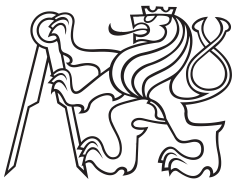
Bachelor Thesis



Czech
Technical
University
in Prague

F8

Faculty of Information Technology
Department of Software Engineering

StableGen:
Blender Plugin for AI-Based 3D Texturing

Ondřej Sakala

Supervisor: Ing. Radek Richtr, Ph.D.
Field of study: Informatics
Subfield: Computer Graphics
May 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	StableGen: blender plugin for AI-based 3D texturing
Student:	Ondřej Sakala
Supervisor:	Ing. Radek Richtr, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Graphics 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2026/2027

Instructions

Develop a Blender plugin for AI-based 3D texturing by leveraging existing AI frameworks. Evaluate both the technical quality of the generated textures and the plugin's usability compared to existing tools.

1) Conduct a research of latent diffusion 3D texturing methods and existing Blender based implementations.
2) Propose and prototype multiple approaches for generating consistent images from different angles and merging them to achieve seamless results.
3) Implement the most promising methods into a functional Blender plugin.
4) Perform testing of the developed approaches against each other and existing solutions.
5) Compare the plugin's features and usability against existing Blender texturing solutions.
6) Prepare and publish the plugin as an open-source project.

Electronically approved by Ing. Radek Richtr, Ph.D. on 18 February 2025 in Prague.

Acknowledgements

The journey to complete this thesis has been both challenging and rewarding, and I am deeply indebted to those who supported me along the way.

I extend my foremost gratitude to my supervisor, Ing. Radek Richtr, Ph.D. His expert direction, perceptive comments, and sustained encouragement were pivotal. His mentorship has not only guided this project but also contributed significantly to my academic development.

My heartfelt thanks also go to my family, whose unwavering support, patience, and understanding have been a constant pillar of strength. To my father, Ing. Michal Sakala, I offer particular thanks for his substantial and timely assistance, and for the insightful discussions that greatly benefited this work.

Their collective belief and help have been truly instrumental in bringing this thesis to fruition.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague, 16 May 2025

Abstract

This thesis presents StableGen, a Blender plugin designed to significantly enhance 3D texturing workflows and expand creative and production capabilities for artists. It enables the efficient creation of detailed and consistent textures across entire scenes by embedding state-of-the-art diffusion models directly within Blender. These are managed via a flexible ComfyUI backend, which allows the use of a wide range of custom models for achieving tailored artistic outcomes.

StableGen streamlines demanding tasks such as concept art, look development, and game asset prototyping. This is achieved through various multi-view generation strategies that effectively maintain the consistency of resulting textures. Diffusion models are guided by ControlNet, which ensures textures perfectly adhere to complex 3D surfaces, while IPAdapter, enabling the use of image references, serves as a powerful tool for controlling style and details. Thanks to its deep integration into Blender, the plugin offers immediate visualization. Combined with integrated tools for essential texture processing and baking, it substantially accelerates creative iteration. StableGen thus serves as a valuable addition to the 3D artist's toolkit, naturally complementing established production workflows by integrating advanced artificial intelligence.

Keywords: StableGen, Blender, generative AI, diffusion models, 3D texturing

Abstrakt

Tato práce představuje StableGen – rozšíření pro Blender navržené tak, aby výrazně zefektivnilo pracovní postupy při 3D texturování a rozšířilo tvůrčí i produkční možnosti umělců. Umožňuje snadno a konzistentně vytvářet detailní textury napříč celými scénami díky integraci špičkových difuzních modelů přímo do prostředí Blenderu. Tyto modely jsou řízeny flexibilním backendem ComfyUI, který umožňuje využívat vlastní modely k dosažení specifických výsledků.

StableGen usnadňuje náročné úkoly, jako je tvorba konceptů nebo prototypování herních modelů, a to díky strategii generování z více pohledů, která zajišťuje konzistenci výsledných textur. Difuzní modely jsou přesně naváděny pomocí ControlNetu, jenž zaručuje přesné přilnutí textur i ke složitým 3D povrchům. Funkce IPAdapter navíc umožňuje využití obrazových předloh při současné kontrole stylu a úrovně detailu. Díky plné integraci do prostředí Blenderu umožňuje StableGen okamžitou vizualizaci spolu s vestavěnými nástroji pro úpravy a zapékání textur, čímž výrazně zrychluje iterativní proces tvorby. StableGen tak představuje cenný nástroj pro 3D umělce, který přirozeně doplňuje zavedené produkční workflow a přináší výhody pokročilé umělé inteligence přímo do jejich každodenní praxe.

Klíčová slova: StableGen, Blender, generativní umělá inteligence, difúzní modely, 3D texturování

Překlad názvu: StableGen: Blender plugin pro AI 3D texturování

Contents

Figures

Tables

Introduction

The field of three-dimensional computer graphics plays an increasingly vital role across various industries, from entertainment and gaming to architecture and product design. A cornerstone of creating realistic and compelling 3D visuals is the process of texturing – applying surface detail, color, and material properties to 3D models. Traditionally, texturing has been a meticulous and often time-consuming task, requiring significant artistic skill and technical knowledge using specialized software. However, the rapid advancements in Artificial Intelligence, particularly in the domain of generative models capable of creating novel images from descriptions, are opening up new possibilities for automating and enhancing creative workflows, including 3D texturing.

While the potential for AI to revolutionize texturing is immense, harnessing this potential effectively presents its own set of challenges. Generating textures that are not only visually appealing but also consistent across the complex surface of a 3D object, controllable by the artist, and seamlessly integrated into existing 3D software environments like Blender remains a significant hurdle. Early explorations into AI texturing often involve complex setups, require substantial computational resources, or produce results that lack the coherence needed for professional use. There is a clear need for tools that bridge the gap between the power of state-of-the-art generative models and the practical needs of 3D artists.

This thesis addresses this need by exploring the application of modern AI image generation techniques, specifically those based on diffusion models, to the problem of 3D texturing directly within the Blender software environment. The core contribution of this thesis is the design, implementation, and evaluation of *StableGen*, a Blender plugin created to provide artists with an accessible and integrated tool for AI-powered texturing. The aim is to simplify the process of applying complex, AI-generated details to 3D models, making these advanced techniques more readily available within a standard production pipeline.

It is important to define the scope of this thesis. The primary focus is on the practical integration and adaptation of existing generative AI models and control mechanisms into a user-friendly plugin. This thesis does not aim

to develop novel core diffusion algorithms or fundamental generative model architectures, nor does it involve training large-scale AI models from scratch. Instead, it leverages readily available pre-trained models and established control frameworks. Furthermore, while various approaches to AI texturing exist, this thesis concentrates on implementing specific strategies deemed suitable for a plugin context and evaluating their effectiveness. The goal is to provide a functional tool within Blender, rather than an exhaustive academic survey of every possible AI texturing method or a standalone texturing application. The focus is on enabling artists within their familiar software, using techniques that balance quality, control, and accessibility.

Chapter 1

Objective

The principal aim of this thesis is the development and rigorous evaluation of StableGen, a Blender plugin for AI-facilitated 3D texturing based on extant AI frameworks, emphasizing both technical output quality and user utility.

The specific objectives guiding this work are:

1. To investigate contemporary latent diffusion texturing methodologies and existing Blender-integrated solutions.
2. To formulate and prototype multiple strategies for achieving multi-view textural consistency and seamless integration.
3. To implement the most viable methods within the functional StableGen plugin for Blender.
4. To systematically evaluate the implemented texturing methods based on output quality, comparing them internally and against established solutions.
5. To compare the feature set and operational usability of StableGen with existing Blender-based texturing solutions.
6. To prepare and disseminate the StableGen plugin as an open-source project.

This investigation seeks to provide a well-validated and accessible tool for AI-enhanced 3D texturing within a professional workflow.

Part I

Theoretical Part

Chapter 2

Background

2.1 Generative model

A **generative model** is a statistical model that learns the underlying probability distribution of a dataset. Let X denote the observed data (and, in supervised settings, Y denote the corresponding labels). A generative model aims to approximate the joint probability distribution $P(X, Y)$ such that new data samples $\tilde{X}$ can be generated by sampling from this learned distribution. These synthesized samples exhibit statistical properties similar to those in the original training data, preserving the complex patterns and structures present in real-world datasets [11]. This approach contrasts with *discriminative models*, which directly models the conditional probability $P(Y \mid X)$ to classify or predict outputs [24, 17].

Common examples of generative models include:

- **Variational Autoencoders (VAEs)** [19], which learn a latent representation of the data and generate new samples by decoding from this latent space.
- **Generative Adversarial Networks (GANs)** [12], where a generator network produces synthetic data that a discriminator network attempts to distinguish from real data.
- **Autoregressive models**, such as Transformer-based language models, which generate sequences by modeling the conditional probability of each element given its predecessors.
- **Diffusion Models** [14], which generate data by iteratively adding and then removing noise from an initial random sample, thereby modeling the data distribution.

These models are particularly useful for tasks such as image synthesis, text generation, and data augmentation, where the ability to produce new, realistic data samples is of significant value.

2.2 Diffusion model

A **diffusion model** [33] is a generative model that approximates the data distribution by simulating a gradual noising process followed by a learned reverse denoising process. In the forward process, a data sample x_0 is gradually corrupted by adding Gaussian noise over T time steps, ultimately producing a nearly isotropic Gaussian sample x_T [33]. A neural network is then trained to reverse this process by iteratively removing the noise, thereby recovering x_0 from x_T. The training objective is derived from a variational lower bound on the data likelihood [33].

In practice, this simplifies to training a neural network to predict the noise (ϵ) added at each step, rather than directly reconstructing $\mathbf{x}_0$. This ϵ-prediction formulation is central to the efficiency and effectiveness of diffusion models [14].

Diffusion models have demonstrated impressive results in image synthesis, generating high-quality and diverse samples. However, the iterative nature of the denoising process makes sampling computationally expensive, as each sample requires multiple forward passes through the model.

Forward Process (Diffusion)

In the forward process, an original data sample $\mathbf{x}_0$ (e.g., an image) is gradually transformed into pure noise over T time steps according to defined **variance schedule** $\{\beta_t\}_{t=1}^{T}$. At each step t ($t = 1, \ldots, T$), the process is defined as a Markov chain:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\Big(\mathbf{x}_t; \sqrt{1-\beta_t}\,\mathbf{x}_{t-1},\, \beta_t\,\mathbf{I}\Big)\,. \tag{2.1}$$

Here:

- $\mathbf{x}_t$ is the data (e.g., image) at time step t,
- β_t is the noise variance at time t determined by the variance schedule,
- $\mathbf{I}$ is the identity matrix (ensuring the noise is isotropic),
- $\mathcal{N}(\cdot; \mu, \Sigma)$ denotes a Gaussian distribution with mean μ and covariance Σ.

The variance schedule $\{\beta_t\}$ is chosen to gradually inject noise such that after T steps the original data is transformed into nearly pure Gaussian noise $\mathbf{x}_T$.

Reverse Process (Denoising)

The reverse process learns to approximate the data distribution by iteratively denoising $\mathbf{x}_t$. It is modeled as a Markov chain with Gaussian transitions [33]:

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)), \tag{2.2}$$

where:

- θ represents the parameters of the reverse (denoising) model,
- $\mu_\theta(\mathbf{x}_t, t)$ is the predicted mean for the denoised data at step t,
- $\Sigma_\theta(\mathbf{x}_t, t)$ is the covariance at step t

Ho et al. [14] employed a fixed variance schedule for $\Sigma_\theta(\mathbf{x}_t, t)$, simplifying the model's architecture and showing that setting $\Sigma_\theta(\mathbf{x}_t, t)$ to either β_t or $\tilde{\beta}_t$ yielded excellent results.

Training Objective

Training involves minimizing the variational bound on the negative log likelihood [33], which compares the learned reverse process to the true forward process:

$$L = \mathbb{E}_q \left[-\log p_\theta(\mathbf{x}_0) \right] \leq \mathbb{E}_q \left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \right], \tag{2.3}$$

where:

- L is the variational bound on negative log likelihood,
- $p_\theta(\mathbf{x}_{0:T})$ is the joint distribution of the reverse process,
- $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$ is the joint distribution of the forward process.

Ho et al. [14] proposed a simplified, unweighted loss function that directly trains the model to predict the noise ϵ at randomly sampled timesteps:

$$L_{simple} = \mathbb{E}_{t,\mathbf{x}_0,\epsilon}[\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2]. \tag{2.4}$$

This objective bypasses explicit covariance estimation and improves training efficiency.

Sampling

To generate data, the model starts with pure noise $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$ and iteratively applies the learned reverse process:

$$\mathbf{x}_{t-1} = \mu_\theta(\mathbf{x}_t, t) + \sqrt{\Sigma_\theta(\mathbf{x}_t, t)} \cdot \mathbf{z}, \quad \mathbf{z} \sim \mathcal{N}(0, \mathbf{I}). \tag{2.5}$$

The covariance $\Sigma_\theta(\mathbf{x}_t, t)$ is frequently derived from the noise schedule $\{\beta_t\}$ rather than learned, as this reduces computational overhead while maintaining sample quality [14].

2.2.1 Subsequent Advancements

The work of Ho et al. [14] significantly advanced diffusion models, demonstrating their ability to generate high-quality images comparable to GANs. Subsequent research has focused on improvements such as:

- **Sampling Efficiency:** Methods like DDIMs [34] and learning the variance schedule [25] have reduced the number of sampling steps.
- **Latent Diffusion Models (LDMs):** Operating in a latent space for improved efficiency on high-resolution data [28].

Diffusion models have found applications in image generation, audio synthesis, reinforcement learning, and computational biology, showcasing their versatility and power.

2.3 ControlNet

ControlNet is a recent extension to diffusion models that enhances controllability by incorporating additional conditioning inputs into the image synthesis process. While traditional diffusion models generate images by iteratively denoising a random sample based solely on a learned data distribution [14, 34], ControlNet introduces a dedicated neural branch that processes external control signals-such as edge maps, segmentation masks, keypoint sketches, or depth maps-to guide the reverse diffusion process. This additional conditioning mechanism allows for more precise and user-directed generation, making it possible to dictate the structural and spatial characteristics of the output. [39]

In the context of this thesis, ControlNet serves as a foundational technology. Its ability to condition the diffusion process on geometric information extracted from the 3D scene (such as depth maps, normal maps, or line art rendered from specific viewpoints) is leveraged extensively within the proposed Stablegen plugin. This allows the generated textures to be structurally

consistent with the underlying mesh geometry, which is crucial for achieving coherent 3D texturing results.

2.3.1 Architecture

ControlNet is designed to work in tandem with pre-trained diffusion models (e.g., Stable Diffusion). In practice, the weights of the original diffusion backbone are often kept frozen to preserve the high-quality generative performance. A parallel, trainable module-the ControlNet branch-is then integrated into each stage of the denoising network. This branch processes the control inputs and injects learned structural information into the generation pipeline, effectively steering the model toward outputs that satisfy both the learned distribution and the user-defined conditions [39].

2.3.2 Training and Sampling

During training, ControlNet is optimized using an objective similar to that of conventional diffusion models. The network is trained to predict the noise added during the forward process; however, it does so with access to both the noisy image and its associated control signal. This dual-input strategy enables the model to learn a conditional dependency between the control signal and the image content. At inference time, the control signal is applied at each denoising step, guiding the reverse diffusion process so that the synthesized image adheres to the desired structural layout while still maintaining the qualities of the diffusion model. [39, 14].

In practice, users can balance the influence of the textual prompt and of the controlnet by adjusting parameters such as the *guidance scale* (also referred to as *strenght* in some implementations) and the *start* and *end* parameters, which determine at which point of the denoising process to apply the guidance.

2.3.3 Advancements

Recent advancements have further expanded this framework.

- For example, **ControlNet++** [23] focuses on improving the quality and consistency of conditional image generation through efficient consistency feedback mechanisms.
- In **Cocktail: Mixing Multi-Modality Control for Text-Conditional Image Generation** [15] presented at NeurIPS 2023, the authors propose a pipeline to mix various modalities into one embedding, combined with a generalized ControlNet, controllable normalization, and spatial guidance for multi-modal and spatially refined control.

- **Uni-ControlNet** [40] unifies diverse conditioning mechanisms into one model, streamlining the training process and reducing computational demands. These innovations represent significant steps forward in controlled image synthesis, offering improved control over image generation while maintaining high fidelity.

2.3.4 Obtaining control signals

The control images used to guide the generation process can be obtained through various methods, depending on the desired control type and application:

- **Preprocessing of Existing Images:** Control signals can be derived from images using traditional algorithms (e.g., for edge and line detection) and specialized AI models (e.g., for semantic segmentation, depth/normal map and pose estimation, line extraction).
- **Direct Export from 3D Software (e.g., Blender):** 3D software is able to directly render specific control information such as *depth maps*, *normal maps*, *segmentation masks*, and *lineart*.

Additionally, users can directly create control images like *scribbles* or simple *segmentation masks* using image editing software to guide the generation.

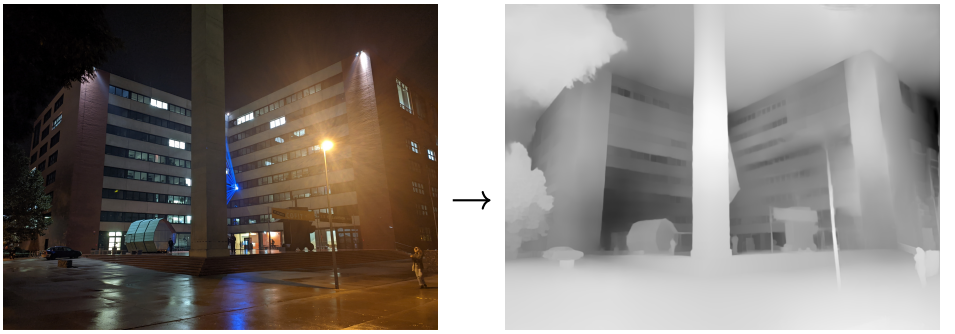
2.3.5 Application

In this section, we demonstrate several applications using different control types. These examples illustrate how structural information from various sources can guide the reverse diffusion process to generate images that adhere to certain kinds of control signals.

Depth Map Control

For an initial demonstration of ControlNet's capabilities, the depth control signal was selected due to its relevance to subsequent analyses within this thesis. In this instance, an RGB image was utilized as the source and subsequently processed into a depth map using a dedicated deep learning model.

A depth map (see Fig. 2.1) provides information about the spatial relationships in the scene. Using a depth map as conditioning input helps preserve perspective and three-dimensional structure in the generated image.



(a) : Source Image **(b) :** Depth Map (Processed)

(c) : Generated Image 1 **(d) :** Generated Image 2

Figure 2.1: Illustration of image generation guided by a depth map: The source image (a) is processed into a depth map (b). This depth map then conditions the diffusion model to produce new images (c and d) that maintain the original scene's spatial relationships and depth cues while adhering to new stylistic prompts.

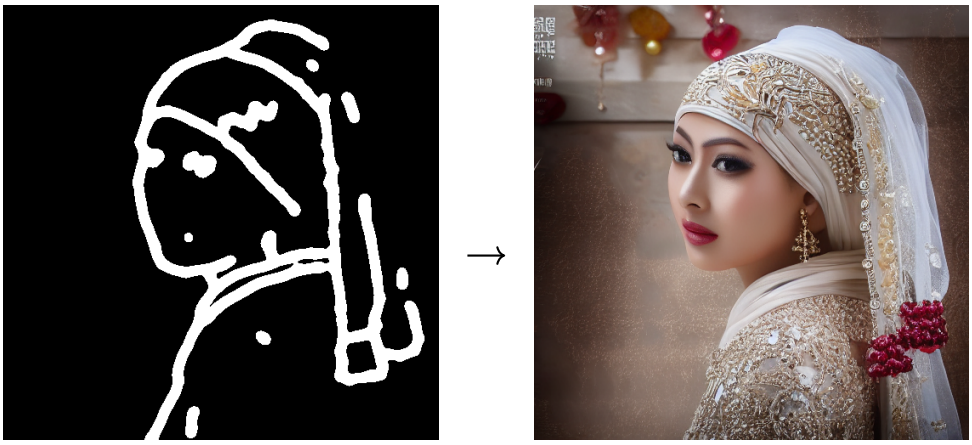
Line Art/Scribble Control

ControlNet can also convert simple sketches or line art into detailed, photorealistic images. This approach is useful for artistic applications, where a rough drawing is refined into a complete image.

OpenPose Control

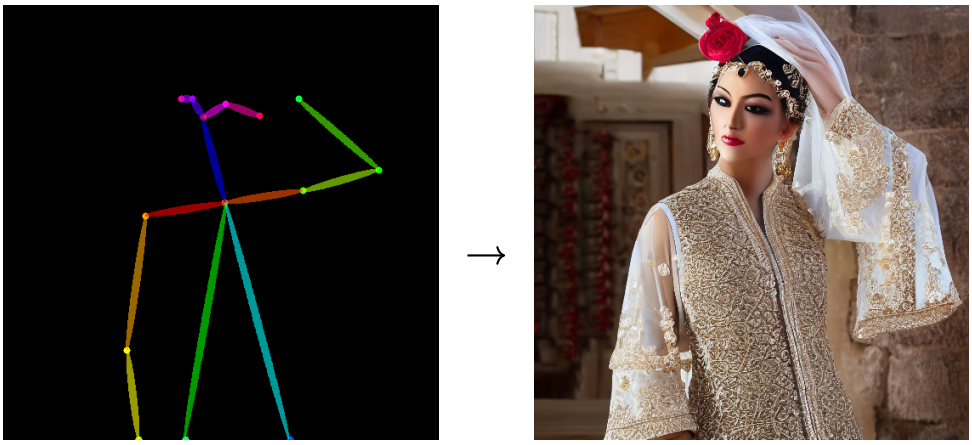
OpenPose control allows for guiding image generation based on the pose of one or more individuals. By providing a skeletal representation of the desired pose, ControlNet can generate images of people adhering to that specific posture.

These examples showcase the ability of ControlNet to utilize different forms of structural input to guide the image generation process.



(a) : Scribble Input **(b) :** Generated Output

Figure 2.2: Generation from a scribble input: (a) a basic sketch is transformed into (b) a refined artwork. Images adapted from ControlNet-Scribble examples [39].



(a) : OpenPose Input **(b) :** Generated Output

Figure 2.3: Process of image generation guided by OpenPose: (a) A skeletal representation dictates the pose of (b) the generated individual. Images adapted from ControlNet-OpenPose examples [39].

2.4 IP-Adapter

While text prompts are powerful, conveying complex visual concepts or specific styles through text alone can be challenging and often requires extensive prompt engineering. Image prompts offer a more direct way to provide visual guidance. However, previous methods for enabling image prompting in text-to-image diffusion models often involved computationally expensive full fine-tuning, which could compromise the model's original capabilities and compatibility, or used simpler adapter techniques that struggled to capture fine-grained details from the reference image.

To overcome these issues, IP-Adapter (Image Prompt Adapter) was introduced as an effective and lightweight adapter that enables pretrained text-to-image diffusion models to utilize image prompts. It achieves this while keeping the original diffusion model frozen, adding only a small number of trainable parameters (approx. 22M). [37]

In this thesis, IP-Adapter serves as a valuable tool integrated into the proposed texture generation workflows. It is primarily used to enforce stylistic consistency across multiple viewpoints, leveraging its ability to condition the generation process on a reference image representing the desired visual style or key features.

2.4.1 Architecture

The IP-Adapter architecture integrates image features into the diffusion process without modifying the pretrained model weights. [37] Key components include:

- **Image Encoder:** A frozen pretrained CLIP image encoder (e.g., OpenCLIP ViT-H/14) extracts semantic features from the input prompt image. Typically, the global image embedding is used.
- **Projection Network:** A small trainable network, consisting of a linear layer and Layer Normalization, projects the image embedding into a sequence of features suitable for the diffusion model's cross-attention mechanism.
- **Adapted UNet Modules with Decoupled Cross-Attention:** This is the core of IP-Adapter. Instead of feeding image features into the existing cross-attention layers designed for text, IP-Adapter adds *new* cross-attention layers specifically for image features alongside the original ones. These new layers share the query (Q) projections with the text attention layers but have their own trainable key (K') and value (V') projections. The outputs from the text attention and image attention layers are then combined (typically summed). This decoupled approach allows the model to process both modalities effectively.

2.4.2 Training and Inference

During training, only the parameters of the projection network and the new image cross-attention layers (K' and V' weights) are optimized, while the original diffusion model and image encoder remain frozen. The training objective is typically the same noise prediction loss used for the base diffusion model, conditioned on both text and image features. To enable classifier-free guidance, image conditions are randomly dropped during training. [37]

At inference time, the IP-Adapter allows for flexible control:

- **Image-Only Prompting:** Text prompts can be omitted (e.g., set to an empty string), allowing generation guided solely by the image prompt.
- **Multimodal Prompting:** Both image and text prompts can be used simultaneously, leveraging the decoupled cross-attention mechanism.
- **Guidance Scales:** Standard classifier-free guidance can be applied for text, and the influence of the image prompt can often be adjusted by scaling the output of the image attention layers (controlled by a weight factor λ) before combining it with the text attention output.

2.4.3 Key Advantages and Capabilities

IP-Adapter offers several significant advantages:

- **Lightweight:** It adds minimal parameters compared to the base model.
- **Compatibility:** Trained IP-Adapters can often be used with various custom models fine-tuned from the same base model, preserving the original model's capabilities. It is also compatible with structural control tools like ControlNet, allowing for combined image prompt and structural guidance.
- **Flexibility:** Supports image-only, text-only (by setting $\lambda = 0$), and multimodal prompting.
- **Performance:** Achieves comparable or better results than many fully fine-tuned image prompt models.

