LayoutDiffusion: Improving Graphic Layout Generation by Discrete Diffusion Probabilistic Models
– Supplementary Metearial –

Contents

A. Additional Implementation Details **13**
A.1. More Implementation Details for LayoutDiffusion 13
A.2. Implementation of other Diffusion Baselines 13
A.3. Settings on Conditional Generation Tasks 14
A.4. Details about Classification Model for FID Evaluation 14

B. Discussion on Diversity **15**
B.1. Metric SelfSim 15
B.2. Algorithm of SelfSim 15
B.3. Case Study of SelfSim 15
B.4. SelfSim Comparison with Existing Methods 15

C. Additional Experiments on Conditional Generation **16**
C.1. Refinement 16
C.2. Generation Conditioned on Type 17

D. Additional Ablation Studies **18**
D.1. Additional Ablation Studies on Conditional Generation Tasks 18
D.2. Additional Ablation Studies on Noise Schedule of Type Tokens 19
D.3. Ablation Studies on Sequence Ordering 19

E. Qualitative Results of LayoutDiffusion **20**
E.1. Unconditional Generation 20
E.2. Refinement 21
E.3. Generation Conditioned on Type 22

F. Fine-Grained Visualization of the Forward Diffusion Process **24**

G. Fine-Grained Visualization of the Reverse Generation Process **26**

A. Additional Implementation Details

A.1. More Implementation Details for LayoutDiffusion

Noise Schedule. We investigate the effectiveness of our proposed schedule $\beta_t = g/(T-t+\epsilon)^h$ for the discretized Gaussian transition matrix by comparing with the original linear schedule $\beta_t = bt/T$ used in [1]. Notably, here β_t is not a variance term bounded in $[0,1]$[1], and with the growth of forward steps, the cumulative matrix $\overline{\mathbf{Q}}_t^{coord}$ converges to uniform distribution. Thus, we attempt to analyse the noise process by observing the standard derivation of the cumulative matrix. A higher std. indicates a more sparse matrix and hence a lower transition probability to other coordinate tokens. As shown in Fig. 6, our schedule presents a gentler noising process and a more stable convergence state compared to the original linear schedule (as suggested by a higher std. at the beginning of the forward process, and a lower std. at the very end of the process).

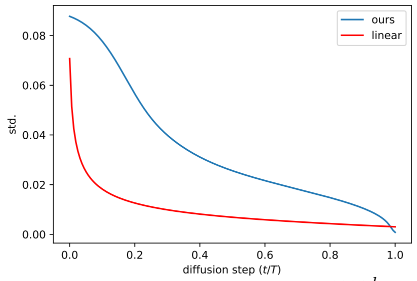

Figure 6. Standard derivation of the cumulative discretized Gaussian matrix $\overline{\mathbf{Q}}_t^{coord}$ of our proposed β_t and the original linear schedule.

Denoising Model. We present the model architecture as in Fig. 7. We embed the input sequence, input timestep, and positions with 768, 128, and 768 dimensions respectively. The dropout rate is set as 0.1. For the transformer encoder, we simply adopt the same hyperparameters of BERT-base [5] encoder, i.e., 12 layers, 12 attention heads, 768 hidden size, and 3,072 dimensions for the feed forward layer.

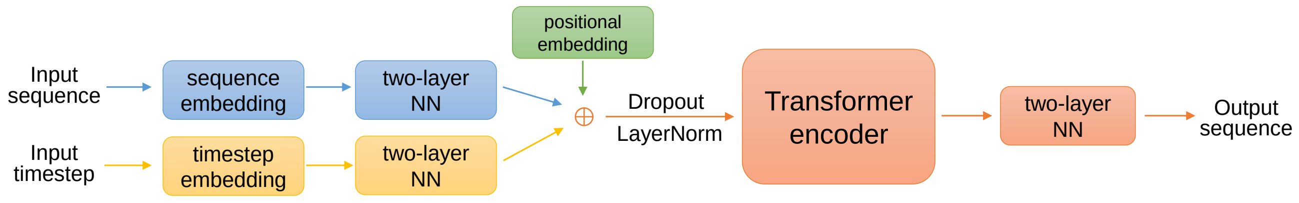

Figure 7. Model architecture of $p_\theta(x_0|x_t)$.

Hyperparameters. We train our model using AdamW optimizer [11] with $lr = 0.00004$, $betas = (0.9, 0.999)$, and zero $weight\_decay$. We also apply an exponential moving average (EMA) over model parameters with a rate of 0.9999. We set the batch size as 64. For RICO [3], we train the model with 2 V100 GPUs for 175,000 steps to achieve the best results; and for PublayNet [16], we train the model with 4 V100 GPUs for 350,000 steps. For the schedule of type tokens, we set $\tilde{T} = 160$ in Eq. (12). For the schedule of coordinate tokens, we set $T = \tilde{T} = 160$, $g = 12.4$, $h = 2.48$, and $\epsilon = 0.0001$ in $\beta_t = g/(T-t+\epsilon)^h$. We also provide the hyperparameters for the implementation of different timesteps apart from 200 steps (we implement these variants in the Tab. 3 of the main paper), as shown in Tab. 4.

A.2. Implementation of other Diffusion Baselines

Diffusion-LM [10]. We implement Diffusion-LM based on the official repository[2]. We realize the layout generation task via Diffusion-LM by feeding the layout as a sequence and reconverting the output sequence to a layout. For the hyperparameters, we simply adopt the default setting that Diffusion-LM adopts on E2E [12] dataset, since only relatively small vocabulary size (~ 150) is required for the tokens that represent layout sequence.

For conditional generation tasks, we apply the similar idea as in LayoutDiffusion. Specifically, for Gen-Type, we fix the type tokens by feeding the target in each timestep and run the whole reverse denoising process. For refinement task,

[1]In fact, as β_t tends to positive infinity, $\mathbf{Q}_t^{coord}$ will approach a transtion matrix for uniform noise as described by Sohl-Dickstein *et al.* [15].
[2]https://github.com/XiangLi1999/Diffusion-LM

Total timesteps	$\tilde{T}$ for type schedule	β_t for coordinate schedule
100	80	$20.0/(80-t+0.0001)^{2.96}$
200	160	$12.4/(160-t+0.0001)^{2.48}$
500	400	$6.2/(400-t+0.0001)^{2.00}$
1000	800	$3.5/(800-t+0.0001)^{1.76}$
2000	1600	$2.0/(1600-t+0.0001)^{1.52}$

Table 4. Hyperparameters for variants of different timesteps.

we embed the layout sequence and set the embedded latent as the input of start timestep T_{refine}, and then run the remaining reverse process with type fixed. For the choice of T_{refine}, we traverse through [250,500,750,1000,1250,1500] of the total 2000 steps, and find the best result is achieved when $T_{\text{refine}} = 1000$.

D3PM uniform [1]. We implement D3PMs based on both the official repository of D3PMs[3] and the official repository of another method concerning discrete diffusion model, i.e., VQ-Diffusion[4], since our implementation is based on PyTorch [13] rather than JAX [2]. We realize the layout generation task in a similar way as LayoutDiffusion, i.e., feeding the layout as a sequence of tokens and treating each token as a discrete state.

For the hyperparameters of the diffusion framework, we set the total diffusion timesteps $T = 1000$, the schedule $\beta_t = (T-t+1)^{-1}$, and the auxiliary loss weight $\lambda = 0.0001$, all follow the setting as reported in D3PMs paper. For the denoising model, we apply the similar model as in LayoutDiffusion and Diffusion-LM for a fair comparison.

For conditional generation tasks, we implement the Gen-Type and refinement the same way as in our implementation of Diffusion-LM, since both methods are replace-based diffusion methods. For the start timestep for refinement task T_{refine}, we sweep from [200,400,600,800] of the total 1000 steps, and find $T_{\text{refine}} = 400$ is the optimal choice.

D3PM absrobing [1]. We implement the D3PM (absorbing) the same way as in D3PM (uniform). All settings except for the model are also referenced from the original D3PMs paper.

One major difference lies in the implementation of conditional generation tasks. For the Gen-Type task, we apply the similar idea as in LayoutDiffusion. To be more specific, we feed the given type set at the beginning step T, and run the whole reverse process. It is noteworthy that, for D3PM (absorbing), all coordinates start to recover strictly from timestep T. Hence, it cannot save steps as the same strategy in LayoutDiffusion by picking a timestep $T_{\text{Gen-Type}}$ which is smaller than T. Besides, in the reverse process of D3PM (absorbing), the sampled tokens cannot transition into `MASK` or other tokens, thus, it is unable to perform the refinement task as in LayoutDiffusion.

A.3. Settings on Conditional Generation Tasks

We present below the settings on conditional generation tasks in the main paper. For further experiments on conditional generation tasks, please refer to Sec. C.

