Animating Street View

Mengyi Shan
shanmy@cs.washington.edu
University of Washington
Seattle, WA, US

Steve Seitz
seitz@cs.washington.edu
University of Washington
Seattle, WA, US

Brian Curless
curless@cs.washington.edu
University of Washington
Seattle, WA, US

Ira Kemelmacher-Shlizerman
kemelmi@cs.washington.edu
University of Washington
Seattle, WA, US

ACM Reference Format:
Mengyi Shan, Steve Seitz, Brian Curless, and Ira Kemelmacher-Shlizerman. 2023. Animating Street View. In *SIGGRAPH Asia 2023 Conference Papers (SA Conference Papers '23), December 12–15, 2023, Sydney, NSW, Australia.* ACM, Sydney, Australia, 5 pages. https://doi.org/10.1145/3610548.3618230

In this supplementary file, we provide more implementation details of our system not included in the main paper.

A RECONSTRUCTION

We mostly describe how the scene is set up in Unity after estimation of major scene properties. These functionalities are implemented in scripts so that we don't have to manually create and edit the objects through graphical user interface.

A.1 Inpainting

We refer to the `Pedestrian` and `Vehicle` classes from the segmentation map as intended inpainting regions. We first divide the regions into connected components for every single pedestrian and/or car. Then we cluster the objects into groups based on adjacency, crop a local 512×512 patch, and take the smallest rectangle with a 10% margin that covers the inpainted region as the mask. We use *"A photo of an empty street"* as the inpainting prompt, and *"Human, pedestrian, vehicle, car"* as the negative prompt. Finally we compose the inpainted cropped patches back onto the original image.

A.2 Walk/Drive Region Segmentation

We start with deciding if the figure is pedestrian-only or vehicle-only by segmenting the image with keywords "drive" and "walk" using [Lüddecke and Ecker 2022]. If the resulting walk(drive) region is less than 1% of the image area, we make the scene vehicle(pedestrian)-only. With a vehicle(pedestrian)-only scene, all the `Sidewalk` and `Road` pixels are identified as drive(walk) regions. In scenes with both pedestrians and vehicles, `Sidewalk` pixels are "walk" regions and `Road` pixels are "drive" regions. To handle objects on street that "blocks" some of the ground pixels, we always dilate the walk regions by 20 pixels and then erode it by 20 pixels (for a 4032×3024 image) thus keeping the main region unchanged but adding on the blocked regions. At the same time, we automatically compute the lowest grounded pixel p and radius r of each objects (poles, pole groups, etc) on street by considering each of them a cubiod mesh. When projecting to BEV map, each object is approximated as a circle with center at p and radius r.

We decide the start and target points for each pedestrian by taking all the "walkable" pixels at the boundary of the image, and the farthest 5% of all "walkable" pixels. When initializing a pedestrian, we randomly choose one point from the "intersection" pool and one point from the "farthest" pool.

We compute the car lanes by taking an average car lane width of 11 feet. We divide the approximated road region into $n = 1, 2, 3$ or 4 lanes of the same width so that the width is closest to the average car lane width of 11 feet. Our system is not able to handle more car lanes, extreme car lane width, or curved car lanes.

A.3 Ground Plane Estimation

After estimation the plane equation $aX + bY + cZ = 1$, we set up a `Plane` GameObject in Unity with scale 100×100m following the equation $aX + bY + cZ = 1$, with the camera pointing towards positive Z-axis. This GameObject is set to be a rigid body without gravity so that it automatically stays at the estimated position as a playground for the characters.

We additionally take advantage of existing characters in the scene to help adjust the scale of ground plane equation if the given scene is not fully empty. In that case, we compute a constant s with the pre-estimated a, b, c so that $aX + bY + cZ = s$ instead of $aX + bY + cZ = 1$. This fixes the ground plane normal but adjust the scale of objects.

When there are `Building` wall pixels next to the ground region as indicated by the semantic segmentation results, we build a wall Plane GameObject in Unity that is perpendicular to the wall/ground boundaries. This is prepared for rendering shadow cast on the building regions.

A.4 Sun Estimation

The script automatically sets up a directional light as the sun after estimating the direction and intensity. Because we are only using the light source in Unity, we don't bother estimate a full environment light map but instead attempt to predict one sun light intensity value and one environment (ambient) light intensity value. In practice, we find that subtle intensity difference is not too much noticeable as long as in a reasonable region. We divide the scene into three weather categories based on the variance v of sun direction prediction bins, and set I_{sun} and I_{ambient} according to Table 1.