2.5 Texture Generation for 3D Meshes

Generating high-quality, controllable, and consistent textures for 3D meshes remains a significant challenge in computer graphics. The field is rapidly advancing, with recent research exploring diverse techniques to address this challenge. Various approaches, prominently including diffusion models, but

also Generative Adversarial Networks (GANs) and Neural Radiance Fields (NeRFs), are being actively developed to create detailed and realistic surface appearances.

Many of these innovative methods leverage common underlying strategies to achieve better results, such as utilizing multi-view information from the 3D model, employing depth conditioning to improve geometric consistency, or learning texture distributions from large datasets of images and shapes. This ongoing research promises to significantly enhance the creation of realistic 3D assets. This section categorizes and discusses several notable papers representing these various techniques, providing an overview of the current landscape in AI-driven 3D texture generation.

2.5.1 Diffusion Model Based Methods

Various papers leverage diffusion models as the core of their texture generation pipelines. Many of these methods utilize information from multiple viewpoints of the 3D model to ensure consistency and completeness of the generated textures.

Methods Using Multi-View Diffusion and Guided Generation

This category includes methods that generate textures by considering multiple views of the 3D shape and employing some form of guidance or masking during the diffusion process.

- **Text2Tex: Text-driven Texture Synthesis via Diffusion Models [7]**

  Text2Tex incorporates inpainting into a pre-trained depth-aware image diffusion model to progressively synthesize high-resolution partial textures from multiple viewpoints. To avoid accumulating inconsistent artifacts across views, it dynamically segments the rendered view into a *generation mask* (can be seen in Figure 2.5) that guides the inpainting model to generate and update textures for corresponding regions. An overview of the process is shown in Figure 2.4.

- **Paint3D: Paint Anything 3D with Lighting-Less Texture Diffusion Models [38]**

  Paint3D proposes a coarse-to-fine generative framework. It first uses a pre-trained depth-aware 2D diffusion model with ControlNet to generate view-conditional images and performs multi-view texture fusion to create an initial coarse texture map. Then, it refines this texture map using specialized UV Inpainting and UVHD diffusion models to handle incomplete areas and remove illumination artifacts, resulting in lighting-less

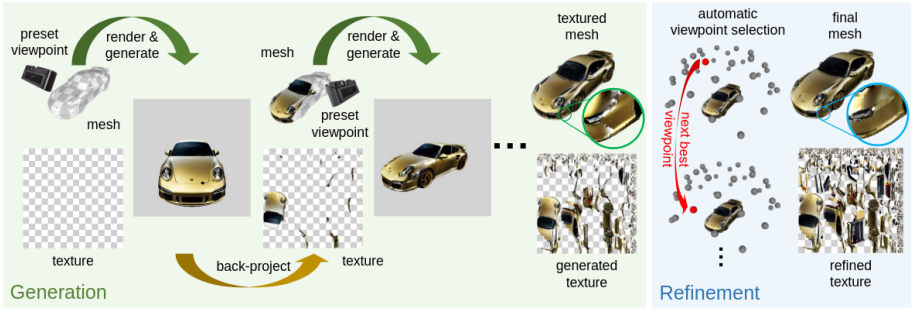


Figure 2.4: Text2Tex: Overview [7]

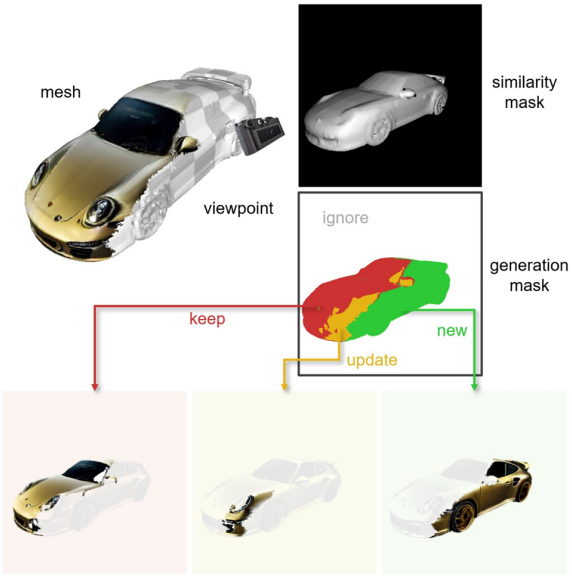


Figure 2.5: Text2Tex: Generation mask graphic [7]

textures. A graphic from the paper showing the texture generation and refinement can be seen in Figure 2.6.

- **TEXTure: Text-Guided Texturing of 3D Shapes [27]**

  TEXTure uses a pre-trained depth-to-image diffusion model and applies an iterative scheme to paint a 3D model from different viewpoints. To address inconsistencies, they dynamically define a *trimap partitioning* of the rendered image into progression states and introduce an elaborated diffusion sampling process that uses this trimap representation to generate seamless textures from different views. The trimap serves a similar role to the generation mask in Text2Tex by guiding the diffusion process based on the current state of the texture. The general process of texture creation using TEXTure can be seen in Figure 2.7.

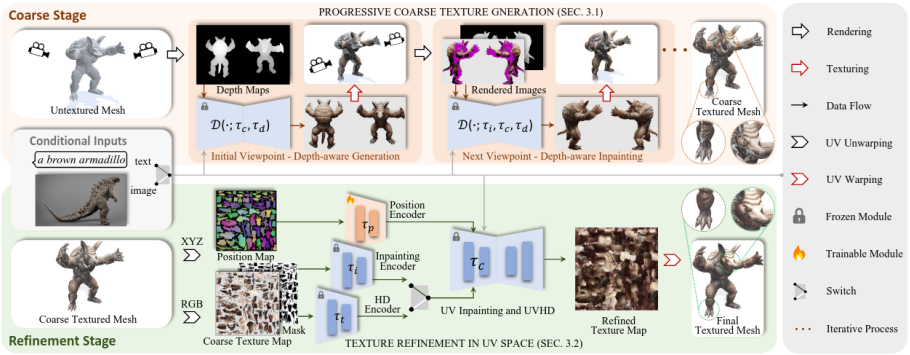


Figure 2.6: Paint3D: Gallery of generated texture results using their coarse-to-fine generative framework [38].

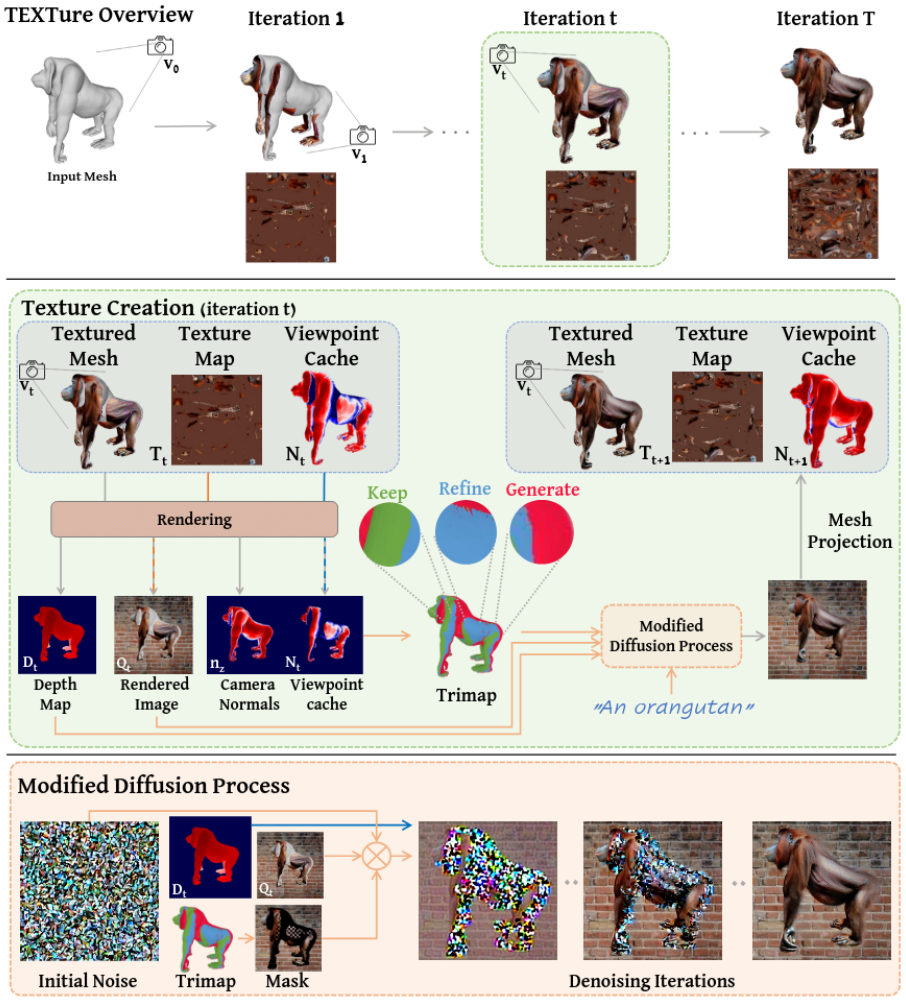


Figure 2.7: TEXTure: Overview [27]

Methods Using Depth-Conditioned Diffusion

- **EucliDreamer: Fast and High-Quality Texturing for 3D Models with Depth-Conditioned Stable Diffusion [21]**

  EucliDreamer introduces a method to generate textures for 3D models from text prompts using a depth-conditioned Stable Diffusion model. The texture is represented as an implicit function on the 3D surface and optimized using Score Distillation Sampling (SDS) and differentiable rendering. Depth conditioning is used to improve texture quality and speed up convergence by removing ambiguities and enhancing cross-view consistency. A sample visualisation from the paper is provided in Figure 2.8.

Figure 2.8: EucliDreamer: 3D objects textured using depth-conditioned Stable Diffusion from text prompts [21].

- **MatAtlas: Text-driven Consistent Geometry Texturing and Material Assignment [6]**

  MatAtlas presents a method for consistent text-guided 3D model texturing that also uses a form of depth conditioning. Their RGB texturing pipeline leverages a grid pattern diffusion, driven by depth and edges. They also propose a multi-step texture refinement process to enhance the quality and 3D consistency of the output. A visualisation off a generated texture can be seen in Figure 2.9.

2.5.2 GAN-Based Methods

Generative Adversarial Networks (GANs) offer another powerful framework for generative modeling. In the context of texture generation, GANs typically involve training a generator network to produce realistic textures that can fool a discriminator network trained to distinguish between generated and real textures.

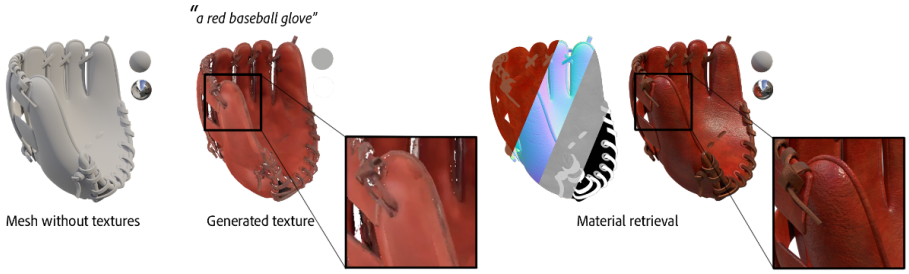


Figure 2.9: MatAtlas: Overview of their method for consistent text-guided 3D model texturing and material assignment [6].

- **texturify: Generating Textures on 3D Shape Surfaces [31]**

  Texturify is a GAN-based method that learns to generate geometry-aware textures for untextured collections of 3D objects. It trains from a collection of images and untextured shapes without requiring explicit 3D color supervision or shape-image correspondence. A graphic representing the process is shown in Figure 2.10. Textures are created directly on the surface of a given 3D shape using face convolutional operators on a hierarchical 4-RoSy parameterization, employing differentiable rendering and adversarial losses to match the distribution of real image observations. This approach, as suggested by its evaluation on car and chair datasets, tends to specialize in the categories of objects it is trained on. An overview graphic from the paper is provided in Figure 2.11.

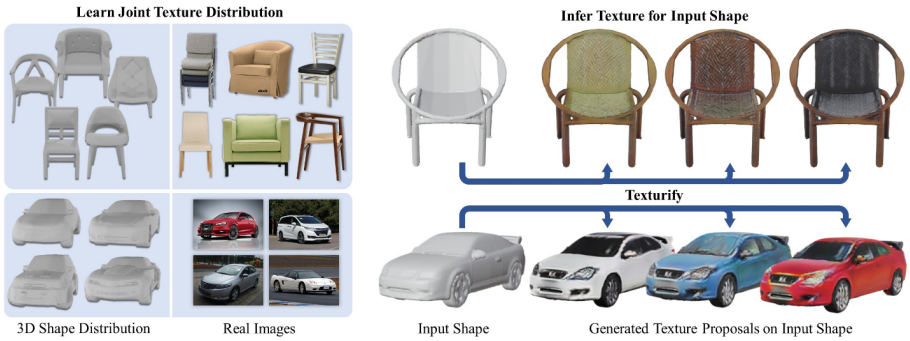


Figure 2.10: Texturify: Learning joint texture distribution and inferring texture for an input shape using a GAN-based approach [31].

2.5.3 Methods Using Neural Radiance Fields

Neural Radiance Fields (NeRFs) have emerged as a compelling approach for representing 3D scenes from a collection of 2D images. While primarily used for novel view synthesis, NeRFs can also be adapted for texture synthesis by learning a continuous volumetric representation of the object's appearance.

- **NeRF-Texture: Texture Synthesis with Neural Radiance Fields [16]**

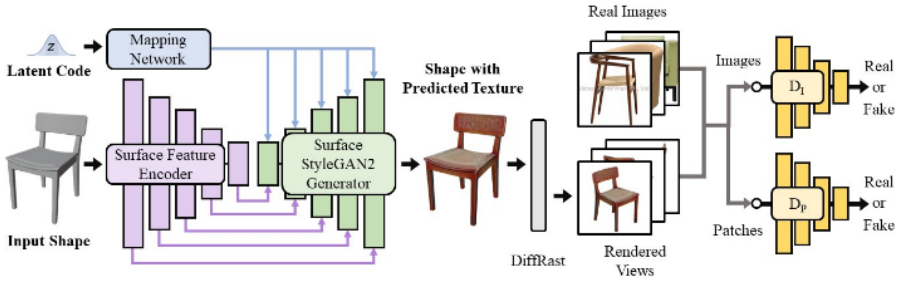


Figure 2.11: Texturify: Overview [31]

NeRF-Texture proposes a novel texture synthesis method using Neural Radiance Fields (NeRF) to capture and synthesize textures from multi-view images. It disentangles a scene with fine geometric details into meso-structure textures and an underlying base shape. This allows textures with meso-structure and view-dependent appearance to be effectively learned as latent features situated on the base shape, which are then fed into a NeRF decoder. A graphic showing a synthetized texture on different objects can be seen in Figure 2.12.

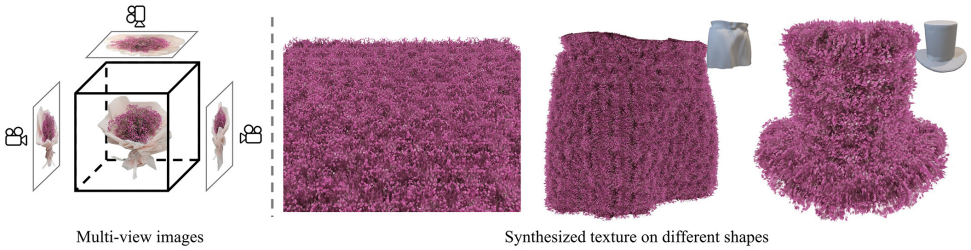


Figure 2.12: NeRF-Texture: Synthesizing textures on different shapes using Neural Radiance Fields from multi-view images [16]

2.5.4 Methods Learning Texture Manifolds

Learning a texture manifold involves capturing the underlying distribution of textures for a specific category of objects. This allows for generating novel textures by sampling from the learned manifold, often conditioned on input geometry or other cues.

- **Mesh2Tex: Generating Mesh Textures from Image Queries [3]**

  Mesh2Tex learns a realistic object texture manifold from uncorrelated collections of 3D object geometry and photorealistic RGB images. It leverages a hybrid mesh-neural-field texture representation that supports high-resolution texture generation on various shape meshes. The learned texture manifold enables effective navigation to generate an object texture for a given 3D object geometry that matches an input RGB image, with robustness to inexact geometry. This approach is category-specific,

meaning the model learns a texture manifold for a particular category of objects based on the training data it receives. A showcase graphic from the paper can be seen in Figure 2.13 and the generation process is visualized in Figure 2.14.

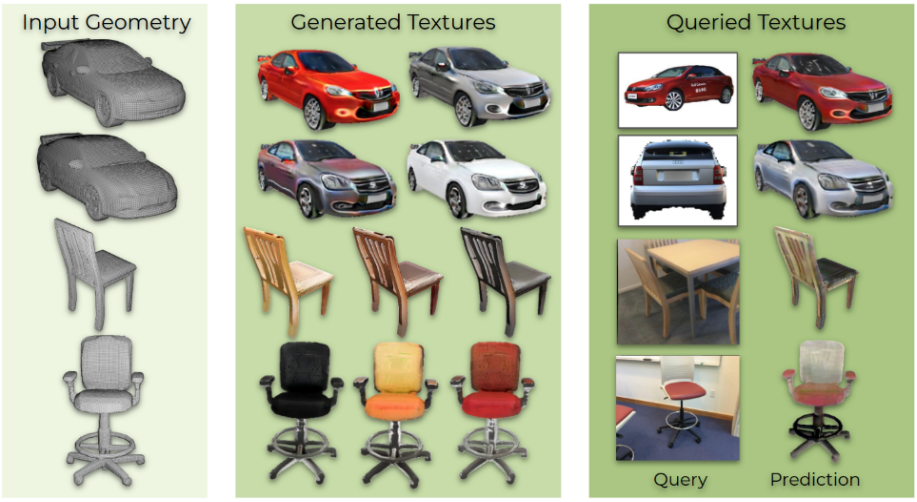


Figure 2.13: Mesh2Tex: Generating mesh textures from image queries by learning a realistic object texture manifold [3].

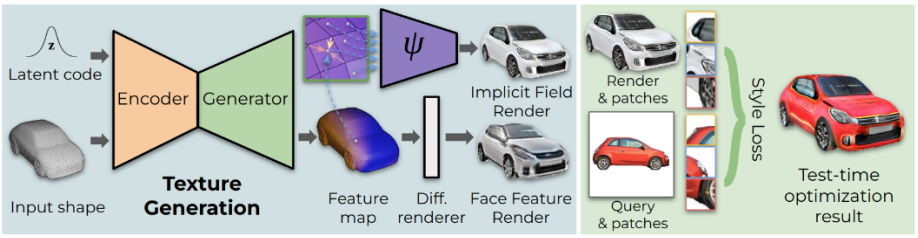


Figure 2.14: Mesh2Tex: Texture generation overview [3]

Chapter 3

Analysis

3.1 Comparative Evaluation of Texture Generation Categories

In this section, we aim to provide a comparative analysis of several prominent categories of texture generation methods for 3D meshes, considering their suitability for the goals of this project.

3.1.1 Diffusion Model Based Methods

Diffusion models have shown remarkable success in generative tasks, including texture synthesis [21], and offer several compelling advantages. A key benefit is the broad availability of pre-trained models like Stable Diffusion XL and FLUX.1, which can often be used for inference on relatively affordable hardware. Furthermore, a large and active ecosystem of techniques such as IPAdapter and ControlNet provides users with significant control and flexibility in the generation process. The existence of numerous specialized models and LoRAs fine-tuned for specific styles or object types further enhances their versatility. However, it's important to note that many diffusion-based texturing methods inherently operate in 2D, requiring strategies to maintain consistency across different views of the 3D model [38, 27]. Generating textures for non-convex shapes can also present challenges, often necessitating additional steps to handle potentially occluded or invisible areas [6, 7].

3.1.2 Methods Using Radiance Fields (NeRFs)

Neural Radiance Fields (NeRFs) excel at synthesizing novel views of complex 3D scenes with high fidelity [16]. Traditionally, NeRFs have been more suited for representing scenes with relatively homogeneous material properties. However, recent advancements have explored *promptable NeRFs*, allowing for some level of user control through text or image prompts, and enabling object manipulation within the scene. While this offers exciting possibilities, the

application of NeRFs to texturing entire scenes with multiple distinct objects and materials might still present complexities compared to other methods. A key advantage of NeRFs lies in their ability to create view-consistent representations and handle intricate geometric details.

3.1.3 Manifold-Based Methods

Approaches based on learning texture manifolds aim to capture the underlying distribution of textures for specific categories of objects [3]. These methods can learn realistic and diverse texture patterns from data and, as research suggests, can also be influenced by user prompts to generate novel textures within the learned manifold. While offering the potential for high-quality, category-specific texture generation, the inherent focus on specific categories might limit their direct applicability to a project aiming for a universal texturing solution across diverse and potentially novel objects within a scene. Training individual manifolds for a wide range of object types could be a significant undertaking.

3.1.4 GAN-Based Methods

Generative Adversarial Networks (GANs) offer a powerful framework for learning complex data distributions and have been successfully applied to texture synthesis, capable of generating diverse and high-resolution textures [31]. While the ecosystem and availability of pre-trained models are not as widespread as in the diffusion model space, GANs remain a viable option. However, training GANs can sometimes be challenging [29] and requires careful tuning to achieve stable and high-quality results for 3D texture generation, especially in ensuring consistency across different viewpoints. Similar to manifold-based methods, GAN approaches like Texturify tend to specialize in the categories of objects they are trained on.

3.1.5 Comparison Summary

Based on the comparative analysis presented in Tables 3.1-3.5, several key insights emerge for our project goals of developing an accessible, flexible texture generation system for diverse 3D scenes.

3.1.6 Method Choice

Diffusion-based methods offer the most compelling combination of advantages, with *very high* model availability and control flexibility, making them particularly suitable for our accessibility requirements. While GANs provide fast generation times after training, their training instability [29] and less mature

Table 3.1: Comparison of advantages for texture generation methods

Method	Advantages
Diffusion Models	High-quality results, large and active ecosystem, relatively accessible inference, significant control.
GAN-Based Methods	Generate realistic and detailed textures, fast generation after training, learn complex distributions.
Manifold-Based Methods	Learn realistic textures, potential for diversity within categories, adaptable for texture transfer.
Radiance Fields (NeRFs)	Excellent view synthesis, captures fine details, view-consistent, increasingly promptable.

Table 3.2: Comparison of disadvantages for texture generation methods

Method	Disadvantages
Diffusion Models	High training cost, 2D-based requiring consistency, challenges with non-convex shapes.
GAN-Based Methods	Training instability, less mature ecosystem, difficulty in view consistency, specialization in object categories.
Manifold-Based Methods	Category-specific, requires large datasets per category, struggles with novel textures.
Radiance Fields (NeRFs)	High training and rendering cost, traditionally for single objects, challenging for diverse scenes.

Table 3.3: Comparison of model availability for texture generation methods

Method	Model Availability
Diffusion Models	**Very high**. Numerous pre-trained models, LoRAs widely available.
GAN-Based Methods	**Moderate**, training often requires significant resources and expertise, potential specialization.
Manifold-Based Methods	**Moderate**, often specific to object categories.
Radiance Fields (NeRFs)	**Moderate** and growing, increasing research and availability for specific applications.

Table 3.4: Comparison of control & flexibility for texture generation methods

Method	Control & Flexibility
Diffusion Models	**Very high** through prompting, image guidance, spatial control, and model combinations.
GAN-Based Methods	**Moderate** through prompting, but precise control can be challenging.
Manifold-Based Methods	**Moderate**, primarily guided by input within the learned manifold, limited for novel styles.
Radiance Fields (NeRFs)	Increasing through prompt engineering and editing techniques, but less direct for detailed texture control.

Table 3.5: Comparison of universality for texture generation methods

Method	Universality
Diffusion Models	**High** potential for various objects and scenes with consistency techniques.
GAN-Based Methods	**Moderate**, challenging for diverse scenes with different material properties, tends to specialize in trained object categories.
Manifold-Based Methods	**Low to Moderate**, primarily for specific object categories.
Radiance Fields (NeRFs)	**Low**, more for individual scenes/objects, diverse scene texturing is an active research area.

ecosystem, along with a tendency for specialization [31], present implementation challenges. Manifold-based approaches, though effective for specific categories [3], lack the universality needed for diverse scene texturing. NeRFs excel at view consistency and fine detail rendering [16], but their traditionally single-object focus and high computational demands conflict with our goal of a lightweight, broadly applicable solution. Given these considerations, our implementation will primarily leverage diffusion models [21, 38, 7, 27, 6], while incorporating supplementary techniques to address their inherent challenges, such as ensuring multi-view consistency, mitigating baked-in lighting effects, enabling finer control over details, and potentially handling more complex material properties beyond color.

3.2 Analyzing Issues Regarding Diffusion-Based Approach

This section analyzes issues regarding the diffusion-based approach for 3D texture generation, focusing on existing methods employed to overcome these.

3.2.1 Multi-View Consistency

Definition

Ensuring that textures generated from multiple viewpoints of a 3D model are consistent and seamless is a critical challenge.

