out the Gaussians that is far than `0.05 * min_z + 0.05` depth.

3.4 Union-Find Clustering (`pruning/cluster.py`)

`build_clusters` uses Union-Find method to group the Gaussians into clusters, and return the dictionary `cluster` that maps each cluster into its Gaussians list. Firstly, we build a `KDTree` for N Gaussian means. For each batch of points, we use the `delta_dist` as the threshold for the geometric constraint, if they pass this constraint, we form two points pair `(i, j)`. Then for these pairs, we do the semantic consistency check, check their `mask_ids[:, i]` and `mask_ids[:, j]` across all views. In views where both are visible (`!= -1`), disagreements are counted. If the conflict ratio exceeds `semantic_tol`, the pair is not merged. The larger `semantic_tol` is, more pairs are likely to merge together, leading to larger clusters.

3.5 G* Scoring (`pruning/score.py`)

`compute_scores` compute each cluster's scored with three components:

1. **Spatial variance**: Do the normalization by substracting the cluster Gaussians' center coordination to each Gaussians in that cluster. Then compute its variance regularly.
2. **Depth variance**: For each cluster, compute the variance of z-depths across its visible gaussians per view. Then average the variance among all the views.
3. **Multi-view visibility ratio (MVR)**: `view_coverage` is the fraction of views with any cluster member, while `mean_visible_frac` is the average fraction of visible Gaussians per view. We use the `view_coverage` as the weight to calculate the final result. This combination prevent the extreme case, for example, a cluster could be seen in all views but only little fraction could be seen each time.

Finally we normalize these three values to [0, 1], and compute the final score: `G* = -alpha * norm(spatial_var) - beta * norm(depth_var) + gamma * norm(mvr)`.

3.6 Pruning (`pruning/prune.py`)

`prune_checkpoint` constructs a boolean keeping mask of size N. For each cluster whose score falls below the threshold, all member Gaussian indices are set to `False`. Then use this keeping mask to prune the Gaussians and save the remaining Gaussians to a new checkpoint file `./pruned_model.pt`.

4. Technical Challenges

4.1 SAM Output Dependency

In the cluster operation, we sample at most 50 camera views (`max_views`) that are generated from the SAM operation. This sampling worked well for most scenes (bonsai, kitchen) but produced chaotic results for the **garden** and **Bicycle** scenes.
We investigated on this issue and found the causes: 1. Some sampled views are poor, they don't even capture the main object that we want; 2. SAM produced poor segmentations for some complex object such as bicycles. When we used these low-quality masks, we failed to form the reasonable clusters.

Figure 2: Bicycle SAM (Bad)

Figure 3: Bicycle SAM (Good)

Solution: We extracted the rendered images and ran SAM manually, then visually picked the `cam_poses` with the best SAM result. Only views with clean segmentation boundaries were manually selected as camera indices. For garden, we picked 8 manually curated view indices instead of 50 uniform samples, and we got the accurate pruning result.

Discussion: The pipeline's pruning quality is heavily dependent on SAM segmentation quality. In a production setting, an automated SAM quality filter could be introduced to accomplish this requirement. In real-time capture scenarios, guiding camera poses to ensure diverse, high-quality viewpoints is an effective approach.

4.2 Occlusion Checking

Our initial implementation of `project_gaussians` function didn't consider the occlusion case: an occluded Gaussian behind a wall would get the foreground mask ID assigned as well. This caused Gaussians from geometrically separate objects to be falsely assigned the same SAM region, corrupting our clustering result.

Solution: A per-pixel depth buffer was introduced in `./pruning/project.py`. For each view, we compare the depth of each Gaussian. We initialize a `depth_buffer` to compute and store the minimum possible depth of the gaussian that map to the pixel, then use it as the threshold to filter out the Gaussians that is far than `0.05 * min_z + 0.05` depth.

4.3 Distance Threshold Parameter Sensitivity

The `delta_dist` parameter for `cKDTree.query_ball_point` has a great impact on runtime and correctness. Setting it too high exponentially creates a large number of candidate pairs, causing CPU memory exhaustion and the program would get killed. Setting it too low increase the number of singleton clusters, affecting the clustering quality.

Solution: We prints nearest-neighbor distance statistics (median, mean, 95th percentile) from a 1000-point sample at the start of `run_cluster`, this step allows us to choose the proper`delta_dist` value that is relative to the scene's point density.

4.4 Coordinate Frame Consistency

The `gsplat` training code uses `similarity_from_cameras`, `align_principal_axes`, and conditional z-flip method to normalize the camera and points coordination. If we loaded the cameras without applying the same normalization step, our rasterization pixel and SAM's image pixel would be inconsistent, corrupting our clustering result.

Solution: We imports the same normalization steps from `gsplat_source/examples/datasets/normalize.py` to the `./pruning/camera.py`.