Gen-Type. We follow the convention in [7, 9]. To be more specific, for a layout in the test set, we extract its type set as the input and let the model generate the bounding box attributes of each element .

Refinement. In the real scenario, the noise level of the user-given flawed layout cannot be known in advance. Besides, different flawed layouts may have different noise levels. To simulate the real scenario, we improve the setting used in RUITE [14]. Specifically, the setting in RUITE is to construct a test set by adding random noises to the position and size of each element, where the noise is sampled from a normal distribution with mean 0 and the standard variance 0.01. In our improved setting, we modify the standard variance of the noise to be uniformly sampled from $[0.005, 0.01, 0.015, 0.02, 0.025]$. Besides, for the baselines, we train the model with input noise of 0.01 standard variance; for LayoutDiffusion, we apply the same inference steps T_{refine} for different levels of noise, unlike the settings in Sec. C.1.

A.4. Details about Classification Model for FID Evaluation

We use the same layout classification model as LayoutGan++ [8] and LayoutFormer++ [7]. Specifically, the model includes an encoder and a decoder, both of which are based on the Transformer architecture. The encoder takes in bounding box coordinates and corresponding labels and produces a feature representation, while the decoder uses the feature representation

[3]`https://github.com/google-research/google-research/tree/master/d3pm`
[4]`https://github.com/microsoft/VQ-Diffusion`

to predict the class probabilities and bounding box coordinates for each layout element. We implement the model based on the official repository of LayoutGan++[5] and train it using the methods described in their paper. Our evaluation results using this model are consistent with those reported in the LayoutFormer++.

B. Discussion on Diversity

B.1. Metric SelfSim

Diversity is a key but often overlooked aspect in layout generation tasks. In this section, we propose a new metric called *SelfSim* to measure the self-similarity of generated layouts, which serves as an indicator of diversity. The intuition behind this metric is that more diverse generated layouts should be less self-similar. Specifically, we calculate the average Intersection over Union (IoU) between any pairs of generated layouts with the same set of element types.

B.2. Algorithm of SelfSim

Inspired by the metrics for evaluating diversity in NLP (*e.g.*, diverse 4-gram [4] and self-BLEU [17]), we propose to assess the diversity of generated layouts by measuring the self-similarity of the generated layout set. Specifically, we partition the generated layouts into different subsets based on their type sets and then count the similarity of the layouts within each subset. The similarity is calculated by averaging the intersection of union (IoU) of the bounding boxes for each pair of layouts in the subset. We present the algorithm for calculating SelfSim as in Algo. 1.

Algorithm 1: Calculation of the Self-Similarity score

```
Input: A set of graphic layouts $\mathbb{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$
Output: The Self-Similarity score of the given layout set
1  partition $\mathbb{X}$ by the layouts' type set, denote the partition as $\mathbb{P} = \{\mathbb{X}_1, \mathbb{X}_2, \dots, \mathbb{X}_n\}$;
2  for $i \leftarrow 1$ to $n$ do                    /* traverse each subset $\mathbb{X}_i \in \mathbb{P}$ */
3      count the number of elements in subset $\mathbb{X}_i$, denoted as $l_i$;
4      if $l_i = 1$ then                            /* only one layout in the subset */
5          set the Self-Similarity score of subset $\mathbb{X}_i$ as $S_i = 0$;
6      else                                         /* more than one layout has this type set */
7          for each pair $(x^i_j, x^i_k)(j \neq k)$ in $\mathbb{X}_i$ do
8              calculate the IoU of bounding boxes between $x^i_j$ and $x^i_k$, denoted as $U^i_{jk}$;
9          average the $U^i_{jk}$ of the total $\binom{l_i}{2}$ pairs to get the mean $S_i$;
10 return the weighted average of all subsets' Self-Similarity score $\frac{\sum_{i=1}^{n} l_i S_i}{\sum_{i=1}^{n} l_i}$;
```

B.3. Case Study of SelfSim

To visually demonstrate the effectiveness of SelfSim, we show some subsets with different SelfSims (i.e., subsets with different S_i) in Fig. 8. We observe that subsets with higher SelfSims tend to have more similar layouts, while those with lower SelfSims have more diverse layouts. This further supports the effectiveness of the SelfSim metric for assessing the diversity of generated layouts.

B.4. SelfSim Comparison with Existing Methods

Tab. 5 compares LayoutDiffusion with existing layout methods and the diffusion-based method using SelfSim. While LayoutFormer++, Diffusion-LM, and LayoutTransformer have advantages in certain aspects of quality (as shown in Tab. 1 in the main paper), they suffer from obvious diversity issues, which aligns with our user study findings (as shown in Fig. 4). On the other hand, although D3PM performs slightly better in diversity on the PubLayNet dataset, it lags behind in terms of quality (see Tab. 1). These results suggest that our proposed method achieves a better quality-diversity trade-off.

[5]https://github.com/ktrk115/const_layout

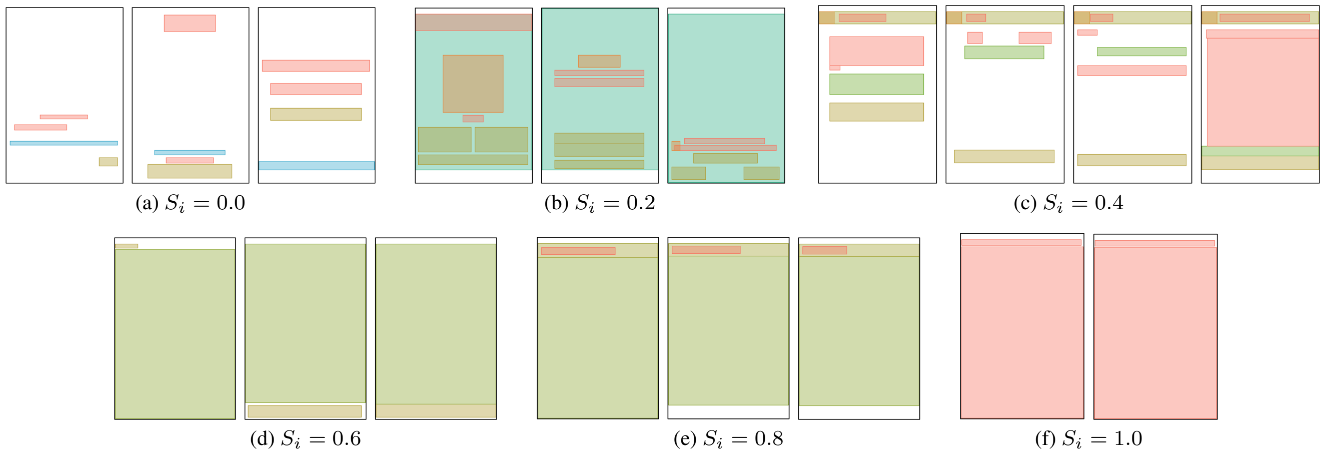
(a) $S_i = 0.0$ (b) $S_i = 0.2$ (c) $S_i = 0.4$ (d) $S_i = 0.6$ (e) $S_i = 0.8$ (f) $S_i = 1.0$

Figure 8. Examples of subsets with different SelfSim. As SelfSim goes from 0 to 1, the layouts in the subset go from totally different to completely identical.

SelfSim↓	LayoutTransformer	LayoutFormer++	Diffusion-LM	D3PM (absorbing)	D3PM (uniform)	LayoutDiffusion
RICO	0.318	*0.581*	*0.326*	0.157	0.165	0.157
PublayNet	*0.314*	*0.328*	0.222	0.194	0.189	0.198

Table 5. Comparison of SelfSim scores for unconditional generation on RICO and PublayNet datasets. Lower SelfSim scores indicate better diversity. *Italic font* denotes the two worst-performing methods.

C. Additional Experiments on Conditional Generation

In this section, we design two sets of experiments to investigate LayoutDiffusion's robustness in the refinement task and its diversity performance in the generation conditioned on type (Gen-Type) task.

C.1. Refinement

Noise level	Methods	mIoU ↑	Overlap→	Align. →	FID ↓
std.=0.005	RUITE	0.743	0.473	0.126	1.244
	LayoutFormer++	0.722	0.479	0.119	1.043
	LayoutDiffusion (30 steps)	**0.787**	**0.467**	**0.095**	**0.499**
std.=0.01	RUITE	0.716	0.483	0.139	1.475
	LayoutFormer++	0.704	0.487	0.123	1.124
	LayoutDiffusion (40 steps)	**0.759**	**0.467**	**0.098**	**0.500**
std.=0.02	RUITE	0.611	0.507	0.203	13.633
	LayoutFormer++	0.621	0.514	0.157	4.981
	LayoutDiffusion (50 steps)	**0.748**	**0.469**	**0.097**	**0.496**
	Real Data	-	0.466	0.093	-

Table 6. Qualitative comparison under different noise levels on RICO. The content in the brackets denotes for the number of inference steps. The best results are **bold**.

