Nitty-gritty details of the cuda_rasterizer

Annada Behera

NISER, Bhubaneswar

Background I

- **Radiance field** Represents the light distribution in three-dimensional space.

$$L(x, y, z, \theta, \phi) : \mathbb{R}^5 \to \mathbb{R}^+ \tag{1}$$

  where (x, y, z) is the location in space and (θ, ϕ) is the spherical co-ordinates.

- **Implicit radiance field** Light distribution without explictly defining the geometry of the scene. Eg. NeRF.

$$L(x, y, z, \theta, \phi) = \mathrm{NN}(x, y, z, \theta, \phi) \tag{2}$$

  - Differentiable and compact representation
  - High computational load at rendering

Background II

- **Explicit radiance field** Uses geometry information for light distribution in a discrete spatial structure.

$$L(x, y, z, \theta, \phi) = \mathrm{Struct}(x, y, z) \cdot f(\theta, \phi) \tag{3}$$

where "Struct" is point cloud, voxel or triangle mesh.

  - Direct access (often faster)
  - High memory usage (sometimes lower resolution)

- **Gaussian splatting** Explicit radiance field with learnable parameter.

$$L(x, y, z, \theta, \phi) = \sum_i N(x, y, z, \mu_i, \Sigma_i) \cdot c_i(\theta, \phi) \tag{4}$$

where N is the Gaussian distribution function with mean μ_i, co-variance Σ_i, and c_i is the view-dependent color.

Table of contents

Spherical harmonics I

Spherical harmonics are constucted are the eigenfunctions of the angular part of the Laplacian in spherical 3D co-ordinates,

$$\nabla^2 = \frac{1}{r^2 \sin(\theta)} \left(\frac{\partial}{\partial r} r^2 \sin(\theta) \frac{\partial}{\partial r} + \frac{\partial}{\partial \theta} \sin(\theta) \frac{\partial}{\partial \theta} + \frac{\partial}{\partial \phi} \csc(\theta) \frac{\partial}{\partial \phi} \right) \tag{5}$$

The Laplacian of a function $\nabla^2 f = 0$ can be solved by variable separation,

$$f(r, \theta, \phi) = R(r) Y(\theta, \phi) \tag{6}$$

Spherical harmonics II

The angular part, $Y_{l,m}$ is generally defined in terms of integer parameters $l \in \mathbb{N}$ and $-l \leq m \leq l$,

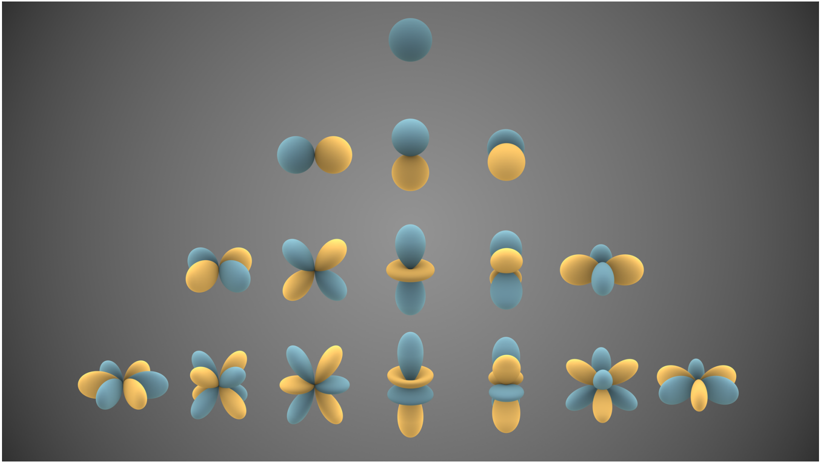
- For $m < 0$, $(-1)^m \sqrt{2} \sqrt{\frac{(2l+1)(l+m)!}{4\pi(l-m)!}} P_l^{-m}(\cos(\theta)) \sin(-m\phi)$
- For $m = 0$, $\sqrt{\frac{2l+1}{4\pi!}} P_l^m(\cos(\theta))$
- For $m > 0$, $(-1)^m \sqrt{2} \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}} P_l^m(\cos(\theta)) \cos(m\phi)$

where $P_l^m(\cos(\theta))$ is the associated Legendre polynomials,

$$P_\ell^m(\cos\theta) = (-1)^m (\sin\theta)^m \ \frac{d^m}{d(\cos\theta)^m} \left(P_\ell(\cos\theta)\right) \tag{7}$$

Spherical harmonics are used in quantum mechanics, classical electrodynamics, black hole physics.