5. Engineering Decisions

5.1 Post-Training Pruning

We chose post-training method rather than in-training or pre-training:

1. During early training, Gaussian positions are unstable and spatial/depth variance metrics are unreliable. `gsplat`'s default split and clone operations continuously change the Gaussian count and indices, making it difficult to integrate SAM inference during training. SAM inference is also computationally expensive.
2. Filtering out background pixels based on SAM segmentation before training is non-trivial; clustering must be performed after the training process.

5.2 Rendered Images Using Camera Poses for SAM

As for SAM's input, We use the current camera poses and trained model to render corresponding images, not the original provided images. This ensures that the 2D masks and the projected Gaussian positions are produced from the same camera parameters, so that they are aligned perfectly.
The tradeoff is that SAM quality depends on training model's performance. For undertrained models, SAM masks may degrade.

5.3 Multi-View Visibility as the Dominant Score Signal

We did the ablation testing by setting up `alpha=1` only, `beta=1` only, and `gamma=1` only on multiple scenes. Surprisingly, we found that **multi-view visibility alone** (`alpha=0, beta=0`) is a sufficient metric for effective pruning. See the print out the normalized values for each component from the garden scene:

Percentile	Spatial Var	Depth Var	Multi-view Visibility Ratio
0%	0.0000	0.0000	0.0000
25%	0.0000	0.0000	0.0000
50%	0.0000	0.0000	0.0000
75%	0.0848	0.0000	0.0000
90%	0.2874	0.0000	0.4286
95%	0.4805	0.0000	0.7143
100%	1.0000	0.0000	1.0000

Table 4: Statistics Summary

1. **Spatial variance only** (`alpha=1, beta=0, gamma=0`): The clusters experiences detail loss similarly. We have kept clusters small (lower `delta_dist`) to enable precise pruning. However, the metric cannot distinguish a compact foreground cluster from a compact background cluster.
2. **Depth variance only** (`alpha=0, beta=1, gamma=0`): The garden scene's depth variance is near-zero for 95%+ of clusters (see percentiles above). After normalization, the all the values are becoming zero, so that the metric provides no signal.
3. **MVR only** (`alpha=0, beta=0, gamma=1`): Clean separation between foreground (high visibility across views) and background/artifacts (visible in few views or overal visibility portion is low).

Based on the tests, We adopt the final parameters `alpha=0.1, beta=0.1, gamma=1`.

5.4 Geometric Awareness Only (No Object Recognition)

We chose not to integrate any object recognition or semantic labeling model (e.g., CLIP), since we want to focus on a general scanning situation without label prompting. The SAM segmentation output provides instance masks that is not object aware. Therefore, we use them and a series of geometric score formulas (spatial spread, depth consistency, multi-view presence) to filter out the noise (backgroud). Our algorithm isolates the geometric contribution for each cluster instead of combining it with semantic classification. Our results demonstrate that geometry-based multi-view visibility is sufficient for effective foreground/background separation in the tested scenes.
In the future, we could introduce object-level semantics prompting (e.g., "keep only the bonsai tree"), this could be integrated in the pre-processing step, and it is crucial to choose an accurate and computing-friendly model.

5.5 Serialized Stage Outputs for Iterative Tuning

Each pipeline stage writes its output a `.pt` file, such as `cluster_result.pt`, `cluster_scores.pt`, `pruned_model.pt`. Therefore we can easily examine with different parameters such as `semantic_tol`, `alpha`, `beta`, `gamma` with the minimum required overhead. We don't need to re-run computing-heavy steps again. For example, we would not need to re-score if we just want to prune at a different ratio. During development, this saved significant iteration time when tuning parameters for each scene.

5.6 Small Cluster Size by Design

We set the `delta_dist` parameter relatively small (default 0.01) to produce fine-grained clusters. While larger clusters would be faster to form and score, they reduce pruning precision: a single large cluster might get blocked at some scale in the captured images, so we can't guarantee its multi-view ratio is the highest at all times. Small clusters can be individually evaluated and removed with less risk. The cost is that this solution pruduces a larger number of clusters and more singleton clusters (individual Gaussians), but the scoring step is GPU-vectorized and we handle large cluster counts efficiently.

6. Critical Risk Analysis: Failure, Misuse, Bias, and Scale

6.1 Failure

A major weakness of our system is that out system replies on the performance of a 2D segmentation model (SAM) to determine which 3D geometric structures should be retained. If SAM misses a critical object, the entire pipeline will unhesitatingly discard it.

6.2 Misuse

Our project may be used for evidence tampering. The reality might be twisted if someone use our pipeline to delete unwanted background. Only disclosing part of the reality could twist the public opinion.

6.3 Bias

Since our preprocessing used SAM, it inherits SAM's biases. SAM could may have inconsistent performance for different context. If SAM struggles to accurately segment individuals with darker skin tones, our algorithm will classify these individuals as background and remove them from the scene.

6.4 Scale

If deployed at scale, running the SAM model for every individual scenario will have a significant computational overhead. Also, a large number of candidate pairs will result in a massive number of KD-Tree queries. This will consume an large amount of computing power. Given these energy consumption concerns, optimizing computational efficiency is of paramount importance.