As introduced in Sec. A.3, in the main paper, our experiments for the refinement task apply a mixture of different levels of noise as input. We suppose that the excellent results achieved by LayoutDiffusion are due to its capability of handling various levels of noise. To investigate the model's robustness to the noise, in this section, we compare with the two strongest baselines and further study the performance of the methods under each specific noise levels.

Specifically, we evaluate the performance of different methods under the conditions that the standard deviation of the noise is 0.005, 0.01, and 0.02, respectively. Plus, for a fair comparison, we train the baselines with the input noise of 0.01 standard deviation, and apply the same model for inference.

Quantitative results. As shown in Tab. 6, for the two baselines (RUITE [14] and LayoutFormer++ [7]), the models exhibit favorable performances when dealing with noise levels less than or equal to the training level (std.=0.005 and std.=0.01). However, when the testing noise level is greater than the training's (std.=0.02), the models suffer a significant performance drop. For LayoutDiffusion, it not only surpasses the baseline in all 12 competitions (3 levels $\times$ 4 metrics), but also consistently presents excellent performance as the noise level varies, indicating that LayoutDiffusion is highly robust to noise levels.

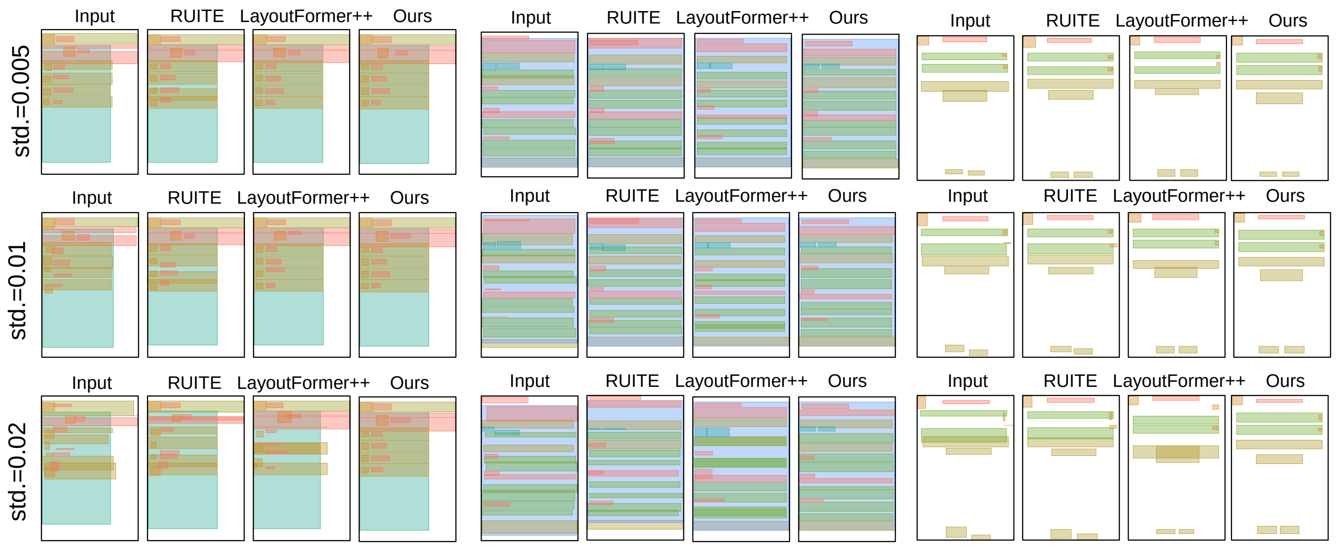

Figure 9. Qualitative comparison under different noise levels on RICO. Each row shares the same noise levels while each column shares the same method. For more quantitative result of LayoutDiffusion on Refinement, please refer to Sec. E.2.

Qualitative results. We provide the quantitative results in Fig. 9. We can conclude that as the input gets more chaotic, LayoutDiffusion consistently produces pleasing layouts, while other baselines fail to achieve so, which is in line with the quantitative results.

C.2. Generation Conditioned on Type

As described in Sec. A.3, in the main paper, our experiments for the Gen-Type task only generate one sample for each input type set. In this section, we study whether they can generate multiple diverse layouts for a given type set to further explore the diversity performance of each method under Gen-Type task.

Specifically, we first find all the different type sets in the layouts of the test set, and then equally generate 5 layouts for each given type set[6]. We compare to the baseline with the best quality performance (i.e., LayoutFormer++ [7]), which can generate multiple layouts with top-k sampling [6]. LayoutDiffusion can generate multiple layouts by simply running the inference process multiple times. We apply the metric SelfSim (as discussed in Sec. B) for the evaluation of diversity.

Dataset	Methods	mIoU ↑	Overlap↓	Align. ↓	FID ↓	SelfSim ↓
RICO	LayoutFormer++	**0.375**	0.563	0.125	9.786	0.536
	LayoutDiffusion	0.357	**0.490**	**0.062**	**8.973**	**0.268**
PublayNet	LayoutFormer++	**0.315**	0.025	0.030	31.121	0.224
	LayoutDiffusion	0.312	**0.007**	**0.029**	**21.522**	**0.189**

Table 7. Qualitative comparison under new sampling strategy (5 samples for each type set). Since the type distribution of the generated layouts differs from that of the test set, we simply assume here that the less misalignment and overlap is better.

Quantitative results. The quantitative results is given in Tab. 7. Compared to LayoutFormer++, LayoutDiffusion performs significantly better in diversity (as suggested by SelfSim), while achieving comparable quality performance (as suggested by Overlap and Align.). We hypothesize that the gap between diversity is due to the probability accumulation of the autoregressive model while LayoutDiffusion samples each layout from independent noise.

[6]In practice, we find 2714 different type sets out of 3729 layouts in the test set of RICO, and 1339 type sets out of 10998 layouts in PublayNet's test set.

Qualitative results. As show in Fig. 10, despite equipped with top-k sampling, LayoutFormer++ still suffers severe diversity problem (duplication occurs in the first three row and the last row. Besides, the generated layouts in the fourth row share similar patterns). While for LayoutDiffusion, all the 5 samples of each type set are both pleasing and in great diversity, further demonstrating the superiority of LayoutDiffusion on Gen-Type task.

