RenGen: Generating Images with as Many Tokens as the Scene Needs

Savya Khosla Jay Mahajan Akin Tuna Sakalli Derek Hoiem David Forsyth Alex Schwing
University of Illinois Urbana-Champaign

Abstract

We present RenGen, a latent diffusion model that generates simple scenes from a few tokens and busy scenes from many, saving time and compute on simple scenes without losing realism. Most latent generators represent every image as a fixed grid of patch tokens, so a simple image costs as much to generate as a complex one. Prior methods that vary the token count through content-adaptive tokenization lose image quality as the count shrinks. RenGen instead trades scene complexity, rather than image quality, for fewer tokens. It does so by generating in the latent space of a pretrained region encoder, where each token represents one region of the image, and a decoder renders these region tokens back into pixels. Unlike patch tokens, region tokens cover areas of varying shape and position, so each token must capture both what its region contains and where it lies on the image canvas. We encode this spatial extent by adding positional embeddings to the encoder features before pooling, and we generate it by anchoring each token to a single randomly sampled point, from which the generator infers the region's shape, size, and content. At sampling time, the number of anchor points sets the trade-off between scene complexity and compute. On ImageNet 256×256*, RenGen reaches a gFID of 1.34 with 88 tokens per image on average, the best among methods that vary their token count. It approaches the quality of RAEv2, a state-of-the-art patch-based model, at* $2.5\times$ *lower latency, advancing the Pareto frontier of generation quality versus compute. RenGen's gFID stays flat across scene complexity while latency varies by* $2.4\times$ *between simplest and busiest scenes, and its autoencoder reconstructs nearly as well as RAEv2 (rFID 0.32 vs. 0.30) with* $2.9\times$ *fewer tokens.*

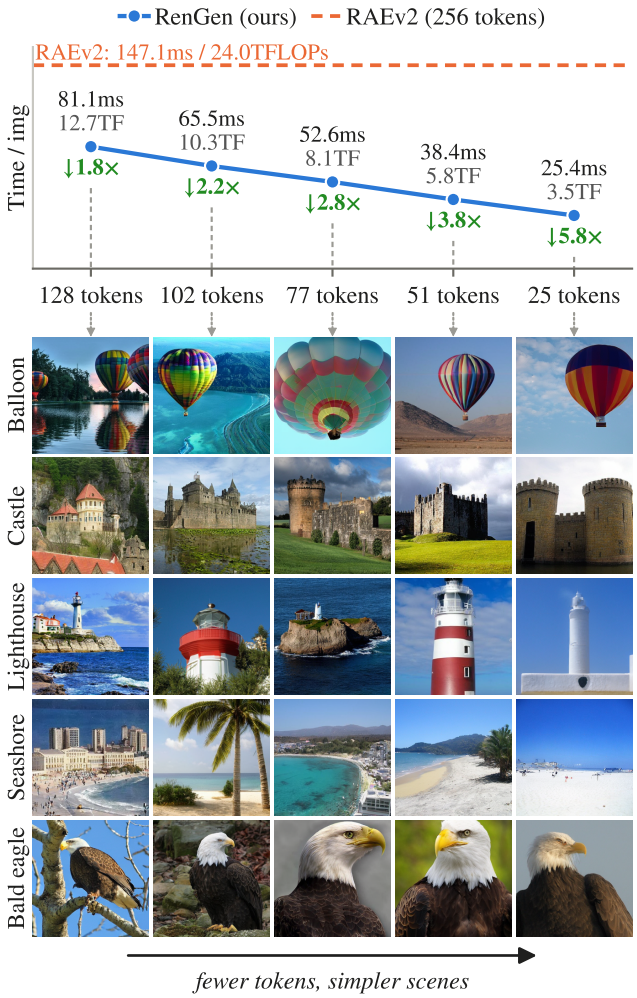

Figure 1. **RenGen's generation cost follows image content.** *Top:* Wall-clock time and TFLOPs per image for end-to-end generation at different token counts for RenGen. Green values give the speed-up over RAEv2, which always uses a fixed grid of 256 patch tokens (dashed line). *Bottom:* 256×256 samples from a single RenGen model. Each row uses the same class label, and each column uses the token count of the point above it, so fewer tokens yield simpler scenes at lower cost.

1. Introduction

We show how to generate realistic images from a small number of tokens, thereby reducing generation time and compute (Fig. 1). We do so by designing a generative model whose latents are region tokens from a pretrained image encoder, each representing one region of the image [19, 22, 38]. The number of tokens then controls scene complexity rather than image quality: fewer tokens yield simpler scenes and more tokens yield busier ones, both

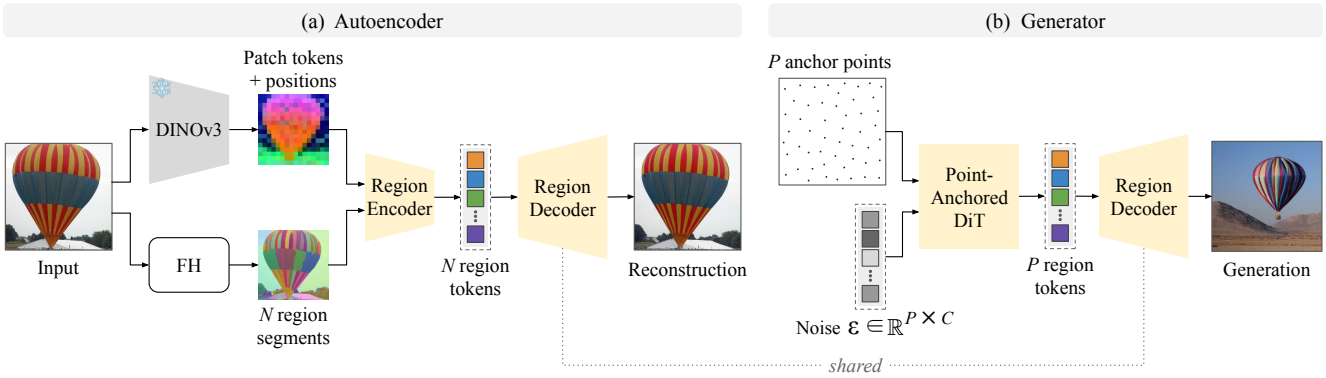

Figure 2. **RenGen overview.** (a) The region encoder partitions an image into N regions with Felzenszwalb–Huttenlocher (FH) segmentation and pools frozen DINOv3 patch features within each region into region tokens, which the decoder renders back into an image. (b) For generation, P random anchor points fix the region positions. A DiT denoises one region token per anchor point, using the anchor coordinates as 2D RoPE positions, and the shared decoder renders the generated tokens into an image.

equally realistic. The key difficulty in generating with region tokens is that each token must specify which pixels it covers in addition to what it contains. We show that anchoring each token to a single randomly sampled point suffices for the generator to infer the region's full extent (shape and size) and content.

Most current generators tokenize every image into a uniform grid of patches [25, 33, 36, 53], requiring the same generation time and compute for simple and complex images at a given resolution (dashed line in Fig. 1). Generators whose compute instead scales with scene complexity would let a single model serve devices with different compute budgets and latency requirements. Prior work pursues this with content-adaptive tokenizers [1, 31, 37] and pixel-space generators that use a variable number of tokens [18, 49]. In both, however, fewer tokens come at the cost of quality. Adaptive tokenizers lag behind leading patch-based autoencoders in reconstruction (Tab. 3), and variable-token generators lose image quality as the token count shrinks (Fig. 5).

We introduce RenGen, a latent generative model built on region tokens. RenGen derives these tokens from the patch features of a pretrained image encoder such as DINOv3 [39], which RAE [53] identifies as effective diffusion latents. Following region encoders [19, 38], it pools these features within image segments, so that the token count follows image content. A decoder renders the resulting region tokens into pixels, and a diffusion transformer (DiT) [33] generates them from noise (Fig. 2). Since a region token covers an arbitrary set of pixels, both decoder and generator face a design question that patch-based models avoid.

How can the decoder recover a region's spatial extent from a single token? Region tokens from prior region encoders [19, 22, 38] do not retain enough positional information to decode a region's spatial extent (Sec. 4.2). RenGen addresses this by adding random Fourier positional embeddings to the patch features before pooling, so that each region token records where its region lies along with what it contains. The decoder then reconstructs the image from patch-grid queries that carry Fourier features of the same random directions and cross-attend to the region tokens, so that each output location learns to gather detail from the regions that cover it, regardless of their number or layout.

How can the DiT generate region tokens when nothing fixes where each region lies? Unlike patch-based generators, where each token's position is fixed in advance to a 16×16 grid cell and the model only needs to generate its content, a region-token generator lacks any implicit spatial prior. We find that generating region layouts jointly with content degrades quality, while generating them in a separate stage erodes the efficiency gained from fewer tokens (Sec. 4.3). Instead, we anchor each token to a single point sampled uniformly on the image canvas, which serves as its 2D RoPE position. From this anchor point our DiT generates the region's shape, size, and content. The number of anchors P thus becomes a sampling-time control over the trade-off between scene complexity and compute.

In summary, our contributions are:

- We introduce RenGen, a latent diffusion model over region tokens that trades scene complexity, not image quality, for compute. Its token budget is chosen at sampling time, and on ImageNet 256×256 a single model reaches a gFID of 1.34 with 88 tokens on average, the best among methods that vary their token count, at $2.5\times$ lower latency than an identically configured patch-based model.
- We make region tokens decodable into images. Tokens formed by simply pooling pretrained image encoder features, as in prior region encoders [19, 22, 38], carry too little position information for the decoder to recover which pixels each one covers. Studying several designs, we find a simple fix that future region encoders can readily adopt, which is to add positional embeddings to the patch features before pooling them into region tokens.

The resulting model reconstructs nearly as well as RAEv2 (rFID 0.32 vs. 0.30) with 2.9× fewer tokens.

- We show that placing each region token at a random point on the image is enough for the generator to produce realistic regions around it. Unlike patch tokens, region tokens have no fixed grid that tells the generator where each one belongs, and generating their positions alongside their content is either slow or degrades quality. Anchoring region token generation to random points avoids both problems at no extra cost.

2. Related Work

Region encoders. Recent works show that region tokens pooled from pretrained vision foundation models are a strong alternative to patch tokens [19, 22, 38]. These representations use far fewer tokens and perform well across tasks, including semantic segmentation [19, 26, 42, 43], one-shot object localization and retrieval [20, 28], vision-language modeling [12, 21, 35, 47], and video understanding [20, 48]. While effective for understanding tasks, we find that region tokens from prior methods are insufficient for reconstruction, because pooled tokens carry little information about where a region lies and what shape it has. We make region tokens decodable by adding positional embeddings to patch features before pooling. Object-centric models also decode images from region-like latents [17, 41], but they use a fixed number of slots per image, whereas our token count follows the number of segments in each image.

Representation-based generation. Recently, there has been a push towards training diffusion transformers with representation autoencoders (e.g., DINO, SigLIP, MAE) [6, 40, 53] instead of standard VAEs [23] used in modern text-to-image and text-to-video models [3, 13, 34, 45]. These works show the promise of improved generation as these autoencoders are believed to have a better semantic understanding of images. Our work follows a similar motivation as we use a representation encoder, but aim to reduce the required token count by focusing on the regions of the images themselves rather than the patches.