Existing Solutions

Several diffusion-based methods address the multi-view consistency issue through different strategies. **Text2Tex** [7] incorporates inpainting guided by a dynamically generated mask to update corresponding regions across views, thus maintaining consistency. **Paint3D** [38] utilizes a coarse-to-fine framework that begins with multi-view texture fusion from view-conditional images and refines the result using UV inpainting. **TEXTure** [27] employs an iterative painting scheme guided by a dynamically defined trimap partitioning of the rendered image, ensuring seamless texture generation from different viewpoints.

3.2.2 Handling Non-Convex Shapes and Occlusions

Definition

Generating complete and coherent textures for 3D models with complex, non-convex shapes and potential occlusions poses another significant challenge.

Existing Solutions

MatAtlas [6] addresses the challenge of handling non-convex shapes and occlusions by using a grid pattern diffusion conditioned on depth and edges, which helps in generating textures consistent with the underlying geometry. The depth-aware inpainting technique used in **Text2Tex** [7] also aids in inferring and generating plausible textures for partially occluded areas.

3.2.3 Mitigating Baked-in Lighting

Definition

Baked-in lighting effects, often present in training data, can limit the relightability of generated textures.

Existing Solutions

Paint3D [38] specifically aims to generate lighting-less textures through its coarse-to-fine framework and specialized UV diffusion models designed to remove illumination artifacts. **EucliDreamer** [21] explores the use of negative prompts during the diffusion process to suppress unwanted shadows and highlights, thereby reducing the impact of baked-in lighting.

3.2.4 Control over Fine Details and Specific Features

Definition

While diffusion models excel at generating realistic textures, achieving precise control over fine details or specific features can be challenging when relying solely on text prompts.

Existing Solutions

Some methods explore incorporating additional conditioning mechanisms to address the challenge of control over fine details. For instance, the use of **image guidance** in methods like Paint3D [38] allows users to provide reference images to influence the style and details of the generated texture. **TEXTure** [27] demonstrates the ability to perform localized scribble-based editing, offering a way to refine specific areas of the texture. Furthermore, the development of techniques like ControlNet in the broader diffusion model ecosystem provides mechanisms for spatial control over the generated content, which could be adapted for precise texture placement and feature manipulation on 3D models.

3.2.5 Handling of Complex Material Properties

Definition

The majority of diffusion-based texture generation methods primarily focus on producing RGB color textures. However, real-world objects exhibit complex

material properties beyond color, such as specular highlights, normals, and roughness, which are crucial for realistic rendering.

Existing Solutions

While some methods like **MatAtlas** [6] explore assigning parametric materials based on the generated RGB texture using Large Language Models, directly generating these complex material maps using diffusion models is a more intricate task. Future work may involve developing specialized diffusion model architectures or conditioning strategies that can output multiple texture maps representing different material properties, potentially guided by material-specific prompts or data.

Part II

Practical Part

Chapter 4

System Design

The proposed solution aims to facilitate the integration of AI-driven texture generation within the Blender environment.

4.1 Blender Plugin Architecture

The Stablegen addon is implemented following standard Blender addon practices, utilizing Python and the `bpy` API. Architecturally, it is designed with a modular approach to separate distinct areas of functionality, enhancing maintainability and clarity. The key components of this architecture, illustrated in Figure 4.1, include:

- **User Interface Component:** Responsible for rendering the plugin's controls within Blender's N-panel. It handles the layout definition, presentation of parameters using Blender's UI widgets, management of user interaction (button clicks, value changes), and the logic for loading, applying, and saving configuration presets.
- **Core Operators:** These are the primary entry points for user-initiated actions, implemented as `bpy.types.Operator` classes. Key operators manage the main texture generation process (handling state, progress feedback, and asynchronous execution), texture baking, preset management, and utility functions like camera setup. The main generation operator, in particular, orchestrates calls to various other components.
- **Properties and Configuration Management:** Plugin settings, user parameters, and operational state are stored using Blender's property system (`bpy.props.*`). Most properties are attached to Blender's `Scene` data block for persistence within `.blend` files and easy binding to UI elements. Structured data, like ControlNet unit configurations, utilizes `bpy.types.PropertyGroup`. Addon preferences store global paths and settings.
- **Backend Communication Layer:** A dedicated component handles all interaction with the external ComfyUI backend. Its responsibilities

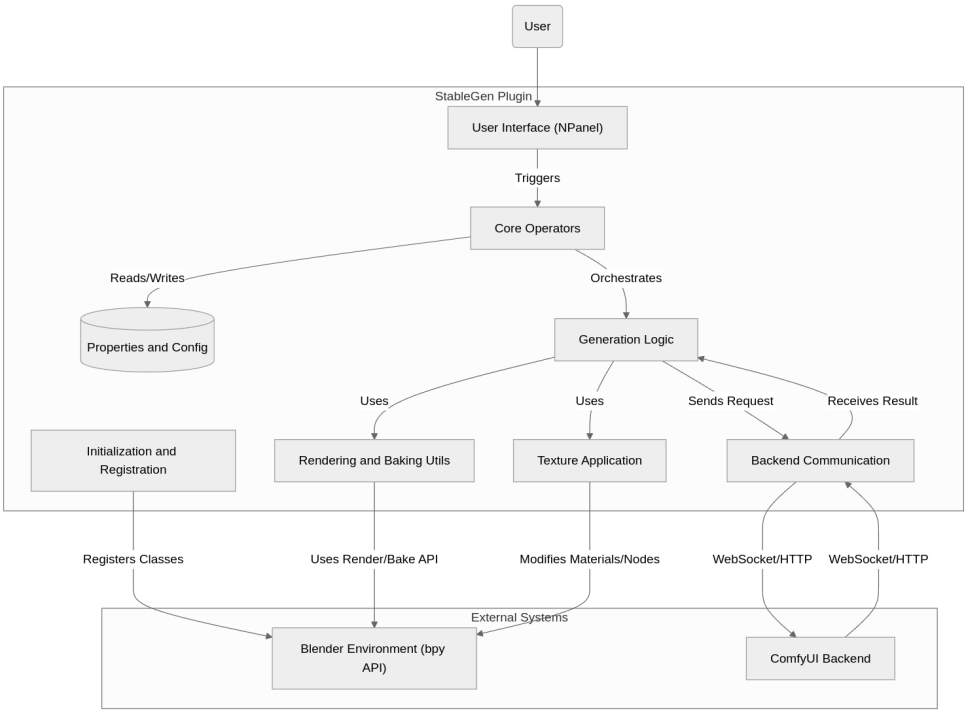


Figure 4.1: High-level block diagram illustrating the architecture of the StableGen plugin and its main components within the Blender environment and interaction with the ComfyUI backend.

include constructing the appropriate workflow JSON based on current settings, establishing WebSocket/HTTP connections, serializing and transmitting request data (including parameters and potentially image data references), receiving results, monitoring progress, and handling communication errors.

- **Generation Logic Component(s):** This part contains the core algorithms specific to the different generation modes (Grid, Inpainting). It prepares data for the backend, including coordinating the export of necessary ControlNet signals and managing the state machine or iterative process required for sequential inpainting. It also incorporates the logic for integrating features like ControlNet and IPAdapter into the generation parameters sent to the backend.

- **Rendering and Baking Utilities:** Functions and operators leveraging Blender's rendering engine (Cycles/Eevee) and compositor are grouped to handle tasks like exporting control signals (depth maps, normal maps, visibility masks) and performing final texture baking using Blender's bake API.

- **Texture Application Component:** This component is responsible for applying the textures received from the backend onto the 3D models. It manages the creation and manipulation of Blender materials and node

trees, sets up UV projection via modifiers or nodes, and implements the complex weighted blending logic (using MixRGB nodes and OSL scripting) to combine results from multiple viewpoints.

- **Initialization and Registration Core:** The standard addon entry point handles the registration and unregistration of all panels, operators, properties, and other necessary components with Blender upon activation and deactivation.

In general operation, user interaction with the UI component triggers specific Operators. These Operators read configuration from the Properties system, utilize Generation Logic and Rendering/Baking Utilities to prepare data, pass instructions and data via the Backend Communication Layer, and finally use the Texture Application Component to integrate the results back into the Blender scene. This modular structure aims to keep different functionalities decoupled where possible.

4.2 User Interface Design

The user interface is designed to accommodate both novice and advanced users. Implemented as a dedicated tab labeled *Stablegen* within Blender's N-panel, the interface is structured into basic and advanced sections. Novice users can leverage pre-configured presets located at the top of the interface for immediate use. Advanced users seeking granular control can expand the advanced parameter section to individually adjust each setting. This dual-tiered design ensures accessibility for users with varying levels of expertise.

4.3 Interaction with diffusion model

The plugin communicates with the external ComfyUI backend [8] via its application programming interface (API) to perform the actual diffusion model processing. ComfyUI, a node-based framework, executes graphical workflows defined by the plugin based on user settings (selected models, prompts, ControlNet units, etc.). This backend choice allows for leveraging complex diffusion pipelines and potentially offloading computation to separate hardware.

The designed interaction pattern involves the plugin constructing a workflow definition and sending it to the ComfyUI server to initiate the generation task. Subsequently, the plugin asynchronously receives status updates and the final generated image result via standard network communication protocols. This asynchronous approach is a key design choice to prevent Blender's user interface from freezing during potentially long generation times. The overall

interaction flow, including how data is conceptually exchanged, is visualized in Figure 4.2.

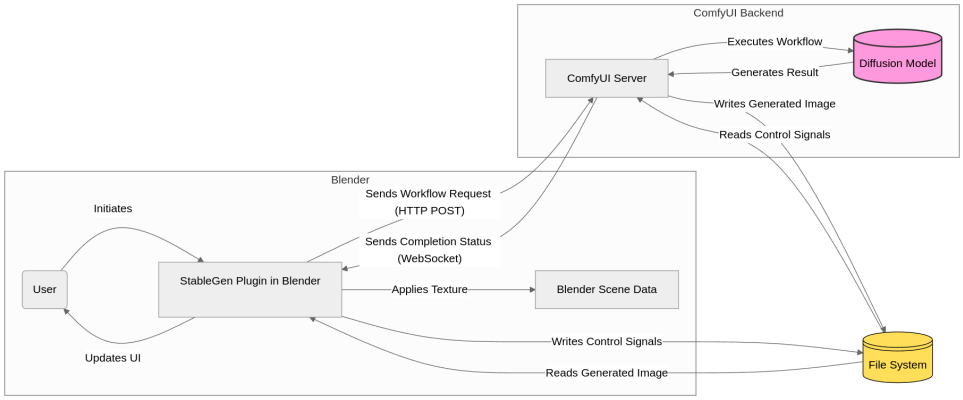


Figure 4.2: Overview of the interaction flow between the StableGen plugin (within Blender) and the ComfyUI backend server during a generation task, including file system interactions.

4.4 Core Generation Principle

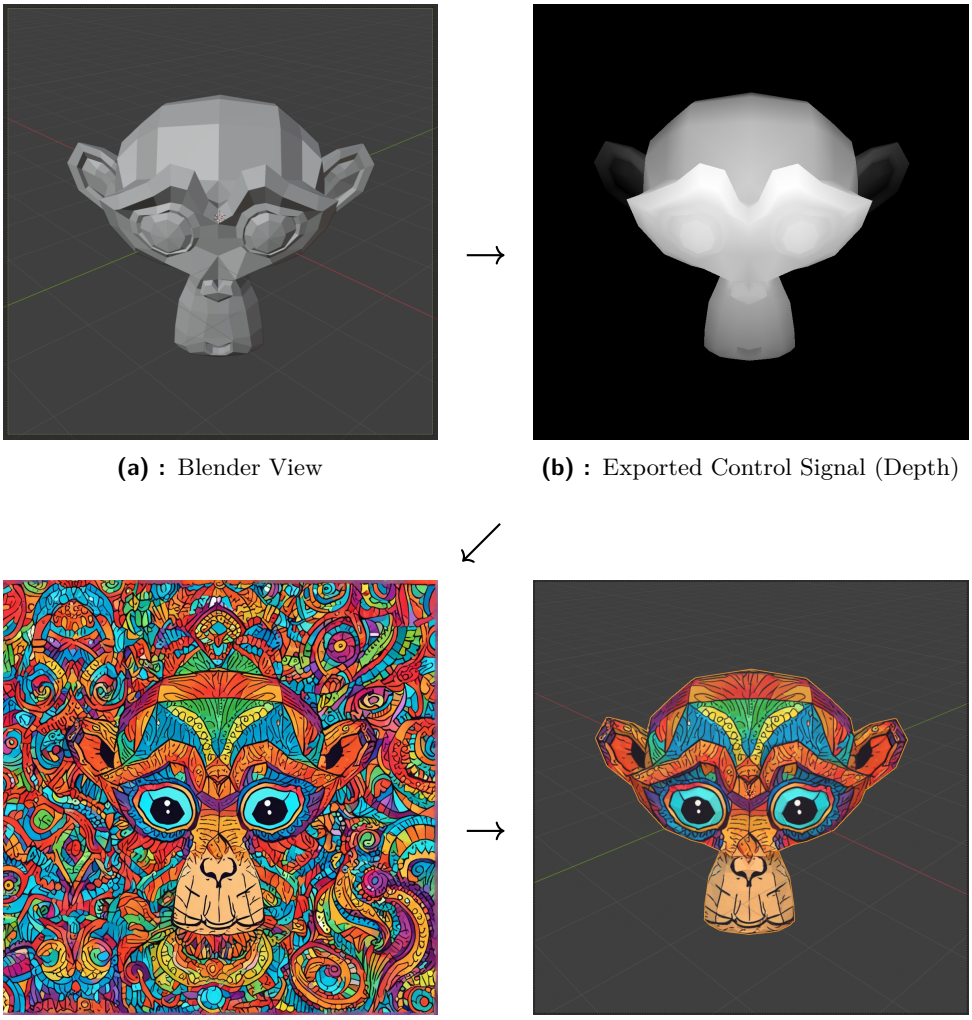
The general workflow consists of generating textures from multiple viewpoints using a diffusion model equipped with ControlNet. This principle is directly taken from existing methods that use it. These existing methods were referenced in Section 2.5.1.

The user first sets up the scene for the generation by adding any number of cameras to it. Then the user configures the parameters by selecting a preset or adjusting parameters manually. The user is then required to select a diffusion model to use and to enter a textual prompt, a description of what the model is supposed to generate.

The necessity of adding the cameras manually is a simplification of the existing implementations previously discussed in Section 3.2.1. Some of the discussed implementations propose an automatic algorithm for optimal coverage of the model [7]. We opted for a semi-manual approach, where the viewpoints are automatically arranged in a circle around the model and the user can move them according to their needs.

The cameras serve two purposes. They are first used to export the control signals required for ControlNet-driven generation, as well as any other outputs necessary for specialized methods (mentioned below). After generating the texture for a given viewpoint, the camera's position and parameters are then used to project the texture onto the scene.

Figure 4.3 depicts this basic process for a given viewpoint.



(a) : Blender View

(b) : Exported Control Signal (Depth)

(c) : Generated Image

(d) : Image Projected onto Mesh

Figure 4.3: The core image generation workflow for a single viewpoint within StableGen. A control signal, such as a depth map (b), is exported from the Blender scene (a). This guides the diffusion model to generate a new texture (c), which is then projected back onto the 3D mesh (d).

4.5 Proposed strategies for multi-view consistency

As previously discussed in the analysis Section 3.2.1, achieving consistency across different viewpoints proves to be difficult without employing additional strategies. In this section, we show two methods derived from existing work [27, 7, 38] and simplified to be used without the need for specialized models, which could reduce the versatility of the solution.

The proposed solutions leverage existing technologies from the diffusion-based ecosystem, namely ControlNet [39] and IPAdapter [37]. Some of the implementations discussed in Section 3.2.1 use dedicated depth-conditioned diffusion [21, 7]. We chose to employ general text-to-image models along with various ControlNet models for 3D model awareness, since they are more readily available and there is a larger diversity among them, which finally leads to better flexibility of our implementation.

4.5.1 Grid Mode

Description

The grid mode operates by combining ControlNet conditioning images into a unified grid, enabling the simultaneous generation of textures for multiple viewpoints.

Process

The Grid Mode process unfolds as follows:

1. **Conditioning Image Generation:** For each selected viewpoint around the 3D model, corresponding ControlNet conditioning images (such as depth maps, normal maps, etc.) are generated.
2. **Grid Formation:** These individual conditioning images are systematically arranged into a single, unified grid image.
3. **Simultaneous Texture Generation:** A diffusion model is employed, using the composite grid image as conditioning input (along with any text prompts). This generates a single output image which is also a grid, containing the textures for all viewpoints simultaneously.
4. **Grid Decomposition:** The generated texture grid is computationally split back into separate, individual images, with each image corresponding to one of the original viewpoints.

5. **Texture Projection:** Each individual texture image, obtained from the decomposition step, is then projected onto the 3D model from its respective viewpoint, contributing to the final texture map.

6. **(Optional) Refinement Pass:** To counteract potential resolution loss inherent in the grid format, an optional second pass can be performed on the individual images obtained in Step 4. Each individual view's texture is partially noised (diffused) and subsequently denoised (reverse diffusion) at its full target resolution, guided by the original high-resolution ControlNet conditioning image for that specific viewpoint.

Figure 4.4 illustrates the core flow from a source image and its derived control signal (depth map) to the generated texture grid and its subsequent projection onto the mesh (implicitly involving the decomposition step).

Advantages and Limitations

This approach leverages the inherent generative capabilities of the diffusion model to potentially maintain consistency across different viewpoints. However, the resulting texture quality is directly contingent upon the model's ability to achieve such consistency.

This approach can offer faster speed at the expense of texture quality. Given the resolution limitations of current state-of-the-art diffusion models, typically around one megapixel (1 MP), generating textures for multiple viewpoints within a single image necessitates a proportional reduction in resolution, and therefore quality, for each individual view.

Limitation Mitigation Strategies

To address the inherent limitations of the Grid Mode, particularly concerning resolution and consistency, the following strategies are implemented:

- **Two-Pass Refinement Strategy:** The primary method to counteract the resolution constraint involves a two-pass approach.

  1. An initial texture grid is generated encompassing all viewpoints simultaneously, as described in Section 4.5.1.
  2. Subsequently, each individual viewpoint's texture is extracted from this grid. It then undergoes a dedicated refinement pass at its full target resolution. This involves partially adding noise (diffusion) to the extracted view and then removing the noise (reverse diffusion), strongly conditioned on the high-resolution ControlNet signals specific to that viewpoint. This significantly improves the detail and sharpness compared to the initial low-resolution grid output.

- **Parameter Presets:** To simplify configuration for users, presets are available. These presets help manage parameters such as the grid layout, diffusion settings for the initial pass, and the strength and specifics of the refinement pass, catering to different types of models and desired outcomes.
- **IPAdapter:** To enforce a consistent overall style or ensure specific features appear across all views in the grid, IPAdapter (Image Prompt Adapter) can be utilized during the initial grid generation phase (and potentially during refinement). In the context of Grid Mode, IPAdapter is conditioned using a single, user-provided reference image. This guides the diffusion model to generate textures across all viewpoints that align with the provided image, ensuring a cohesive aesthetic defined by the user.

Combining these strategies allows Grid Mode to offer a balance between generation speed and acceptable texture quality, particularly for assets where rapid texturing of multiple views with a consistent user-defined style is desired.

Scene Suitability

This method is best suited for scenes which don't require a large number of viewpoints. Empirical evidence suggests promising results for scenes with up to six viewpoints. Having a further amount results in deteriorated texture detail and consistency. Furthermore, the efficacy of this method is influenced by the nature of the objects within the scene, demonstrating particular suitability for texturing distinct and non-homogeneous objects such as characters.

4.5.2 Inpainting Mode

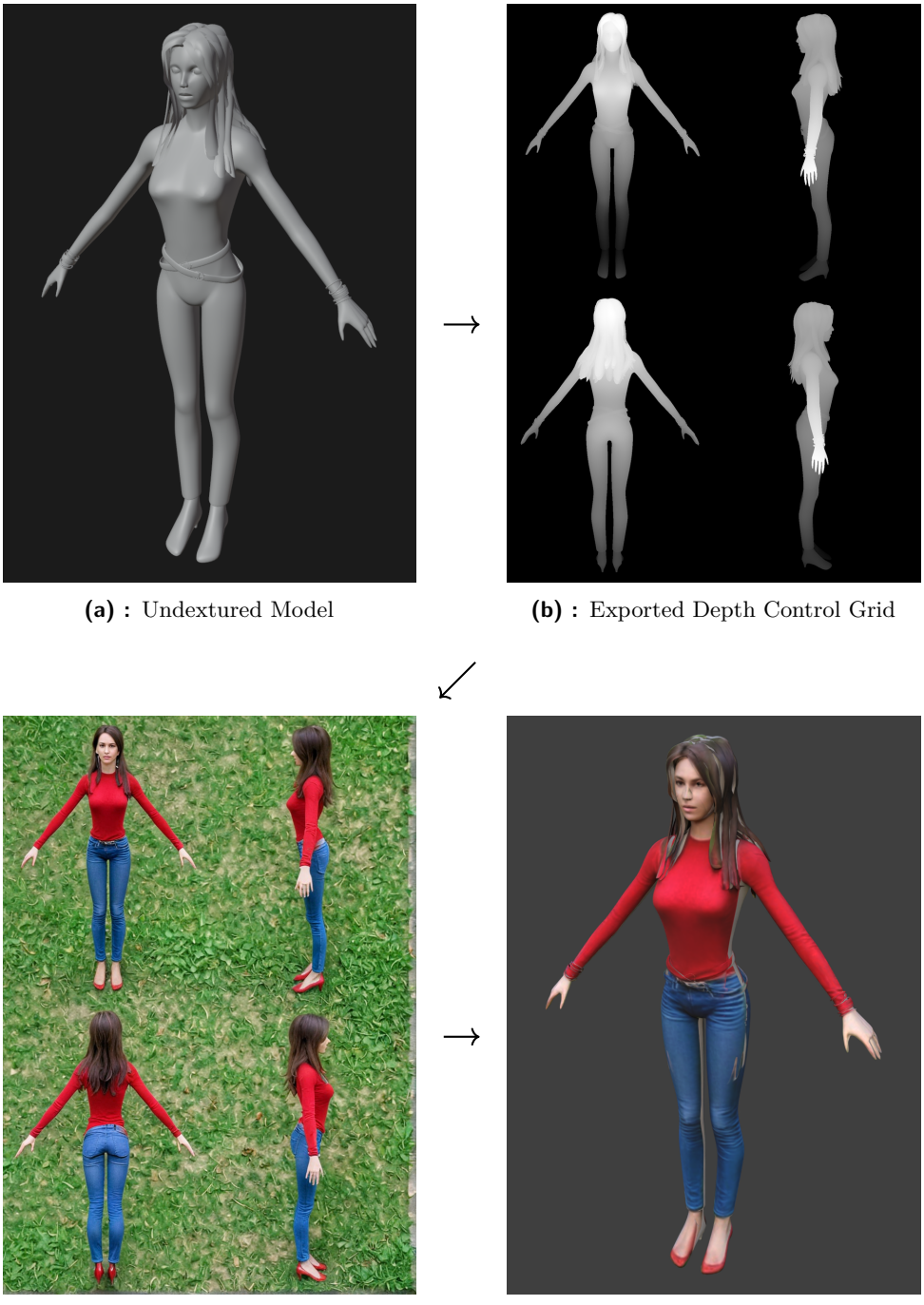
Description

The inpainting mode generates textures sequentially, one viewpoint at a time. It uses information from previously generated views to help maintain consistency in the final texture.

Process

The first viewpoint is generated normally. For all subsequent viewpoints, the process is as follows:

1. A *visibility mask* is created. This mask identifies which parts of the 3D model, as seen from the current viewpoint, were already textured in previous steps.



(a) : Undextured Model

(b) : Exported Depth Control Grid

(c) : Generated Image Grid

(d) : Grid Images Projected onto Mesh

Figure 4.4: Overview of the Grid mode generation process. Control signals from multiple viewpoints, such as depth maps, are combined into a grid (b) from source views (a shows one example). A single diffusion pass generates a corresponding grid of textures (c), which are then split and projected onto the 3D mesh (d).

2. An *RGB render* containing the projected textures from all previously generated viewpoints is exported. This provides the visual context for the next step.

3. The diffusion model performs inpainting directly within the exported *RGB render*. It targets the areas indicated by the *visibility mask* (regions not visible previously), using the surrounding existing projected textures as context to maintain consistency.

4. The newly generated texture patch from the current viewpoint is projected back onto the model's texture map, updating it for use as context in subsequent viewpoint generations.

The key steps involving the visibility mask and RGB context for generating subsequent viewpoints are illustrated in Figure 4.5.

Advantages and Limitations

This method generally produces high-quality textures. It is also scalable, as adding more viewpoints does not inherently reduce the resolution of individual views. (A specific Blender-related issue affecting this scalability will be discussed in the Implementation chapter). However, the quality of the inpainting is sensitive to the accuracy of the visibility mask; improper masking can lead to artifacts. Fine-tuning parameters for different use cases is often necessary, meaning a single preset may not work well for all situations.

Limitation Mitigation Strategies

Several approaches can be employed to mitigate the limitations of the inpainting mode and enhance texture consistency:

- **Parameter Presets:** To simplify setup for users, several presets are provided that configure the inpainting parameters for common use cases and model types.
- **IPAdapter for Consistency:** To further enforce visual consistency across viewpoints, especially when geometric overlap is insufficient or fine details need to be maintained, IPAdapter (Image Prompt Adapter) can be utilized. This technique conditions the diffusion model's generation not just on text prompts and ControlNet signals, but also on a reference image. Users can configure IPAdapter to use one of the following as the reference:
  - The texture generated for the very first viewpoint.
  - The texture generated for the most recently completed viewpoint.