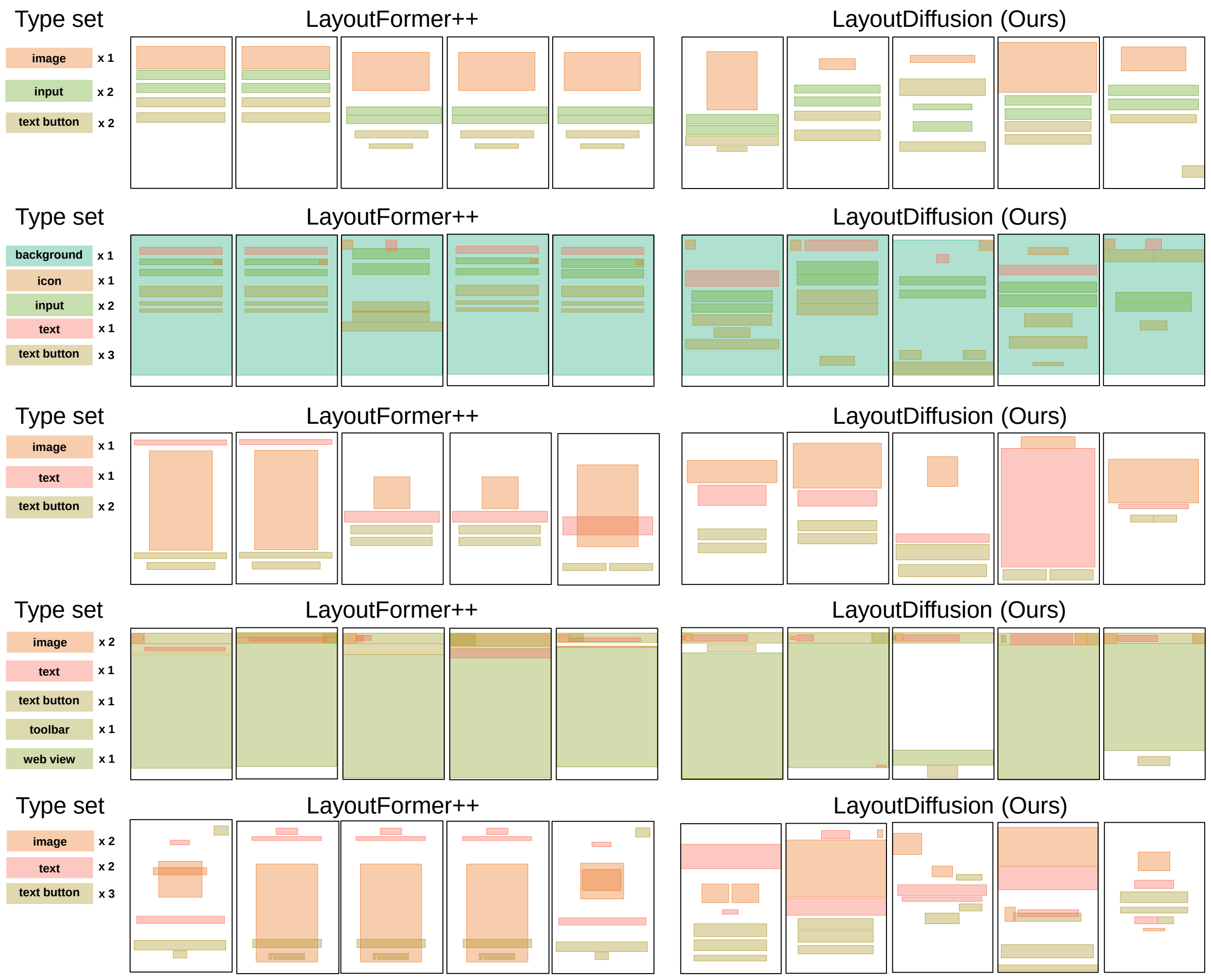

Figure 10. Qualitative comparison of diversity on RICO. Each type set corresponds to five samples from LayoutFormer++ (left) and five samples from LayoutDiffusion (right). For more qualitative results of LayoutDiffusion on Gen-Type, please refer to Sec. E.3.

D. Additional Ablation Studies

D.1. Additional Ablation Studies on Conditional Generation Tasks

In addition to the ablation studies on unconditional layout generation discussed in the main paper (see Tab. 2), we also conducted ablation studies with the variations on two conditional generation tasks, i.e., Gen-Type and Refinement, to further investigate the effectiveness of our design. The quantitative comparison is shown in Tab. 8.

LayoutDiffusion consistently outperforms all the variations on almost all metrics in both tasks. This result indicates that our design is superior and the reverse generation process is well-suited to both tasks, allowing the model to better leverage the given conditions. Notably, the comparison between LayoutDiffusion and Uniform $\mathbf{Q}_t^{\text{type}}$ as well as Linear $\overline{\gamma}_t$ highlights the importance of our handling of the type tokens, which considers type corruption factor. This factor leads to better utilization of type information in the Gen-Type task. Moreover, the comparison between LayoutDiffusion and the two variations of $\mathbf{Q}_t^{\text{coord}}$ as well as Linear β_t in the Refinement task demonstrates the importance of our design for the coordinate tokens, which helps us model the precise details of the layout and achieve better performance in the Refinement task.

Methods	Gen-Type mIoU ↑	Gen-Type Overlap ↓	Gen-Type Align. ↓	Gen-Type FID ↓	Refinement mIoU ↑	Refinement Overlap ↓	Refinement Align.↓	Refinement FID↓
Uniform $\mathbf{Q}_t^{\text{coord}}$	0.324	0.601	0.141	2.944	0.645	0.487	0.199	4.312
Absorbing $\mathbf{Q}_t^{\text{coord}}$	0.336	0.587	0.137	2.846	-	-	-	-
Uniform $\mathbf{Q}_t^{\text{type}}$	0.320	0.532	0.188	3.070	0.698	0.477	0.167	2.443
Linear $\overline{\gamma}_t$	0.308	0.513	0.164	2.768	0.667	**0.467**	0.133	1.451
Linear β_t	0.317	0.527	0.191	2.273	0.659	0.491	0.185	1.835
LayoutDiffusion (ours)	**0.345**	**0.491**	**0.124**	**1.557**	**0.719**	0.469	**0.102**	**0.549**

Table 8. Quantitative results on conditional generation tasks for LayoutDiffusion and its ablations on RICO. The variation of absorbing $\mathbf{Q}_t^{\text{coord}}$ do not support refinement, as the coordinates are fixed during generation. The best result is in **bold**.

D.2. Additional Ablation Studies on Noise Schedule of Type Tokens

In this section, we further investigate the effectiveness of our type schedule by experimenting other different $\overline{\gamma}_t$ schedules.

Recall that in the main paper, we follow the insight that type changes in the early stage may bring large semantic shift to the layout, thus, we set the noise schedule for $\overline{\gamma}_t$ as:

$$\overline{\gamma}_t = \begin{cases} 0, & t < \tilde{T} \\ (t - \tilde{T})/(T - \tilde{T}), & t \geq \tilde{T} \end{cases} \tag{12}$$

We denote this kind of schedule as "late absorb $\tilde{T}$", since under this schedule, all type tokens stay unchanged until timestep $\tilde{T}$ when they start to absorb, and at the terminal step T, all type tokens reach the absorbed state. Follow this idea, we can come to a similar noise schedule, "early absorb $\tilde{T}'$", where the type tokens start to absorb at the beginning and fully adsorbed in the early stage, and it can be defined as follows:

$$\overline{\gamma}_t = \begin{cases} t/\tilde{T}', & t < \tilde{T}' \\ 1, & t \geq \tilde{T}' \end{cases} \tag{13}$$

Note that, when $\tilde{T}' = T$ and $\tilde{T} = 0$, two schedules becomes the same and is experimented in the ablation studies of the main paper (denoted as "linear $\overline{\gamma}_t$"). Here, we provide a detailed experiment on $\overline{\gamma}_t$, including different choices of "late absorb $\tilde{T}$" and "early absorb $\tilde{T}'$".

Experiments	Methods	mIoU ↑	Overlap→	Align.→	FID ↓
Type schedule	early absorb $\tilde{T}' = 40$	0.574	0.512	0.160	3.107
	early absorb $\tilde{T}' = 100$	0.590	0.496	0.143	2.952
	late absorb $\tilde{T} = 0$ (early absorb $\tilde{T}' = 200$)	0.580	0.522	0.156	2.846
	late absorb $\tilde{T} = 100$	0.599	0.495	0.121	2.612
Sequence ordering	ltwh+random	0.585	0.505	0.136	3.166
	ltwh+position	0.577	0.491	0.128	3.055
	ltwh+lexico	0.578	0.501	0.125	3.111
	ltrb+random	0.619	0.504	0.089	2.505
	ltrb+position	0.613	0.482	0.115	2.446
Ours	ltrb+lexico, late absorb $\tilde{T} = 160$	0.620	0.502	0.069	2.490
Real Data		-	0.466	0.093	-

Table 9. More ablation studies under unconditional generation task on RICO.

As shown in the first group of Tab. 9, we can conclude that as the type starts absorb later (from top to bottom in the table), the overall generation performance becomes better. Specifically, with $\tilde{T}$ for the late absorb decreases, the quality of the generated layouts gets worse (as suggested by mIoU, Align., and FID). When it comes to early absorb, the performance drops as $\tilde{T}'$ decreases. The experiment empirically supports the insight we discuss above.

D.3. Ablation Studies on Sequence Ordering

In the main paper, we sort the layout sequence according to the alphabetical order of the elements' type in the layout (denoted as "lexico"). Other choices are the positional ordering of the elements' bounding boxes (denoted as "position") or simply randomly sorting the elements (denoted as "random"). Besides, for each element in the layout, we represent its bounding box by the left, top, right, bottom coordinates (denoted as "ltrb"). One can also represent the bounding box using an element's left coordinate, top coordinate, width and height (denoted as "ltwh"). Here, we provide the results of all these options for sequence ordering.

As shown in the second group of Tab. 9, for the format representing the coordinates, the ltrb group exhibits better overall performance than the ltwh group. One explanation is that ltrb format may provide more straightforward information for the precise alignment of the bounding boxes. For the ordering of elements, the alphabetical ordering and positional ordering are slightly better than the random ordering. We hypothesize that the model can exploit the additional ordering information with positional embedding. However, note that for conditional generation, the positional ordering of the elements is unknown, so to enable both unconditional and conditional generation, we apply the alphabetical ordering to sort the elements.

E. Qualitative Results of LayoutDiffusion

In this section, we provide more generated samples covering three unconditional and conditional tasks on two datasets.