Efficient token generation. Several works explored efficient ways to generate an image with fewer tokens. RTI [49] employs a Hilbert curve to detect which features change the most and uses LoRA fine-tuning to handle variable region token budgets. Token Merging (ToMe) [4] groups tokens by feature similarity within a forward pass. BudgetPix [18] employs an adaptive encoder and decoder with multiple resolution paths and fine-tunes an existing DiT. All of these works either adapt an existing pixel-based or latent-based flow model to complete this task. Our work differs as we directly model a region space that is semantically aligned and flexible to handle a variable number of tokens.

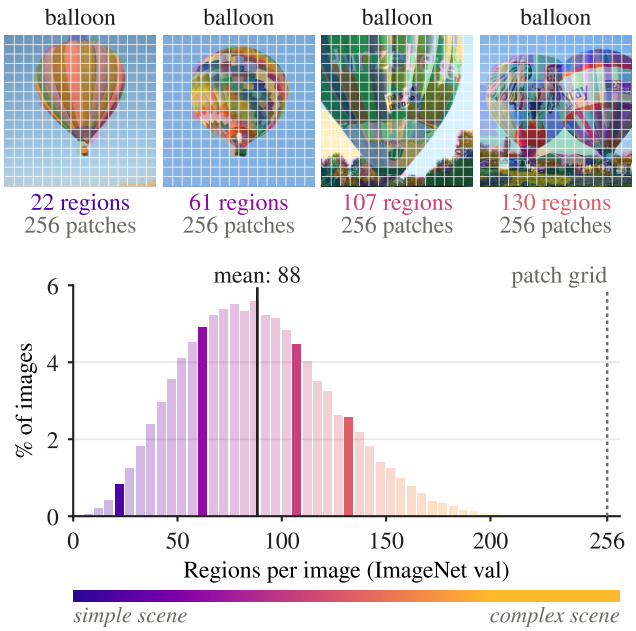

Figure 3. **Image complexity varies, but patch tokenizers spend the same budget on every image.** *Top:* ImageNet images of increasing complexity, overlaid with the 16×16 patch grid of patch-based latent diffusion models and with their region partitions. The simplest of the four scenes has only 22 regions, while the busiest has 130. *Bottom:* Regions per image across ImageNet validation set. The mean of 88 is 2.9× fewer than the 256 tokens of a patch grid (dashed line).

3. RenGen

Images vary widely in complexity, yet patch-based latent generative models represent every image with the same number of tokens (Fig. 3). RenGen instead uses a variable-size latent representation that adapts to image content. RenGen partitions each image into N regions and represents each with a region token obtained by pooling frozen DINOv3 features within that region (Sec. 3.1). A decoder maps these tokens back to pixels through a grid of patch queries that cross-attend to the region tokens (Sec. 3.2). For generation, we place P anchor points on the image canvas and use a DiT to generate one region token per anchor, which the decoder renders into an image (Sec. 3.4). The number of anchors P sets both the complexity of the generated image and the generator's compute. Fig. 2 shows an overview of RenGen.

3.1. Region Encoder

Content-adaptive partitioning. We partition each image into regions using the Felzenszwalb–Huttenlocher (FH) algorithm [11]. FH greedily merges pixels into segments whose internal color variation is small relative to the differences across their boundaries. We run FH with scale

k=100, Gaussian smoothing σ=0.8, and a minimum segment size of 32 pixels. The resulting segments are then aligned to the backbone's patch grid. Each patch is owned by the smallest segment covering at least 25% of its area, or otherwise by the segment covering the largest patch area. Any segment that owns no patches is merged into the owners of the patches it overlaps. The resulting regions average 88 per image on the ImageNet validation set, with the 5th and 95th percentiles at 34 and 151, respectively. For each region, we store a coverage mask m_i that records the fraction of each patch the region covers.

Pooling patch features into region tokens. We use a lightweight cross-attention module to pool frozen DINOv3 [39] patch features within each region. Following RAEv2 [40], we use DINOv3 features from seven layers, sampled at every other layer starting from the final one. We add random Fourier positional embeddings to these features before pooling, so that each region token records both what its region contains and where it lies. The cross-attention query for each region starts as the mean of its patch features weighted by the coverage mask m_i and is refined by L_e=4 cross-attention blocks. Attention logits are biased by $\log m_i$, which restricts each query to its region and down-weights partially covered patches. We ablate different ways of encoding position in the token in Sec. 4.2.

3.2. Region Decoder

Query grid. Unlike patch tokens, region tokens vary in number and have no fixed location in the output image, yet they must be decoded into an image of a specified resolution. To do this, the decoder places one query on each 16×16 output patch, with each query reading from the region tokens through cross-attention. A G×G query grid therefore produces a $16G$×$16G$ image, making the output resolution independent of the number of region tokens. Each query is initialized with Fourier features of its patch position, averaged over the patch area to suppress frequencies finer than the patch grid.

Decoding by cross-attention. The decoder consists of L_d transformer layers (L_d=6 for ViT-B and 14 for ViT-XL). Each layer applies cross-attention from the queries to the region tokens, followed by self-attention among the queries, with an MLP after each attention operation. Following DETR [5], we add each query's Fourier features before every attention operation. These features share their random directions with the Fourier embeddings added to the patch features in the encoder, which helps cross-attention associate each query with the regions covering its position. Finally, a linear head maps each query to the RGB values of its corresponding output patch.

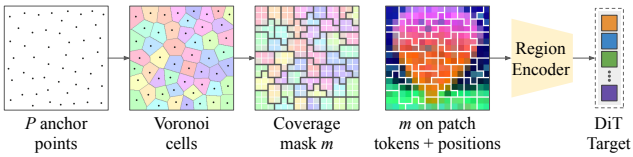

Figure 4. **Constructing DiT targets.** During training, we place P anchor points on the image canvas, partition the canvas into their Voronoi cells, compute each cell's coverage mask on the patch grid, and encode the image with the region encoder to obtain one target region token per anchor.

3.3. Autoencoder Training

We train the encoder's pooling module end-to-end with the decoder on the reconstruction objective. The DINOv3 backbone is kept frozen. Given an image x and its reconstruction $\hat{x}$, we minimize

$$\mathcal{L} = \|x - \hat{x}\|_1 + \lambda_{\text{p}}\,\mathcal{L}_{\text{LPIPS}}(x, \hat{x}) + \lambda_{\text{adv}}\,\mathcal{L}_{\text{adv}}(\hat{x}), \qquad (1)$$

where $\mathcal{L}_{\text{LPIPS}}$ is the perceptual loss [50] and $\mathcal{L}_{\text{adv}}$ is the adversarial loss. The training recipe follows RAEv2 [40], with details in Appendix.

3.4. Point-Anchored DiT

Unlike patch tokens, region tokens have no fixed number, positions, sizes, or shapes, so a generator must decide both the *spatial layout* (the position, shape, and size of each region) and the *content* (the semantics and appearance of each region). We explore several ways of generating layout and content (Sec. 4.3) and find that conditioning on real layouts requires training images at sampling time and leads to memorization, generating layouts jointly with content underperforms, and generating them with a separate DiT adds a second sequential stage that erodes the efficiency gained from generating fewer tokens. We instead fix only the region positions before generation, as described next.

Placing anchor points. We fix a coarse layout by placing P image-independent anchor points on the image canvas, sampled uniformly and spread evenly with three iterations of Lloyd's algorithm [29]. During training, these anchors define the DiT targets (Fig. 4). Concretely, we (1) set P to the image's FH region count randomly varied by $\pm 20\%$ and clipped to $[16, 196]$; (2) place P anchor points on the image canvas and partition it into their Voronoi cells; (3) compute a coverage mask for each cell on the backbone's patch grid; and (4) encode the image with the region encoder, yielding one region token per anchor as the DiT's target. During sampling, we set P to the FH region count of a random training image of the target class (88 on average), or choose it manually to trade scene complexity for compute.

Denoising region tokens at anchor points. We use a DiT with a DDT head following RAE [53]. Its input is the set of P noisy tokens, with no patch grid. Each token's

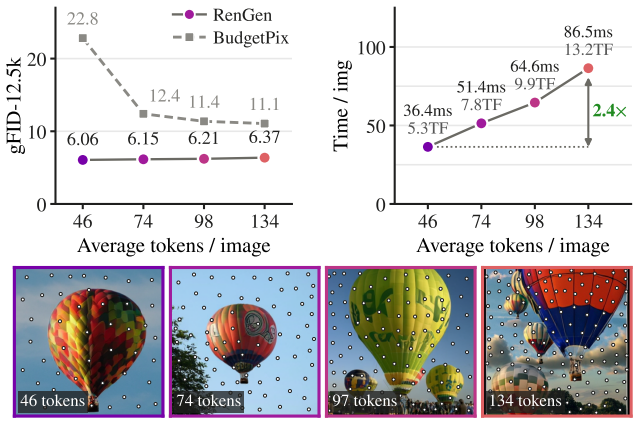

Figure 5. **RenGen adapts its compute to scene complexity without losing quality.** gFID stays flat across four complexity buckets, so in low-resource settings RenGen can generate simple, realistic scenes 2.4× faster and with 2.5× less compute than complex ones. Samples of the same class (bottom; anchor points in white) grow busier as the token budget increases.

only spatial input is its anchor point, which we use as 2D RoPE [14] in every attention layer. The timestep, the class label, and the token count P enter as additional in-context tokens. During sampling, we integrate the ODE with $T{=}50$ Euler steps under the timestep shift described below, using internal guidance [54] from an auxiliary head on an intermediate block. The region decoder (Sec. 3.2) renders the image from the denoised region tokens.

Training. We normalize the target tokens per channel and train with flow matching [27] on the interpolant $x_t = (1-t)\,x + t\,\epsilon$, predicting x and computing the loss on the implied velocity. To batch token sets of different sizes, we pad them to a common length, mask the padded tokens in attention, and average the loss over real tokens only. We drop the class label with probability 0.1 to enable classifier-free guidance [16]. We sample t from a logit-normal distribution and apply the resolution-dependent shift $t \mapsto st/(1+(s-1)t)$ with $s{=}8$.

4. Experiments

We first show that RenGen adapts its compute to scene complexity while maintaining image quality (Sec. 4.1). We then ablate how region tokens encode spatial layout (Sec. 4.2) and how region layout and content are generated (Sec. 4.3). Next, we compare RenGen with representation autoencoders (RAEs) and fixed-length, variable-length, and content-adaptive tokenizers on reconstruction and generation (Sec. 4.4). Finally, we show that RenGen scales to higher resolutions more efficiently than fixed-grid models (Sec. 4.5). All experiments use ImageNet [7] at 256×256. Implementation details, evaluation metrics, and the timing protocol are provided in the Appendix.