- An external image supplied by the user, defining the target style or key features.

By referencing a consistent source image, IPAdapter helps maintain stylistic coherence, color palettes, and specific details across the sequentially generated texture patches.

Fine-tuning the interplay between the standard inpainting context (visibility mask and RGB render) and the IPAdapter conditioning allows for robust texture generation even in challenging scenarios.

Scene Suitability

The effectiveness of this method depends on the placement and number of viewpoints. Consistency relies on having sufficient overlap between views to provide context for the inpainting model. If viewpoints are too far apart or too few are used, the model may lack the necessary context, potentially leading to inconsistencies or complete failure in maintaining a coherent texture across the model. Careful camera placement is therefore important for achieving good results with this mode.

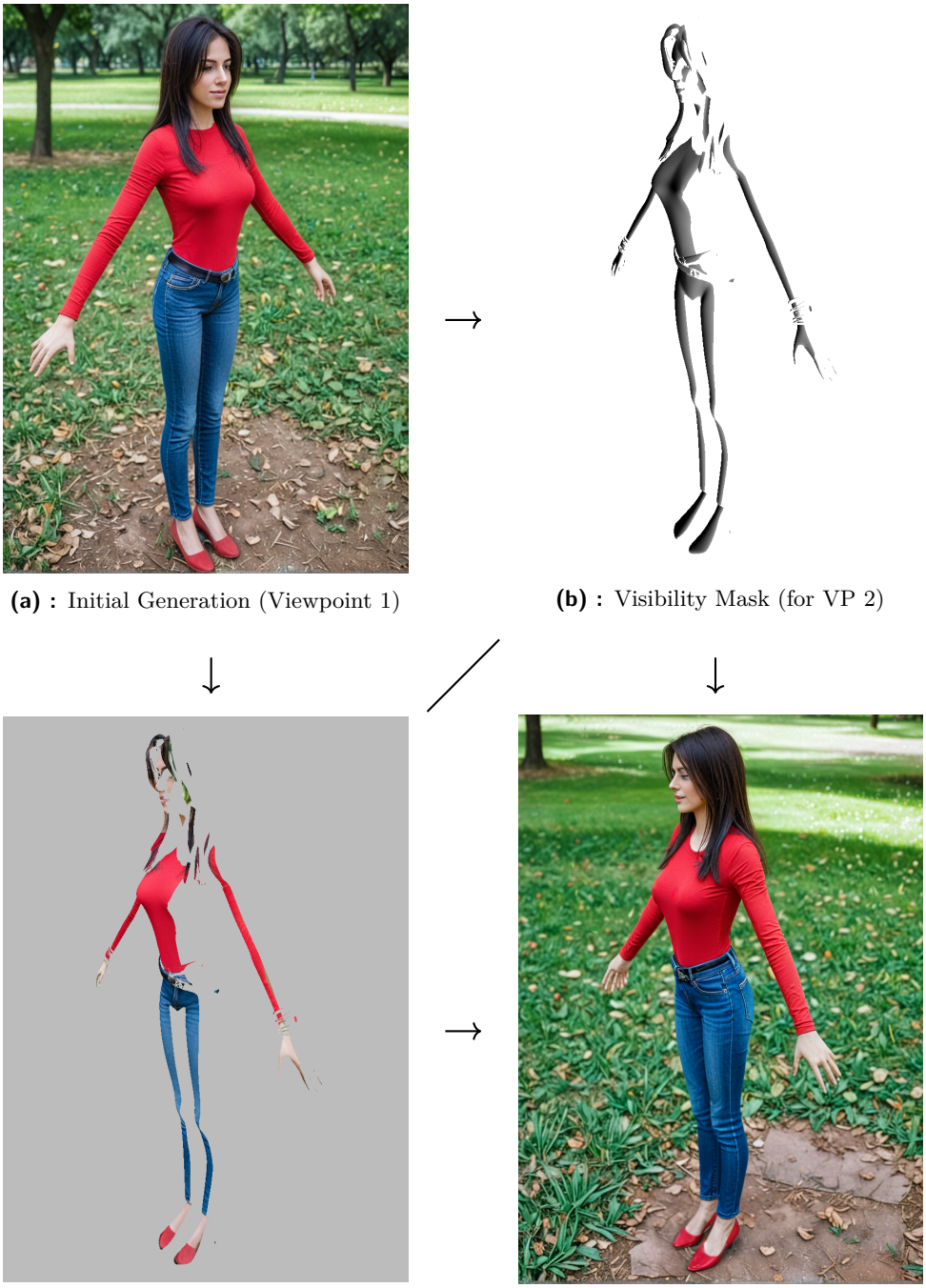
4.6 Proposed strategies for occlusion handling

Generating textures from multiple viewpoints requires mechanisms to handle areas that might be visible from some angles but occluded from others, or missed entirely. Leaving these areas unaddressed can result in visual artifacts or incomplete textures. Two primary strategies are proposed to manage these challenges: dynamically weighting viewpoint contributions based on visibility and surface angle, and post-process inpainting directly in the texture's UV space to fill remaining gaps.

4.6.1 Weighted Blending via Ray Tracing

To seamlessly combine textures from multiple viewpoints, especially handling overlaps and occlusions, a *Weighted Blending* strategy is employed. This approach dynamically calculates the contribution of each viewpoint's generated texture at every point on the model's surface.

The core of this strategy lies in calculating a per-viewpoint *weight* for each surface point. This weight determines how much influence a specific viewpoint's texture has during the final blending process. The calculation is performed computationally, utilizing a shader, which considers several factors:



(a) : Initial Generation (Viewpoint 1)

(b) : Visibility Mask (for VP 2)

(c) : RGB Context (for VP 2)

(d) : Inpainted Result (Viewpoint 2)

Figure 4.5: The sequential Inpainting mode workflow. After an initial generation (a) for Viewpoint 1, subsequent views are generated using context from previously textured areas. This involves creating a visibility mask (b) and an RGB render of the current texture state (c) from the perspective of the new viewpoint (Viewpoint 2). These guide the inpainting process to produce the result for Viewpoint 2 (d).

1. **View Frustum Culling:** Firstly, a check determines if the surface point lies within the viewing frustum of the specific camera viewpoint. This involves transforming the point's position and the view direction into the camera's coordinate system and comparing it against the camera's field of view (FOV) and aspect ratio. Points falling outside the camera's view cone receive a weight of zero for that viewpoint.

2. **Occlusion Detection via Ray Tracing:** To handle occlusions by other parts of the geometry, a ray is cast from the surface point towards the camera's origin. If this ray hits another object or part of the same object before reaching a certain threshold distance, the original point is considered occluded from that viewpoint. In such occlusion cases, the weight assigned for that viewpoint is zero, ensuring that hidden surfaces do not incorrectly receive texture data from an occluded camera angle.

3. **Surface Angle Attenuation:** The angle θ between the surface normal at a point and the incoming view direction from the camera is evaluated. Instead of only using a simple threshold, the contribution is weighted using the formula $W_{\text{angle}} = |\cos(\theta)|^{\text{Exponent}}$, where the `Exponent` is a user-configurable parameter *(Weight Exponent)*. This allows for fine-tuning the blend falloff: higher exponent values significantly prioritize viewpoints nearly perpendicular to the surface ($\theta \approx 0°$), creating sharper transitions between views, while an exponent of 1.0 replicates a standard weighting based directly on $|\cos(\theta)|$. Points where the angle exceeds a maximum threshold (e.g., 90 degrees) still receive a weight of zero, ensuring that back-facing or extremely oblique surfaces do not contribute.

The shader outputs a final weight for the given point and viewpoint, combining these checks. A weight of zero signifies no contribution due to being outside the view frustum, occluded, or exceeding the angle threshold. A positive weight indicates visibility, with the value often reflecting the viewing angle's suitability.

These calculated weights, one for each viewpoint, are then used in a blending process (conceptually similar to a weighted average) to combine the corresponding generated textures. Viewpoints with higher weights contribute more significantly to the final texture color at that specific surface point. This ensures smooth transitions between textures from different views and correctly handles occlusions by giving zero influence to viewpoints from which a surface point is hidden.

Figure 5.6 in the Implementation chapter provides a visual demonstration of how these weights might be distributed across a simple sphere from multiple viewpoints.

4.6.2 UV Inpainting for Gap Filling

While the Weighted Blending strategy effectively manages texture contributions based on visibility and angle, surface areas that are completely occluded across all viewpoints will still lack generated texture after the initial blending process. To address these remaining gaps and ensure complete texture coverage, a *UV Inpainting* method is employed as a subsequent post-processing step.

This process involves unwrapping the model's geometry onto a standard 2D UV map. A visibility map is then calculated in this UV space, identifying the texels (texture pixels) corresponding to the occluded or untextured areas. Diffusion-based inpainting is subsequently performed directly on the unwrapped UV texture, specifically targeting these identified regions. This allows the diffusion model to intelligently fill in the missing texture information based on the surrounding textured areas and potentially text prompts. To further enhance the quality of the inpainted regions for scenes with multiple objects, users can optionally provide distinct text prompts for each object during this stage.

4.7 Utility Methods

Beyond the core texture generation modes, the system incorporates supplementary methods to address common challenges and provide necessary functionalities for a complete texturing workflow integrated within Blender.

4.7.1 Texture Baking

While the generation process might initially utilize multiple rendered images projected onto the geometry using blending techniques, a final, standardized texture format is often required, particularly for exporting models to other software or game engines. *Texture Baking* serves this purpose by consolidating the generated visual information into a single texture image file per object, mapped according to its UV layout.

The process transforms the projected textures into the standard UV map format. The system can utilize a pre-existing UV map provided with the model. However, if a model lacks a UV map, or if the existing map has issues like overlapping islands or incomplete coverage, alternative UV unwrapping options are provided within the tool to generate a suitable layout before baking. Initially, the scene's appearance comes from projecting and blending the n images generated by the diffusion model for each viewpoint; the baking process then converts this into a discrete UV-mapped texture for each individual object.

4.8 Workflow Examples

Stablegen is designed to integrate AI-driven texture generation into various Blender workflows, offering flexibility for different use cases, from rapid prototyping to refining existing scenes.

For users seeking simplicity, a common workflow involves setting up cameras in the Blender scene to define the desired viewpoints for texture generation. Within the Stablegen interface, the user selects a preset, which predetermines the underlying generation method (**Grid Mode** or **Inpainting Mode**) and associated parameters. The user then chooses the desired diffusion model, provides a text prompt describing the texture, and initiates the generation process. This streamlined approach applies the generated textures across all objects currently visible within the scene from the defined viewpoints. Upon generation completion, Stablegen applies the resulting textures by creating or modifying Blender materials for the scene objects. The subsequent functionalities are crucial integration points, allowing users to address generation artifacts or transform the output into standard UV texture maps compatible with Blender's conventional texturing tools and external applications. This ensures that while the generation method is novel, the output can seamlessly integrate with established 3D asset workflows.

Alternatively, Stablegen offers flexibility for more advanced workflows. Users can start with an existing preset and then modify various individual parameters to achieve finer control over the output. Furthermore, these customized settings can be saved as new user presets for quick loading in the future.

The plugin integrates into different creative pipelines:

- **Concept Art and Look Development:** Users can quickly apply stylistic textures to entire scenes or collections of assets based on text prompts. This allows for rapid visualization of different looks and moods without manual texturing, accelerating the creative iteration process.
- **Game Asset Prototyping:** Initial textures can be generated for low-poly or high-poly game assets directly within Blender. These initial textures, applied scene-wide, can then be refined using *UV Inpainting* for any gaps and subsequently processed using the *Texture Baking* feature. This produces standard UV-mapped textures suitable for testing in game engines or as a base for further manual refinement.
- **Texturing Existing Scenes:** For scenes containing untextured models, Stablegen provides a way to apply materials based on a unified prompt across all objects, establishing a consistent base appearance. It can also be used to experiment with re-texturing existing scenes by generating new materials based on prompts.

- **Refining Existing Textures:** Stablegen can be utilized to modify textures that are already applied to models. Users can employ text prompts, potentially combined with image guidance (e.g., via IPAdapter) to guide the diffusion model in restyling, improving resolution, altering details, or generally remastering existing texture maps. This offers a way to iterate on or update previously textured assets non-destructively or as a basis for new variations.

Chapter 5

Implementation

5.1 Introduction and Environment

This chapter details the technical realization of the Stablegen plugin, translating the concepts outlined in the System Design chapter into a functional Blender addon. It covers the development environment, the plugin's architecture, the implementation of core functionalities using the Blender Python API, and the interaction with the ComfyUI backend.

5.1.1 Development Tools and Software Versions

The development was initially conducted using Blender version 4.0 and subsequently updated for compatibility up to version 4.3. The plugin utilizes the Python 3.10 interpreter bundled with these Blender versions. The backend processing relies on a running instance of ComfyUI, accessible via its web API. Key Python dependencies beyond Blender's standard library include Pillow (PIL) for image manipulation, OpenCV (`cv2`) for specific image processing tasks like Canny edge detection, and `websocket-client`/`websockets` for ComfyUI communication, which are automatically installed into the Blender interpreter's environment if missing.

5.1.2 Key Technologies Employed

The implementation primarily utilizes the Python programming language (version 3.10). Integration with Blender is achieved exclusively through Blender's Python Application Programming Interface (API), commonly referred to as `bpy`. Communication with the generative backend leverages the aforementioned WebSocket libraries to interact with the ComfyUI web API, exchanging workflow data formatted as JSON and handling image data transfer. Standard Python libraries are used for tasks like HTTP requests (e.g., for interrupting the queue) and file handling.

5.2 Plugin Structure and Code Organization

5.2.1 Addon Initialization and Registration

As a standard Blender addon, Stablegen includes a main `__init__.py` file containing the necessary addon information (`bl_info`) dictionary. This file implements the mandatory `register()` and `unregister()` functions. The `register()` function is crucial as it defines a large number of addon properties directly onto Blender's Scene type (`bpy.types.Scene`) using `bpy.props.*`, registers all custom classes (Operators, Panels, PropertyGroups like `ControlNetUnit`), and sets up application handlers (e.g., `load_handler`). The `unregister()` function cleans up by deleting these scene properties and unregistering all classes. Addon preferences (`StableGenAddonPreferences`) defining paths and ComfyUI mappings are also managed here.

5.2.2 Module Breakdown and Responsibilities

To maintain modularity and readability, the plugin's codebase is organized into several Python modules, each responsible for a specific aspect of functionality:

- `__init__.py`: Handles addon registration, defines addon preferences, core scene properties, `ControlNetUnit` PropertyGroup, and operators for managing ControlNet units. Also includes helper functions for populating model lists based on preferences.
- `stablegen.py`: Defines the main user interface (`StableGenPanel`) presented in the N-panel, organizing all settings and triggering operators. It also implements operators for managing presets (`ApplyPreset`, `SavePreset`, `DeletePreset`) and relies on global variables (`PRESETS`, `GEN_PARAMETERS`) and helper functions (`get_preset_items`, `update_parameters`) for preset logic.
- `generator.py`: Contains the main generation operator (`ComfyUIGenerate`) which orchestrates the entire process. This includes preparing control signals, managing modal operation and progress updates, running the core generation logic in a separate thread (`async_generate`), and defining the methods (`generate`, `refine`, `generate_flux`, `refine_flux`) that build the ComfyUI JSON workflows and handle communication. It also includes helpers for building ControlNet chains, image grid manipulation using PIL, and exporting common control maps like depth and normals.
- `render_tools.py`: Provides operators and utility functions related to rendering, baking, UV unwrapping, image processing (using OpenCV), material manipulation, and camera setup within the addon. These tools support the main generation process by preparing input data (like

renders, Canny maps, visibility masks) or handling outputs (applying textures, baking).

- `project.py`: Handles the complex process of applying generated textures back onto the scene objects. The main function (`project_image`) manages UV projection modifier setup, material node tree creation/modification, including hierarchical blending of textures from multiple viewpoints using MixRGB nodes controlled by OSL script-based weights, and integrates with the baking process (`simple_project_bake`).
- `utils.py`: Contains general utility operators and functions, such as setting up HDRI lighting (`AddHDRI`) and potentially helper functions like retrieving data stored unconventionally in node trees (`get_last_material_index`).
- `util/install_testing.py`: Provides a utility function (`install`) to check for and install required Python dependencies (`websocket-client`, `websockets`) into Blender interpreter's environment.

5.2.3 Core Classes and Data Structures

Key custom data structures include the `ControlNetUnit` PropertyGroup defined in `__init__.py`. The main operational logic is encapsulated within Blender Operators (bpy.types.Operator) rather than in many standalone classes.

5.2.4 User Interface Implementation (bpy UI API)

The plugin's user interface is primarily defined in the `stablegen.py` module within the `StableGenPanel` class, which inherits from `bpy.types.Panel` and is registered to appear in Blender's N-panel sidebar under the *StableGen* tab. The overall layout, as depicted in Figure 5.1, is constructed within the `draw()` method using Blender's standard UI layout system accessed via the `layout` object passed by the context.

The interface is organized into logical sections for intuitive workflow. At the top, primary action buttons such as *Add Cameras, Collect Camera Prompts, Generate,* and *Bake Textures* are provided. Below this, a *Preset* system allows users to save and load configurations. The *Main Parameters* section exposes essential settings like the textual *Prompt, Model* selection (e.g., SDXL checkpoints), *Architecture* (SDXL or FLUX), and *Generation Mode* (e.g., Sequential or Grid). Further customization is available through a series of collapsible *Advanced Parameters* sections, which include *Core Generation Settings* (e.g., steps, CFG), *Viewpoint Blending Settings, Output & Material Settings, Image Guidance (IPAdapter & ControlNet), Inpainting Options,* and *Generation Mode Specifics.* Finally, a *Tools* section at the bottom offers utility

functions such as *Switch Material, Add HDRI Light, Apply All Modifiers, Convert Curves to Mesh,* and *Export Orbit GIF/MP4.*

The interface utilizes standard Blender UI elements like rows, columns, boxes (`layout.box()`), and collapsible panels (`layout.operator("wm.operator_with_panel",text="Section").panel="PANEL_TYPE_ID"`) for clear organization. Key parameters, defined as scene properties, are exposed using `layout.prop(context.scene, "property_name")`, and buttons trigger `bpy.types.Operator` classes. The UI also dynamically adapts by displaying certain sections based on the selected generation mode (`generation_method` scene property) and shows progress bars during operations by polling properties of active modal operators.

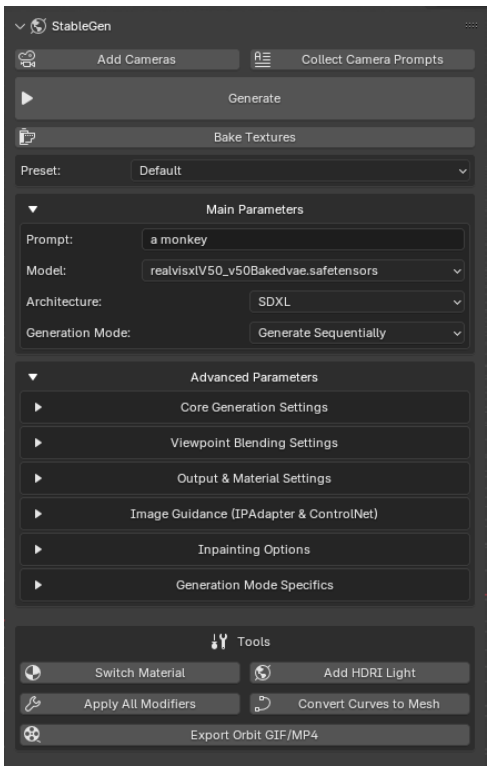


Figure 5.1: Overview of the StableGen plugin's main user interface within Blender's N-panel. Key sections visible include primary action buttons (Add Cameras, Generate, Bake Textures), preset management, main generation parameters (Prompt, Model, Architecture, Mode), collapsible advanced parameter groups (Core Settings, Blending, Image Guidance, Inpainting, etc.), and a utility tools section at the bottom.

5.2.5 Parameter Handling and Preset System

Core generation parameters and addon settings are defined as custom properties attached directly to Blender's scene type (`bpy.types.Scene`) within the main `__init__.py` file, utilizing various `bpy.props.*` types (e.g., String-

Property, FloatProperty, EnumProperty, CollectionProperty for ControlNet units). Many of these properties employ an `update` callback linked to the `update_parameters` function (defined in `stablegen.py`). This function compares the current scene settings against known presets and updates the `stablegen_preset` EnumProperty accordingly, ensuring the UI reflects whether the current state matches a saved preset or is *custom*.

The preset system, primarily implemented in `stablegen.py`, relies on a global Python dictionary named `PRESETS` which stores both built-in default configurations and user-saved presets. A global list, `GEN_PARAMETERS`, defines exactly which scene properties are included when saving or loading a preset. The UI dropdown for selecting presets is populated by the `get_preset_items` function, which lists keys from the `PRESETS` dictionary. The `ApplyPreset` operator retrieves the dictionary for the selected preset name and applies its values to the corresponding scene properties. The `SavePreset` operator invokes a dialog for naming, gathers current settings defined by `GEN_PARAMETERS` (and optionally ControlNet units), and adds a new entry to the `PRESETS` dictionary. The `DeletePreset` operator removes user-defined presets from the dictionary.

5.3 Viewpoint and Control Signal Implementation

This section details the implementation related to handling viewpoints (defined by Blender cameras) and generating the various control signal images required by ControlNet during the texture generation process.

5.3.1 Camera Handling (bpy Scene/Object API)

Viewpoints for texture generation are defined by standard Blender Camera objects placed within the scene. The main generation operator (`ComfyUIGenerate` in `generator.py`) iterates through the scene's objects to identify all active cameras using `bpy.context.scene.objects` and checking object types. Their positions, orientations, and lens settings are accessed via standard `bpy.data.objects[name].location`, `.rotation_euler`, and `.data` attributes for use in rendering control signals and projecting textures.

To facilitate viewpoint setup, the `AddCameras` operator (implemented in `render_tools.py`) provides a utility to automatically create a specified number of cameras. These cameras are arranged in a circle around a central point (derived from the active object or view center) and inherit settings like lens and sensor size from the previously active camera. This operator enters a modal state, allowing the user to interactively refine the position and orientation of each newly added camera using Blender's Fly Navigation before finalizing their placement.

5.3.2 Control Signal Generation/Export (bpy Data/Render API)

The plugin implements functions to automatically generate and export various image types commonly used as ControlNet conditioning signals. These functions leverage Blender's rendering capabilities (primarily Cycles for specific passes) and compositor nodes, often supplemented by external libraries like OpenCV for image processing.

- **Depth Maps:** The `export_depthmap` function within `generator.py` handles depth map generation. It utilizes Blender's compositor nodes to extract Z-depth information directly from the render passes. The output is saved as an image file (examples shown in Figure 5.2).
- **Normal Maps:** World-space normal maps are generated by the `export_normal` function, also in `generator.py`. This function configures compositor nodes to output the Normal pass, ensuring a neutral background, and saves the result (examples shown in Figure 5.3).
- **Canny Edge Maps:** Canny map generation is implemented in the `export_canny` function within `render_tools.py`. This function first calls `export_render` (also in `render_tools.py`) to produce a simple base render of the scene from the specified viewpoint. It then uses the OpenCV library (`cv2`) to read this rendered image, apply the Canny edge detection algorithm (`cv2.Canny`) with user-defined thresholds, and save the resulting black-and-white edge map (examples shown in Figure 5.4).
- **Other Maps:** Additional functions like `export_emit_image` and `export_visibility` (both in `render_tools.py`) are used internally for specific purposes, such as generating emission-only renders or creating visibility masks for the inpainting workflow, often involving temporary material modifications and specific compositor setups. The basic `export_render` provides a standard color render when needed.

These export functions ensure that the necessary conditioning images, corresponding precisely to the user-defined camera viewpoints, are available for the diffusion model during the generation process.

5.4 Generation Initiation and Backend Communication

This section details the implementation of the processes responsible for initiating a texture generation task based on user configuration and managing the communication with the ComfyUI backend server. The core logic resides primarily within the `generator.py` module.

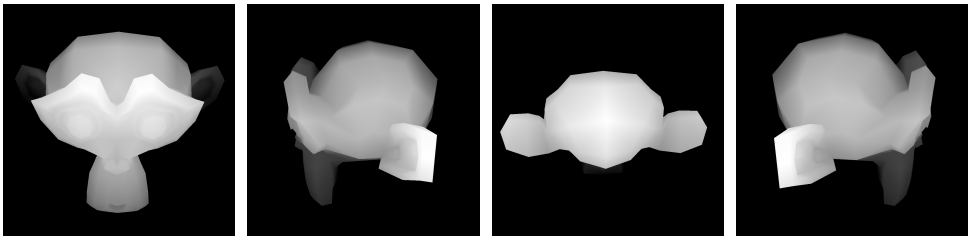


Figure 5.2: Examples of exported Depth Maps for the Suzanne model from different viewpoints, used as ControlNet input.