E.1. Unconditional Generation

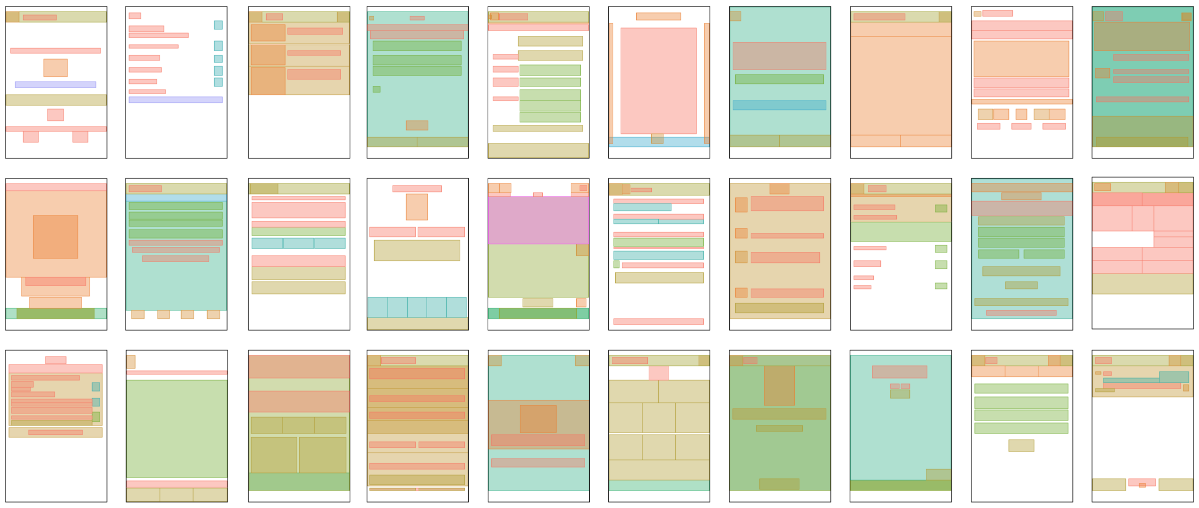

Figure 11. Examples of unconditional generation on RICO dataset.

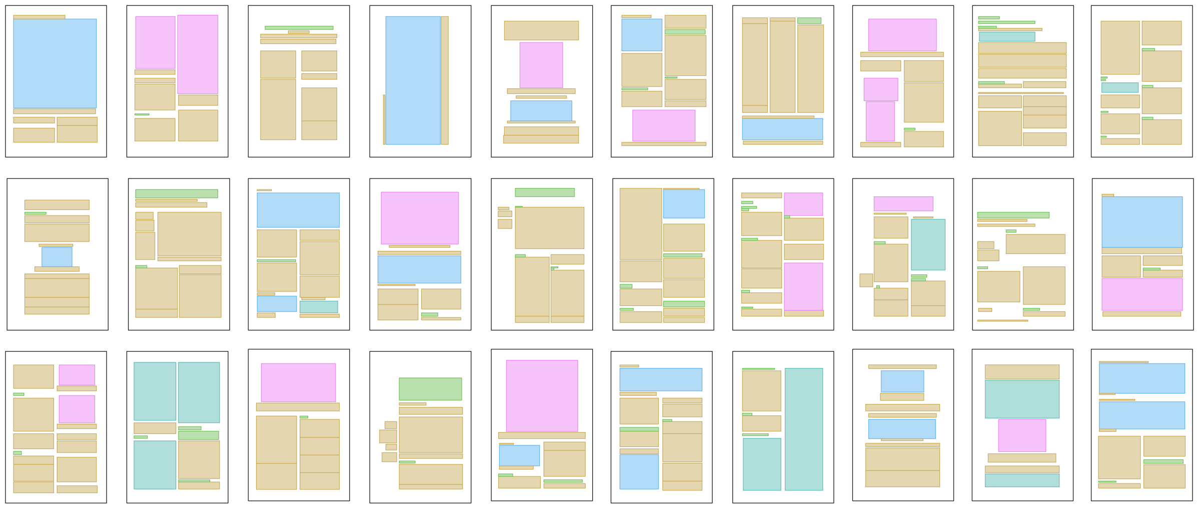

Figure 12. Examples of unconditional generation on PublayNet dataset.

E.2. Refinement

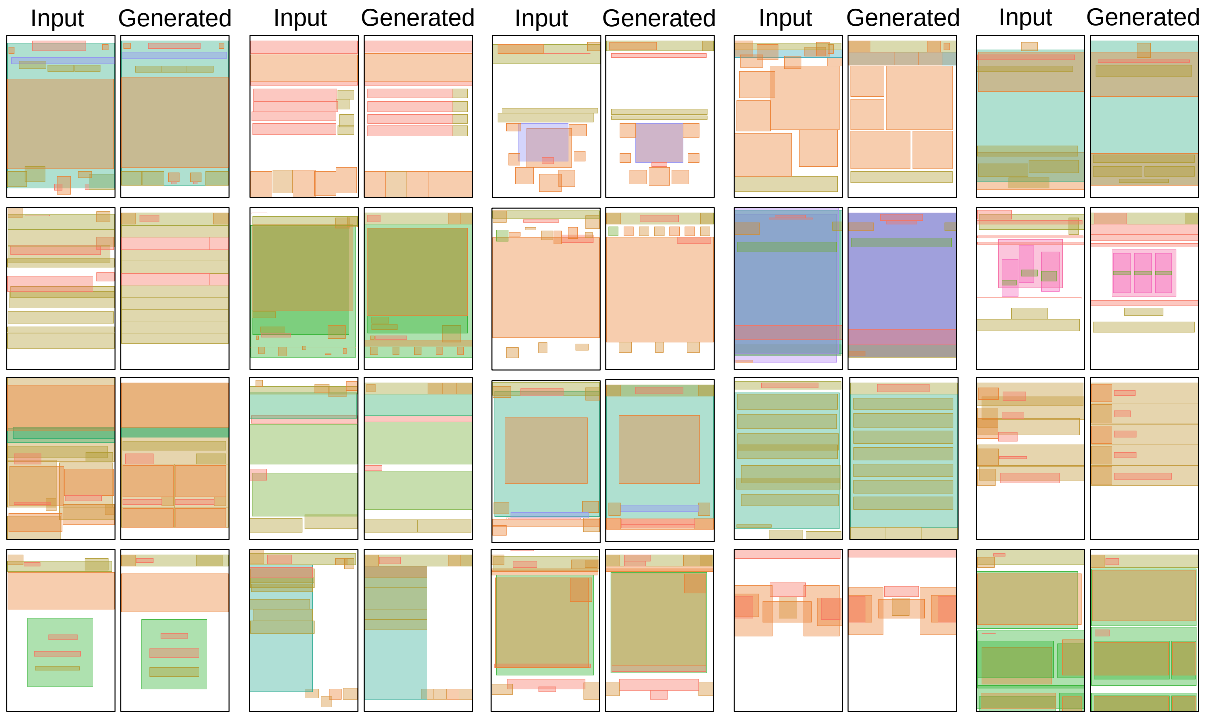

Figure 13. Examples of refinement on RICO. The left side of each pair is the input layout while the right side is the generated layout.

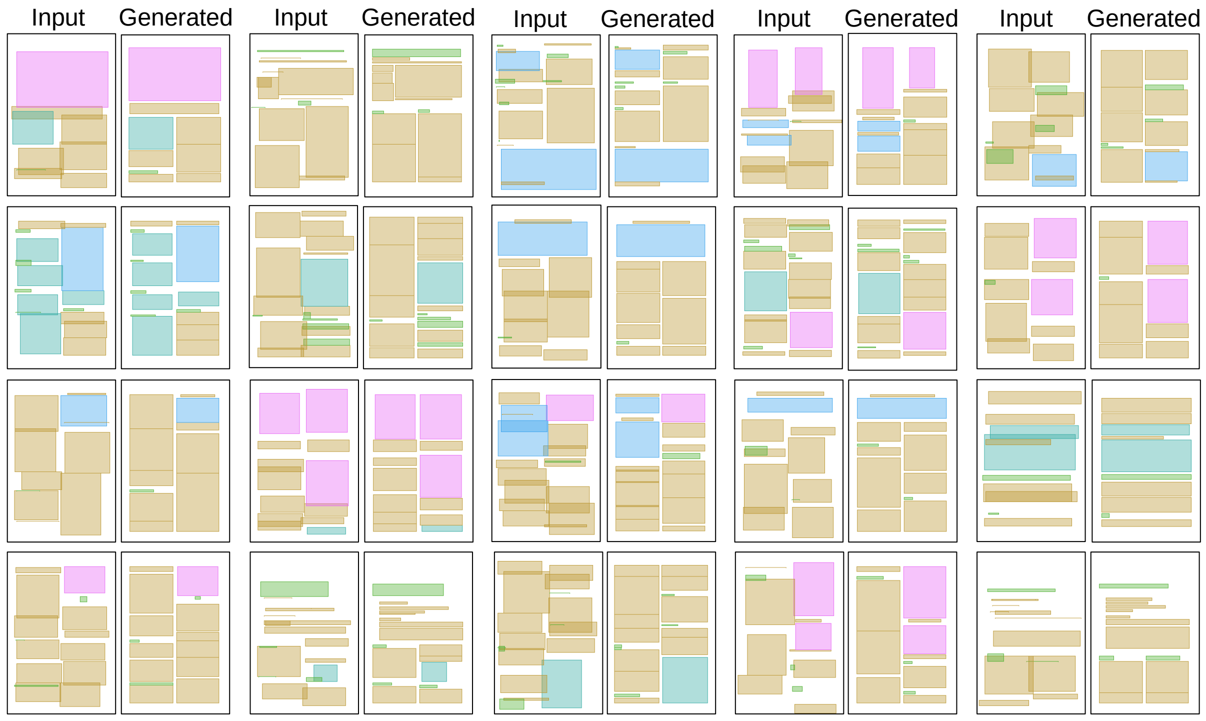

Figure 14. Examples of refinement on PublayNet. The left side of each pair is the input layout while the right side is the generated layout.