Spherical harmonics III



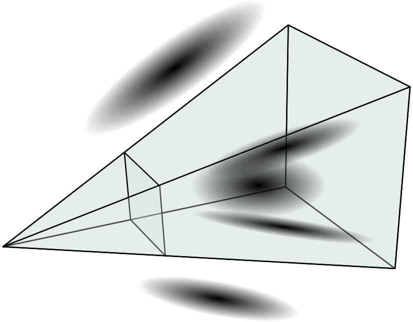
Blue represents positive and yellow represents the negative values, and they form a complete set of basis functions to represent functions on the surface of a sphere S^2.

Spherical harmonics IV

```
__device__ glm::vec3 computeColorFromSH(int idx, int deg, int
max_coeffs, const glm::vec3* means, glm::vec3 campos, const
float* shs, bool* clamped)
```

- `__device__` (cuda specific) run this on the GPU.
- `glm::vec3` is OpenGL 3D vector of RGB color coordinates.
- `idx` is the index of the Gaussian in the Gaussians array
- `deg` is the l value in $Y_{l,m}$
- `max_coeffs` is the maximum coefficient calculated from $Y_{l,m}$
- `means` the mean of the Gaussian μ
- `campos` is the camera position
- `shs` value of $Y_{l,m}(\theta, \phi)$
- `clamped` if truthy then clamp the intensity value to $[0, 1]$

Frustum culling I



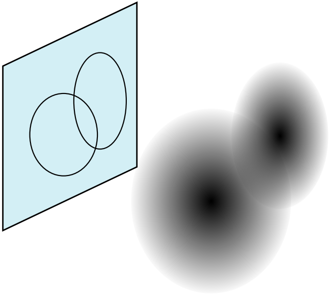
- Given a camera pose, determine which Gaussians are outside the camera frustum and exclude the Gaussians that are not in the frustum from subsequent computation.

Frustum culling II

```
__forceinline__ __device__ bool in_frustum(int idx, const float*
orig_points, const float* viewmatrix, const float* projmatrix,
bool prefiltered, float3& p_view)
__global__ void checkFrustum(int P, const float* orig_points,
const float* viewmatrix, const float* projmatrix, bool* present)
```

- `__forceinline__` same as C++ `inline` but for CUDA
- `__global__` can be called from both CPU and GPU, interface.
- `orig_points` flat array of all points $(N_p, 3)$
- `viewmatrix` world to camera space transformation matrix
- `projmatrix` camera to image space transformation matrix
- `prefiltered` sanity check
- `p_view` ¯_(ツ)_/¯
- `present` output

“Splatting” I



The 2D projected covariance matrix Σ' is computed as,

$$\Sigma' = JW\Sigma W^\top J^\top \tag{8}$$

where J is the “Jacobian of affine approximation of the projective transformation” and W is the world to camera space transformation matrix.

"Splatting" II

```
__device__ float3 computeCov2D(const float3& mean, float focal_x, float focal_y, float tan_fovx, float tan_fovy, const float* cov3D, const float* viewmatrix)
```

- `focal_x, focal_y` The focal length of the camera, f_x, f_y
- `tan_fovx, tan_fovy` The tan of field of view F_x, F_y.

$$J = \begin{bmatrix} f_x/t_z & 0 & -(f_x t_x)/t_z^2 \\ 0 & f_y/t_z & -(f_y t_y)/t_z^2 \\ 0 & 0 & 0 \end{bmatrix} \tag{9}$$

where $t = \mathbf{lookAt} \cdot \mu$ and t_x and t_y is clamped by $1.3 \tan(F_x)$ and $1.3 \tan(F_x)$ respectively.

Rendering the pixels I

Given the position of the pixel x and it's distance to overlapping Gaussians N, (*sorted according to their depth which was computed from W*), the final color of the pixel is,

$$C = \sum_{i \in N} c_i \alpha'_i \prod_{j=1}^{i-1} (1 - \alpha'_j) \tag{10}$$

where the color c_i comes from the spherical harmonics and the final opacity comes from,

$$\alpha'_i = \alpha_i \exp\big(-\frac{1}{2}(x' - \mu'_i)\Sigma_i'^{-1}(x' - \mu'_i)\big) \tag{11}$$

where x and μ are from projected image space.

Bottlenecks

- Parallel sorting on the GPU. (Bitonic, Radix?)
- Large number of pixels and Gaussians.

Rendering the pixels II

This final color against the background and opacity,

```
for (int ch=0; ch<CHANNELS; ch++)
    out_color[ch*H*W + pix_id] = __fmaf_rn(T,bg_color[ch],C[ch]);
```

- `__fmaf_rn` (CUDA Math API) Fused multiply-add, $xy + z$ and round-to-nearest-even mode.
- `T` is the transmittance.