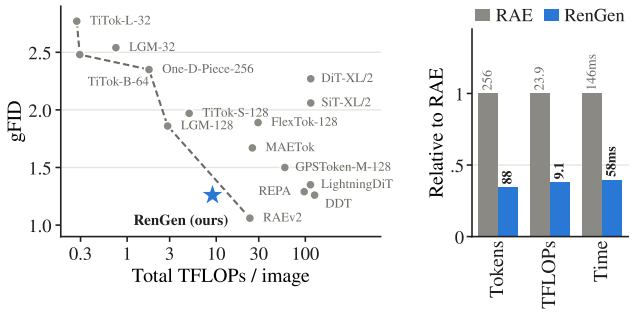

Figure 6. **RenGen sets a new quality–compute trade-off.** *Left:* gFID versus total TFLOPs per image on ImageNet 256×256. RenGen lies below the Pareto frontier of prior methods (dashed). *Right:* The source of this gain. At matched model size, RenGen generates 2.9× fewer tokens than RAEv2, which translates directly into 2.6× fewer TFLOPs and 2.5× less wall-clock time per image.

4.1. Compute and quality across scene complexity

We show that RenGen allocates compute in proportion to scene complexity without sacrificing image quality. We use FH region count [11] as a proxy for scene complexity because FH places a boundary wherever the color difference across it exceeds the variation within the adjacent segments, so a single object on a plain background yields few regions, while a cluttered scene yields many.

We split the 50k ImageNet validation images into four equal-sized buckets by FH region count [11]. For each bucket, RenGen generates 12.5k images, drawing class labels from the bucket's class distribution and P from its region-count distribution. We compute gFID against the corresponding real-image bucket, ensuring that generated images are evaluated against scenes of similar complexity (Fig. 5). From the most to the least complex bucket, the average token count drops from 134 to 46, reducing TFLOPs by 2.5× and wall-clock time per image by 2.4×. Despite this reduction, gFID remains nearly constant, ranging from 6.37 to 6.06. Thus, RenGen generates simpler scenes 2.4× faster than complex ones at comparable image quality. In contrast, BudgetPix [18], evaluated under the same protocol with matched token counts, sees its gFID degrade from 11.06 to 22.82 as its budget shrinks, indicating that it trades image quality, rather than scene complexity, for compute.

We also verify that the generated images reflect the complexity of their target buckets. The samples shown in Fig. 5 illustrate this trend: scenes become progressively busier as the token count increases, while remaining realistic. Quantitatively, the mean FH region count of generated images is within 8% of that of real images in every bucket, and within 3% in all but the busiest bucket. It also correlates strongly with P (Spearman $\rho = 0.94$), confirming that the number of anchor points largely determines the complexity of the generated scene.

4.2. How to encode spatial extent in region tokens?

A region token must carry both *what* a region contains and *where* it is, so the decoder can reconstruct the image by placing its content at the correct spatial location. We compare region tokens from a leading region encoder, REN [19], with three ways of constructing them ourselves (Tab. 1).

REN tokens. Training a decoder on frozen REN tokens yields poor reconstructions, which we attribute to two properties of these tokens: (1) REN adds no positional encoding to the patch features it pools, so each token captures its region's extent only weakly via the image encoder's position embedding; and (2) its regions often overlap, so pixels are ambiguously assigned across tokens. Fine-tuning REN's cross-attention jointly with the decoder improves reconstruction only slightly, but the regions lose their spatial coherence and the number of tokens grows sharply during training.

Pooling patch features within FH segments. To fix the region count and obtain non-overlapping tokens, we replace REN's learned regions with Felzenszwalb–Huttenlocher (FH) segments [11], following RVLM [21], since FH is fast and covers every pixel with exactly one region. We pool DINOv3 features within each segment using the cross-attention module of Sec. 3.1, without positional embeddings. This resolves REN's overlap but not its weak position signal, because DINOv3 encodes position only through rotary embeddings in attention [14, 39], leaving its patch features with little absolute position information. Reconstruction quality therefore still lags behind RAEv2 (Tab. 1).

Explicit mask encoding. To strengthen the position signal, we add a learned linear projection of each region's mask to its pooled token. This closes most of the reconstruction gap to RAEv2, but generation suffers (Tab. 1). The decoder learns to read each region's exact extent from the mask code, which simplifies reconstruction but shifts the burden to the DiT, since it must now generate these mask codes in addition to each region's content. We find that the DiT struggles to generate mask codes precise enough for the decoder to place regions correctly, so small errors in the codes become visible errors in the generated layout (Sec. 4.3).

Pooling patch features with embedded position (ours). Instead of a separate mask code, we add random Fourier positional embeddings to the patch features before pooling, so that each region token encodes where its region lies together with what it contains. The decoder queries carry Fourier features of the same random directions, so cross-attention can match each query to the regions covering its position rather than reading their extent from an explicit code. This matches the reconstruction quality of explicit mask encoding without extra parameters. Because position is now part of the token content, the DiT no longer has to generate a separate, precise layout code, and it generates substantially better images.

Region tokens	Regions	Tokens	rFID↓	gFID↓
RAEv2 (for reference)	n/a	256	0.30	1.06
REN, frozen [19]	Learned	77	0.86	–
REN, fine-tuned [19]	Learned	77→185*	0.47	–
Pooling features without position	FH	88	0.51	–
Explicit mask encoding	FH	88	**0.32**	3.10
Pooling patch features with position	FH	88	**0.32**	1.34

Table 1. **Adding positional embeddings to patch features before pooling them into region tokens gives the best reconstruction and generation.** REN tokens reconstruct poorly because their regions overlap and their features encode little position. Fine-tuning REN makes the number of tokens grow during training (*mean token count at the start→end of training). Pooling patch features within fixed FH regions keeps the token count stable, but the tokens still encode little position. Adding an explicit mask code to each token improves reconstruction but hurts generation, because the DiT must generate these codes precisely.

4.3. How to generate without implicit spatial prior?

Patch tokens occupy fixed grid positions, given to the DiT through 2D RoPE. Region tokens, in contrast, vary in position, shape, and size from image to image, so the DiT must generate both *what* each region contains and *where* it lies. We compare multiple ways of generating region tokens, from treating them as an unordered set to anchoring them at random points, as discussed below (Tab. 2).

Generating region tokens as an unordered set. In the simplest design, the DiT receives no explicit spatial prior and generates region tokens that jointly encode content and position. Since the DiT must denoise both where each region belongs and what it contains, regions can shift across denoising steps, disrupting the generation of consistent content at each location.

Conditioning on a given layout. We provide the DiT with a layout and train it to generate only the pooled DINOv3 features. Concretely, for each training image, we use its FH segmentation to obtain region centers for 2D RoPE and mask embeddings that are added to the tokens being denoised. This follows the explicit mask encoding design discussed in Sec. 4.2. At sampling time, borrowing the layout of a random training image produces remarkably good results, even outperforming RAEv2. However, layouts borrowed from validation images cause a sharp drop in quality, indicating that the DiT memorizes training layouts rather than generalizing to new ones. The approach therefore does not yield a standalone generator, since it requires training-set layouts at sampling time.

Learning to generate layouts. To remove the dependence on borrowed layouts, we split generation across two DiTs.

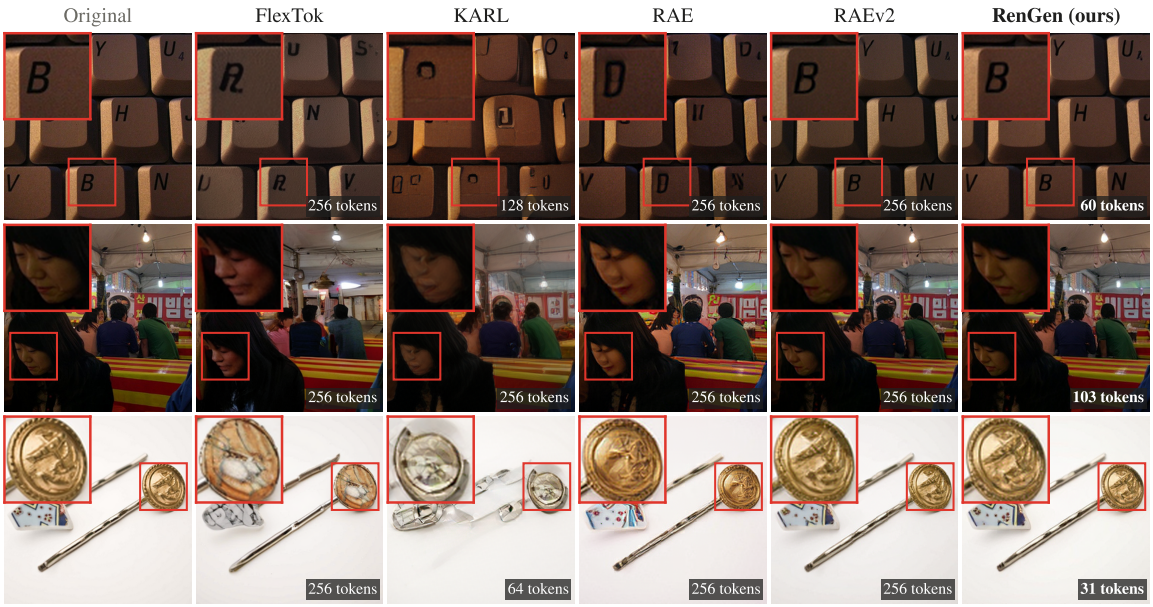

Figure 7. **RenGen reconstructs fine detail as faithfully as RAEv2 with far fewer tokens.** Each tile shows its token count; RenGen's follows image complexity, from 31 tokens for the hairpin on a plain background to 103 for the crowded scene.

The first DiT generates layout for the image, and the second generates the region tokens conditioned on the generated layouts. This works, but the two sequential stages nearly eliminates the efficiency advantage of RenGen over RAEv2, even though RenGen generates 2.9× fewer tokens. To avoid the two-stage setup, we also try a single DiT that generates the layout and region tokens jointly, conditioning each step on the layout predicted in the previous step, but this still does not work well.

Random anchor points (ours). When pooling patch features with embedded positions (Sec. 4.2), we find that although the autoencoder is trained only on FH segments, it reconstructs nearly as well from other partitions: FH segments yield an rFID of 0.320, compared with 0.332 for Voronoi cells and 0.323 for SLIC-style superpixels, both computed from randomly sampled points. We therefore need not predict or borrow an FH layout for generation, and can instead define the region layout from random anchor points. For each training image, we (1) sample P points on the canvas, with P drawn uniformly within $\pm 20\%$ of the image's FH segment count; (2) partition the image into their Voronoi cells or SLIC-style superpixels; (3) encode the image under this partition to get the target tokens for the DiT (Fig. 4); and (4) use the anchor positions as 2D RoPE for the DiT. At sampling time, we draw fresh random anchors, with P chosen by the user or drawn from the class's token counts, and the DiT generates region tokens that carry both content and position. This requires no layout model, no training images, and no layout guidance. The DiT thus retains the spatial prior that the unordered set lacks, without having to generate a layout explicitly.

Spatial input at sampling	**Training regions**	**Sampling time/img (ms)**	**gFID↓**
none (unordered set)	FH	57.3	6.32
FH layout of a train image (memorized)	FH	57.9	1.02
FH layout of a val image	FH	58.0	3.60
layout generated by a separate DiT	FH	128.0	2.47
layout generated jointly	FH	135.9	2.29
random anchor points	SLIC	57.9	1.36
random anchor points (ours)	Voronoi	57.9	1.34

Table 2. **Randomly fixing region positions with anchor points enables strong generation at low cost.** Without any spatial prior, the DiT trains poorly. Conditioning on real layouts separately leads to memorization and requires real images at sampling time, while generating layouts roughly doubles sampling time and still degrades quality. Random anchor points avoid both and outperform held-out real layouts with a single DiT. At sampling, the DiT receives only the points; the training regions determine how the targets are partitioned around them. Sampling time is measured per image at batch size 64 on one NVIDIA GH200.