E.3. Generation Conditioned on Type

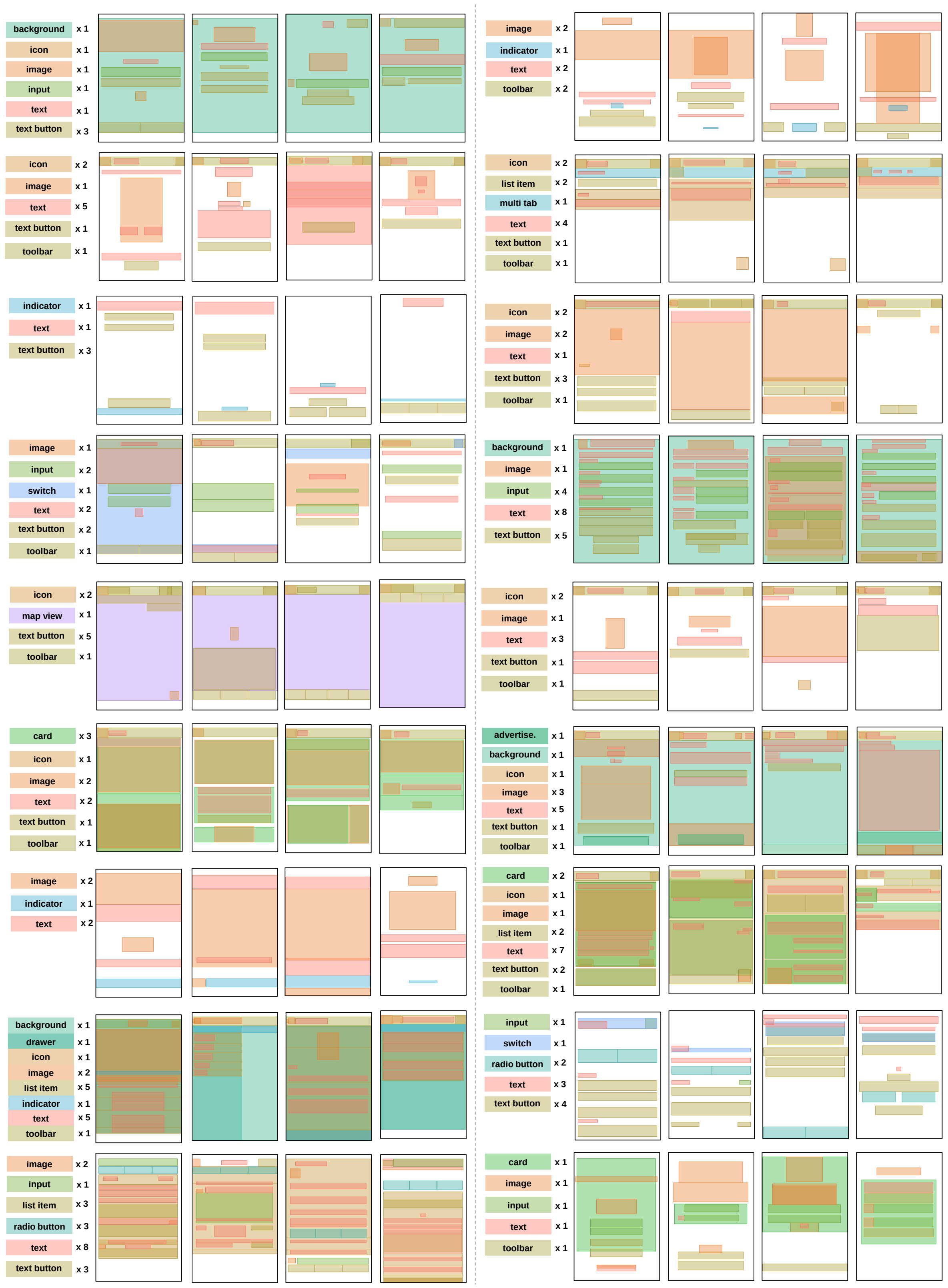

Figure 15. Examples of Gen-Type on RICO. Each given type set corresponds to four generated layouts.

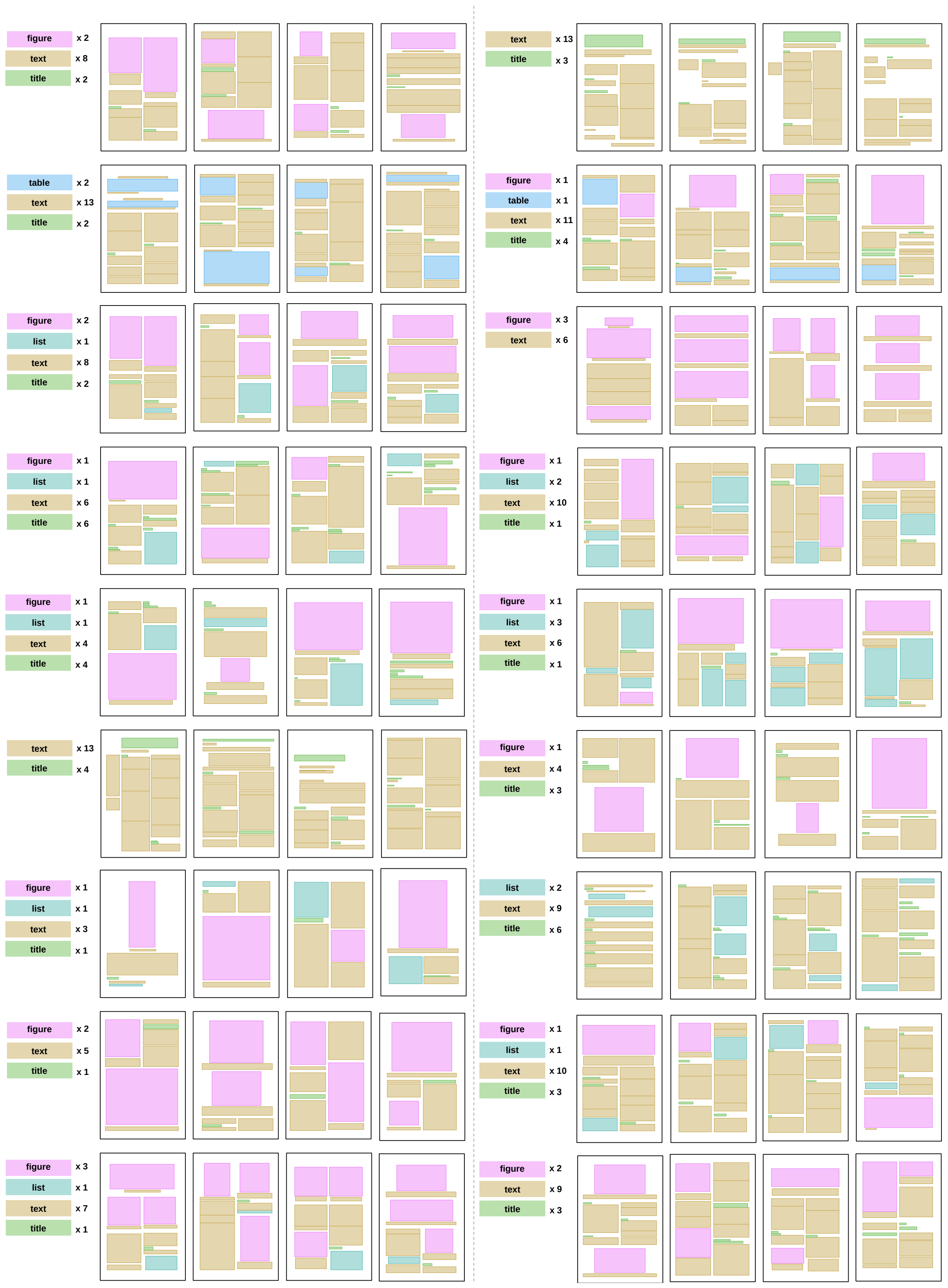

Figure 16. Examples of Gen-Type on PublayNet. Each given type set corresponds to four generated layouts.