The transmittance is calculated as,

```
float power = __fmaf_rn(-con_o.y * d.x, d.y, __fmaf_rn(-0.5f *
↪  con_o.x, d.x * d.x, __fmaf_rn(-0.5f * con_o.z, d.y * d.y,
↪  0.f)));
// Eq. (2) from 3D Gaussian splatting paper.
float alpha = fminf(0.99f, __fmaf_rn(con_o.w, expf(power), 0.f));
if (alpha < ALPHA_THRESHOLD_FWD) continue;
float test_T = __fmaf_rn(T, -alpha, T);
if (test_T < ALPHA_MIN_FWD) { done = true; continue; }
```

The color from the Gaussian, `C[ch]` is,

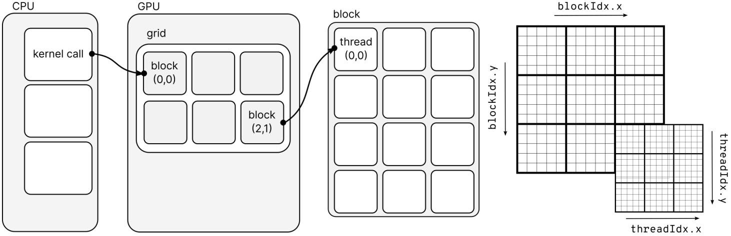
Rendering the pixels III

```
// Eq. (3) from 3D Gaussian splatting paper.
for (int ch = 0; ch < CHANNELS; ch++)
    C[ch] = __fmaf_rn(features[collected_id[j] * CHANNELS + ch],
    ↪  __fmaf_rn(alpha, T, 0.f), C[ch]);
// Compute 2D screen-space covariance matrix
float3 cov = computeCov2D(p_orig, focal_x, focal_y, tan_fovx,
↪  tan_fovy, cov3D, viewmatrix);

// Invert covariance (EWA algorithm)
float det = (cov.x * cov.z - cov.y * cov.y);
if (det == 0.0f)
    return;
float det_inv = 1.f / det;
float3 conic = {cov.z * det_inv, -cov.y * det_inv, cov.x *
↪  det_inv};
```

Tiles I

The CUDA streaming multiprocessor (SM) hierarchy,




```
renderCUDA<NUM_CHANNELS><<<(P + 255) / 256, 256>>>(...)
preprocessCUDA<NUM_CHANNELS><<<(P + 255) / 256, 256>>>(...)
```

The processing if done not as a whole but over a set of non-overlapping patches called tiles of size 16×16.

```
// Allocate storage for batches of collectively fetched data.
__shared__ int collected_id[BLOCK_SIZE];
__shared__ float2 collected_xy[BLOCK_SIZE];
__shared__ float4 collected_conic_opacity[BLOCK_SIZE];
```

Covariance matrix

$$\Sigma = \mathbf{R}\mathbf{S}\mathbf{S}^{\mathbf{T}}\mathbf{R}^{\mathbf{T}} \tag{12}$$

where, $\mathbf{R}$ and $\mathbf{S}$ are the rotation and scaling matrix obtained from q and s respectively.

```
__device__ void computeCov3D(const glm::vec3 scale, float mod,
const glm::vec4 rot, float* cov3D)
```

Computational graph: $(q, s) \mapsto \Sigma \mapsto \Sigma' \mapsto \alpha$

Backward pass I

```
__device__ void computeColorFromSH(int idx, int deg, int
max_coeffs, const glm::vec3* means, glm::vec3 campos, const
float* shs, const bool* clamped, const glm::vec3* dL_dcolor,
glm::vec3* dL_dmeans, glm::vec3* dL_dshs)
```

- `dL_dcolor` is $\partial L / \partial C$
- `dL_dmeans` is $\partial L / \partial \mu$
- `dL_dshs` is $\partial L / \partial Y$, the co-efficients of the spherical harmonics.

```
__global__ void computeCov2DCUDA(...  const float* dL_dconics,
float3* dL_dmeans, float* dL_dcov)
```

- `dL_dconics` is $\partial L / \partial C$
- `dL_dcov` is $\partial L / \partial_{\text{2DCov}}$

Backward pass II

```
__device__ void computeCov3D(int idx, const glm::vec3 scale,
float mod, const glm::vec4 rot, const float* dL_dcov3Ds,
glm::vec3* dL_dscales, glm::vec4* dL_drots)
```

- `dL_dscales` and `dL_drots` is derivative w.r.t. q and s.
- `dL_dcov` is $\partial L / \partial_{\text{3DCov}}$