Differentiable Rendering

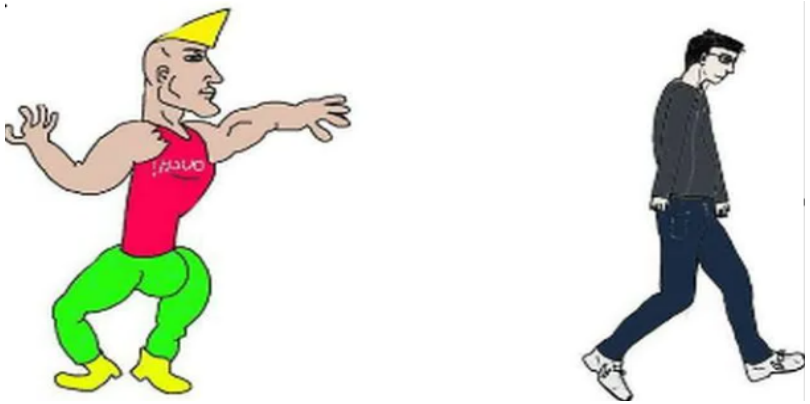
Matej Lorinc

- **novel-view synthesis**
- simulating a new view at a scene
  - using ground-truth views
- view <=> render <=> image <=> picture

- classical neural networks:
  - computational graphs
  - gradient of loss function with respect to paremeters - weights
- lets generalize
  - input: camera position
  - paremeters: color, opacity of pixels
  - forward pass: render
  - still differentiable
- lets apply to novel-view synthesis

- first attempt: **radiance fields**
  - each volumetric pixel (voxel) in 3d is a set of parameters
  - render, compute loss (with some diff lossfunc), backprop
  - optimize the whole volume tensor field
  - implicit repres. => shooting a ray per pixel and sampling along the way
  - extremely inefficient (dense sampling / evaluations)

- second attempt: **splatting**
  - explicit geometric objects (can be thought of as neurons)
  - each has its own parameters (position, orientation, size etc.)
  - each contributes to the render in some way
  - advantages:
    - less memory
    - primitives => rasterization
    - rasterization > ray marching (in practice)



- **3D gaussians**
  - parameters: gaussian function in 3d
  - opacity, color
- rendering: **rasterization**
  - project to 2d ellipses
  - bin into tiles (bounding boxes - great)
  - sort by depth
  - alpha-blend

- **training**
  - randomly initialize gaussians
  - render and backprop
- **optimization**
  - where needed divide geometry (high detail regions)
  - where needed join geometry (low detail)
  - + million other improvements
    - where and how to split gaussians
    - how to initialize them
    - anti-aliasing

- python implementation: **gsplat**
- minimal torch-based library
- **example**
  - showing: parameters, gradients, optimization loop
  - aula UK 2D image
  - 100 000 initial gaussians
  - 1000 iterations
  - pinhole camera

```python
def _init_gaussians(self):
    bd = 2

    self.means = bd * (torch.rand(self.num_points, 3, device=self.device) - 0.5)
    self.scales = torch.rand(self.num_points, 3, device=self.device)
    d = 3
    self.rgbs = torch.rand(self.num_points, d, device=self.device)

    u = torch.rand(self.num_points, 1, device=self.device)
    v = torch.rand(self.num_points, 1, device=self.device)
    w = torch.rand(self.num_points, 1, device=self.device)

    self.quats = torch.cat(
        [
            torch.sqrt(1.0 - u) * torch.sin(2.0 * math.pi * v),
            torch.sqrt(1.0 - u) * torch.cos(2.0 * math.pi * v),
            torch.sqrt(u) * torch.sin(2.0 * math.pi * w),
            torch.sqrt(u) * torch.cos(2.0 * math.pi * w),
        ],
        -1,
    )
    self.opacities = torch.ones((self.num_points), device=self.device)
```

```
        self.viewmat = torch.tensor(
            [
                [1.0, 0.0, 0.0, 0.0],
                [0.0, 1.0, 0.0, 0.0],
                [0.0, 0.0, 1.0, 8.0],
                [0.0, 0.0, 0.0, 1.0],
            ],
            device=self.device,
        )
        self.background = torch.zeros(d, device=self.device)

        self.means.requires_grad = True
        self.scales.requires_grad = True
        self.quats.requires_grad = True
        self.rgbs.requires_grad = True
        self.opacities.requires_grad = True
        self.viewmat.requires_grad = False
```

```
        # ...
        optimizer = optim.Adam(
            [self.rgbs, self.means, self.scales, self.opacities, self.quats], lr
        )
        mse_loss = torch.nn.MSELoss()
        # ...
        for iter in range(iterations):
            start = time.time()

            renders = rasterize_fnc(
                self.means,
                self.quats / self.quats.norm(dim=-1, keepdim=True),
                self.scales, #  ...
            )[0]
            out_img = renders[0]
            #  ...
            loss = mse_loss(out_img, self.gt_image)
            #  ...
            loss.backward()
            #  ...
            optimizer.step()
```


UNIVERZITA KOMENSKÉHO