F. Fine-Grained Visualization of the Forward Diffusion Process

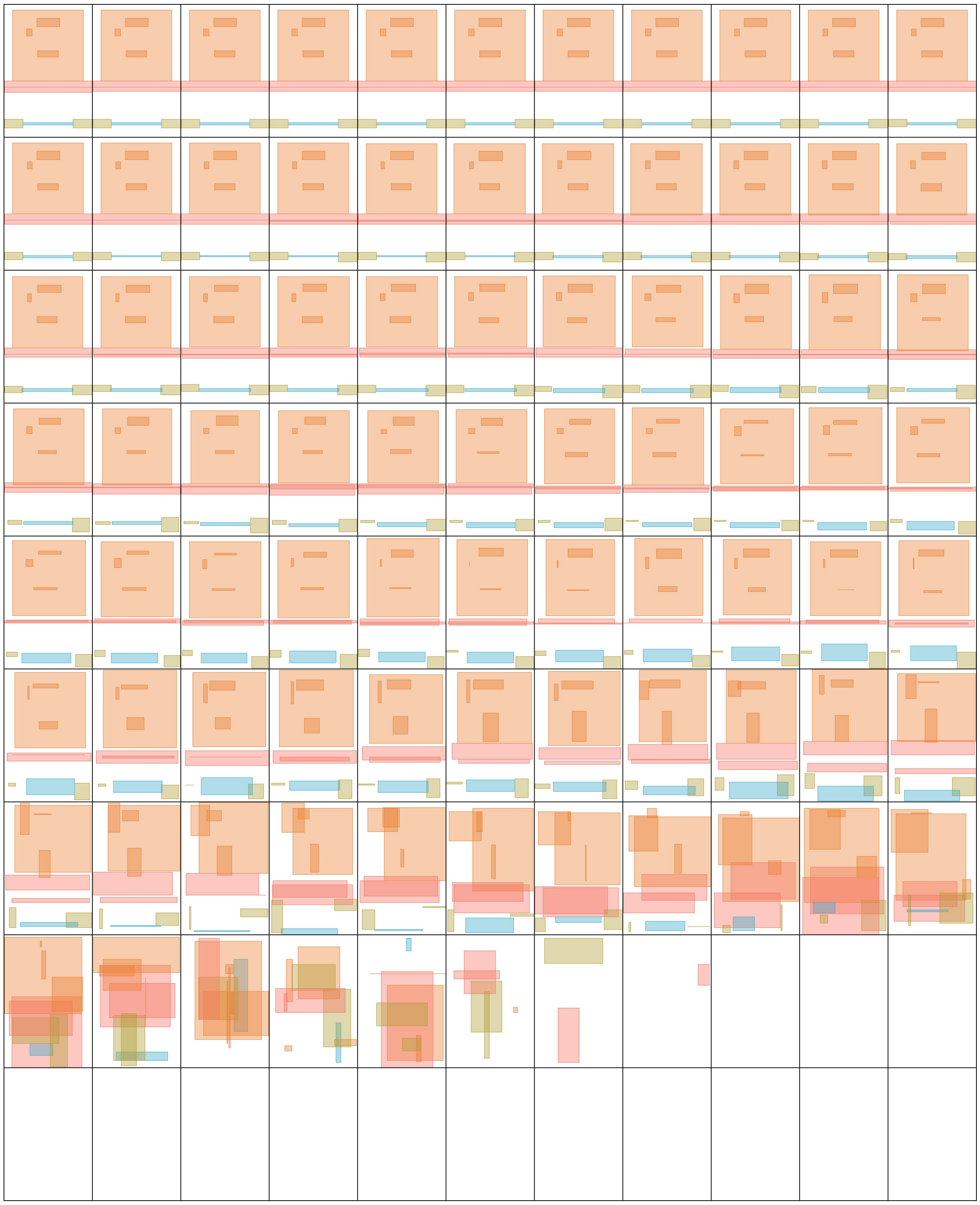

Figure 17. An example of the forward process of LayoutDiffusion on RICO. We sample 99 steps uniformly from the total 200 timesteps.

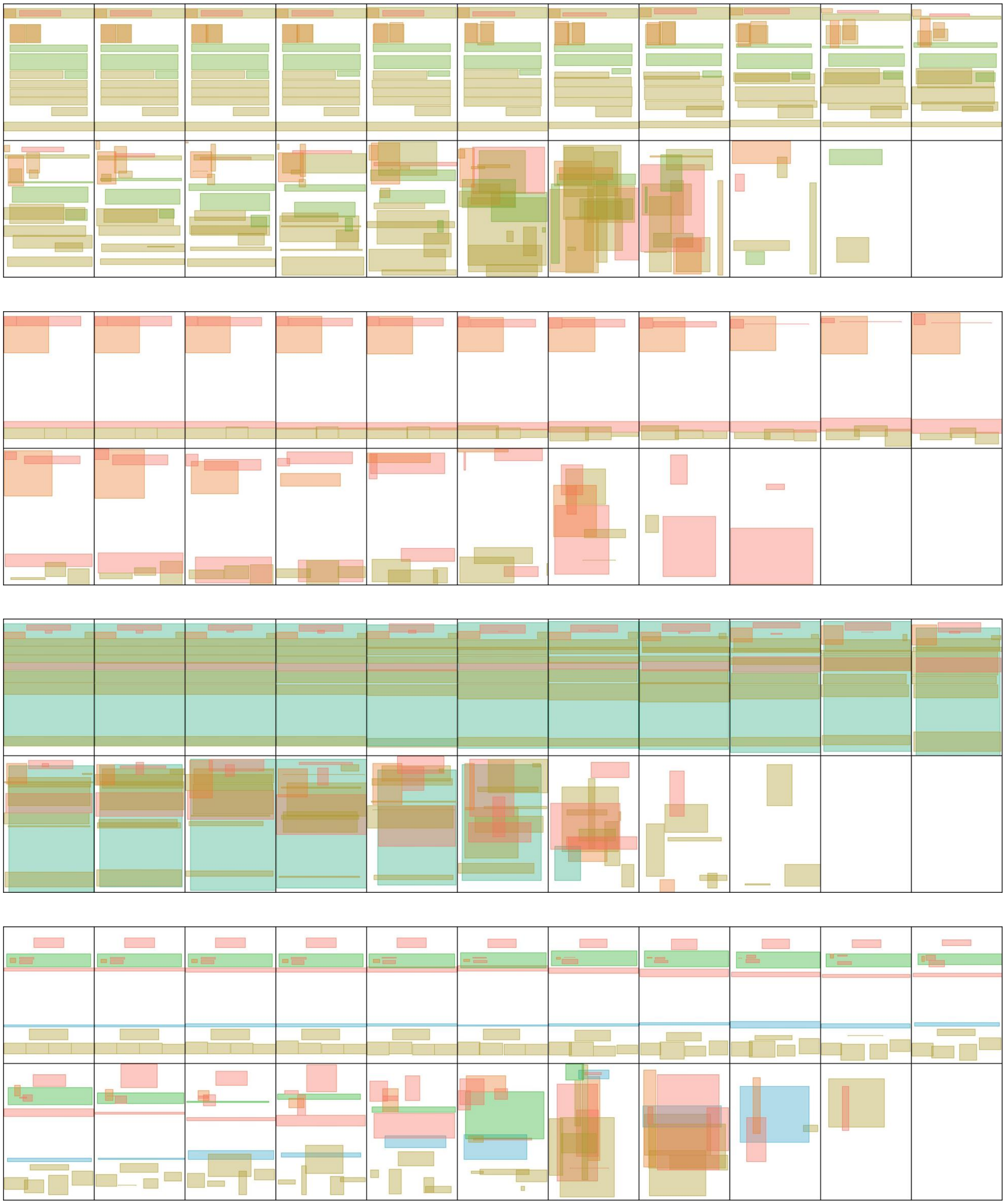

Figure 18. Examples of the forward process of LayoutDiffusion on RICO. We sample 22 steps uniformly from the total 200 timesteps.