4.4. Comparison with latent generative models

We show that RenGen reconstructs and generates images on par with leading latent generative models at substantially lower compute and latency. Fig. 6 relates generation quality to compute, and Tab. 3 compares reconstruction and generation in detail against (1) representation autoencoders (RAEs) that use a pretrained DINOv3 encoder, (2) compact tokenizers that encode every image into a fixed number of tokens below the 16×16 grid, (3) variable-length tokenizers

Method	Content adaptive	Controllable complexity	Frozen enc.	Reconstruction		Generation	
				rFID↓	PSNR↑	gFID↓	IS↑
Compact fixed-length tokenizers							
TiTok [46]	✗	✗	✗	1.71	17.73	1.97	281.8
MAETok [6]	✗	✗	✗	0.48	**23.61**	1.67	**311.2**
Variable-length tokenizers							
FlexTok [1]	✗	✓	✗	1.20	15.90	1.89	275.6
One-D-Piece [31]	✗	✓	✗	1.77	17.07	–	–
AdaTok [30]	✗	✓	✗	1.63	17.92	–	–
GPSToken [51]	✗	✓	✗	0.65	24.06	1.50	–
LGM [24]	✗	✓	✗	0.59	22.80	1.86	290.7
Content-adaptive tokenization							
ALIT [9]	✓	✓	✗	3.18	18.55	–	–
KARL [8]	✓	✓	✗	4.05	20.78	–	–
AdaTok [30]	✓	✗	✗	1.50	17.78	–	–
BudgetPix§ [18]	✓	✓	✗	–	–	4.75	284.0
Representation autoencoders							
RAEv2 [40]	✗	✗	✓	**0.30**	22.58	**1.06**	255.3
RenGen	✓	✓	✓	0.32	22.23	1.34	253.7

Table 3. **RenGen achieves the best rFID and gFID among methods that vary their token count.** It also reconstructs and generates nearly as well as RAEv2 with 2.6× fewer TFLOPs and 2.5× less wall-clock time (Fig. 6). The checkmark columns indicate whether a method's token count varies with image content (*content adaptive*), whether a user can choose the token budget of a single model at inference (*controllable complexity*), and whether its latents come from a frozen pretrained representation encoder (*frozen enc.*). §Included as the closest prior generator that adapts its compute to image content; it operates in pixel space, so reconstruction metrics do not apply.

that support several token budgets, and (4) content-adaptive tokenizers whose token count varies per image.

Reconstruction. RenGen reconstructs nearly as well as RAEv2 (rFID 0.32 vs. 0.30 with DINOv3-L) while using 2.9× fewer tokens. Among tokenizers that vary their token count, it achieves the lowest rFID, 4.7× lower than AdaTok (1.50) and 1.8× lower than LGM (0.59). RenGen's PSNR is lower than that of tokenizers trained end-to-end for pixel reconstruction, such as MAETok (22.2 vs. 23.6). Qualitatively, however, RenGen preserves fine structure as well as RAEv2 with far fewer tokens (Fig. 7).

Generation quality and cost. RenGen reaches a gFID of 1.34 with 88 tokens on average, 0.28 behind RAEv2 (1.06), and is the strongest model among those that vary their token count. Following RAEv2 [40], we sample with internal guidance (scale 2.0 on $t \in [0.6, 1]$, 50 Euler steps, no CFG). Fig. 6 places this result against compute. RenGen lies below the Pareto frontier of prior methods, reaching its gFID at 9.1 TFLOPs per image, while RAEv2 needs 23.9. The right panel shows the source of this gain. At matched model size, RenGen generates 2.9× fewer tokens than RAEv2, which yields 2.6× fewer TFLOPs and 2.5× less wall-clock time per image. These savings come entirely from the shorter token sequence.

Method	Encoded resolution	Decoded resolution	Tokens	DiT GFLOPs↓	rFID↓
RAEv2 (for reference)	256	256	256	135.2	0.30
RAEv2	256	512	256	135.2	0.50
RenGen (for reference)	256	256	88	47.5	0.32
RenGen	256	512	**88**	**47.5**	0.48
RenGen	512	512	**88**	**47.5**	**0.39**

Table 4. **RenGen scales to higher resolutions more efficiently than RAEv2.** At 512×512, RenGen reconstructs slightly better than RAEv2 while using about a third of the tokens and DiT compute. Because its token count depends on image content rather than resolution, RenGen can also encode the full 512×512 image into the same 88 tokens, improving reconstruction further at no extra cost. A single RenGen model handles both 256×256 and 512×512 inputs here.

4.5. Efficient scaling to higher resolutions

Being a region-based model, RenGen encodes a 512×512 image into the same number of tokens as a 256×256 image, so the DiT cost of generating region latents stays the same across resolutions. Only the decoder's cost grows with resolution, and it only runs once per image rather than at every denoising step. In contrast, for patch-based models like RAEs, doubling the image side quadruples the number of tokens and increases the DiT's attention cost by 16×. To scale efficiently to higher resolutions, RAE trains a decoder with twice the encoder's patch size (32 vs. 16), so that a 256×256 image is encoded into 256 tokens and decoded at 512×512 [53]. As Tab. 4 shows, RAE's approach is both less efficient and less effective than RenGen. Moreover, since RenGen's decoder queries are 2D Fourier positional embeddings of the output patch grid, a single model can render images at multiple resolutions (here, 256×256 and 512×512) by varying only the number and frequencies of the queries. Patch-based models, in contrast, must retrain (at least the decoder) for each new resolution. We leave a fuller study of multi-resolution decoding and generation to future work.

5. Conclusion

We introduce RenGen, a latent diffusion model built on region tokens. We demonstrate that RenGen is content adaptive allowing control over scene complexity. On ImageNet 256×256, RenGen reconstructs images with rFID similar to state-of-the-art representation autoencoders using 2.9× less tokens and generates a gFID of 1.34 with 2.5× lower latency. For future work, we plan to explore text-to-image training, more efficient training, and decoding at more resolutions and aspect ratios from a single model. Region-based generation also enables finer control, such as editing the layout of a scene or resizing and rotating individual objects.

LLM Usage Statement

We used generative AI tools, such as LLMs, for agentic coding (Claude), paper writing, paper editing, assist with translation, creating figures, and search for relevant related work. We have not used generative AI to develop theoretical model, formulate math claims, proofs, create hypotheses, design experiments or provide feedback on research methodology. We have reviewed all AI-assisted work for correctness.

References

[1] Roman Bachmann, Jesse Allardice, David Mizrahi, Enrico Fini, Oğuzhan Fatih Kar, Elmira Amirloo, Alaaeldin El-Nouby, Amir Zamir, and Afshin Dehghan. FlexTok: Resampling images into 1D token sequences of flexible length. In *ICML*, 2025. 2, 8

[2] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *ICCV*, 2021. 17

[3] Black Forest Labs. FLUX. https://github.com/black-forest-labs/flux, 2024. 3

[4] Daniel Bolya and Judy Hoffman. Token merging for fast stable diffusion. In *CVPRW*, 2023. 3

[5] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *ECCV*, 2020. 4

[6] Hao Chen, Yujin Han, Fangyi Chen, Xiang Li, Yidong Wang, Jindong Wang, Ze Wang, Zicheng Liu, Difan Zou, and Bhiksha Raj. Masked autoencoders are effective tokenizers for diffusion models. In *ICML*, 2025. 3, 8

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009. 5

[8] Shivam Duggal, Sanghyun Byun, William T. Freeman, Antonio Torralba, and Phillip Isola. Single-pass adaptive image tokenization for minimum program search. In *NeurIPS*, 2025. 8

[9] Shivam Duggal, Phillip Isola, Antonio Torralba, and William T. Freeman. Adaptive length image tokenization via recurrent allocation. In *ICLR*, 2025. 8

[10] Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis. In *CVPR*, 2021. 17

[11] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *IJCV*, 2004. 3, 5, 6

[12] Qiushan Guo, Shalini De Mello, Hongxu Yin, Wonmin Byeon, Ka Chun Cheung, Yizhou Yu, Ping Luo, and Sifei Liu. RegionGPT: Towards region understanding vision language model. In *CVPR*, 2024. 3

[13] Yoav HaCohen, Nisan Chiprut, Benny Brazowski, Daniel Shalem, Dudu Moshe, Eitan Richardson, Eran Levin, Guy Shiran, Nir Zabari, Ori Gordon, Poriya Panet, Sapir Weissbuch, Victor Kulikov, Yaki Bitterman, Zeev Melumian, and Ofir Bibi. LTX-Video: Realtime video latent diffusion. *ArXiv*, 2024. 3

[14] Byeongho Heo, Song Park, Dongyoon Han, and Sangdoo Yun. Rotary position embedding for vision transformer. In *ECCV*, 2024. 5, 6

[15] Amir Hertz, Ron Mokady, Jay Tenenbaum, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. Prompt-to-prompt image editing with cross-attention control. In *ICLR*, 2023. 11

[16] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance. In *NeurIPSW*, 2021. 5

[17] Jindong Jiang, Fei Deng, Gautam Singh, and Sungjin Ahn. Object-centric slot diffusion. In *NeurIPS*, 2023. 3

[18] Ozgur Kara, Yujia Chen, Daniel Watson, David Forsyth, James Matthew Rehg, Wen-Sheng Chu, and Du Tran. BudgetPix: Compute-adaptive tokenization for pixel-space image diffusion. *ArXiv*, 2026. 2, 3, 5, 8

[19] Savya Khosla, Sethuraman T V, Barnett Lee, Alexander Schwing, and Derek Hoiem. REN: Fast and efficient region encodings from patch-based image encoders. In *NeurIPS*, 2025. 1, 2, 3, 6

[20] Savya Khosla, Sethuraman T V, Alexander Schwing, and Derek Hoiem. RELOCATE: A simple training-free baseline for visual query localization using region-based representations. In *CVPR*, 2025. 3

[21] Savya Khosla, Chris Dongjoo Kim, Alexander Schwing, Derek Hoiem, Christopher Clark, and Sangho Lee. RVLM: Fewer tokens, better resolution scaling, fine-grained spatial grounding. *ArXiv*, 2026. 3, 6

[22] Savya Khosla, Sethuraman T V, Aryan Chadha, Alexander Schwing, and Derek Hoiem. T-REN: Learning text-aligned region tokens improves dense vision-language alignment and scalability. In *ECCV*, 2026. 1, 2, 3

[23] Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *ICLR*, 2014. 3

[24] Dong Hoon Lee and Seunghoon Hong. Variable-length tokenization via learnable global merging for diffusion transformers. *ArXiv*, 2026. 8

[25] Tianhong Li and Kaiming He. Back to basics: Let denoising generative models denoise. In *CVPR*, 2026. 2

[26] Feng Liang, Bichen Wu, Xiaoliang Dai, Kunpeng Li, Yinan Zhao, Hang Zhang, Peizhao Zhang, Peter Vajda, and Diana Marculescu. Open-vocabulary semantic segmentation with mask-adapted CLIP. In *CVPR*, 2023. 3