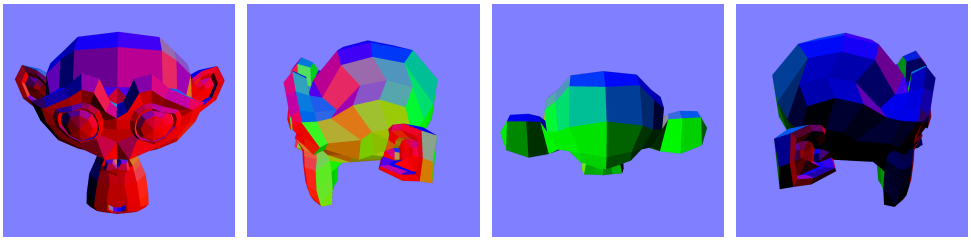


Figure 5.3: Examples of exported Normal Maps for the Suzanne model from different viewpoints, used as ControlNet input.

5.4.1 Backend Communication Details (ComfyUI API)

Communication with the ComfyUI backend involves two primary protocols: an initial HTTP request to queue the generation task, followed by WebSocket communication for real-time progress updates and final image transfer. This communication flow is visually summarized in Figure 4.2.

API Connection and Response Handling

The process begins by sending the dynamically generated workflow JSON to the ComfyUI server's `/prompt` endpoint via an HTTP POST request, handled by the `_queue_prompt` helper function in `generator.py`. This action queues the job on the backend. Subsequently, a persistent WebSocket connection is established to the server's `/ws` endpoint using standard Python libraries (`websocket-client` or `websockets`, managed by `_connect_to_websocket`). The plugin listens on this WebSocket connection for messages from ComfyUI, processing message types like `'executing'` (to track the current node) and `'progress'` (to update the UI progress bar). Error messages from the backend are also received via WebSocket. Additionally, to handle user cancellations, a separate HTTP POST request is sent to the `/interrupt` endpoint, implemented in the `cancel_generate` method.

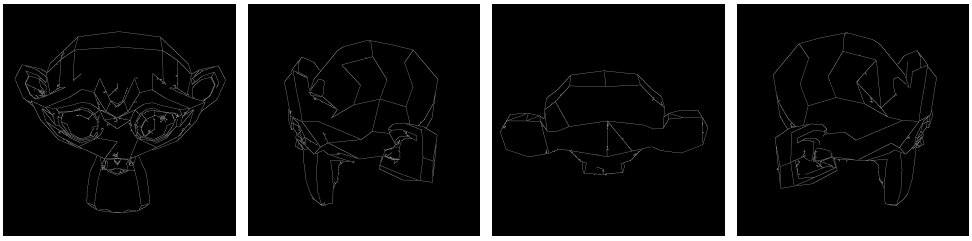


Figure 5.4: Examples of exported Canny Edge Maps for the Suzanne model from different viewpoints, used as ControlNet input.

Workflow/API Call Construction

The plugin constructs the ComfyUI workflow dynamically as a JSON object before sending it to the backend. This task is handled by specific methods within `generator.py`: `generate()`, `refine()`, `generate_flux()`, and `refine_flux()`. These methods utilize helper functions like `_create_base_prompt()`, `_create_img2img_base_prompt()`, and their FLUX-specific counterparts (`create_base_prompt_flux()`, `_create_img2img_base_prompt_flux()`) to build the initial JSON structure based on predefined templates stored in `util/helpers.py`. They then programmatically modify this JSON, adding nodes and configuring their inputs based on user settings (e.g., selected models, prompts, steps, CFG, sampler, scheduler, architecture). For debugging purposes, these generated JSON prompts are optionally saved to disk using `_save_prompt_to_file()`.

Data Transfer and Formatting

Parameters such as text prompts, sampler settings, and model names are embedded directly into the fields of the JSON workflow definition. Input image data (ControlNet signals, initial images for img2img, masks for inpainting) are handled by saving the required images to temporary files first. The JSON workflow then includes `LoadImage` nodes configured with the file paths to these images, allowing ComfyUI to access them. Generated output images are handled by a `SaveImageWebSocket` node within the ComfyUI workflow; the plugin receives the final image as binary data via the WebSocket connection (after stripping an 8-byte header) once the node executes.

Asynchronous Operation Handling

To prevent Blender's user interface from freezing during potentially long generation times, the core generation process, including backend communication and waiting, is executed in a separate background thread managed by Python's `threading` module. The main `ComfyUIGenerate` operator initiates this thread, targeting the `_async_generate` method. The operator then en-

ters Blender's modal state (`RUNNING_MODAL`), allowing its `modal()` method to run periodically via a timer (`bpy.app.timers`). This `modal()` method checks status variables (e.g., `_is_running`, `_progress`, `_error`, `_stage`) updated by the background thread, enabling UI updates (via `redraw_ui`) and handling completion or errors without blocking the main thread. Tasks requiring main thread execution after background processing, such as final image projection in sequential mode, are scheduled using `bpy.app.timers.register` from within the background thread, often using `threading.Event` objects for synchronization.

Integrating ControlNet and IPAdapter

ControlNet and IPAdapter functionalities are integrated by dynamically adding and configuring the necessary nodes within the ComfyUI JSON workflow before execution. For ControlNet, the `build_controlnet_chain_extended` helper function in `generator.py` iterates through the active `ControlNetUnit` properties defined in the scene collection. For each unit, it programmatically adds unique `LoadImage`, `ControlNetLoader`, and `ControlNetApplyAdvanced` nodes to the JSON workflow, connecting their respective positive, negative, and VAE inputs/outputs sequentially to form a processing chain. It also handles specific configurations for union-type ControlNet models by connecting through a `SetUnionControlNetType` node if required.

Similarly, the core generation and refinement methods (`generate`, `refine`, etc.) contain logic, often encapsulated in helper methods like `_configure_lora()`, `_configure_lora_for_refinement()`, `_configure_ipadapter()`, and `_configure_ipadapter_refine()`, to conditionally add and configure LoRA loader nodes and the relevant IPAdapter nodes (`ipadapter`, `ipadapter_image`) based on user selections in the UI, connecting them appropriately within the workflow graph.

5.4.2 Initiating Generation Tasks

The generation process is initiated when the user invokes the main `ComfyUIGenerate` operator. For the UV inpainting mode, if the corresponding option (`ask_object_prompts`) is enabled, the operator's `invoke()` method first manages the collection of object-specific prompts using Blender's `invoke_props_dialog`, cycling through mesh objects before proceeding.

Otherwise, the operator's `execute()` method performs the main setup. This includes performing checks (e.g., preference paths, UV slot availability), identifying active cameras, and triggering the export of all required control signal maps (depth, normal, canny, etc.) by calling functions like `export_depthmap`, `export_normal`, `export_canny`. Once setup is complete, `execute()` launches the primary generation logic in a separate thread by

calling the `_async_generate` method and then enters the modal state to monitor its progress. The `_async_generate` method acts as the main orchestrator within the background thread, determining which core generation or refinement method (`generate`, `refine`, `generate_flux`, `refine_flux`) to call based on the selected `generation_method` scene property and other configuration settings like the model architecture.

5.5 ComfyUI workflow design

This section outlines the structure of the underlying ComfyUI node workflows that are dynamically constructed and executed by the Stablegen plugin via the backend API. These workflows form the core of the image generation process. Understanding their node-based structure clarifies the generation pipeline executed by ComfyUI.

5.5.1 Overview

The plugin generates different ComfyUI workflows depending on the user's selected architecture (SDXL or FLUX), generation mode, and enabled features. The Python code described previously builds these workflows as JSON objects. The following subsections describe the fundamental structures for these variations.

5.5.2 Base Text-to-Image Workflow (SDXL)

For initial generation using standard SDXL models, the workflow consists of core nodes for model loading, prompt encoding, optional LoRA application, ControlNet chain processing, latent image generation, sampling, decoding, and saving (Figure 5.5). The key components include:

- Model Loading: A `CheckpointLoaderSimple` node loads the main diffusion model.
- Prompt Encoding: Two `CLIPTextEncode` nodes process positive and negative prompts. A `CLIPSetLastLayer` node adjusts CLIP Skip.
- LoRA (Optional): A `LoraLoader` node modifies the model and CLIP outputs.
- ControlNet Chain: Each controlnet image is proccessed using multiple nodes (`LoadImage`, `ControlNetLoader` -> `ControlNetApplyAdvanced`). These *ControlNet units* are constructed and chained together by `build_controlnet_chain_extended`. This chain is included in all modes except UV inpainting.

- Latent Image Generation: An `EmptyLatentImage` node creates initial latent noise.
- Sampling: A sampler node (e.g., `KSampler`) performs diffusion using the ControlNet-modified conditioning, latent image, and sampling parameters.
- Decoding and Saving: A `VAEDecode` node converts latent to pixel space, and `SaveImageWebSocket` sends the result back.

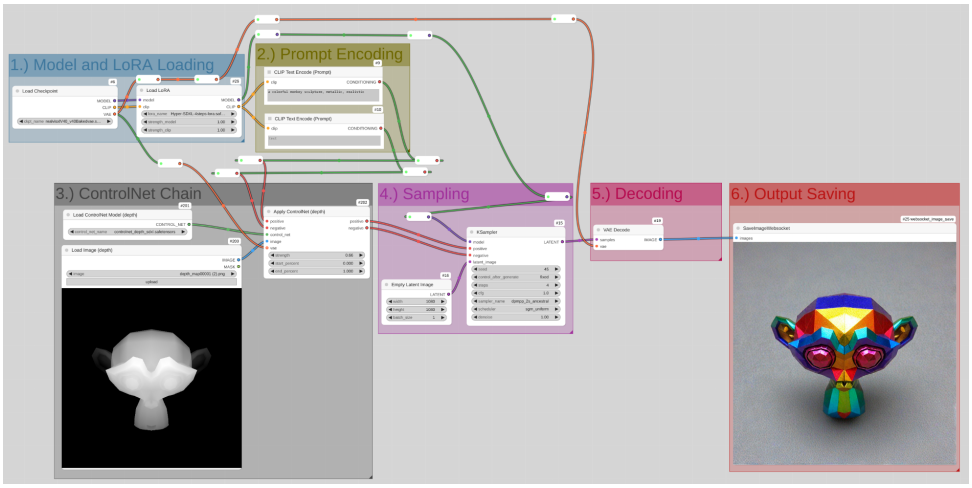


Figure 5.5: Node graph structure for the base SDXL Text-to-Image workflow. (ComfyUI workflow)

5.5.3 Base Image-to-Image/Inpainting Workflow (SDXL)

Workflows involving refinement or inpainting share the core SDXL structure but include extra nodes for image loading, encoding, and mask processing. Key aspects include:

- Image/Mask Loading: `LoadImage` nodes load the initial image and (for inpainting) the mask.
- VAE Encoding: `VAEEncode` (refinement) or `VAEEncodeForInpaint` (inpainting) encodes the input image to latent space.
- Mask Processing (Inpainting): The loaded mask is processed through `GrowMask` and optionally `ImageBlur` before use.
- ControlNet Chain (Refinement): Included as in text-to-image for standard refinement; omitted for UV inpainting mode.
- Sampler Configuration: Receives the encoded latent. The `denoise` parameter controls the influence of the original latent.
- Other core nodes function similarly to the text-to-image workflow.

5.5.4 FLUX Architecture Workflows

When the FLUX architecture is selected, the workflow structure is distinct and does not utilize a negative text prompt. Core components are:

- Core Components: Key nodes include `VAELoader`, `DualCLIPLoader`, `UNETLoader`, `BasicScheduler`, `BasicGuider`, and `FLUXGuidance`.
- No Negative Prompt: Only the positive prompt is encoded and used.
- ControlNet Chain: The ControlNet chain modifies the positive conditioning signal, connecting its final output to the `FLUXGuidance` node.
- Sampling Nodes: Sampling involves nodes like `SamplerCustomAdvanced` configured via the `BasicScheduler` and `BasicGuider`.
- Inpainting/Refinement: Specific FLUX-compatible nodes (e.g., `VAEEncodeForInpaint`, potentially `DifferentialDiffusion`, `InpaintModelConditioning`) are employed when needed.

5.5.5 Integration of Optional Control Features

Features like IPAdapter and advanced inpainting techniques are integrated by inserting specific nodes:

IPAdapter Integration

When enabled, a `LoadImage` node loads the reference image. An `IPAdapter` node receives this image and the model conditioning. Its output model conditioning replaces the original input to the sampler, influencing generation according to IPAdapter settings.

Advanced Inpainting Techniques (Differential Diffusion)

Differential Diffusion is an advanced, inference-time technique applicable to both SDXL and FLUX architectures, designed for enhanced image editing and inpainting control. Instead of a binary mask, it utilizes a grayscale *change map* where pixel intensity dictates the desired amount of change (edit strength) for that specific location. [22]

The core mechanism operates during the diffusion denoising loop. At each timestep t, a binary mask is derived from the change map based on a threshold related to t. Regions where the change map value exceeds the threshold continue denoising from the previous step's latent state (z_{t+1}), while regions below the threshold are reset using a noised version of the original image's latent state corresponding to timestep t (z'_t). [22] This effectively

means areas designated for less change (darker in the change map) retain information from the original image for more steps, while areas designated for more change (brighter) start the generative process earlier.

This per-pixel strength control allows for gradual spatial transitions in edits, better integration of inpainted elements (soft-inpainting) with reduced artifacts, and finer control over the extent to which different image regions are modified, all without requiring model retraining. Enabling this technique involves specific nodes in the ComfyUI workflow, such as `VAEEncodeForInpaint`, `InpaintModelConditioning`, and potentially a custom `DifferentialDiffusion` sampler or node setup that implements the described timestep-based latent mixing logic.

5.6 Texture Application and Post-Processing Implementation

This section details the implementation of methods responsible for applying the generated viewpoint images onto the 3D models within Blender, as well as post-processing steps like UV inpainting to fix occlusions and texture baking to create standardized UV maps.

5.6.1 Texture Projection and Material Management (bpy API)

Once images are generated for each viewpoint by the ComfyUI backend, they need to be applied to the surfaces of the scene objects visible from those viewpoints. The core logic for this resides in the `project_image` function within `project.py`. This function dynamically creates or modifies Blender materials to achieve a weighted blend of the projected textures.

UV Projection Setup

For each camera-object pair considered in a generation pass, the `project_image` function generates a unique UV map using Blender's UV Project modifier (`bpy.ops.object.modifier_apply` with a temporary `UVProject` modifier). The modifier is configured to use the specific camera as the projector object, with aspect ratios automatically calculated based on the render resolution. These generated UV maps (e.g., `ProjectionUV_<i>_<mat_id>`) are then used within the material node tree to map the corresponding generated textures.

Because Blender limits meshes to a maximum of 8 UV map slots, this projection method faces a constraint when many viewpoints are desired. If the number of viewpoints exceeds the available slots, the plugin cannot store all unique projection maps simultaneously. While the `project.py` module

contains a helper (`simple_project_bake`) for intermediate baking during generation (primarily intended for the sequential mode workflow to mitigate this), enabling this feature impacts performance due to repeated baking operations. Therefore, for workflows involving a large number of viewpoints, careful consideration of this limit or reliance on the final Texture Baking step is necessary.

Material Node Tree Construction

The `project_image` function constructs a potentially complex material node tree for each object's material slot. For each generated viewpoint image, a `ShaderNodeTexImage` is created, loaded with the image, and mapped using a `ShaderNodeUVMap` node pointing to the corresponding unique projection UV map created earlier.

Weighted Blending Logic

To seamlessly blend textures from different viewpoints based on surface angle and visibility, a weighted blending approach is implemented using material nodes:

1. **Weight Calculation (OSL):** A per-viewpoint weight is calculated using a `ShaderNodeScript` loading an external `raycast.osl` script. This script takes various inputs, including geometry data, camera parameters (accessed via intermediate nodes), and a new `Power` parameter linked to the user's *Weight Exponent* setting. The core calculation within the script now computes the weight as $W = \text{pow}(\texttt{orthogonality}, \texttt{Power})$, where `orthogonality` represents $|\cos(\theta)|$ and θ is the angle between the view direction and the surface normal. The script still incorporates checks for camera frustum visibility and occlusion, outputting zero if the point is not directly visible from the viewpoint or if the angle exceeds a maximum threshold.

2. **Hierarchical Mixing:** The individual projected textures and their corresponding calculated weights are fed into the recursive `build_mix_tree` helper function. This builds a binary tree of `ShaderNodeMixRGB` nodes. Each node blends two inputs using a dynamically calculated factor derived from their weights. This process continues recursively until a single blended color output remains. This blending is illustrated conceptually in Figure 5.6.

3. **Fallback and Final Output:** The final output of the mix tree is blended one last time with a user-defined fallback color. The factor for this final mix is determined by the total accumulated weight processed through a

`ShaderNodeValToRGB` (ColorRamp), effectively masking areas not covered sufficiently by any projection. This final color is then connected to a `ShaderNodeBsdfPrincipled` node and the `ShaderNodeOutputMaterial`.

This OSL-based dynamic weighting ensures smooth transitions between viewpoint projections without hard seams, as demonstrated on a simple model in Figure 5.7.

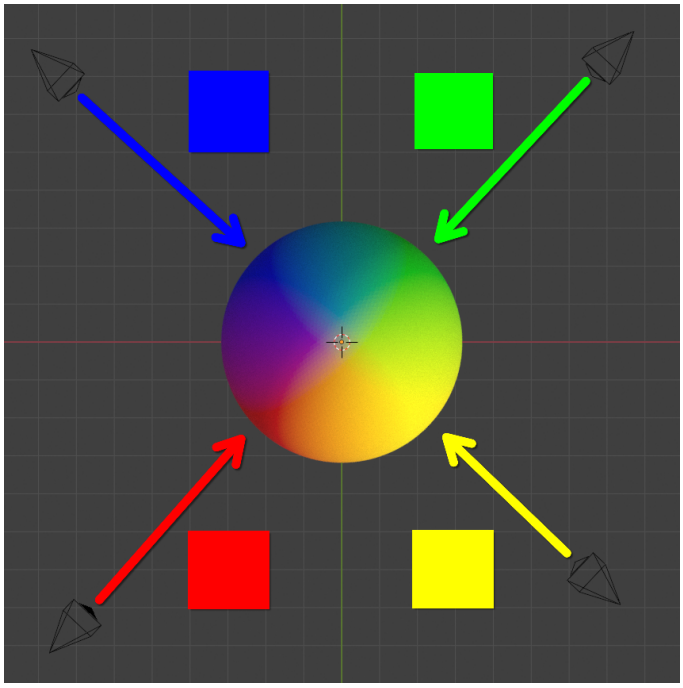


Figure 5.6: Demonstration of the OSL-based weighted blending on a sphere using solid colors projected from multiple viewpoints.

5.6.2 Texture Baking Implementation (bpy Bake API)

Texture baking converts the complex, projection-based material setup into a standard, single texture image per object, suitable for export or simpler rendering setups. This functionality is primarily implemented in the `BakeTextures` modal operator within `render_tools.py`. An example result is shown in Figure 5.8

The operator iterates through mesh objects and performs several phases:

1. **Unwrap Phase (Optional):** The `unwrap` utility function ensures a suitable target UV map (e.g., 'BakeUV') exists, unwrapping if needed based on user settings.

2. **Bake Phase:** For each object, the `bake_texture` utility function is called. This function sets up Cycles bake settings, creates a target image,

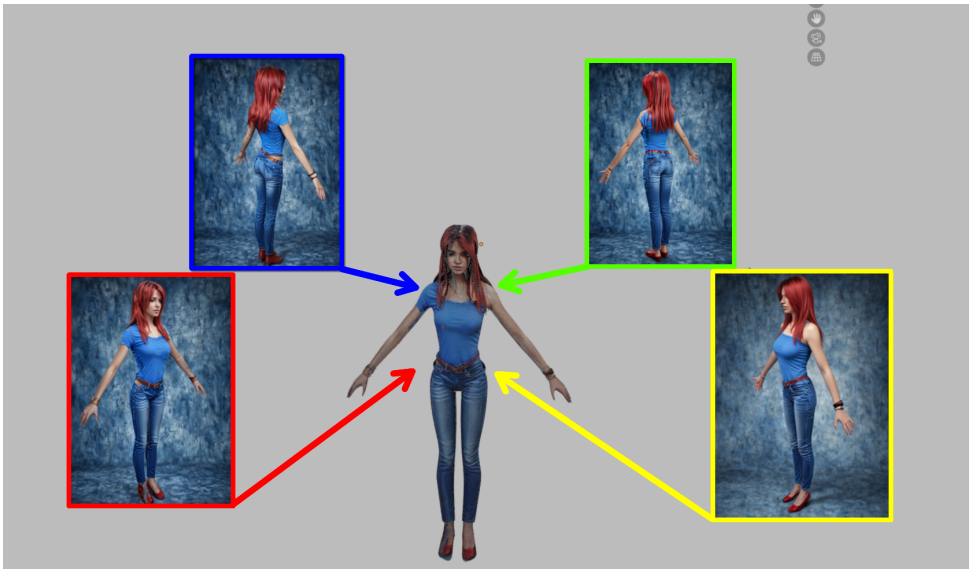


Figure 5.7: Example of generated textures applied to a simple model using the projection method.

selects it in the active material's node tree, executes `bpy.ops.object.bake(type='DIFFUSE')`, saves the resulting image, and cleans up. It effectively bakes the final color output of the complex projection material.

3. **Apply Material Phase (Optional):** The `add_baked_material` helper can create a new, simple material using the baked texture image mapped via the target UV map.

(a) : Generated Image (Grid)

(b) : Images Projected onto Model

(c) : Final Baked Texture (UV Map)

Figure 5.8: The texture baking process is shown, where generated images (a), exemplified here by Grid mode output, are first projected onto the 3D model (b) via the dynamic material setup. Finally, this projected appearance is consolidated into a single texture map (c) that aligns with the object's standard UV coordinates.

5.6.3 UV Inpainting Implementation

This process addresses areas missed during multi-view generation due to occlusion, operating directly in the UV space of a baked or standard texture map. The steps are visualized in Figure 5.9. A showcase of a model before and after the UV inpainting process is depicted in Figure 5.10

1. **UV Map Requirement:** A standard, non-overlapping UV map is necessary. The `unwrap` utility can prepare this if needed.
2. **Visibility Mask Generation:** The `export_visibility` function calculates visibility based on the projection material's blend weights and bakes this onto the standard UV map to create a mask image.
3. **Inpainting Execution:** The main generation logic (`refine` or `refine_flux`) uses the existing baked texture as input and the generated UV visibility map as the mask for the ComfyUI inpainting workflow. Object-specific prompts enhance results.
4. **Applying Result:** The `apply_uv_inpaint_texture` function updates the material node tree, connecting the newly inpainted texture to replace the fallback color input.

(a) : Original Baked Texture

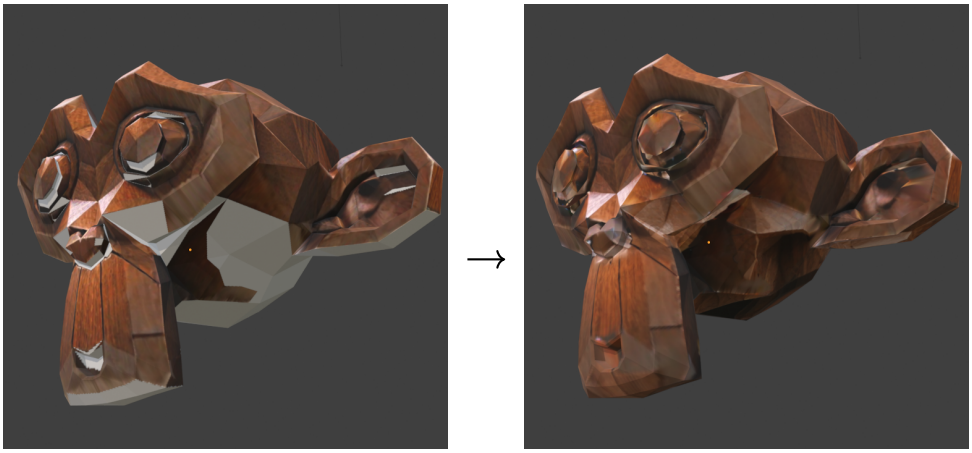
(b) : Visibility Mask (UV space)

(c) : Final Inpainted Texture

Figure 5.9: This figure illustrates the UV Inpainting process. An original baked texture (a) might exhibit gaps due to occluded areas during initial generation. A visibility mask (b) is then generated in UV space to precisely identify these missing regions. Finally, diffusion-based inpainting is applied to fill these gaps, resulting in a more complete final texture (c).

5.6.4 Texture Refinement

Beyond initial generation, Stablegen supports refining textures using another generation pass. This is primarily useful for adding localized detail or for adjusting/improving existing textures. The behavior depends on the *Preserve Original Textures* setting (`context.scene.refine_preserve`), which



(a) : Model Before UV Inpainting **(b) :** Model After UV Inpainting

Figure 5.10: This figure shows the effect of UV Inpainting on a 3D model. Initially, areas not covered by any viewpoint projection display the fallback color, often gray (a). After the UV Inpainting process is applied, these previously untextured areas are filled with generated content (b).

dictates how the new generation pass interacts with the previous material node setup. In both modes, the existing visual appearance (rendered via `export_emit_image`) is used as the input image for the img2img refinement process initiated by the `refine` or `refine_flux` functions in `generator.py`.