G. Fine-Grained Visualization of the Reverse Generation Process

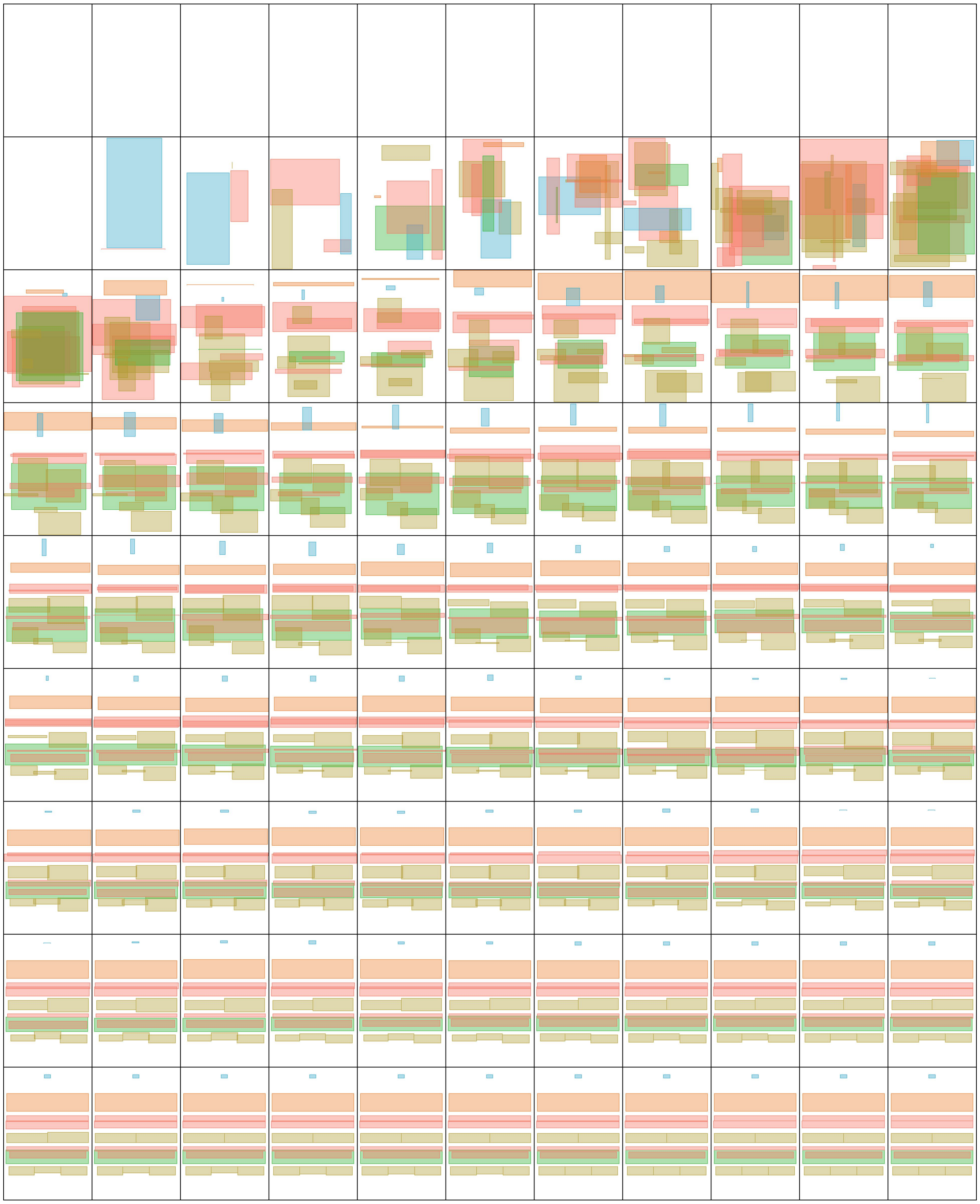

Figure 19. An example of the reverse process of LayoutDiffusion on RICO. We sample 99 steps uniformly from the total 200 timesteps.

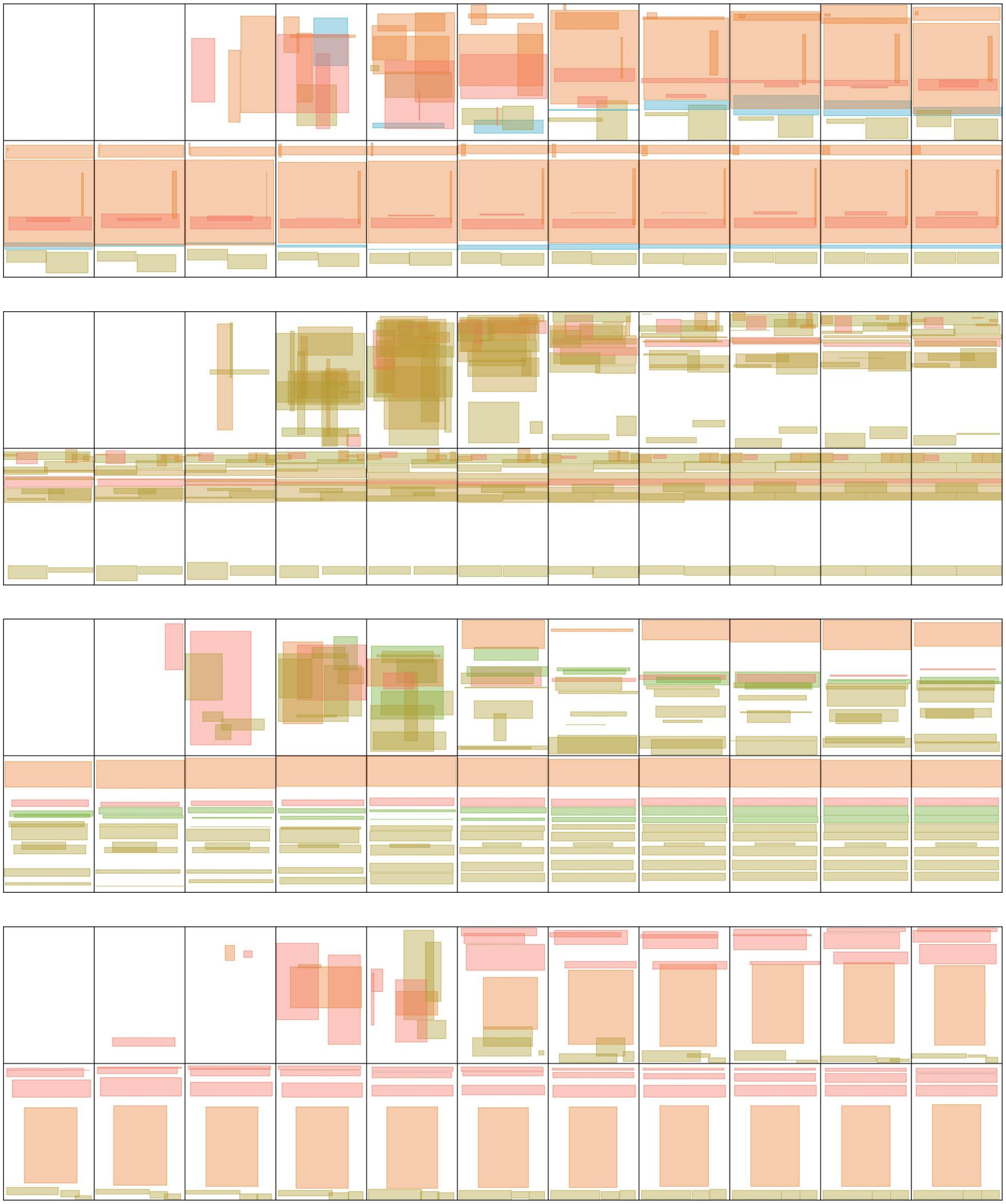

Figure 20. Examples of the reverse process of LayoutDiffusion on RICO. We sample 22 steps uniformly from the total 200 timesteps.

References

[1] Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems*, 34:17981–17993, 2021. 13, 14

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. 14

[3] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 845–854, 2017. 13

[4] Aditya Deshpande, Jyoti Aneja, Liwei Wang, Alexander G Schwing, and David Forsyth. Fast, diverse and accurate image captioning guided by part-of-speech. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10695–10704, 2019. 15

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 13

[6] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, Melbourne, Australia, July 2018. Association for Computational Linguistics. 17

[7] Zhaoyun Jiang, Jiaqi Guo, Shizhao Sun, Huayu Deng, Zhongkai Wu, Vuksan Mijovic, Zijiang James Yang, Jian-Guang Lou, and Dongmei Zhang. Layoutformer++: Conditional graphic layout generation via constraint serialization and decoding space restriction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18403–18412, 2023. 14, 17

[8] Kotaro Kikuchi, Edgar Simo-Serra, Mayu Otani, and Kota Yamaguchi. Constrained graphic layout generation via latent optimization. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 88–96, 2021. 14

[9] Xiang Kong, Lu Jiang, Huiwen Chang, Han Zhang, Yuan Hao, Haifeng Gong, and Irfan Essa. Blt: Bidirectional layout transformer for controllable layout generation. *arXiv preprint arXiv:2112.05112*, 2021. 14

[10] Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B Hashimoto. Diffusion-lm improves controllable text generation. *arXiv preprint arXiv:2205.14217*, 2022. 13

[11] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. 13

[12] Jekaterina Novikova, Ondřej Dušek, and Verena Rieser. The e2e dataset: New challenges for end-to-end generation. *arXiv preprint arXiv:1706.09254*, 2017. 13

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 14

[14] Soliha Rahman, Vinoth Pandian Sermuga Pandian, and Matthias Jarke. Ruite: Refining ui layout aesthetics using transformer encoder. In *26th International Conference on Intelligent User Interfaces-Companion*, pages 81–83, 2021. 14, 17

[15] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning*, pages 2256–2265. PMLR, 2015. 13

[16] Xu Zhong, Jianbin Tang, and Antonio Jimeno Yepes. Publaynet: largest dataset ever for document layout analysis. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 1015–1022. IEEE, 2019. 13

[17] Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. Texygen: A benchmarking platform for text generation models. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 1097–1100, 2018. 15