[27] Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matthew Le. Flow matching for generative modeling. In *ICLR*, 2023. 5

[28] Yang Liu, Muzhi Zhu, Hengtao Li, Hao Chen, Xinlong Wang, and Chunhua Shen. Matcher: Segment anything with one shot using all-purpose feature matching. In *ICLR*, 2024. 3

[29] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Theory*, 1982. 4, 17

[30] Xiaocheng Lu, Yuxi Chen, Jie Zhang, Jian Liu, Jingcai Guo, Fangqi Zhu, Tao Han, and Song Guo. AdaTok: Self-budgeting image tokenization with quality-preserving dynamic tokens. *ArXiv*, 2026. 8

[31] Keita Miwa, Kento Sasaki, Hidehisa Arai, Tsubasa Takahashi, and Yu Yamaguchi. One-D-Piece: Image tokenizer meets quality-controllable compression. *ArXiv*, 2025. 2, 8

[32] Ron Mokady, Amir Hertz, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. Null-text inversion for editing real images using guided diffusion models. In *CVPR*, 2023. 11

[33] William Peebles and Saining Xie. Scalable diffusion models with transformers. In *ICCV*, 2023. 2

[34] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. SDXL: Improving latent diffusion models for high-resolution image synthesis. In *ICLR*, 2024. 3

[35] Hanoona Rasheed, Muhammad Maaz, Sahal Shaji, Abdelrahman Shaker, Salman Khan, Hisham Cholakkal, Rao M. Anwer, Eric Xing, Ming-Hsuan Yang, and Fahad S. Khan. GLaMM: Pixel grounding large multimodal model. In *CVPR*, 2024. 3

[36] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *CVPR*, 2022. 2

[37] Junhong Shen, Kushal Tirumala, Michihiro Yasunaga, Ishan Misra, Luke Zettlemoyer, Lili Yu, and Chunting Zhou. CAT: Content-adaptive image tokenization. In *NeurIPS*, 2025. 2

[38] Michal Shlapentokh-Rothman, Ansel Blume, Yao Xiao, Yuqun Wu, Sethuraman T V, Heyi Tao, Jae Yong Lee, Wilfredo Torres, Yu-Xiong Wang, and Derek Hoiem. Region-based representations revisited. In *CVPR*, 2024. 1, 2, 3

[39] Oriane Siméoni, Huy V. Vo, Maximilian Seitzer, Federico Baldassarre, Maxime Oquab, Cijo Jose, Vasil Khalidov, Marc Szafraniec, Seungeun Yi, Michaël Ramamonjisoa, et al. DINOv3. *ArXiv*, 2025. 2, 4, 6, 16

[40] Jaskirat Singh, Boyang Zheng, Zongze Wu, Richard Zhang, Eli Shechtman, and Saining Xie. Improved baselines with representation autoencoders. In *NeurIPS*, 2026. 3, 4, 8, 16

[41] Ziyi Wu, Jingyu Hu, Wuyue Lu, Igor Gilitschenski, and Animesh Garg. SlotDiffusion: Object-centric generative modeling with diffusion models. In *NeurIPS*, 2023. 3

[42] Yao Xiao, Qiqian Fu, Heyi Tao, Yuqun Wu, Zhen Zhu, and Derek Hoiem. TextRegion: Text-aligned region tokens from frozen image-text models. *TMLR*, 2025. 3

[43] Jiarui Xu, Sifei Liu, Arash Vahdat, Wonmin Byeon, Xiaolong Wang, and Shalini De Mello. Open-vocabulary panoptic segmentation with text-to-image diffusion models. In *CVPR*, 2023. 3

[44] Binxin Yang, Shuyang Gu, Bo Zhang, Ting Zhang, Xuejin Chen, Xiaoyan Sun, Dong Chen, and Fang Wen. Paint by example: Exemplar-based image editing with diffusion models. In *CVPR*, 2023. 11

[45] Zhuoyi Yang, Jiayan Teng, Wendi Zheng, Ming Ding, Shiyu Huang, Jiazheng Xu, Yuanming Yang, Wenyi Hong, Xiaohan Zhang, Guanyu Feng, Da Yin, Yuxuan Zhang, Weihan Wang, Yean Cheng, Bin Xu, Xiaotao Gu, Yuxiao Dong, and Jie Tang. CogVideoX: Text-to-video diffusion models with an expert transformer. In *ICLR*, 2025. 3

[46] Qihang Yu, Mark Weber, Xueqing Deng, Xiaohui Shen, Daniel Cremers, and Liang-Chieh Chen. An image is worth 32 tokens for reconstruction and generation. In *NeurIPS*, 2024. 8

[47] Yuqian Yuan, Wentong Li, Jian Liu, Dongqi Tang, Xinjie Luo, Chi Qin, Lei Zhang, and Jianke Zhu. Osprey: Pixel understanding with visual instruction tuning. In *CVPR*, 2024. 3

[48] Yuqian Yuan, Hang Zhang, Wentong Li, Zesen Cheng, Boqiang Zhang, Long Li, Xin Li, Deli Zhao, Wenqiao Zhang, Yueting Zhuang, Jianke Zhu, and Lidong Bing. VideoRefer suite: Advancing spatial-temporal object understanding with video LLM. In *CVPR*, 2025. 3

[49] Eduard Zamfir, Christian Reisswig, Zongwei Wu, Yongqin Xian, and Radu Timofte. Elastic token compression for pixel-space diffusion transformers. *ArXiv*, 2026. 2, 3

[50] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018. 4

[51] Zhengqiang Zhang, Rongyuan Wu, Lingchen Sun, and Lei Zhang. GPSToken: Gaussian parameterized spatially-adaptive tokenization for image representation and generation. In *NeurIPS*, 2025. 8

[52] Shengyu Zhao, Zhijian Liu, Ji Lin, Jun-Yan Zhu, and Song Han. Differentiable augmentation for data-efficient gan training. *ArXiv*, 2020. 17

[53] Boyang Zheng, Nanye Ma, Shengbang Tong, and Saining Xie. Diffusion transformers with representation autoencoders. In *ICLR*, 2026. 2, 3, 4, 8, 17

[54] Xingyu Zhou, Qifan Li, Xiaobin Hu, Hai Chen, and Shuhang Gu. Guiding a diffusion transformer with the internal dynamics of itself. In *CVPR*, 2026. 5, 17

Appendix

Sec. A shows how RenGen's design gives users direct scene control. Users can add detail to a generated image (Sec. A.1), transplant objects between images (Sec. A.2), move content within an image (Sec. A.3), and inpaint regions (Sec. A.4). Sec. B presents additional ablations. Finally, we provide implementation details for the autoencoder (Sec. C) and the point-anchored DiT (Sec. D), and describe the evaluation and timing protocols (Sec. E).

A. Direct scene control with anchor points

A RenGen image is a set of anchor points paired with region tokens, so users can control an image directly by changing this set. Adding anchor points adds detail (Fig. A2), replacing tokens transplants objects (Fig. A3), moving points moves content (Figs. A4 and A5), regenerating tokens from noise inpaints missing regions (Fig. A6), and interpolating tokens blends two objects (Fig. A7). All controls use the trained model as is, with no extra training, pixel masks, attention manipulation, or inversion. **These controls follow from two properties that patch-based generators lack: (1) token positions are inputs to the generator rather than fixed grid indices, and (2) the number of tokens is not fixed by the architecture.** Notably, generating an image and then editing it with RenGen is faster than just generating an image with RAEv2 (Fig. A1). We demonstrate these controls qualitatively and leave their systematic evaluation, and extensions such as text-guided control, to future work.

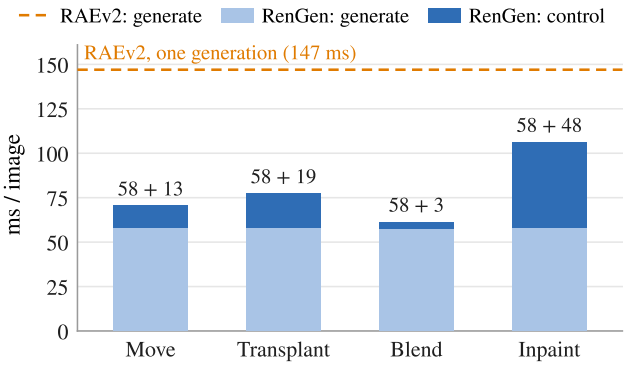

Figure A1. **Generating an image and then editing it with RenGen takes less time than a single RAEv2 generation.** Wall-clock time per image (batch 64, bf16, one GH200). Each bar stacks a RenGen generation (light, 50 steps, 88 anchor points) and one control (dark). Moving and transplanting run 10 denoising steps, blending only encodes and decodes, and inpainting regenerates the masked tokens over 50 steps. The dashed line marks one RAEv2 generation (256 tokens, 50 steps, official guidance).

A.1. Preview cheaply, then refine

Users can sample many cheap previews and pay for full detail only on the one they choose (Fig. A2). We generate images with only 32 tokens, then add anchor points to a chosen image while keeping its existing points fixed. Each new token starts as a copy of its nearest existing token, and we re-noise all tokens to $t_0 \approx 0.9$ before finishing sampling with up to 128–148 tokens. The composition, pose, and palette of the preview stay fixed while detail accumulates; e.g., more spots on the ladybug, more feathers on the birds, and more toppings on the pizza. To preview cheaply, patch-based models generate at a lower resolution since their token count is tied to image size, and refining the preview requires a model trained to generate across resolutions. RenGen instead previews at full resolution and refines by adding tokens.

A.2. Transplanting objects between real images

Users can move an object from one real photo into another (Fig. A3). We encode both images at the same 128 anchor points, after aligning the source object's bounding box with the destination's. We then replace the destination tokens inside the object with the source tokens, re-noise to t_0=0.67, and denoise with the source class. The transplanted object adopts the destination's lighting and context. For example, in Fig. A3, the apple gains a shadow on the dark backdrop, the robin perches on the lupins, and the cauliflower sits among cabbage leaves. The edited region is simply a set of tokens selected by their anchor points, so no pixel mask is needed, and real images need no inversion because the encoder gives their tokens directly. Comparable edits with patch-based diffusion models typically rely on inversion [32] and on pixel masks [44] or attention manipulation [15].

A.3. Moving content

Users can move a scene by moving its anchor points. In Fig. A4, we translate every point by the same offset and re-noise to t_0=0.67, so only the last 10 of 50 sampling steps run. The object keeps its identity, pose, and colors, and the generator fills in the strip uncovered at the trailing edge. Because anchor points are continuous, the offset need not align with any grid. As shown in Fig. A5, steps of 5% of the image width (12.8 px, less than one 16 px patch) move the content smoothly. In a patch-based model, token positions are fixed by the grid, so content can at best be shifted by whole patches, and the uncovered strip must be filled separately.