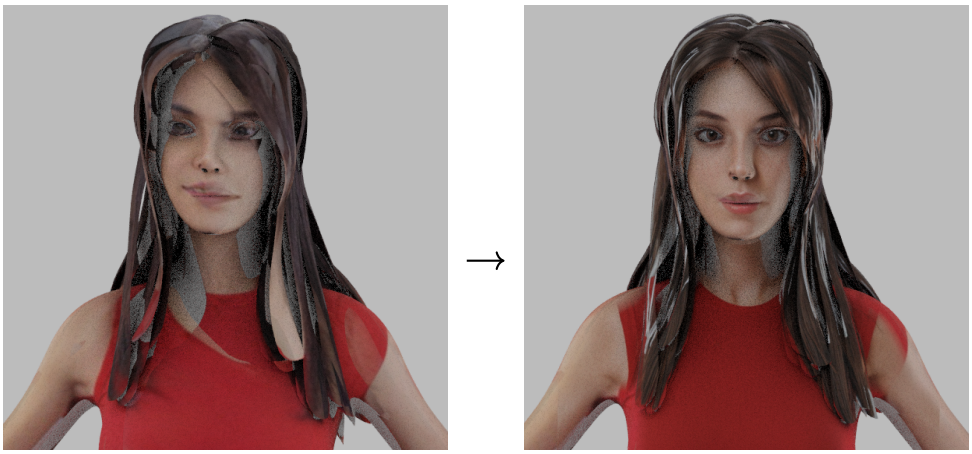
Refinement with Preservation (Layering)

This mode is active when *Preserve Original Textures* is enabled. It is designed for adding details or making localized changes while keeping the underlying texture from previous passes visible in areas not covered by the new pass (e.g., refining a character's face after a full-body texture pass, as conceptualized in Figure 5.11).

This functionality leverages the recursive structure of the material blending tree created by `build_mix_tree` within `project.py`. When a refinement pass is initiated with preservation enabled, `project_image` builds a new node tree for the new viewpoints/prompt. Crucially, the `build_mix_tree` function identifies the final output node (specifically, the `MixRGB` node blending with the fallback color) from the previous material setup (`last_node`). It then connects this previous result to the `Color2` input of the corresponding final `MixRGB` node in the *new* node tree.

The effect is that the newly generated texture (from the current pass, connected to `Color1`) is blended *over* the result of the previous pass (`Color2`) based on the OSL weight calculation for the new viewpoints. Where the new pass has strong coverage (high weight), the new texture dominates. Where the new pass has no coverage (0 weight), the final mix defaults towards

`Color2`, which contains the fully blended result from the previous generation pass(es), thus preserving the old texture in those areas. The blending can be further improved by adjusting the automatically provided *Color Ramp* node, which will make the transition more seamless.



(a) : Texture Before Refinement **(b) :** Texture After Targeted Refinement

Figure 5.11: A conceptual example of texture refinement with preservation is shown. An initial texture (a) provides the base. A subsequent generation pass (b), which can be targeted to a specific area (such as the face in this example) using a new prompt, is then layered over the original texture by means of the node tree blending logic.

Refinement with Replacement (Stylization/Improvement)

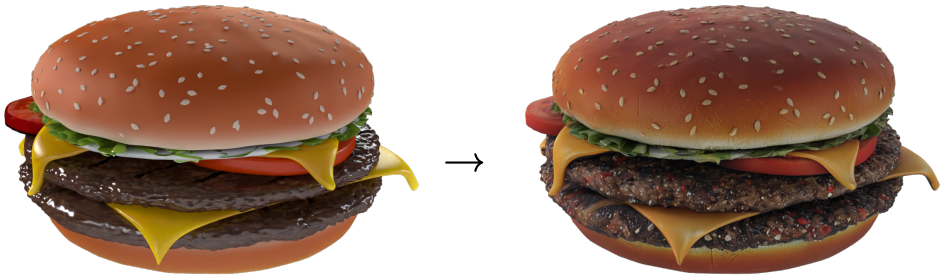
This mode is active when *Preserve Original Textures* is disabled. It is useful for cases where the goal is to completely re-style, adjust, or improve an existing texture based on a new prompt or parameters, without necessarily layering it over the old version in areas of low coverage. For instance, taking a base texture generated with one style prompt and running another pass with a different style prompt to transform its appearance (demonstrated in Figure 5.12).

When preservation is disabled, the `project_image` function still uses the rendered appearance of the existing texture as the input image for the backend img2img process (`refine` or `refine_flux`). However, when constructing the material node tree, the previous node setup is effectively discarded. The `build_mix_tree` function is called without knowledge of the previous tree's output (`last_node` is not connected). It builds a new blending hierarchy based *only* on the viewpoints and generated textures from the current refinement pass.

The resulting material node tree completely replaces the previous one. While the diffusion model used the old texture as a starting point for generation (img2img), the final render in Blender only uses the result of the current

pass, blended with the fallback color in areas of low coverage for *this specific pass*. This allows for significant alteration or complete replacement of the texture's appearance based on the new prompt and generation parameters.

This method also has the advantage of not needing the previous texture to be generated using StableGen, as it doesn't rely on the proprietary node structure.



(a) : Original Texture **(b) :** After Refinement (Replacement)

Figure 5.12: This figure demonstrates texture refinement with replacement. An original texture (a) serves as the input for an image-to-image process guided by a new prompt or settings. The outcome is a restyled or adjusted texture (b) which entirely replaces the previous material setup. In this case, manual textures from Blend Swap [2] were used with denoise parameter at 0.7.

5.7 Implementation of Generation Mode Logic

This section details the specific algorithms implemented within the Stablegen plugin for the different multi-view texture generation strategies: Grid Mode and Inpainting Mode. It builds upon the previously described mechanisms for backend communication and texture projection.

5.7.1 Grid Mode Logic

The Grid Mode processes multiple viewpoints simultaneously in a single initial diffusion pass. This logic is coordinated within the `_async_generate` method in `generator.py` when the scene's `generation_method` is set to `'grid'`.

1. **Control Map Combination:** For each enabled and required ControlNet type (e.g., depth, canny, normal), the corresponding per-viewpoint images, previously exported, are collected. The `combine_maps` helper

function is called for each type. This function uses the Pillow (PIL) library via the `create_grid_image` method to arrange these individual map images into a single, square-like grid image.

2. **Resolution Rescaling:** The combined grid image for each control type is then rescaled by the `rescale_to_1mp` method, again using PIL, to ensure the total resolution is approximately 1 megapixel and the dimensions are divisible by 8, compatible with standard diffusion model requirements. The final dimensions (`_grid_width`, `_grid_height`) are stored for configuring the latent image size.

3. **Single Generation Call:** A single request is made to the ComfyUI backend by calling the appropriate generation function (`generate` or `generate_flux`). The constructed ComfyUI workflow JSON includes an `EmptyLatentImage` node configured with the calculated grid dimensions, and the combined grid images are passed as inputs to the respective `LoadImage` nodes within the ControlNet chain(s).

4. **Result Splitting:** Upon receiving the single generated grid image back from ComfyUI, the `split_generated_grid` helper function is called. This uses PIL to crop the grid image back into individual tile images, corresponding to the original camera viewpoints based on the stored grid dimensions. These individual images are saved to the output directory with appropriate naming (e.g., incorporating the camera index and material ID).

5. **Refinement Pass (Optional):** If the user enables the `refine_images` scene property, the logic proceeds to iterate through the newly split viewpoint images. For each split image, the corresponding refinement function (`refine` or `refine_flux`) is called. This initiates a separate img2img process for each viewpoint, using the original per-viewpoint control map(s) and the corresponding split image as the input image. The refined images then overwrite the initially saved split images.

After the generation, and optional refinement, is complete for all viewpoints in the grid, the `async_generate` function schedules the `image_project_callback`, which in turn calls the `project_image` function (detailed in the previous section) to apply these final individual viewpoint images onto the scene objects using the standard projection and blending material setup. A visualization of this process can be seen in Figure 4.4.

5.7.2 Inpainting Mode Logic

The Inpainting Mode, internally referred to as the `'sequential'` generation method, processes viewpoints one by one. It uses the projected result of the previous step to provide context for generating the current viewpoint via inpainting, aiming for high detail and consistency. This mode involves

coordination between the background generation thread (`_async_generate` in `generator.py`) and Blender's main thread using timer callbacks and synchronization events.

1. **First Viewpoint Generation:** The process starts simply by generating the texture for the first camera viewpoint (index 0) using a standard text-to-image call (`generate` or `generate_flux`) with its corresponding control signals.

2. **Projection and Context Export (Main Thread Callback):** After the image for viewpoint i is generated in the background thread, `_async_generate` schedules the `image_project_callback` function to run in Blender's main thread via `bpy.app.timers.register`. This callback performs crucial steps before the next viewpoint can be generated:

   - It calls the `project_image` function (from `project.py`). This function applies the generated textures for viewpoints 0 through i by setting up the projection UV maps and updating the material node tree blend.
   - It then prepares the context for the *next* viewpoint (i+1) by calling export functions (from `render_tools.py`): `export_visibility` (using render mode) creates the inpainting mask by rendering which parts of the scene already have texture from viewpoints 0 to i when viewed from camera i+1, and `export_emit_image` renders the currently projected texture colors from camera i+1 to serve as the RGB context image.
   - Finally, the callback signals a `threading.Event` (`_wait_event`) to notify the background thread that projection and context export are complete.

3. **Subsequent Viewpoint Inpainting (Background Thread):** The `_async_generate` function, running in the background thread, waits for the `_wait_event` signal. Once received, it retrieves the file paths for the visibility mask and RGB context render corresponding to viewpoint i+1. It then calls the appropriate refinement function (`refine` or `refine_flux`), constructing an inpainting workflow. This workflow uses the exported RGB context render as the input image and the visibility map as the mask input (likely processed further by nodes like `GrowMask` or `ImageBlur` within the ComfyUI workflow). The relevant ControlNet signals for viewpoint i+1 are also included.

4. **Iteration:** The inpainted result for viewpoint i+1 is saved. The background thread increments the viewpoint index and schedules the `image_project_callback` again to project this latest result and export context for the next view (i+2). This loop continues until all viewpoints are processed.

5. **IPAdapter Integration (Sequential):** When the `sequential_ipadapter` option is enabled, the logic within `_configure_sequential_mode` (called by `refine`/`refine_flux`) dynamically sets the input image for the IPAdapter ComfyUI node. Based on the `sequential_ipadapter_mode` setting, it uses either the saved image from the very first viewpoint (index 0) or the image saved from the immediately preceding viewpoint (i-1) to guide the current step (i).

This sequential, stateful process allows for detailed generation at native resolution for each view while actively enforcing consistency through the masked inpainting approach informed by previous projections. The final projection after the last viewpoint applies the complete result. An illustration of the process on two viewpoints can be seen in Figure 4.5.

5.7.3 Providing Context for Inpainting Mode

The Inpainting Mode relies on providing the diffusion model with specific contextual information from previously generated viewpoints to guide the generation process for subsequent views. This context comprises a visibility mask, indicating already textured areas, and an RGB render showing the current state of the projected textures. The implementation leverages functions within `render_tools.py` and coordinates their execution within the main generation logic in `generator.py`.

Visibility Map Exporting (Sequential Mode)

For the Inpainting Mode, the visibility mask identifies which parts of the model are already textured from previous viewpoints (i) when seen from the perspective of the next viewpoint (i+1). This mask is crucial for telling the diffusion model which areas to inpaint in the current step.

The core implementation resides in the `export_visibility` function within `render_tools.py`. This function is called by the main generation operator's callback (`image_project_callback` in `generator.py`) after viewpoint i's texture has been generated and projected. It operates by temporarily modifying the active material(s) on the mesh objects using the `prepare_material` helper function. This modification intercepts the final blended color output from the hierarchical `MixRGB` node tree (which represents the accumulated weight from viewpoints 0 to i) before it reaches the BSDF shader.

Crucially, the way this visibility value is processed depends on the *Use smooth mask* setting (`context.scene.sequential_smooth`):

- **Binary Mask (Smooth Mask Disabled):** If *Use smooth mask* is turned off, a `ShaderNodeMath` node set to the 'COMPARE' operation is

inserted. It compares the incoming blended weight value against a threshold defined by the *Sequential Factor* (`context.scene.sequential_factor`). If the weight is greater than the threshold, the node outputs 1 (white, visible); otherwise, it outputs 0 (black, not visible/to be inpainted). This creates a sharp, binary mask.

- **Smooth Gradient Mask (Smooth Mask Enabled):** If *Use smooth mask* is enabled, a `ShaderNodeValToRGB` (Color Ramp) node is used instead of the 'COMPARE' node. This node receives the blended weight value as input. Its interpolation is set to 'LINEAR', and it's configured with two color stops (black at position 0, white at position 1). The transition point between black and white is controlled by the *Sequential Smooth Factor* (`context.scene.sequential_factor_smooth`). This setup results in a grayscale mask with a smooth gradient in areas where the calculated weight is near the transition threshold, rather than a hard edge.

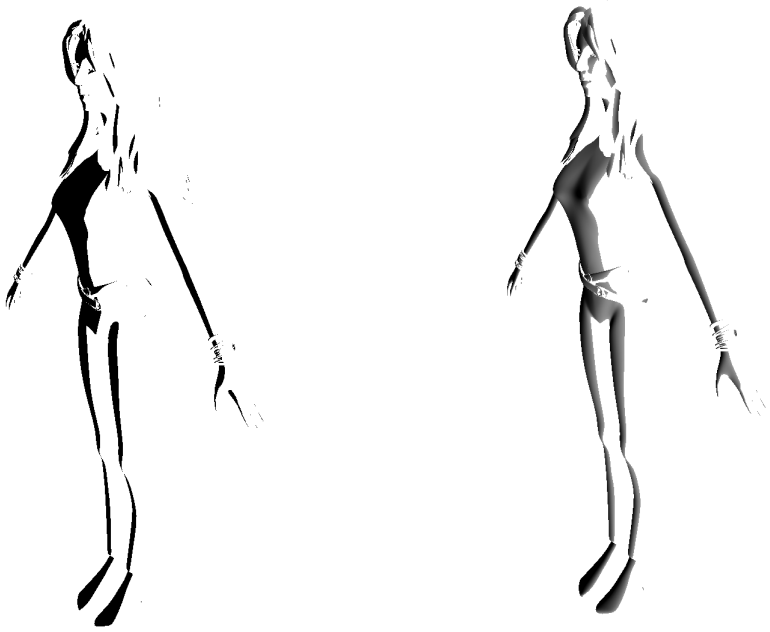
After modifying the material node tree with either the 'COMPARE' or 'Color Ramp' node, the `export_visibility` function calls `export_emit_image`. This renders the scene from the perspective of the next camera (viewpoint i+1) using the temporarily altered material, capturing the calculated visibility map (either binary or smooth gradient) as an image (see Figure 5.13).

Visibility Map Post-Processing (ComfyUI)

Raw visibility masks generated directly from the rendering process can sometimes have sharp edges where textured and untextured regions meet. Using such masks directly for inpainting can occasionally lead to noticeable seams or abrupt transitions in the final texture. To mitigate this and encourage smoother blending, the exported visibility mask often undergoes post-processing within the ComfyUI workflow before being used by the diffusion model.

Once the visibility mask is exported, it is passed to the ComfyUI backend as the mask input for the inpainting workflow (`refine` or `refine_flux` methods in `generator.py`). Within the dynamically constructed ComfyUI JSON workflow, the mask image is loaded using a `LoadImage` node. Before being used by the core inpainting nodes (`VAEEncodeForInpaint` or `InpaintModelConditioning`), the mask undergoes optional post-processing based on user settings configured in the Stablegen UI:

- **Grow Mask:** A `GrowMask` node can expand the boundaries of the masked (inpainting) regions by a specified pixel amount (`context.scene.grow_mask_by`). This creates a small overlap with the existing textured area, giving the inpainting model a transition zone.



(a) : Binary Mask ('Compare' Node)

(b) : Smooth Gradient Mask ('Color Ramp' Node)

Figure 5.13: A side-by-side comparison illustrates different methods for generating visibility masks for sequential inpainting. One approach yields a binary mask (a), created using a 'COMPARE' node when smooth masking is disabled. The alternative produces a smooth gradient mask (b), generated with a 'Color Ramp' node when smooth masking is enabled.

- **Blur Mask:** Subsequently, an `ImageBlur` node can apply a Gaussian blur to the mask if enabled (`context.scene.blur_mask`). The blur intensity is controlled by the `blur_mask_radius` and `blur_mask_sigma` parameters. Blurring the mask edges softens the transition boundary, further helping the diffusion model to seamlessly integrate the newly generated pixels with the surrounding context.

The final, potentially grown and blurred, mask is then connected to the appropriate input of the inpainting VAE encoder or conditioning node, guiding the diffusion model to modify only the designated areas while facilitating a smoother blend at the edges.

RGB Context Render Exporting

For the sequential Inpainting Mode, besides the mask, an RGB image representing the current state of the texture projected onto the model (as seen from the *next* viewpoint) is required as context for the diffusion model.

This RGB context render is generated by the `export_emit_image` function

in `render_tools.py`, also called by the main operator's callback (`image_project_callback`) after projecting the result of viewpoint i and before generating viewpoint i+1. Similar to visibility map generation, this function temporarily modifies the object materials using the `prepare_material` helper. However, instead of outputting a visibility value, the modification ensures that the final blended color from the projection node tree is directly connected to the material output, bypassing the BSDF shading.

The scene is then rendered using the Cycles engine with minimal samples for speed. Specific compositor nodes might be used to combine emission or environment passes if necessary, ensuring the render accurately captures the appearance of the already-applied textures from the perspective of the upcoming camera view. The resulting RGB image (Figure 5.14) is saved and subsequently used as the main input image for the ComfyUI inpainting workflow for the next viewpoint.

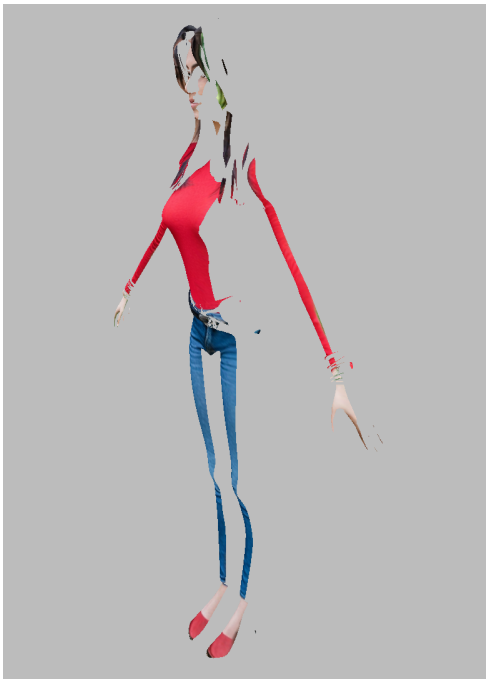


Figure 5.14: Example of an exported RGB context render, showing the state of the projected texture from the perspective of the next viewpoint to be inpainted (corresponding to Step 2b in Figure 4.5).

Chapter 6

Evaluation and Testing

6.1 Evaluation Methodology

To assess the effectiveness of the StableGen plugin and compare its different generation modes, a multi-faceted evaluation approach was employed, combining a perceptual user study with qualitative visual analysis. This section details the methodology used for generating test cases and collecting perceived quality data.

6.1.1 Technical Setup

All texture generations were performed using the StableGen plugin operating within Blender [Specify Blender Version, e.g., 4.1]. The plugin communicated with a ComfyUI backend running the RealVisXL V5.0 SDXL checkpoint [30]. Only a single depth-based ControlNet model was used during this evaluation. IPAdapter conditioning utilized the OpenCLIP ViT-H/14 image encoder. Tests were run on an NVIDIA RTX 3070 Laptop GPU with 8 GB VRAM.

6.1.2 Test Assets

Six distinct 3D assets were selected to evaluate the plugin's performance across a range of complexities, styles, and object types. Sources are provided where applicable:

1. **Woman:** Prompt: "a woman, blue jeans, red shirt, in a park, photo". A detailed character model in an A-pose, testing prompt adherence (clothing color), and handling of surfaces like hair, face, and thin geometry. (Source: Blendswap [9])
2. **Car:** Prompt: "green car". A detailed 1967 GTO model, testing consistency on hard surfaces (paint, windows), handling of different angles, and avoidance of lighting artifacts. (Source: Blendswap [35])

3. **Subway Scene:** Prompt: "subway station". A multi-object scene including machinery, architecture, and furniture within an enclosed space, testing consistency across multiple objects and large surfaces (floor, ceiling). (Source: Blendswap [1])
4. **Anime Head:** Prompt: "anime girl head, red hair". A stylized character head model, testing the ability to generate non-photorealistic styles and handle details like stylized hair and facial features. (Source: Blendswap [36])
5. **Burger:** Prompt: "a cheeseburger, with tomatoes". A complex object with multiple distinct layers and materials (bun, meat, cheese, vegetables), testing the generation of realistic food textures. (Source: Blendswap [2])
6. **Greek House:** Prompt: "antic greek house". An architectural model, testing texture generation for flat surfaces, consistency, and the ability to imply detail (like tiles or bricks) not present in the base geometry. The 3D model was provided by a fellow student, Veronika Soboličová, as part of her work on the Politeia 21 project [32].

6.1.3 Generation Methods Compared

The following configurations of the StableGen plugin were compared. For the Sequential and Separate modes using IPAdapter (**Sequential (IPAdapter)**, **Separate (IPAdapter)**, **Sequential + UV Inpaint**), the first generated image was used as the IPAdapter reference for assets 1-5, while a provided external image was used for asset 6 (Greek House). Notably, the Grid modes (**Grid**, **Grid + Refine**) also utilized IPAdapter, but *only* for asset 6, using the same external reference image. All inpainting steps (Sequential methods, UV Inpaint) utilized a Differential Diffusion workflow.

- **Sequential (IPAdapter):** Inpainting mode (4.5.2) with IPAdapter enabled.
- **Grid:** Basic Grid mode (4.5.1) generation.
- **Grid + Refine:** Grid mode followed by the second refinement pass.
- **Separate (IPAdapter):** Independent generation per view, with IPAdapter enabled.
- **Sequential (No IPAdapter):** Inpainting mode without IPAdapter.
- **Separate (No IPAdapter):** Independent generation per view, without IPAdapter.
- **Sequential + UV Inpaint:** Sequential mode (with IPAdapter) followed by UV inpainting pass (tested on asset 1 only).
- **Manual Textures:** Manually created textures (Available for assets 2 and 5 only).

6.1.4 Generation Parameters

Consistent generation parameters were used across different methods for each test asset where feasible to ensure a fair comparison. Key parameters are detailed in Tables 6.1 and 6.2. The random seed was kept constant at 42 for reproducibility during development but is not detailed further as the focus is on method comparison.

Table 6.1: Standard generation parameters (assets 1, 2, 4, 5, 6)

Parameter	Value
Steps	10
CFG Scale	1.5
Sampler	`DPM++ 2S Ancestral`
Scheduler	`SGM Uniform`
Control After Generate	Fixed
LoRA	SDXL Lighting LoRA (8step) [4]

Table 6.2: Specific generation parameters (asset 3 - subway scene)

Parameter	Value
Steps	25
CFG Scale	5.0
LoRA	None
Other Parameters	*As per Table 6.1*

The number of viewpoints and use of camera-specific prompts also varied, as detailed in Table 6.3.

Table 6.3: Viewpoint configuration per asset

Asset	No. Cameras	Camera-Specific Prompts Used?
Woman (Asset 1)	6	Yes
Car (Asset 2)	6	Yes
Subway Scene (Asset 3)	5	No
Anime Head (Asset 4)	6	Yes
Burger (Asset 5)	5	No
Greek House (Asset 6)	6	No

Minor adjustments to other parameters (e.g., ControlNet strength, refinement settings) were occasionally made per asset to achieve visually better results, but these adjustments were kept consistent across the different generation methods being compared for that specific asset whenever possible.

6.1.5 Perceptual Quality Survey

To gather comparative data on perceived texture quality, an online survey was conducted using the OpinionX platform [26].

- **Methodology:** The survey employed the *Pair Rank* question format [20]. Each of the $N = 22$ participants was presented with a fixed number of randomly selected pairs of visual results. These results consisted of rendered images or spinning GIFs generated by the various configurations of StableGen and, where applicable, manually created textures for each test asset. Participants were asked to select the image in each pair with the higher perceived texture quality. Figure 6.1 provides an example of this Pair Rank interface. While other AI-based texturing solutions for Blender were considered, they were not included due to practical challenges encountered during preliminary evaluations, such as limitations in handling the multi-mesh structure of the test assets or significant difficulties in achieving consistent baseline outputs suitable for direct comparison.
- **Scoring:** OpinionX calculates a score for each method based on its win rate in these head-to-head comparisons (number of wins divided by the total number of pair comparisons involving that method), producing a ranked list with scores from 0-100 [20].
- **Data Presentation:** The primary output is a ranked list of the generation methods for each test asset based on perceived quality according to the survey respondents.

6.1.6 Qualitative Analysis

Alongside the survey results, a qualitative visual analysis of the generated GIFs and renders will be performed. This involves inspecting the results for detail fidelity, prompt adherence, multi-view consistency, style coherence (especially regarding IPAdapter), geometric accuracy (ControlNet effectiveness), presence of artifacts, and overall aesthetic appeal. Visual examples will be used to illustrate the strengths and weaknesses of each method and to provide context for the survey rankings.

6.2 Quantitative Texture Quality Evaluation

This section presents the results derived from the perceptual quality survey conducted using the OpinionX platform, comparing the different texture generation methods based on participant preferences ($N = 22$).