Conclusion & Reflection

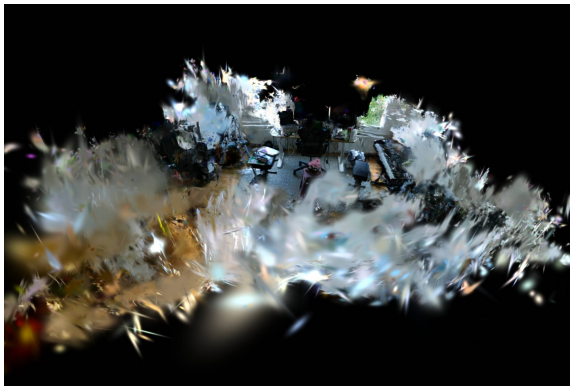
Within this project, we reviewed the computer graphics projection pipeline, and studied how camera poses and world transform matrices work together to represent objects in the world and project them onto frames from different angles. We then examined the 3DGS paper, covering point cloud representation, how 3DGS represents and projects 3D Gaussians onto frames, and how its coloring leverages NeRF-based volume rendering to produce the final color for each pixel. The whole process is accelerated by a tile-based rendering scheme.

We developed a cluster-based pruning method, exploring various segmentation strategies and leveraging the Segment Anything Model (SAM) to generate classification maps across multiple views. By aggregating multi-view labels, we performed clustering and scoring, eventually settling on a multi-view ratio-dominant pruning approach after evaluating different geometric parameters.

Our next steps will be collecting various input datasets, and introducing an input evaluating model or scanning guidance tools to increase the quality of the results and computational efficiency. We might also introduce a semantic-aware model (e.g., CLIP) to guide the pruning, and expand this project to make the objects interactable.

Appendix

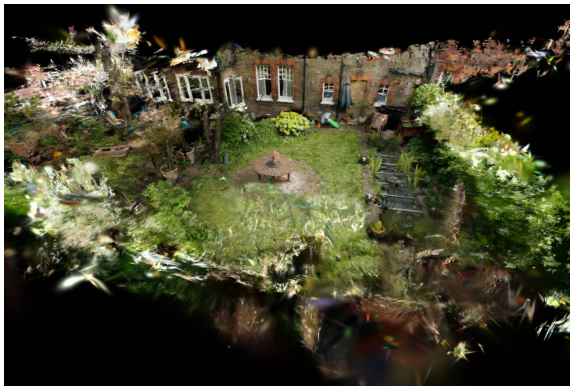
Bonsai



Before Pruning

After Pruning

Garden



Before Pruning

After Pruning

Bicycle

Before Pruning

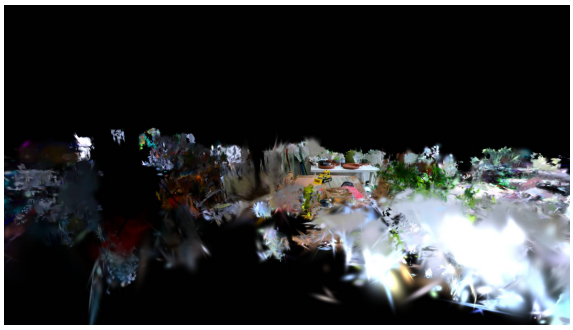
After Pruning

Counter

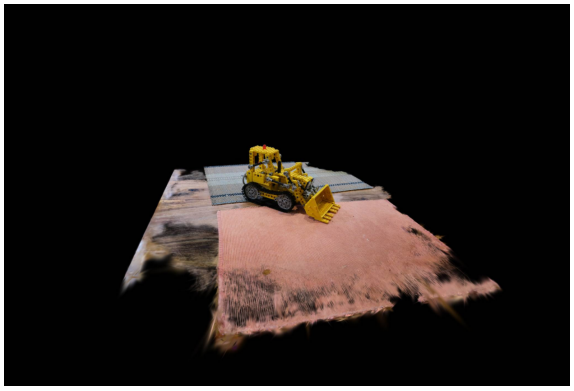
Before Pruning

After Pruning

Kitchen

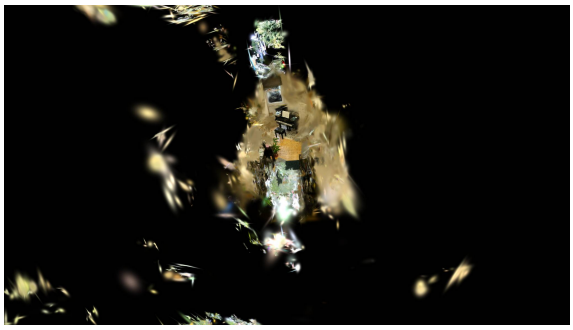


Before Pruning



After Pruning

Room



Before Pruning

After Pruning

Stump

Before Pruning

After Pruning