A.4. Inpainting

Users can replace any part of a real image with new content, including an object of a different class, using the same

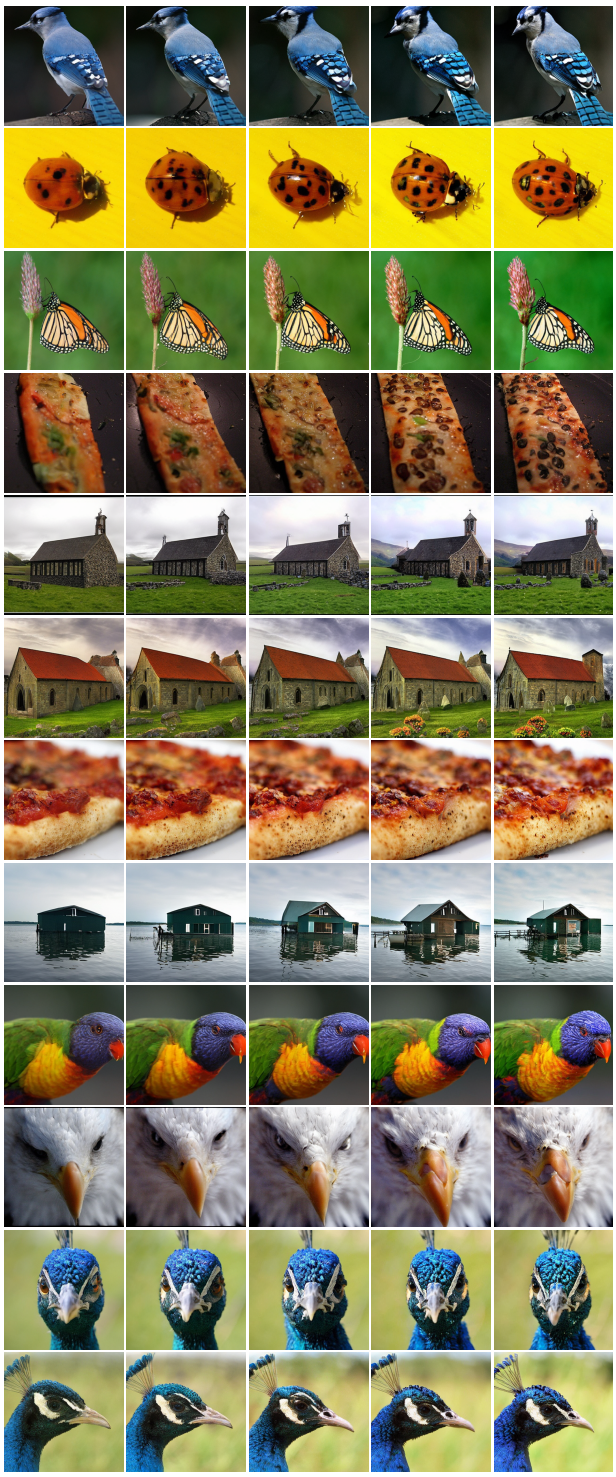

Figure A2. **Users can preview layouts cheaply with few tokens and then add anchor points to refine the chosen one.** The left column shows images generated with only 32 tokens, cheap enough to sample many candidate layouts. Adding anchor points to a chosen image keeps its layout and adds fine detail.

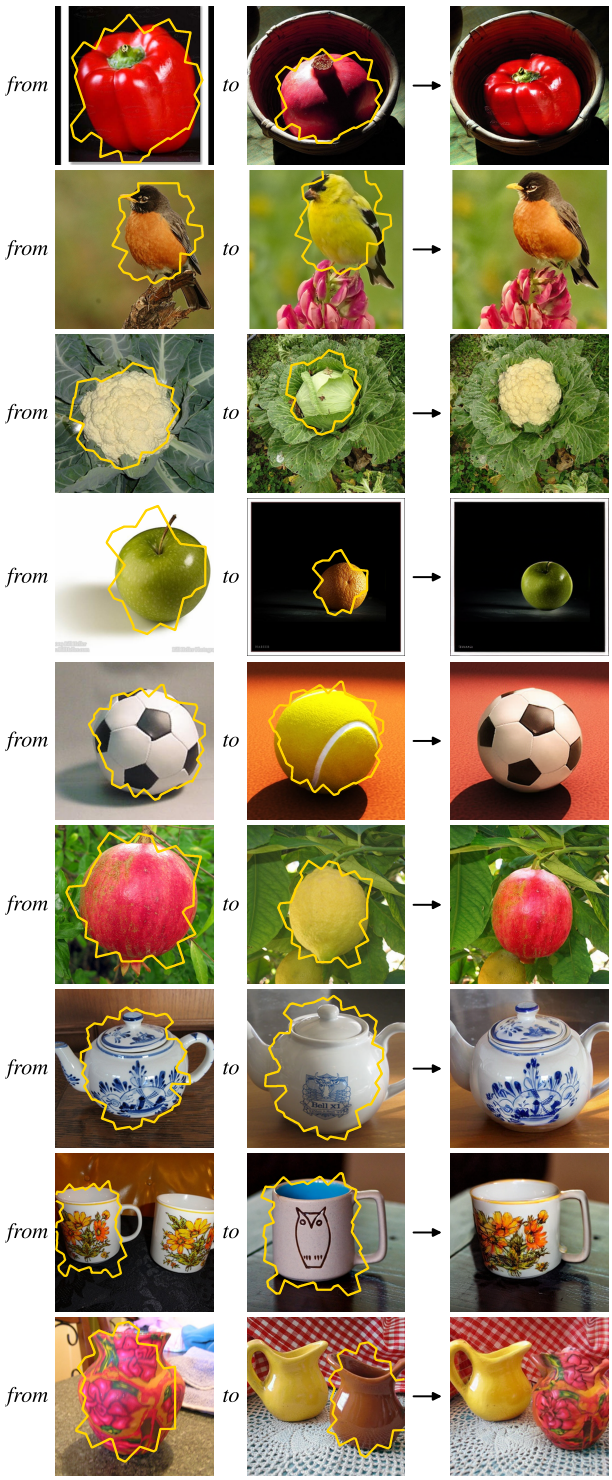

Figure A3. **Swapping region tokens between images lets users transplant objects, training-free.** The tokens inside the outlined region of the destination are replaced with the source's tokens at those points. The transplanted object adopts the destination's lighting, shadows, and surroundings.

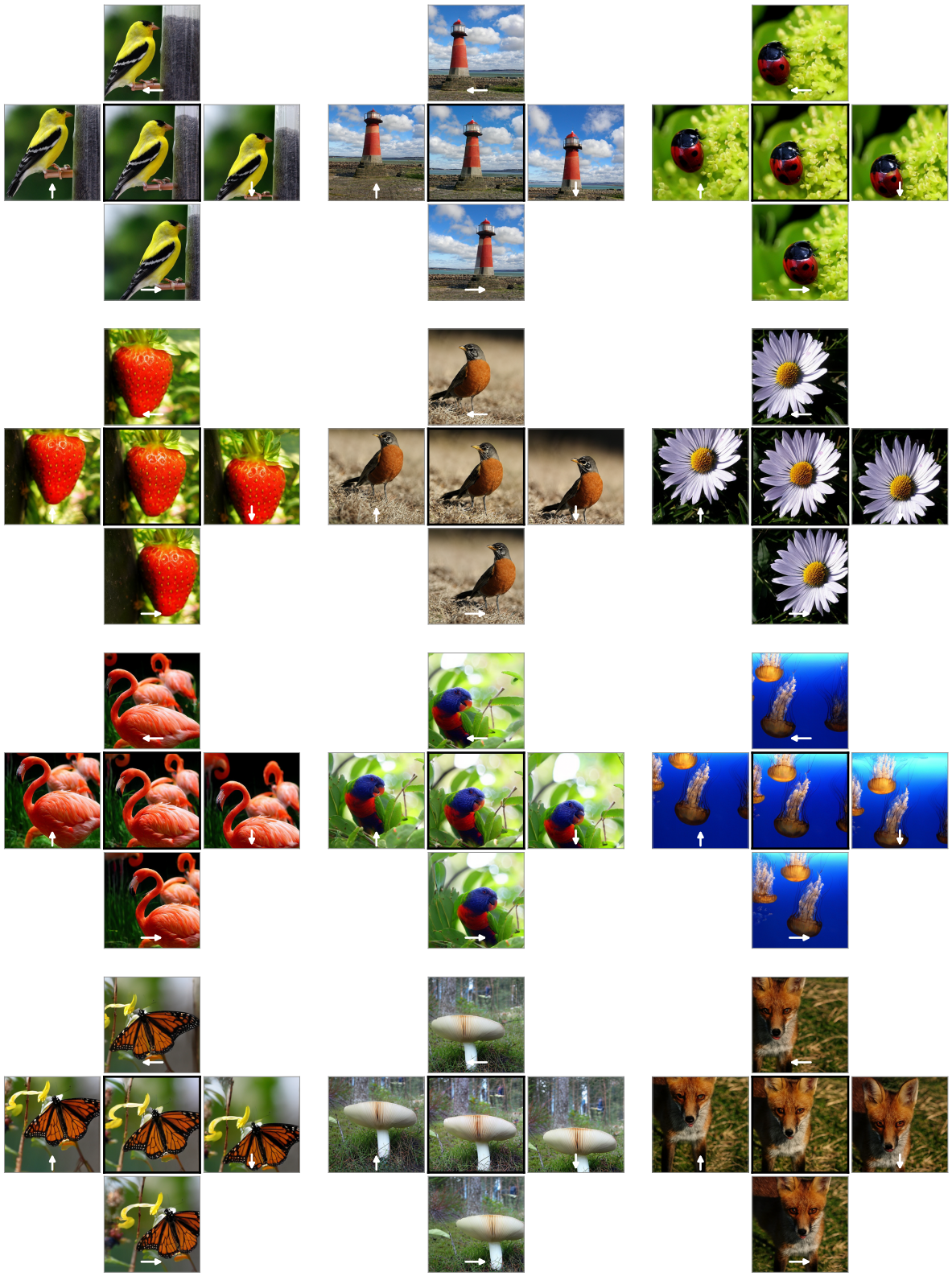

Figure A4. **Moving the anchor points lets users reposition the scene, training-free.** Each panel's center shows a generated image. Shifting its anchor points along the arrow moves the scene accordingly, and RenGen fills in the uncovered border without masks or inpainting.