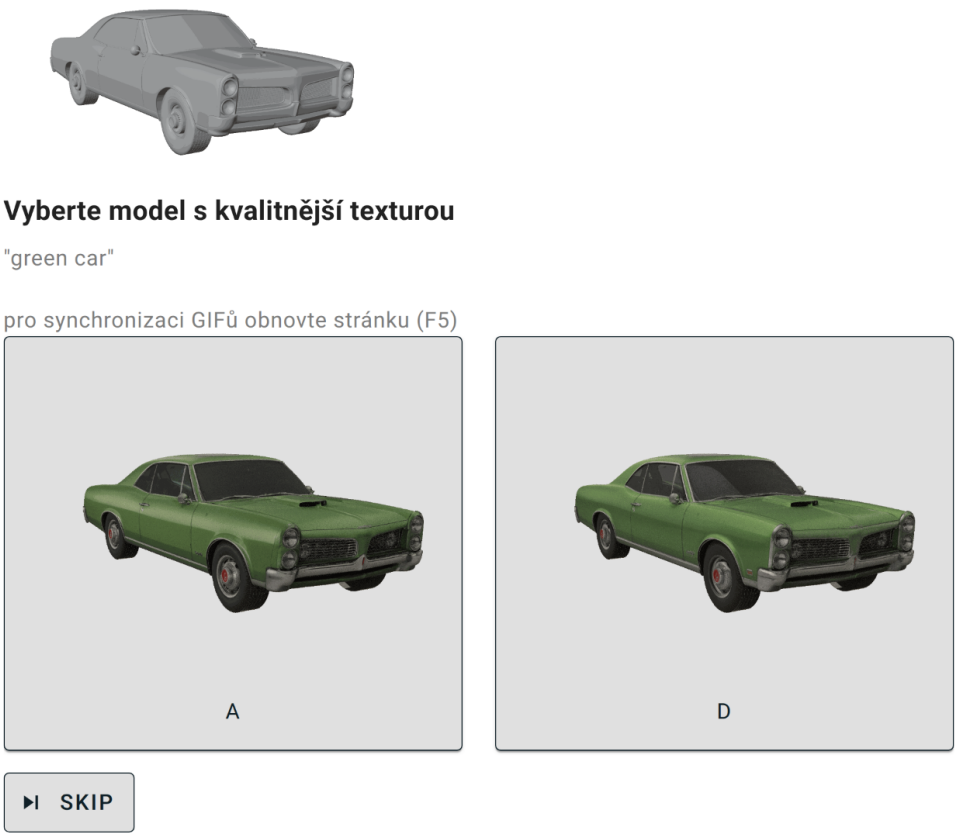


Figure 6.1: An illustrative example of the Pair Rank comparison interface used in the OpinionX survey. Participants were shown two spinning GIFs side-by-side, each representing a 3D asset textured by a different method or configuration, along with an untextured reference, and were asked to select the option with the higher perceived texture quality.

6.2.1 Survey Results

Participants completed pairwise comparisons ("Choose the better texture") between visual results (renders/GIFs) generated by the different methods for each test asset. The OpinionX platform calculated a score for each method based on its win rate (percentage of head-to-head comparisons won), resulting in the following rankings.

Asset 1: Woman

The perceived quality ranking for the Woman asset is presented in Table 6.4. Method 'Sequential + UV Inpaint (IPAdapter)' achieved the highest score, while the basic 'Grid' method ranked lowest.

Table 6.4: Perceptual quality ranking - woman asset

Rank	Method	Score (Win Rate %)
1st	Sequential + UV Inpaint (IPAdapter)	84
2nd	Sequential (No IPAdapter)	61
3rd	Separate (IPAdapter)	56
4th	Sequential (IPAdapter)	53
5th	Grid + Refine	50
6th	Separate (No IPAdapter)	40
7th	Grid	13

Asset 2: Car

The perceived quality ranking for the Car asset, including comparison with manual textures, is presented in Table 6.5. Notably, 'Sequential (No IPAdapter)' ranked highest among the AI methods, slightly ahead of 'Separate (IPAdapter)' and the 'Manual Textures'.

Table 6.5: Perceptual quality ranking - car asset

Rank	Method	Score (Win Rate %)
1st	Sequential (No IPAdapter)	69
2nd	Separate (IPAdapter)	65
3rd	Manual Textures	61
4th	Sequential (IPAdapter)	57
5th	Separate (No IPAdapter)	43
6th	Grid + Refine	30
7th	Grid	5

Asset 3: Subway Scene

The perceived quality ranking for the Subway Scene asset is presented in Table 6.6. The 'Separate (No IPAdapter)' and 'Sequential (IPAdapter)' methods scored very closely at the top.

Table 6.6: Perceptual quality ranking - subway scene asset

Rank	Method	Score (Win Rate %)
1st	Separate (No IPAdapter)	71
2nd	Sequential (IPAdapter)	70
3rd	Separate (IPAdapter)	61
4th	Sequential (No IPAdapter)	52
5th	Grid + Refine	33
6th	Grid	9

Asset 4: Anime Head

The perceived quality ranking for the Anime Head asset is presented in Table 6.7. 'Sequential (IPAdapter)' and 'Grid + Refine' performed best for this stylized model.

Table 6.7: Perceptual quality ranking - anime head asset

Rank	Method	Score (Win Rate %)
1st	Sequential (IPAdapter)	77
2nd	Grid + Refine	73
3rd	Sequential (No IPAdapter)	67
4th	Separate (No IPAdapter)	45
5th	Separate (IPAdapter)	44
6th	Grid	14

Asset 5: Burger

The perceived quality ranking for the Burger asset, including comparison with manual textures, is presented in Table 6.8. 'Grid + Refine' slightly outperformed 'Sequential (IPAdapter)', with both scoring higher than the 'Manual Textures'.

Table 6.8: Perceptual quality ranking - burger asset

Rank	Method	Score (Win Rate %)
1st	Grid + Refine	69
2nd	Sequential (IPAdapter)	67
3rd	Manual Textures	54
4th	Sequential (No IPAdapter)	48
5th	Separate (IPAdapter)	39
6th	Separate (No IPAdapter)	38
7th	Grid	28

Asset 6: Greek House

The perceived quality ranking for the Greek House asset is presented in Table 6.9. For this asset, relevant methods used an external IPAdapter reference image. 'Separate (IPAdapter)' ranked highest.

Table 6.9: Perceptual quality ranking - Greek house asset

Rank	Method	Score (Win Rate %)
1st	Separate (IPAdapter)	76
2nd	Separate (No IPAdapter)	64
3rd	Sequential (IPAdapter)	61
4th	Grid + Refine	60
5th	Grid	14
6th	Sequential (No IPAdapter)	11

6.3 Qualitative Analysis and Visual Showcase

This section complements the quantitative survey results by providing a qualitative examination of the textures generated by the StableGen plugin. Through visual examples and detailed analysis, we explore the strengths and weaknesses of the different generation methods across various criteria, offering deeper insights into their practical performance.

6.3.1 Structural Interpretation and ControlNet Guidance

A foundational challenge in AI-driven 3D texturing is the model's ability to correctly interpret an object's overall structure and its distinct parts. This structural understanding, coupled with how well textures adhere to the perceived geometry under ControlNet guidance, is a prerequisite for achieving

believable results. Failures or ambiguities in this initial interpretation phase significantly complicate the subsequent task of ensuring consistency across multiple viewpoints, as discussed in Section 6.3.2. This section examines these primary challenges related to how the AI perceives and renders the object's form.

Several types of errors were observed where the AI struggled to accurately interpret or render the intended structure for a given (conceptual) viewpoint, even before considering how multiple such views would later be combined. Perhaps the most striking was the *Grid* mode's performance on the Car asset, where it consistently failed to differentiate the vehicle's front from its back, generating two front ends, as can be seen in Figure 6.2. This points to a fundamental failure in object part differentiation for those specific viewpoints within the grid, likely exacerbated by the Grid mode's reliance on a single global prompt without viewpoint-specific instructions. Similarly, issues arose with the interpretation of layered or ambiguous components; on the Burger, the model, possibly misguided by ControlNet on the complex layers, sometimes generated lettuce texture on an area intended as the bottom meat patty. This suggests difficulties in accurately parsing components when visual cues for ControlNet might be ambiguous.

Models also sometimes hallucinated or erroneously generated structural features not present in the original geometry, or failed to render existing complex geometry accurately. For instance, non-IPAdapter *Separate* methods generated extra windows on the Greek House (see Figure 6.3), and the *Separate (No IPAdapter)* variant produced an erroneous headband on the Anime Head from a single viewpoint. These represent cases where the model's interpretation for that specific view deviated significantly from the source geometry.

The effectiveness of ControlNet in ensuring textures faithfully adhere to the perceived local geometric details is also influenced by several factors:

- **ControlNet Strength and Detail Trade-off:** The strength of ControlNet guidance is a critical parameter. For the Car, a relatively low ControlNet strength (0.5) was used to promote photorealism. This likely contributed to issues like slightly misaligned door seams and the generation of plausible but non-existent details (extra lights, vents) not present in the original mesh, as the model had more freedom to interpret and add detail. Increasing ControlNet strength could improve geometric adherence and prevent such hallucinations but might concurrently simplify textures or reduce perceived realism. This trade-off requires careful tuning.
- **Guidance in Areas with Poor Contextual Information:** While completely occluded areas default to the fallback color, issues arose for surfaces that were visible but provided very limited or ambiguous geometric context to ControlNet. For example, distant surfaces seen

through small openings, like parts of the Greek House interior viewed from afar through doorways, sometimes received nonsensical textures or exhibited strong, unexplained baked-in shadows. This occurs because ControlNet has insufficient clear geometric information to robustly guide the diffusion model in such poorly-constrained regions.

In summary, the initial stage of interpreting an object's structure and features for each intended view is fraught with potential for error, from misidentifying major parts to hallucinating details or struggling with complex local geometry. These foundational interpretation issues, along with the nuances of ControlNet application, directly impact the quality of individual view generations before even considering their consistency with other views.

(a) : Front View **(b) :** Back View (Generated as Front)

Figure 6.2: Example of fundamental consistency failure using the base *Grid* method on the Car asset. The generation intended for the back view (b) erroneously replicated the front view's features (a).

(a) : Untextured **(b) :** No IPAdapter **(c) :** IPAdapter

Figure 6.3: Demonstration of IPAdapter's role in preventing model hallucination of structural details on the Greek House. The untextured view (a) shows no windows on this particular facade. When textured using *Separate (No IPAdapter)* (b), additional windows are erroneously generated. In contrast, using *Separate (IPAdapter)* (c), with an external reference image guiding the style, correctly avoids generating these non-existent windows and adheres to the actual geometry. All other parameters remain the same.

6.3.2 Analysis of Multi-view Consistency

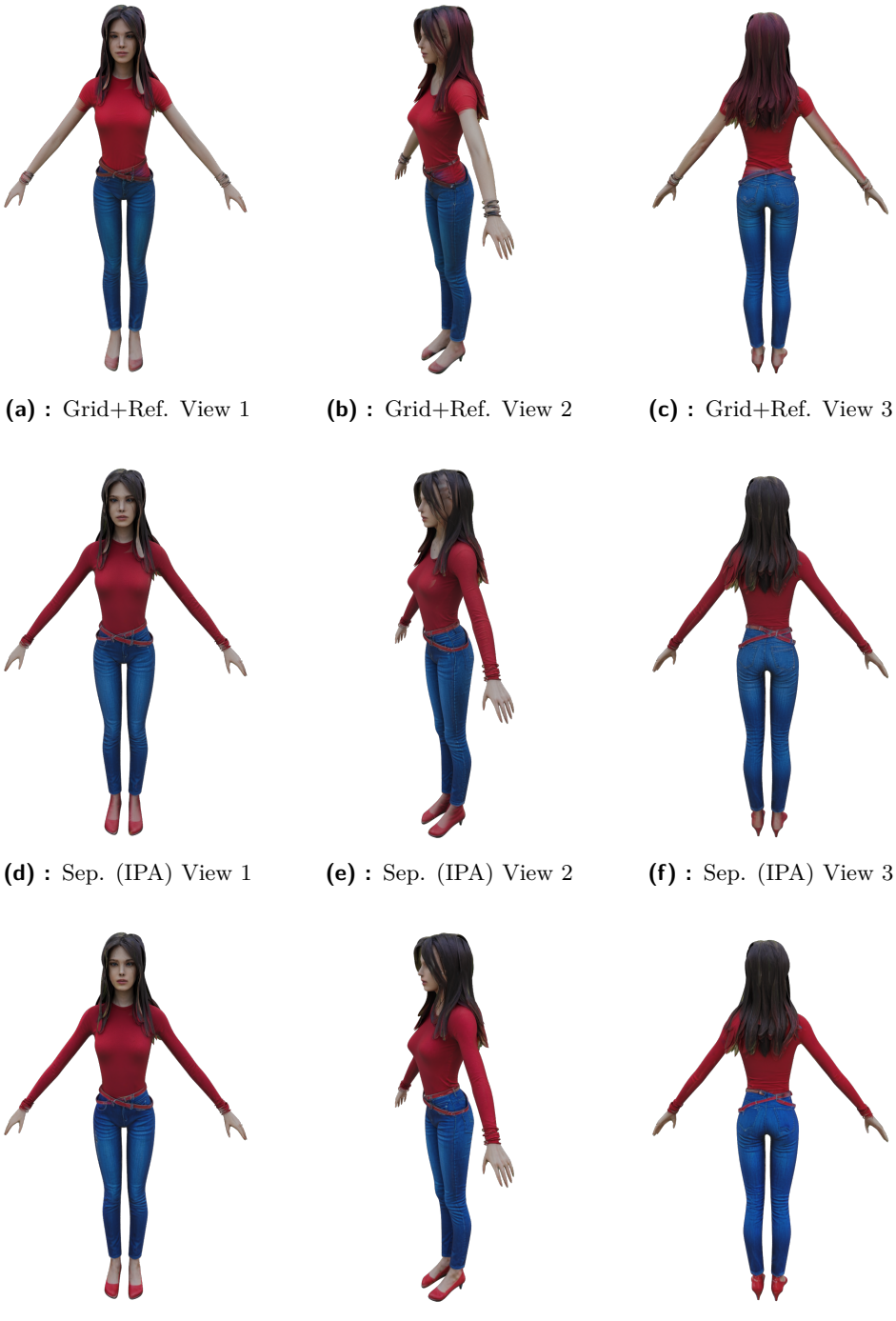
Following the challenges of accurately interpreting an object's structure for individual views, the subsequent task is to ensure consistency across these multiple views. This is particularly complex when initial structural

interpretations are ambiguous or vary from one viewpoint generation to another, requiring robust mechanisms to achieve a cohesive final texture. This subsection analyzes how well the different generation modes maintained consistency in terms of stylistic coherence (e.g., color, material appearance), the alignment of texture features at view boundaries to avoid visible seams, and the prevention of blurring from blending misaligned or slightly differing details.

The *Sequential* methods (both with and without IPAdapter, and including the UV Inpaint variant) generally produced a good degree of visual consistency across views, leveraging iterative inpainting. For instance, in the Subway Scene, *Sequential* modes achieved better alignment for wall textures compared to other methods, though they struggled with consistent floor and ceiling tile alignment due to challenging camera angles. However, stylistic consistency could falter without IPAdapter; *Sequential (No IPAdapter)* exhibited a progressive hair color shift on both the Woman and Anime Head assets (see Figure 6.6), and similarly produced a stylistically inconsistent concrete-like roof texture on one side of the Greek House. While inpainting generally blended transitions well, minor misalignments or reinterpretations could still occur, such as the appearance of a "double pocket" on the Woman's jeans with *Sequential (IPAdapter)*.

In contrast, the basic *Grid* mode, aside from its structural interpretation failures (like the Car's two fronts, discussed previously), also suffered from visual inconsistencies related to alignment across its simultaneously generated views. Textures often appeared blurry, and elements could be poorly defined (e.g., Woman's boots and belt details were indistinct). The *Grid + Refine* method, while improving detail, often inherited these underlying inconsistencies. Aside from that, *Grid* mode maintained reasonable style consistency across the multiple viewpoints; in this regard, it was far better than the *Separate* and *Sequential* methods *without IPAdapter*.

The *Separate* generation methods, producing each view independently, faced significant challenges in maintaining multi-view consistency without strong external guidance. A key issue was the inconsistent interpretation of features across views; for example, the *Separate (IPAdapter)* method struggled with the Woman's boots, where the model's differing interpretation of boot height from various angles led to noticeable blurring when these views were blended. Similarly, *Separate (No IPAdapter)* produced different sleeve lengths for the Woman from different viewpoints. Without IPAdapter, stylistic variations in lighting, color, or texture were more apparent between views, as seen in the Subway Scene where *Separate (No IPAdapter)* produced detailed but stylistically chaotic results. Mechanisms like IPAdapter and careful inpainting strategies are therefore crucial to help enforce or guide consistency when initial view generations might otherwise diverge.



(a) : Grid+Ref. View 1 **(b) :** Grid+Ref. View 2 **(c) :** Grid+Ref. View 3

(d) : Sep. (IPA) View 1 **(e) :** Sep. (IPA) View 2 **(f) :** Sep. (IPA) View 3

(g) : Seq. (IPA) View 1 **(h) :** Seq. (IPA) View 2 **(i) :** Seq. (IPA) View 3

Figure 6.4: Comparison of multi-view consistency on the Woman asset across different generation methods. Rows top to bottom: *Grid + Refine*, *Separate (IPAdapter)*, *Sequential (IPAdapter)*. Columns represent different viewing angles. Note differences in handling of hair boundaries, clothing seams, and boot consistency.

(a) : View 1 (2nd viewpoint)

(b) : View 2 (4th viewpoint)

(c) : View 3 (6th viewpoint)

Figure 6.5: Progressive hair color shift observed on the Woman asset using *Sequential (No IPAdapter)*. Note the potential change in hair tone (e.g., becoming redder) as generation progresses around the model.

(a) : View 1 (Front)

(b) : View 2 (Side)

(c) : View 3 (Back)

Figure 6.6: Progressive hair color shift observed on the Anime Head asset using *Sequential (No IPAdapter)*. Note the potential change in hair tone (e.g., becoming lighter/desaturated) as generation progresses around the model.

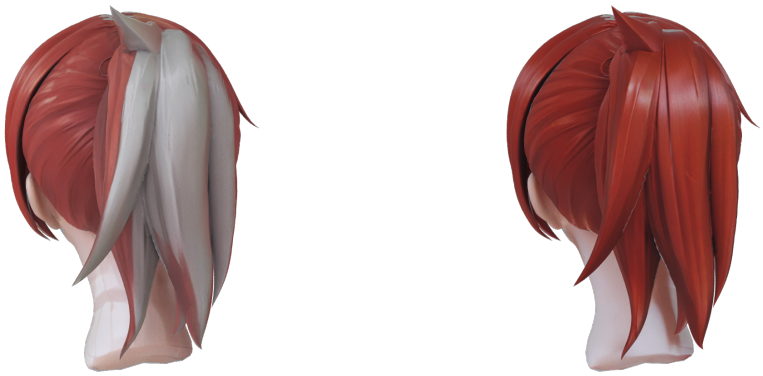
6.3.3 Effectiveness of IPAdapter for Style Coherence

IPAdapter is integrated into StableGen to allow users to guide the generation process using a reference image for stylistic consistency. This subsection assesses its impact based on the visual results.

The use of IPAdapter generally aimed to enforce a consistent artistic style and color palette across views. For assets like the Anime Head, using *Sequential (IPAdapter)* resulted in more consistent lighting and reflections compared to non-IPAdapter variants (*Sequential (No IPAdapter)* and *Separate (No IPAdapter)*). It also crucially mitigated a progressive hair color desaturation artifact seen in the *Sequential (No IPAdapter)* run for that asset. Similarly, for the Greek House, where an external, near lighting-less render was used as reference, IPAdapter (in both *Separate (IPAdapter)* and *Sequential (IPAdapter)* modes) was vital for maintaining architectural consistency and preventing the generation of spurious details like extra windows (see Figure 6.3), which occurred in the non-IPAdapter variants. For the

Subway Scene, *Sequential (IPAdapter)* provided a unified color palette and consistent textures (like wall tiles) throughout the station, contributing to a more coherent feel than the *Sequential (No IPAdapter)* version.

However, IPAdapter's influence wasn't always optimal. For the Car asset, the non-IPAdapter methods (*Sequential (No IPAdapter)* and *Separate (No IPAdapter)*) were slightly preferred in the survey, suggesting IPAdapter might have been too constraining or subtly degraded quality in that context. For the Woman asset, while IPAdapter in *Sequential (IPAdapter)* helped maintain hair color compared to its non-IPAdapter counterpart (*Sequential (No IPAdapter)*), both variants faced some challenges. The effectiveness of IPAdapter seems dependent on the asset, the quality of the reference image (first view or external), viewpoint placement, and potentially the desired level of creative variation versus strict consistency. Using a reference image with strong baked-in shadows, for instance, could propagate those shadows, whereas a lighting-less reference (as used for the Greek House) can help achieve a cleaner result.



(a) : *Sequential (No IPAdapter)* **(b) :** *Sequential (IPAdapter)*

Figure 6.7: Effectiveness of IPAdapter in maintaining hair color consistency for the Anime Head using sequential generation, viewed from the back. Subfigure (a) shows the result using *Sequential (No IPAdapter)*, exhibiting the progressive hair color desaturation/lightening towards the back of the head (as also seen in Figure 6.6). Subfigure (b) demonstrates how *Sequential (IPAdapter)*, using the first generated view as reference, successfully mitigates this artifact and maintains a consistent red hair color across the entire model.

6.3.4 Detail Rendering and Refinement

The level of detail in generated textures and the effectiveness of refinement processes are crucial for visual quality.

Methods generating views at full target resolution, such as the *Sequential*

(a) : Angle 1: *Sequential (IPAdapter)*

(b) : Angle 1: *Sequential (No IPAdapter)*

(c) : Angle 2: *Sequential (IPAdapter)*

(d) : Angle 2: *Sequential (No IPAdapter)*

Figure 6.8: Visual comparison of IPAdapter's effect on the Subway Scene using the *Sequential* generation mode. The top row shows Angle 1 with IPAdapter (a) and without IPAdapter (b). The bottom row shows Angle 2 with IPAdapter (c) and without IPAdapter (d). Note differences in color palette consistency, texture details (e.g., on walls or signage), and overall stylistic coherence between the IPAdapter and non-IPAdapter versions for each angle.

and *Separate* approaches, generally produced textures with reasonable detail levels. For instance, on the Woman asset, the *Separate (IPAdapter)* method successfully rendered the belt distinctly from the jeans, and both it and *Sequential (IPAdapter)* captured detailed jeans textures. Similarly, for the Burger, most methods besides the basic *Grid* managed to render seeds on the upper bun. However, the quality of fine details sometimes varied; for example, the lettuce texture on the Burger was not rendered accurately by any method, possibly due to ControlNet challenges or base model limitations for such organic details.

The *Grid* mode consistently produced blurry results and missed geometric details, such as the hair boundaries and entire boots/belt on the Woman, or the seeds on the Burger bun. The *Grid + Refine* method provided a significant improvement in clarity and detail over the base *Grid* pass, as observed on the Woman and Anime Head. However, this refinement pass (run at 0.8 denoise) could not fix areas entirely missed by the initial grid pass (like the Woman's boots or the belt, see Figure 6.9c) and sometimes introduced new artifacts if the base grid generation was flawed (e.g., weird hair artifacts on the Woman asset). While higher denoise values could be used, setting it to 1.0 would essentially replicate the *Separate* mode, negating the purpose of the grid.

The *Separate* and *Sequential* methods generally offered comparable levels of detail, with specific advantages depending on the asset (e.g., *Separate (IPAdapter)* captured the Woman's belt well, while *Sequential (IPAdapter)* handled her boots more consistently).

6.3.5 Artifact Analysis

Generating textures using AI diffusion models, especially across multiple viewpoints for 3D objects, inevitably introduces various visual artifacts. Understanding these is crucial for evaluating the practical usability of the methods. Several distinct types of artifacts were observed during the visual analysis of the generated assets:

- **Seams, Discontinuities, and Blurring from Inconsistency:** Perhaps the most common challenge in multi-view texturing is ensuring seamless transitions between areas generated from different camera angles. Visible seams or discontinuities can arise where these projections meet, particularly if the blending mechanism or consistency enforcement is imperfect. This was sometimes apparent with the *Separate* methods (like *Separate (No IPAdapter)*) when IPAdapter guidance was weak or absent. Even *Sequential* methods could exhibit subtle seams if view overlap was insufficient. Furthermore, inconsistencies in generated details between overlapping views could lead to blurred regions in the final blended texture. This was observed, for instance, with the sleeve edges

(a) : *Grid* Mode **(b) :** *Grid + Refine* **(c) :** *Sequential (IPAdapter)*

Figure 6.9: This figure compares detail rendering on the lower part (boots and belt) of the Woman asset. The output from the basic *Grid* mode (a) illustrates common issues such as blurriness and a failure to clearly define these features. Subsequently, the result from *Grid + Refine* (b), while sharper, shows how it often inherits the fundamental lack of detail and form for these specific features from the initial grid pass. In contrast, the *Sequential (IPAdapter)* method (c) demonstrates a more successful and detailed rendering of both the boots and the belt.

on the Woman using *Grid + Refine*, the boots generated by *Separate (IPAdapter)* on the Woman, and inconsistent object details within the Subway Scene using *Separate (IPAdapter)*.

- **Blurriness and Low Resolution:** Lack of sharpness or detail is another significant artifact category. This was most pronounced in the basic *Grid* method, an expected consequence of generating multiple views within a limited resolution space (around 1 megapixel total). While the *Grid + Refine* pass significantly improved clarity, some residual softness or inability to resolve fine details captured by other methods often remained. General blurriness could also result from low diffusion step counts or inherent properties of the base model.