Table 1: Sun light and environment light settings

I_{sun}	I_{ambient}	v range	Weather	Strong directional light
2	1.5	$v > 0.3$	Sunny	Yes
1.5	2	$0.1 < v < 0.3$	Sunny	No
1	1.5	$v < 0.1$	Cloudy	No

We prepare data for the sun estimation network by taking the 19093 panoramas in the dataset, and generating 10 perspective images from each panorama, randomly choosing the FOV from (60, 80, 90) degree, camera elevation from (-10, 0, 10) degree, and camera azimuth from (150, 180, 210, -30, 0, 30) degree, so that the data covers a wide enough range of images similar to our test images. The sun estimation network is trained with Adam optimizer with a learning rate of 1e-3. We train the network for 50 epochs with batch size 16. We train on two GeForce RTX 2080 GPUs for four days.

A.5 Shadow Occluders

We start with identifying shaded ground regions by processing the image with [Hong et al. 2022] and taking ground pixels with prediction values greater than 0.1. We project these shaded ground pixels in the image back to a plane of height $h = 10$ with a given sunlight direction. In Unity, we add Cube objects with size 0.1×0.1 meters at each position so that they block the sun light properly to mimic the existing shadow effects in the image. At rendering time, we keep the shadow effects created by these Cube occluders but set themselves invisible. If the figure is identified as without strong directional light source, the shadow occluders are disabled in the rendering stage.

B SIMULATION

B.1 Crowd Simulation

Here we provide an in detail derivation of the crowd simulation algorithm. We would like to compute the optimal path P by

$$\operatorname*{argmin}_P \ \underbrace{\alpha \int_P 1 ds}_{\text{Path Length}} + \underbrace{\beta \int_P 1 dt}_{\text{Time}} + \underbrace{\gamma \int_P g dt}_{\text{Discomfort}} \tag{1}$$

where α, β, γ are weight parameters that could be set by hand. The integral terms ds means that the integral is taken with respect to path length while dt means the integral is taken with respect to time. These two variables satisfy the relationship $ds = vdt$ where v is the speed. This can be further simplified to

$$\operatorname*{argmin}_P \int_P C ds, \quad \text{where} \quad C \equiv \frac{\alpha v + \beta + \gamma g}{v} \tag{2}$$

In practice, we set $\alpha = \beta = 1$ and $\gamma = 2$. To compute the discomfort field, we make the value linearly decrease through the 10 closest grid to obstacle boundaries and sidewalk boundaries. We take advantage of the fast-marching algorithm implemented in scikit-fmm[1] to compute the potential field. And use a step size $\Delta t = 0.1$ to move the characters in the direction of descending potential.

B.2 Traffic Simulation

We store the cars on each lane in a Linked-List style data structure, with each car object pointing towards the previous car (or to null if it's the first car in a row). If the traffic light turns red for cars, the car objects within a distance threshold to the crosswalk will start decelerating so that it can stop within $d = 1$ meter to the crosswalk. At the same time, a car will automatically decelerate no matter what the traffic light is if it becomes too close to the previous car, so that it can stop within $d = 1$ meter to that car. We dynamically compute the threshold distance to start decelerate or accelerate based on speed, deceleration and acceleration values. In practice, the speed is set to 15 meters per second in urban street scenes and 25 meters per second in vehicle-only freeway scenes. The acceleration and deceleration are both set to 5 meters per square seconds. These values could be changed manually to generate diverse traffic dynamic.

C RENDERING

We describe how the rendering and recording process is implemented in Unity as well as post-processing details. These functionalities are implemented in scripts so that all the manual works are replaced by a simple click.

C.1 Shadow

Unity will be able to generate realistic shadow shapes for our objects, but not color because the scene albedo and material are not reconstructed in the game engine. To merely take advantage of the shadow shape information, we record two videos through Unity recorder with fps = 30, one with and one without hard shadow.

To automatically decide the shadow color, we run a local shadow color matching algorithm on the intersection of local shadow and non-shadow regions. We take 10 patches of 200×200 pixels, pick one that has shadow and non-shadow region almost 1:1, and take the average of shadow and non-shadow RGB colors. Then we compute a three channel ratio between these two RGB values. To shade a pixel, we multiply it by this three channel shadow ratio.

To avoid double shadow, we only shade a pixel if it's not in the shadow region. To eliminate the unnaturalness when the existing shadow is soft and blurry, we compute a boundary region by dilating the shadow and non-shadow region each by 10 pixels and take the intersection. Then we apply a Gaussian blur kernel of size 20×20 on the boundary region's shadow factor.