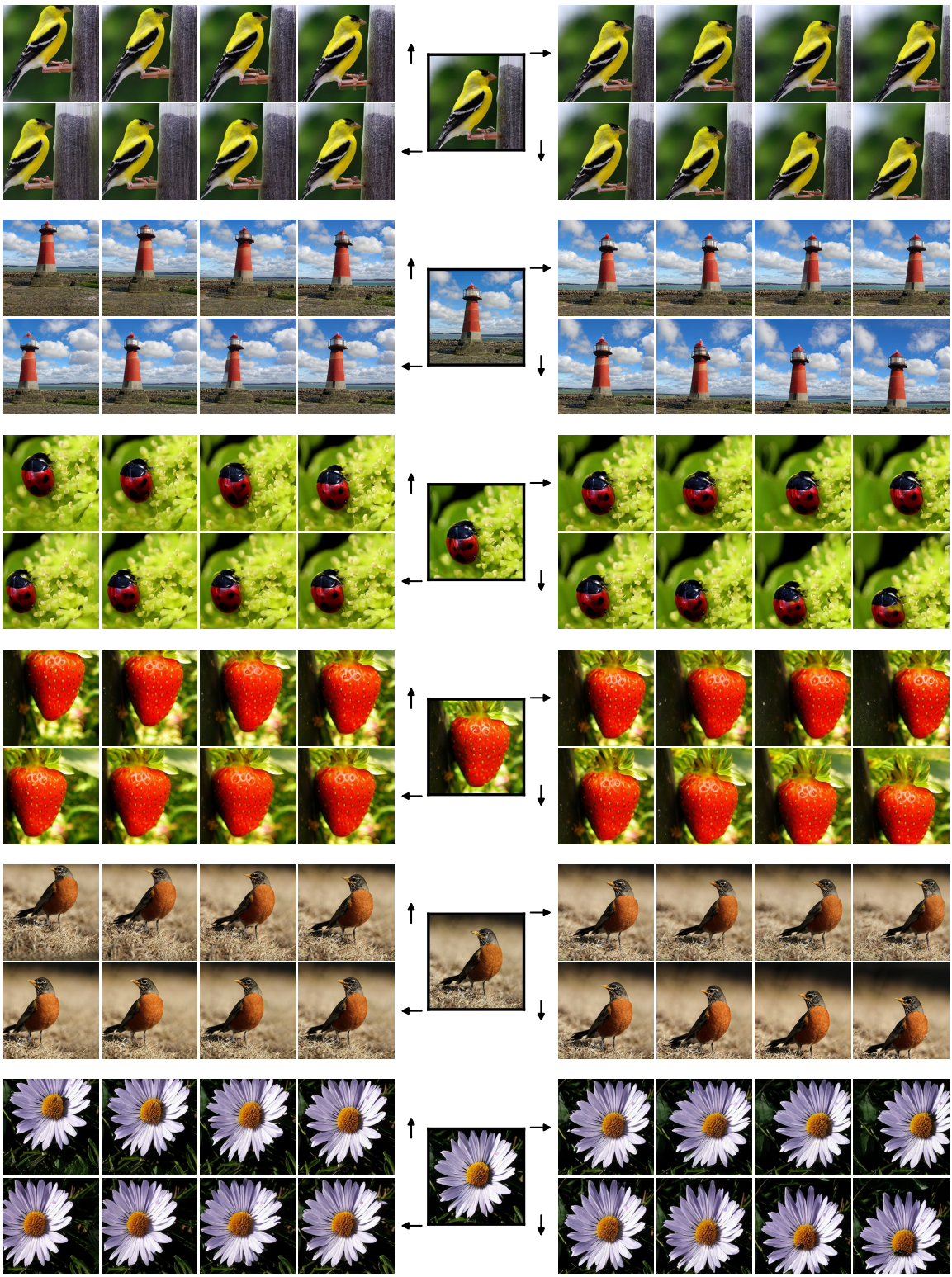

Figure A5. **Moving the anchor points lets users reposition content continuously, training-free.** Each row shifts the anchor points of the outlined center image by 5 to 20% of the image width, horizontally (top) or vertically (bottom), using the same noise for every offset. Even a 5% shift (12.8 px), smaller than one patch, moves the content smoothly.

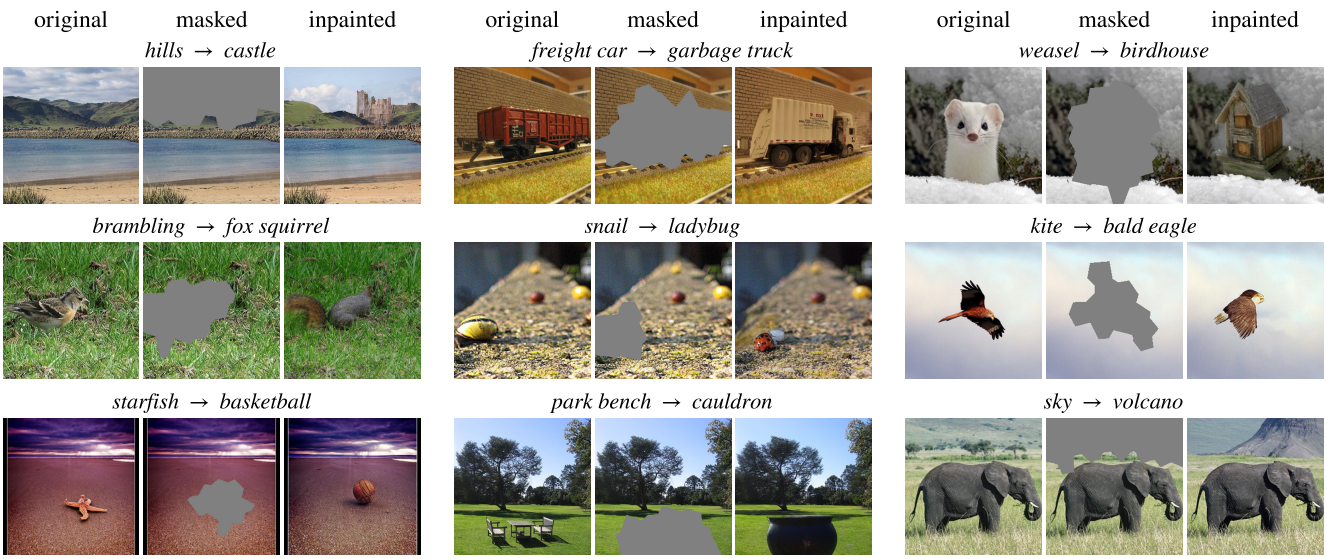

Figure A6. **Inpainting replaces any region of a real image with an object of a chosen class, training-free.** Tokens in the masked cells (gray) are regenerated with the new class while all others stay fixed. The inserted content matches the scene's scale, lighting, and depth of field.

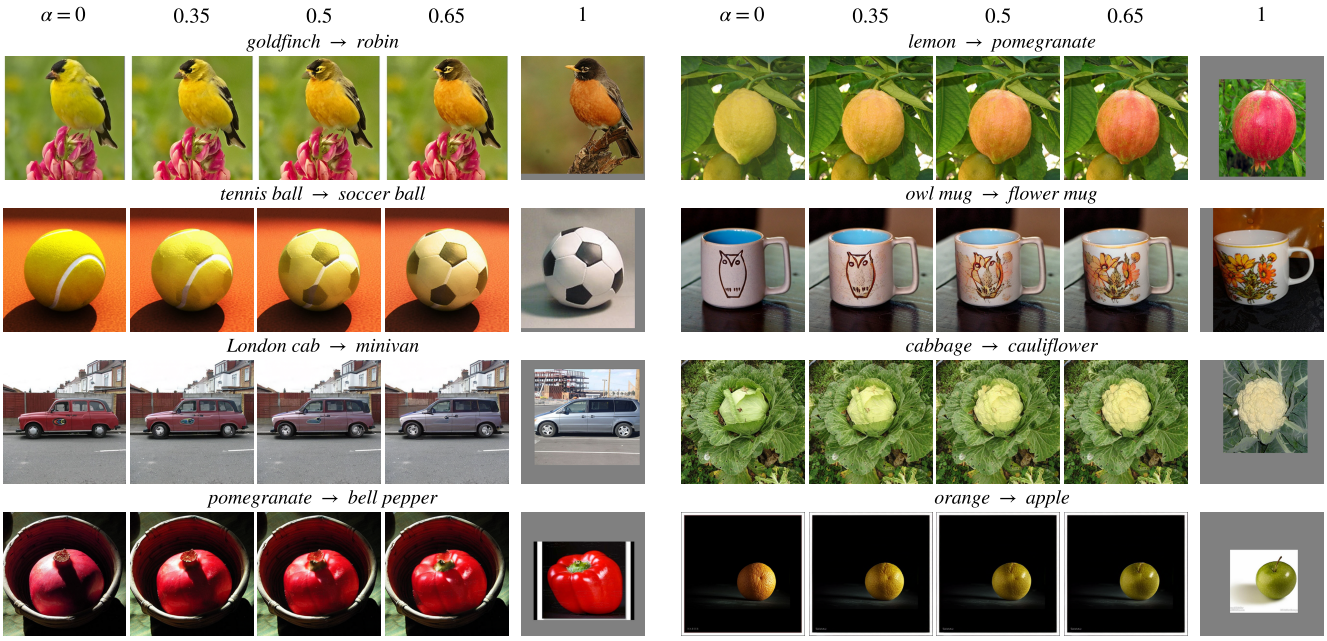

Figure A7. **Interpolating region tokens blends two objects into plausible hybrids rather than cross-fades.** Tokens inside the destination object (α=0) are interpolated toward those of the aligned source image (α=1, right) and decoded directly. Each hybrid keeps the destination's pose, lighting, and background.

trained generator (Fig. A6). We encode the image at P anchor points and select the cells to replace. The selected tokens start from pure noise and are denoised over the full schedule with a chosen class c, while all other tokens are fed in clean at every step and stay unchanged. Through the anchor points, the generator sees both the known content and the exact shape of the hole, and fills the hole to fit its surroundings. In Fig. A6, the inserted objects follow the scene's scale, lighting, and depth of field. Because the hole is a set of tokens rather than a pixel mask, it can take any shape formed by the cells, and no inpainting-specific training or mask conditioning is needed. Patch-based models such as RAEv2 can inpaint in the same way, but they always process the full grid of 256 tokens.

A.5. Blending objects

Users can blend two objects into a single hybrid by interpolating their tokens (Fig. A7). We align the source object with the destination object by scaling and translating the source image so that their bounding boxes coincide, and encode both images at the same anchor points, so tokens at the same point describe the same part of the object. Inside the destination object, we interpolate $z_i = (1-\alpha)\, z_i^{\text{dst}} + \alpha\, z_i^{\text{src}}$, keep all other tokens unchanged, and decode directly without sampling. Intermediate values of α produce a single plausible object that keeps the destination's pose, lighting, and shadow, rather than a cross-fade. Most of the change happens for α between about 0.35 and 0.65, suggesting that interpolating region tokens moves between objects along plausible images rather than averaging their appearance.

B. Additional ablations

B.1. Sensitivity to the region partition

RenGen uses Felzenszwalb–Huttenlocher (FH) segmentation to define regions for the autoencoder. We test how sensitive reconstruction is to the FH parameters by varying them at inference time, one at a time, with the autoencoder trained at k=100, σ=0.8, m=32 (Tab. B1). Reconstruction quality changes little across a wide range of settings, even though the number of tokens per image roughly doubles across this range. The scale k has the largest effect on the token count and gives a natural knob for the token budget. Lowering k yields more, finer regions with similar quality, and quality degrades only once regions become coarse (k≥200). Smoothing σ and the minimum segment size m mainly shift the token count and have a smaller effect on quality. A single autoencoder can therefore operate at different token budgets without retraining, and our default lies in the flat part of this trade-off.

Setting	Tokens	rFID↓	PSNR↑	SSIM↑
Scale k (σ=0.8, m=32)				
k=25	158.4	0.328	21.44	0.604
k=50	124.0	**0.313**	21.88	**0.609**
k=100	88.3	0.321	**22.23**	**0.609**
k=200	56.9	0.390	21.36	0.590
k=400	33.6	0.774	19.05	0.543
Smoothing σ (k=100, m=32)				
σ=0.4	107.7	**0.312**	22.03	0.608
σ=0.8	88.3	0.321	**22.23**	**0.609**
σ=1.2	74.6	0.337	22.18	0.605
σ=1.6	63.4	0.366	21.84	0.598
Minimum size m (k=100, σ=0.8)				
m=8	85.7	0.324	22.15	0.607
m=16	86.7	0.323	22.18	0.608
m=32	88.3	0.321	22.23	**0.609**
m=64	88.3	**0.320**	**22.27**	**0.609**
m=128	71.3	0.335	22.11	0.604

Table B1. **Reconstruction is robust to the FH parameters, so one autoencoder can change its token budget at inference without retraining.** rFID stays within 0.31–0.34 from 74 to 158 tokens per image, and degrades only once regions become coarse (k≥200). The autoencoder is trained with k=100, σ=0.8, m=32 (shaded), and we vary one parameter at a time at inference. Results on ImageNet-val (256×256). Tokens is the mean number of regions per image.