When the scene is classified as without strong sun source, we only set up a top light perpendicular to the ground and render the shadow region with a shadow factor $s = 0.9$ for human and $s = 0.7$ for cars, because cars in general are closer to the ground and may block more light. The shadow masks are run through a Gaussian blur process with kernel size of 20×20 to create the diffuse, soft shadow effect.

[1]https://github.com/scikit-fmm/scikit-fmm

If there is a reconstructed building wall object in the scene, we treat it in the same way as the ground plane and record shadow cast on that plane.

C.2 Occlusion

To acquire the depth values of the added objects, we simulate a depth camera with the exact same parameters as the RGB camera to record a depth video during the rendering process. This is achieved by a customized shader in Unity.

We refine the estimated depth D_{bg} for each pixel belonging to `pole`, `pole group`, `traffic sign`, `traffic light` classes by computing their contact point with ground and depth of that pixel given ground equation. Specifically, we divide the segmentated objects regions into instances by connected components segmentation, and identify the grounding pixel of each objects. Because we know the ground plane equation, we can back project this pixel into 3D to retrieve its depth. This method assumes that the objects are thin and almost perpendicular to the ground plane.

D ASSETS

We directly buy assets for pedestrian and cars from online asset stores. Specifically, the car assets are bought from the official Unity Asset store[2], the pedestrian assets are bought from RenderPeople[3] and additional walking motion clips are bought from ActorCore[4]

We make each pedestrian character a prefab in Unity with a `CharacterController` component and a `Animator` component. The former controls gravity and path following, while the latter controls switching between `Idle` and `Walk` motions with different speed. When playing the characters in Unity, we make each character controller walk on their own pace with a root motion baked into the pose, so that they won’t look like floating on the ground. We build the path following mechanism through controller the rotation of each character controller. In practice, we keep the character always facing towards the next not reached point on the predicted path. Unity’s built-in `Quaternion.RotateTowards` function controls a smooth rotation towards the direction of next point, instead of a hard turn. This in fact provides tolerance to some unnaturalness of our planned path, because Unity will be able to smooth out the path again without violating the main structure.

Similarly, we make each car object a prefab in Unity and directly change their positions at each time step. We implement the accelerating and deccelerating effects by linearly increasing and decreasing the cars’ moving speed.

As a demo, we only include six different character appearances for pedestrians, two walking motion pieces, and a dozen of car models. This could be extended to any number of diverse assets to create even better visual effects.

E PANORAMA

For the reconstruction and rendering stage, we decompose the equirectangular panorama into 6 still images in a cubemap manner,

Figure 1: Assets used in the rendering process. (a) Pedestrian character assets from RenderPeople. (b) Car assets from Unity Store.

and apply all the processing steps (depth estimation, semantic segmentation, inpainting, shadow/occlusion composition) with each direction (up, down, left, right, top, bottom). Crowd simulation is no different from regular images as long as we can project the semantic segmentation information to the BEV representation.

In Unity, because there is no ready-to-use implementation of 360 depth camera, we render six cameras at the same time in a cubemap manner, and convert from cubemap to equirectangular when viewing.

F IMAGE ATTRIBUTION

- **The empty solar spring intersection and crosswalks** by Aleksandr Volkov with standard license at Adobe Stock. Asset ID #156108165. Figure 16 first row first column. Video 00:02-00:12, 02:20-02:42.
- **Mijas white washed street, small famous village in Spain. Charming empty narrow streets with New Year decorations, on houses walls hanging flower pots, sunny day no people. Costa del Sol, Málaga** by Alex Tihonov with standard license at Adobe Stock. Asset ID #319516381. Figure 9 top left. Video 01:00-01:15.
- **Empty street - Sunday afternoon** by dinu_dragomirescu. Wikimedia Commons. CC BY 3.0. Video 04:10-04:25. Figure 16 second row.
- **Empty streets** by Geoff Staubyns. Wikimedia Commons. CC BY 4.0. Video 03:41-03:56. Figure 8 bottom row. Figure 9 bottom left. Figure 14. Figure 2 left column.
- **Empty street in a village in central Crete (Avli)** by Annatsach. Wikimedia Commons. CC BY 4.0. Video 03:26-03:41. Figure 7. Figure 12 bottom row.

REFERENCES

Yan Hong, Li Niu, and Jianfu Zhang. 2022. Shadow Generation for Composite Image in Real-world Scenes. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

Timo Lüddecke and Alexander Ecker. 2022. Image Segmentation Using Text and Image Prompts. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 7086–7096.

[2]https://assetstore.unity.com/packages/3d/vehicles/archviz-car-collection-pack-1-169659

[3]https://renderpeople.com/free-3d-people/

[4]https://actorcore.reallusion.com