B.2. Does generated complexity follow token count?

RenGen is meant to spend fewer tokens on simple scenes and more on busy ones, so the complexity of a generated image should follow the number of anchor points it receives. We test this by splitting ImageNet-val images into four buckets by FH region count, generating 12.5k images per bucket with the class labels and token counts of that bucket's images, and re-segmenting the generated images with the same FH parameters (Tab. B2). The mean region count of generated images closely tracks the requested token count, with a rank correlation of 0.94 across all samples. Most generated images fall inside their bucket's region range. The fraction is lower for the middle buckets because their ranges are narrow relative to the per-image spread. In the most complex bucket, generated images are slightly simpler than real ones. Quality stays nearly constant across buckets, so RenGen trades scene complexity rather than image quality as the token count changes.

Bucket	Real region count	Generated region count	In range
Q1 (1–61)	45.6	$45.9_{\pm 15.6}$	85%
Q2 (62–84)	73.3	$73.3_{\pm 14.5}$	58%
Q3 (85–109)	96.7	$94.3_{\pm 16.5}$	54%
Q4 (110–255)	133.6	$122.8_{\pm 21.4}$	72%

Table B2. **The complexity of generated images matches that of real images.** We report the mean region count of real and generated images (std as subscript) and the fraction of generated images whose region count falls inside the bucket's range (In range).

C. Autoencoder details

Tab. C1 lists the full autoencoder configuration.

Encoder. We use a frozen DINOv3 ViT-L/16 [39], which maps a 256×256 image to a 16×16 grid of patch features. Following RAEv2 [40], we aggregate features from seven blocks (every other block, ending with the last). The region encoder projects these features to width C=1152 and adds random Fourier features $\gamma(\mathbf{u}) = [\sin(2\pi B\mathbf{u}), \cos(2\pi B\mathbf{u})]$ of each patch center $\mathbf{u}$, with $B \in \mathbb{R}^{576\times 2}$ sampled from $\mathcal{N}(0,1)$. Each region query is initialized as the average of

the projected features (without positional features) over the region's patches, and is then refined by four cross-attention blocks. The attention logits are biased by $\log m_i$, where m_i is the region's patch-ownership mask, so each token attends only to its own patches.

Decoder. The decoder uses one query per output patch. Queries are initialized with Mip-NeRF-style integrated positional encodings [2] using frequencies $3B$, which suppress frequencies finer than the patch grid, and the same encodings are added before every attention operation. Each of the L_d=14 layers applies cross-attention to the region tokens, self-attention among the queries, and MLPs. A linear head maps each query to the $16\times16\times3$ pixels of its patch. We scale the initialization of residual output projections by $1/\sqrt{4L_d}$, without which training stalls and the decoder ignores the tokens.

Training. We train the region encoder and decoder jointly on ImageNet-1K with Eq. 1, using λ_p=1 and λ_{adv}=0.75 with VQGAN's adaptive weight [10]. The discriminator follows RAE [53], a frozen DINO ViT-S/8 with trainable heads and DiffAugment [52]. To make the decoder robust to generator errors, we add Gaussian noise to the region tokens during training, with a standard deviation of $\sigma \sim \mathcal{U}(0, 0.8)$ times the per-channel token standard deviation, as in RAE. We train for 48 epochs with AdamW, batch size 512, and bf16 mixed precision. The learning rate follows a cosine schedule from 2×10^{-4} to 2×10^{-6}. Training is completed on a single NVIDIA 4×GH200 node for all autoencoder experiments.

D. Point-anchored DiT details

Tab. D1 lists the full DiT configuration.

Architecture. We use the DiT^{DH}-XL architecture of RAE [53], a 28-block transformer followed by a wide, shallow DDT head. Unlike RAE's patch-based model, the input is a variable-length set of P region tokens, padded within a batch and handled with an attention mask. Each token uses its anchor point as a continuous 2D RoPE position, scaled to units of the 16×16 patch grid, so a point at a patch center receives exactly that patch's rotation angles. The timestep, class label, and token count P are embedded as learnable in-context tokens appended to the sequence. Internal guidance [54] reads an auxiliary prediction from an intermediate block through the same head, so it requires no extra network evaluation.

Training targets. For each training image, we set P to its FH region count, perturbed by a random log-uniform factor and clipped to $[16, 196]$. We sample P points uniformly and refine them with three Lloyd iterations [29]. The frozen region encoder then pools the image's DINOv3 features within each Voronoi cell, giving one target token per anchor point (Fig. 4 of the main paper). Targets are normalized per channel with training-set statistics, and we group images with similar P into batches to reduce padding.

Representation encoder (frozen)	
Backbone	DINOv3 ViT-L/16
Feature blocks	12, 14, …, 24 (7 blocks)
Feature grid	$16\times16\times1024$
Regions	
Partition	FH, k=100, σ=0.8, min. size 32 px
Patch ownership	smallest segment with $\geq$25% coverage
Regions / image	88.3 mean (5th / 95th pct. 34 / 151)
Region encoder	
Width / heads / blocks	1152 / 16 / 4
Positional features	576 random Fourier freqs., $\mathcal{N}(0, 1)$
Parameters	67.6M
Region decoder	
Width / heads / MLP	1152 / 16 / 4096
Layers L_d	14
Layer	cross-attn, MLP, self-attn, MLP
Query encoding	integrated PE, frequencies $3B$
Output	linear, $16\times16\times3$ per query
Parameters	415.4M
Training	
Dataset	ImageNet-1K train, 256×256
Optimizer	AdamW (β_1, β_2= 0.9, 0.95)
Learning rate	$2\times10^{-4} \rightarrow 2\times10^{-6}$, cosine
Warmup	0.25-epoch
Batch size / epochs	512 / 48
Loss weights	λ_p=1, λ_{adv}=0.75 (adaptive)
Discriminator	frozen DINO ViT-S/8, DiffAugment
Token noise	$\sigma \sim \mathcal{U}(0, 0.8) \times$ channel std.
EMA decay	0.9978
Precision	bf16

Table C1. **Autoencoder configuration.**

Training. We train with flow matching on $x_t = (1-t)\,x + t\,\epsilon$, predicting x and computing the loss on the implied velocity over non-padded tokens only. Following RAEv2, timesteps follow a shifted logit-normal distribution. Although RenGen trains for more epochs than RAEv2 (120 vs. 80), each step costs $\sim$2.5× fewer FLOPs, so its total DiT training compute is roughly 40% lower. Training is completed on a single NVIDIA 4×GH200 node for all DiT experiments.

Sampling. At sampling time, we draw P from the region count of a random training image of the target class and place the anchor points with the same uniform sampling and Lloyd refinement used in training. No segmentation is run at sampling time. We integrate with a 50-step Euler solver and apply internal guidance over the noisier part of the trajectory.

Architecture (DiT^{DH}-XL)	
Body width / heads / blocks	1440 / 20 / 28
Head width / heads / blocks	2048 / 16 / 2
In-context tokens (time / class / P)	4 / 8 / 4
Positional encoding	2D RoPE at anchor points
Internal-guidance branch	output of block 8
Parameters	876.5M
Targets	
Token count P	FH count $\times e^u$, $u{\sim}\mathcal{U}(-0.2, 0.2)$ clipped to $[16, 196]$
Anchor points	uniform + 3 Lloyd iterations
Cells	Voronoi, on the $16{\times}16$ patch grid
Normalization	per-channel mean / std
Augmentation	horizontal flip
Optimization	
Objective	flow matching, x-pred., v-loss
Timesteps	logit-normal(0, 1), shift $s{=}8$
Label dropout	0.1
Optimizer	Muon (2D weights) + AdamW $\beta{=}(0.9, 0.95)$, no weight decay
Batch size / grad. clip	1024 / 1.0
Learning rate	$2{\times}10^{-4}$, linear decay to $2{\times}10^{-5}$ over epochs 25–50, then constant
Epochs	120
EMA decay	0.9995
Sampling	
Solver	Euler, 50 steps, shift $s{=}8.0$
Guidance	internal, scale 2.1, $t \in [0.6, 1]$
Anchor points	uniform + 3 Lloyd iterations

Table D1. **Point-anchored DiT configuration.**

E. Evaluation protocol

Generation metrics. Unless noted otherwise, we compute gFID and IS from 50k samples (50 per class) against ImageNet $256{\times}256$ training-set statistics. Sampling settings are listed in Tab. D1.

Reconstruction metrics. We compute rFID, PSNR, and SSIM on the 50k ImageNet validation images, center-cropped to $256{\times}256$. rFID is measured against the statistics of the same images.

Complexity buckets (Fig. 5). We sort the validation images by FH region count and split them into four equal buckets of 12.5k images. For each image, we generate one sample with its class label and region count, and compute each bucket's gFID against its 12.5k real images. Because both the sample and reference sets are smaller than in the standard protocol, these values are higher than 50k gFID and are comparable only across buckets. We evaluate BudgetPix on the same buckets with its released JiT-L/16 checkpoint and sampler, and report its token count averaged over the denoising trajectory.

Timing and FLOPs. We report per-image wall-clock time for end-to-end generation, including anchor placement, 50 guided sampling steps, and decoding. We measure it on one GH200 GPU at batch size 64 in bf16, with $P{=}88$, the mean token count. Baselines are timed with the same setup. We count FLOPs with PyTorch's `FlopCounterMode` over all DiT evaluations and one decoder pass.

Baselines. Numbers in Tab. 3 are taken from the respective papers, except for RAEv2 and BudgetPix, which we evaluate from their released checkpoints. For Fig. 7, we run the released FlexTok, KARL, RAE, and RAEv2 models with their default reconstruction settings.

F. Limitations

RenGen approaches but does not yet match the gFID of the strongest patch-based models. However, compared to the strongest patch-based model, RenGen uses $2.9\times$ fewer tokens and $2.5\times$ lower latency. Closing this gap without giving up this efficiency remains an open direction. At very low token counts, thin structures and small objects can be lost. This trade-off is inherent to any compressed latent, and in RenGen it is easy to avoid, since users can raise the token count for scenes that need it. At sampling time, the token count is either drawn from training images of the class, which keeps generated complexity matched to the data, or set by the user. A model that predicts its own budget from the condition could be more convenient. The controls in Sec. A are training-free from the model, with no task-specific training, and leaving room for refinement. Edits follow the cells of their anchor points rather than exact object outlines, transplanting and blending work best for roughly aligned objects of similar shape, and re-noising can slightly alter details near the edit. Finally, we evaluate only class-conditional generation on ImageNet, the standard benchmark for latent generative models. Scaling region tokens to text-to-image generation requires substantially more data and compute, and is a natural next step.