- **Color and Lighting Inconsistencies:** Maintaining consistent color and lighting across views is vital for realism. Failures here manifested in several ways. A notable example occurred with *Sequential (No IPAdapter)*, where a progressive color shift was observed on the hair of both the Woman (becoming redder, see Figure 6.5) and the Anime Head (becoming desaturated, see Figure 6.6) as generation proceeded around the model. This highlights a potential instability in the inpainting process regarding color when not anchored by IPAdapter. More generally, many methods produced textures with some degree of baked-in lighting or reflections (e.g., on the Car windows and paint, Anime Head features). While IPAdapter sometimes helped make these more consistent, they

can hinder realistic relighting of the asset later.

- **Prompt Bleeding and Misinterpretation:** Diffusion models can sometimes misinterpret prompts or allow elements to "bleed" into unintended areas. Green hints from a "park" background prompt appeared on the hair edges of the Woman. The "red hair" prompt for the Anime Head resulted in red irises as well. The Burger proved challenging, with lettuce texture being generated on the bottom meat patty, and cheese sometimes appearing red (particularly with *Separate* methods). These indicate limitations in the model's scene understanding and compositional control.

- **Geometric and Generation Failures:** More severe failures involve significant deviations from the intended geometry or generation process. Examples include the *Grid* mode generating two front sides for the Car, or completely missing geometry like the Woman's boots. Methods without IPAdaper hallucinated extra windows on the Greek House. Text generation within the Subway Scene was largely illegible (a known limitation of SDXL). Poor texture alignment occurred on challenging surfaces like the Subway floor/ceiling tiles due to oblique camera angles. Inpainting could also fail subtly; on the Woman, the *Sequential (IPAdapter)* method produced an artifact resembling a double pocket on the jeans, likely where context from one view conflicted with the inpainting of the next. Furthermore, UV Inpainting, while filling gaps, showed its own limitations by applying "park" texture to the Woman's skin, indicating poor performance on unwrapped texture synthesis without specialized models.

6.3.6 Performance on Diverse Asset Types

The evaluation assets spanned several categories, revealing varying method suitability.

Assets featuring organic geometry (e.g., Woman, Anime Head, Burger): For the Woman character and the stylized Anime Head, *Sequential* methods generally provided the best consistency for complex surfaces like hair and faces, with IPAdapter proving beneficial for maintaining color and lighting coherence (*Sequential (IPAdapter)* was strong for Anime, *Sequential + UV Inpaint (IPAdapter)* for Woman). *Grid + Refine* also performed well for the Anime head, effectively capturing its simpler, bold features after the refinement pass. For the Burger, an organic object with distinct material layers, *Grid + Refine* and *Sequential (IPAdapter)* produced visually appealing, detailed results preferred over manual textures, despite some inaccuracies (misplaced lettuce, red cheese). This suggests both methods can handle such multi-component objects reasonably well. ControlNet seemed to struggle with accurately texturing some organic elements like the Woman's hair or the Burger's lettuce across all methods.

Hard-Surface Objects (e.g., Car, Greek House): Performance on hard-surface models showed less clear trends. For the Car, methods without IPAdapter (*Sequential (No IPAdapter)*) were slightly preferred, possibly suggesting that IPAdapter's constraints were detrimental to perceived quality or realism on this specific model/prompt combination. Common issues included baked-in lighting/reflections and geometric inaccuracies like misaligned seams, potentially related to the lower ControlNet strength used. For the Greek House architecture, IPAdapter was crucial, with *Separate (IPAdapter)* ranking highest. This method, generating views independently but guided by a strong external style reference, may be particularly suited to architectural models with sharp angles where sequential inpainting offers less benefit and consistent styling is paramount. Non-IPAdapter methods failed significantly here, hallucinating details.

Complex Scenes (e.g., Subway Scene): Texturing the entire Subway Scene presented significant challenges. *Sequential (IPAdapter)* provided the most coherent result with a unified palette and reasonably consistent wall textures. However, issues like poor text generation, difficulty texturing floor/ceiling tiles due to camera angles, and unavoidable gaps behind objects were apparent. The highly-ranked *Separate (No IPAdapter)* produced detailed but inconsistent textures, resulting in a visually distinct "chaotic" aesthetic that some users preferred but lacked overall coherence.

Overall, the analysis suggests *Sequential* methods are robust for organic forms requiring consistency, *Separate* (often with IPAdapter) is strong for style-driven or architectural assets, and *Grid + Refine* is a viable option for objects where the initial grid pass can adequately capture the base form.

6.3.7 Benchmarking against Manual Textures

For two assets, the Car and the Burger, *Manual Textures* were available, providing a direct benchmark for the AI-generated results.

For the Car asset, the top AI methods (*Sequential (No IPAdapter)* and *Separate (IPAdapter)*) were preferred in the perceptual survey over the provided *Manual Textures*. The manual texture appeared stylistically distinct and less photorealistic compared to the AI results, which, despite artifacts like baked-in lighting and potential seam issues, likely achieved a higher level of perceived realism or surface complexity (e.g., paint appearance). The manual texture did, however, correctly implement features like a 3D interior viewable through transparent windows, something the AI methods (generating opaque textures) did not replicate.

In the case of the Burger asset, *Grid + Refine* and *Sequential (IPAdapter)* also outperformed the *Manual Textures*. The manual texture was described as less realistic and more "shiny." While it correctly depicted all parts of the burger, the AI methods generated more intricate and arguably more convincing details for elements like the bun's seeds, meat texture, tomatoes, and

cheese, even if some elements like the lettuce were misplaced or inaccurately textured.

These comparisons indicate that StableGen's AI-driven approach can produce textures that meet or exceed the perceived quality of manual work in certain aspects, particularly regarding photorealistic detail and complexity, achieved with significantly less manual effort. However, manual texturing offers superior artistic control, accuracy in representing specific components, avoidance of AI-specific artifacts, and the ability to handle non-standard material properties like transparency directly.

(a) : *Manual Textures* **(b) :** AI: *Sequential (No IPAdapter)*

Figure 6.10: Comparison of textures for the Car asset. The provided *Manual Textures* (a) appeared stylistically distinct and less photorealistic, though it featured a 3D interior with transparent windows. The top-scoring AI-generated texture from *Sequential (No IPAdapter)* (b) was preferred in the survey, potentially offering more complex surface reflections or a more convincing "painted metal" appearance, despite some baked-in lighting and the AI not replicating the transparent windows.

6.3.8 Summary of Qualitative Insights

The detailed visual examination of the generated textures reveals several key findings about the StableGen plugin and the implemented methods:

Consistency vs. Detail Trade-offs Persist: While *Sequential* methods generally provide the best multi-view consistency, especially for organic models, they are not immune to artifacts, particularly progressive color shifts if IPAdapter is not used. There remains an inherent trade-off between ensuring perfect consistency and allowing for maximum per-view detail and variation.

Grid Refinement is Necessary but Imperfect: The *Grid + Refine* approach is significantly better than basic *Grid*, adding crucial detail. However, it inherits limitations from the initial low-resolution, potentially inconsistent grid pass, which refinement cannot fully overcome.

IPAdapter Offers Powerful but Contextual Control: IPAdapter is a valuable tool for style enforcement and improving consistency (color, lighting), particularly with a good reference image. Its impact varies by asset, however,

(a) : *Manual Textures* **(b) :** AI: *Sequential (IPAdapter)*

Figure 6.11: Comparison of textures for the Burger asset. The *Manual Textures* (a) were noted as being less realistic and more "shiny," although all components were distinctly textured. The AI-generated texture from *Sequential (IPAdapter)* (b), which also ranked highly in the survey and was preferred over manual textures, achieved intricate and seemingly realistic details for elements like the bun seeds and meat patty, and better handled issues like "red cheese" compared to some other AI methods, despite potential minor inaccuracies with other elements like lettuce placement.

and it can sometimes feel overly constraining or less effective than allowing the model more freedom, requiring user judgment.

ControlNet Establishes Geometric Grounding: Depth-based ControlNet successfully grounds the generation in the object's geometry for the most part. Limitations appear with highly complex or non-standard geometry, and its effectiveness interacts with ControlNet strength settings and the diffusion model's interpretive capabilities.

Artifacts are Diverse and Method-Dependent: Various artifacts, including seams, blur, color shifts, prompt bleeding, baked-in lighting, and geometric inaccuracies, affect all methods to some degree. Understanding the typical artifacts associated with each method (*Grid*, *Sequential*, *Separate*) helps anticipate and potentially mitigate them through parameter tuning or workflow choices (e.g., using IPAdapter, adjusting camera placement).

Optimal Method Varies by Asset Type: No single method is universally optimal. *Sequential* shines for organic consistency, *Separate* (often with IPAdapter) for style-driven architecture, and *Grid* can still be preferred for rapid prototyping as it offers superior speed. Complex scenes remain a significant challenge for achieving uniform high quality automatically.

AI Quality is Competitive but Control Remains an Issue: AI-generated textures, particularly from the better-performing methods, can match or exceed the perceived quality of manual textures in aspects like

realism and detail complexity. However, AI methods lack the fine-grained artistic control of manual workflows and introduce unique types of artifacts that require awareness and potential post-processing or careful setup to manage.

In summary, StableGen provides a flexible platform with multiple strategies, but achieving high-quality results requires an understanding of each method's strengths, weaknesses, and typical artifacts, combined with thoughtful application based on the specific asset and desired outcome.

6.4 Comparative Analysis of StableGen with Existing Texturing Solutions

To better situate the StableGen plugin within the existing landscape of 3D texturing methodologies, this section provides a comparative analysis of its features and usability. StableGen will be contrasted with traditional manual texturing workflows prevalent in Blender, as well as with other contemporary AI-powered texturing addons, specifically Dream Textures by Carson Katri and the Diffused Texture Addon by Frederik Hasecke. This comparison aims to illuminate StableGen's unique contributions, highlight its advantages and potential trade-offs, and clarify its overall positioning as a tool for 3D artists.

6.4.1 StableGen in Contrast to Manual Texturing in Blender

The Manual Texturing Paradigm

Traditional manual texturing within Blender, or in conjunction with specialized software like Substance Painter or Photoshop, represents a well-established and highly refined workflow. It typically involves meticulous UV unwrapping of the 3D model, followed by the creation or sourcing of texture maps (diffuse, normal, roughness, metallic, etc.). Artists might paint textures directly onto the model or onto its UV layout, utilize libraries of photographic or procedurally generated materials, and employ Blender's powerful shader node system to construct complex material appearances adhering to Physically Based Rendering (PBR) principles.

This paradigm offers an unparalleled degree of artistic control, allowing for pixel-perfect precision, highly specific stylistic choices, and the creation of intricate material effects. However, achieving high-quality, complex, or photorealistic results manually often requires significant artistic skill, technical knowledge (e.g., of UVs, PBR theory, shader networks), and a substantial time investment, particularly for detailed assets or large scenes.

StableGen: AI-Assisted Workflow and Creative Exploration

StableGen introduces an AI-assisted approach to 3D texturing, leveraging the capabilities of diffusion models to generate textures based on textual prompts and image references. Its core workflow is designed to augment and accelerate the texturing process by automating the creation of initial texture details and stylistic appearances. This is achieved by integrating key components such as ControlNet, which guides the AI to ensure generated textures conform to the 3D geometry; IPAdapter, for providing style guidance through reference images; and a set of distinct multi-view generation strategies, aimed at achieving comprehensive coverage and visual coherence across entire models or scenes.

Showcasing StableGen's flexibility

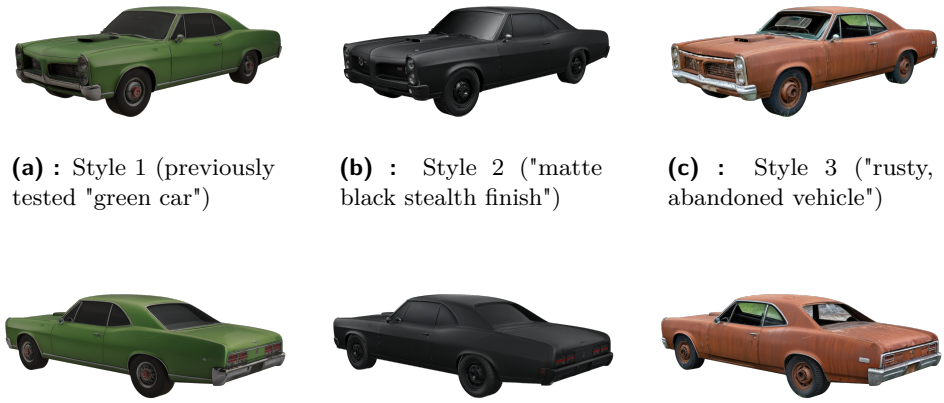
A significant strength of StableGen lies in its ability to facilitate rapid exploration of diverse visual ideas and styles, often with a considerably lower initial time investment compared to achieving similar breadth manually. This is particularly valuable in concept development, look development, and pre-visualization stages, as demonstrated by its capabilities in textual and image-based prompting.

- **Diverse Outputs via Textual Prompting:** The most direct way StableGen offers creative flexibility is through textual prompts. By simply altering the descriptive text, users can guide the AI to generate vastly different surface characteristics and thematic styles for 3D assets. For instance, the Greek House model can be rapidly transformed by prompting for distinct aesthetics such as an *abandoned, rusty concrete building within a city*, a *medieval, overgrown fantasy house*, or a *detailed, sci-fi cyberpunk structure*, with each iteration potentially generated in a matter of minutes. Figure 6.12 illustrates these specific stylistic transformations. Material properties on objects, such as the Car asset, can also be quickly varied to depict, for instance, a *classic green painted finish, a matte black stealth look, or a rusty, abandoned state* (exemplified in Figure 6.13). Furthermore, for character models like the Woman asset, prompts can rapidly change clothing ensembles, hair color, and overall thematic appearance, for example, from a *casual daytime outfit* to a *futuristic leather attire with altered features*, or even a complete transformation into a *steampunk-inspired robotic figure*, as showcased in Figure 6.14. This allows artists to quickly visualize and compare multiple design directions.
- **Style Transfer and Reference-Based Texturing with IPAdapter:** Beyond text, StableGen's integration of IPAdapter unlocks powerful image-based prompting. This allows artists to use existing images as strong visual references to guide the generation process, achieving specific

artistic styles, material appearances, or transferring features from a reference image to the 3D model. For example, the distinct visual style of a famous artwork, such as Van Gogh's *The Starry Night*, can be applied to texture diverse 3D models like the Anime Head, Car, and Greek House, as demonstrated in Figure 6.15. Similarly, a photograph of a real-world object or material could be used to inform the texture generation on an entirely different object. This method offers a more direct and often more intuitive way to control artistic style and material appearance than relying solely on text-prompt engineering.

(a) : Concrete Style **(b) :** Medieval Style **(c) :** Cyberpunk Style

Figure 6.12: Demonstrating rapid stylistic transformation on the Greek House model using StableGen with varying textual prompts. Subfigure (a) shows an "abandoned concrete building" style. Subfigure (b) illustrates a "medieval house, overgrown, fantasy" style. Subfigure (c) depicts a "cyberpunk house, detailed, sci-fi, realistic" style. Each variation showcases the flexibility for concept exploration through prompt engineering.



(a) : Style 1 (previously tested "green car") **(b) :** Style 2 ("matte black stealth finish") **(c) :** Style 3 ("rusty, abandoned vehicle")

(d) : Style 1 (Angle 2) **(e) :** Style 2 (Angle 2) **(f) :** Style 3 (Angle 2)

Figure 6.13: Demonstrating StableGen's flexibility in generating different material finishes and styles for the Car asset using textual prompts. The top row (a, b, c) shows a front-side perspective for prompts evoking a "green car", "matte black stealth finish", and "rusty, abandoned vehicle" respectively. The bottom row (d, e, f) presents the same three styles from a rear-side perspective, illustrating how these material concepts are applied across different views.

(a) : Style 1 ("blue jeans, red shirt, in a park, photo")

(b) : Style 2 ("leather jacket, dark jeans, futuristic, red hair")

(c) : Style 3 ("robot, gold, steampunk antique")

(d) : Style 1 (Angle 2)

(e) : Style 2 (Angle 2)

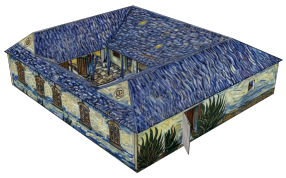
(f) : Style 3 (Angle 2)

Figure 6.14: Demonstrating StableGen's flexibility in generating diverse stylistic and feature variations on the Woman asset, shown from two different angles. The top row displays three distinct styles from a primary angle, generated with prompts for: "a woman, blue jeans, red shirt, in a park, photo" (a); "a woman, red hair, leather jacket and dark jeans, futuristic style" (b); and "a robot, gold elements, steampunk antique style" (c). The bottom row (d, e, f) showcases the same three stylistic prompts respectively, applied to the model from a secondary angle (a side-rear view), illustrating how these visual concepts translate across different perspectives.

(a) : Reference: *The Starry Night* by V. van Gogh [10]

(b) : Anime Head (Angle 1)

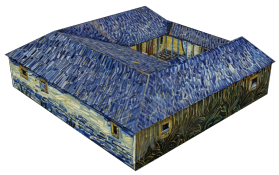
(c) : Car (Angle 1)



(d) : Greek House (Angle 1)

(e) : Anime Head (Angle 2)

(f) : Car (Angle 2)



(g) : Greek House (Angle 2)

Figure 6.15: Demonstration of IPAdapter applying the artistic style of *The Starry Night* by Vincent van Gogh [10] (a) to diverse 3D assets using StableGen. The style is transferred to the Anime Head (b, e), the Car asset (c, f), and the Greek House (d, g). Each asset is shown from two different angles (Angle 1 in top row of results, Angle 2 in bottom row of results) to illustrate how the style is applied across their surfaces. For this style transfer, the original evaluation prompts for these assets were modified by removing explicit color terms and appending the phrase "artistic style" to complement the IPAdapter image guidance.

Comparative Analysis: Workflow, Control, and Output Characteristics

When comparing StableGen's AI-assisted workflow with traditional manual texturing, several key differences emerge:

- **Speed and Efficiency:** While rapid image generation is achievable with standalone AI image tools, StableGen offers a distinct advantage in speed and efficiency by enabling the direct application and iteration of AI-generated textures onto 3D models and scenes from multiple angles, providing immediate in-context visualization. For initial texture passes, conceptual exploration, or applying a consistent style to numerous assets, StableGen can achieve results in minutes, a significant time saving compared to the hours or days often required for comparable manual work. However, refining AI outputs or fixing artifacts can add to this initial time.

- **Artistic Control:** Manual texturing provides direct, pixel-level control. StableGen offers indirect control through prompts, the selection and weighting of ControlNet inputs, IPAdapter reference images, and numerous generation parameters. While this allows for powerful emergent details and complex results, achieving a highly specific, pre-envisioned outcome can sometimes require more iteration and experimentation (prompt engineering) than direct painting.

- **Skill Sets:** Manual texturing demands traditional artistic skills (drawing, painting, understanding light and material) and software proficiency. StableGen, while introducing the need for skills in prompt crafting and understanding AI model behavior, aims to lower the initial barrier to entry for some tasks through its preset system, which offers pre-configured settings. Nonetheless, effectively leveraging its advanced features like multi-view strategies, ControlNet, and IPAdapter still benefits from a learning curve focused on AI-specific control mechanisms rather than direct manipulation.

- **Iteration and Experimentation:** StableGen excels in rapid iteration. The low cost of generating variations encourages experimentation with styles, materials, and concepts that might be too time-consuming to explore manually.

- **Output Characteristics:** AI can generate novel and highly complex details that might be difficult or tedious to create manually. It can also introduce unexpected elements or styles. However, as detailed in Section 6.3, AI-generated textures can also suffer from artifacts, inconsistencies, or misinterpretations that require careful management or post-processing. Manual textures, while potentially more laborious, typically reflect the artist's intent more precisely.

Complementary Roles in a Production Pipeline

Rather than being mutually exclusive, StableGen and manual texturing workflows can be viewed as complementary, each offering distinct advantages for different stages or tasks in a 3D production pipeline.

- **StableGen Use Cases:** Ideal for rapid prototyping, concept art generation, look development, creating initial base textures (which can then be refined manually), quickly texturing background assets or large numbers of similar objects, and exploring a wide range of stylistic variations early in the design process. Its IPAdapter functionality further aids in quickly applying specific visual styles or material references.
- **Manual Texturing Use Cases:** Remains essential for creating final *hero* assets requiring high levels of precision and artistic control, developing highly specific or nuanced artistic styles, authoring complex PBR materials with exact physical properties, detailed decal work, and addressing or refining any artifacts or shortcomings in AI-generated textures.

6.4.2 StableGen in Comparison with Other AI Texturing Addons for Blender

Beyond manual methods, several other Blender addons leverage AI for texturing. This section compares StableGen with two notable examples: Dream Textures and Diffused Texture Addon.

Dream Textures (by Carson Katri)

Dream Textures by Carson Katri [18] is a comprehensive, free, and open-source addon that integrates Stable Diffusion directly into Blender. It serves as a versatile toolkit for a variety of AI-driven tasks, including general image generation from prompts, creating seamless (tileable) textures, robust 2D image inpainting and outpainting utilities, AI upscaling of images, and even re-styling entire animations using a Cycles render pass. For the specific task of applying textures to 3D models, its key feature is *Project Dream Texture*, which utilizes depth-to-image capabilities to generate a texture from the current viewport's perspective and project it onto selected geometry.

When comparing StableGen specifically for the task of applying cohesive textures to 3D objects, particularly from multiple viewpoints:

- **Multi-View Strategy:** Dream Textures' *Project Dream Texture* is primarily designed for single-view projection onto a mesh. While it's possible for a user to manually perform multiple such projections from

different angles, potentially creating and assigning separate materials for each, the addon does not offer an integrated, automated system for generating and seamlessly blending these multiple views into a single, cohesive UV-mapped texture set for an entire object. StableGen, by contrast, is fundamentally built around multi-view strategies (its Grid and Sequential modes), incorporating automated viewpoint processing, sophisticated texture blending via OSL shaders (Section 4.6.1), and inpainting mechanisms (Section 5.7.3) specifically designed to achieve more consistent full-object coverage with a unified material output.

- **AI Control Features for 3D Texturing:** Both addons support ControlNet for guiding image generation. Dream Textures provides convenient built-in pre-processors for common ControlNet types (like depth, edges, OpenPose). StableGen, through its ComfyUI backend, allows users to leverage any ControlNet model compatible with their ComfyUI setup, offering potentially greater flexibility in choosing specific models or combining multiple types of control signals tailored for each viewpoint. Furthermore, StableGen's IPAdapter integration is specifically geared towards influencing style and maintaining consistency across multiple views in its 3D texturing workflows (including internal referencing of previous views), which is a distinct application focus compared to general image-to-image style transfer that IPAdapter might be used for within a broader toolkit.

- **Workflow Focus and Backend Architecture:** Dream Textures serves as an extensive AI image generation toolkit within Blender, where 3D texture projection is one of its many features. StableGen is architected more narrowly as a specialized multi-view 3D texturing pipeline. Its reliance on an external ComfyUI backend (Section 4.3), while requiring an initial setup, is key to its flexibility in model support (SDXL, FLUX.1-dev via StableGen-defined workflows), use of specific LoRAs, and the potential for complex conditioning that ComfyUI's node-based system enables for the texturing process.

It is important to acknowledge that Dream Textures, with its broader scope, offers several functionalities not currently central to StableGen's focused 3D texturing pipeline. These include its dedicated tools for general 2D image editing tasks like outpainting, standalone AI image upscaling features, the capability to re-style animations via a Cycles render pass, and potentially more extensive history management for its diverse operations. These make Dream Textures a very versatile addon for artists looking for a wide range of AI-assisted 2D content creation and image manipulation tools within Blender, alongside its 3D projection capabilities.

Diffused Texture Addon (by Frederik Hasecke)

The Diffused Texture Addon by Frederik Hasecke [13], which served as an early inspiration for some of the multi-view concepts explored in StableGen, also aims to generate textures directly onto meshes in Blender using Stable Diffusion. It features modes like *Text2Image Parallel* and *Image2Image Sequential* and incorporates LoRA, IPAdapter, and ControlNet (depth, Canny, normal based on a *Mesh Complexity* setting). It also operates locally.

Key differences and StableGen's advancements include:

- **Scene and Viewpoint Handling:** The Diffused Texture Addon [13] primarily focuses on texturing a single selected mesh at a time, employing automatically arranged preset camera arrays (with a choice of, e.g., 9 or 16 cameras) for multi-view generation. Its texture application appears to be handled internally. StableGen, in contrast, is designed for scene-wide application, texturing all currently visible meshes based on artist-defined camera viewpoints. This allows for more tailored coverage of complex scenes or specific object arrangements. Furthermore, StableGen leverages Blender's native systems for texture application, using projection modifiers and a sophisticated material node tree, including an OSL shader for weighted blending (Section 4.6.1), which makes the process transparent and potentially modifiable by the artist within Blender's shader editor.
- **Customization Depth:** Both addons provide controls for standard diffusion parameters (prompts, negative prompts, steps, guidance scale) and offer support for LoRAs. They also both integrate IPAdapter for style guidance using an external image. The Diffused Texture Addon allows selection between SD1.5 and SDXL base models and offers a tiered approach to ControlNet usage (Depth, Canny, Normal) based on a *Mesh Complexity* setting, which dictates specific combinations (e.g., *Low* might only use Depth, *High* might use all three).

  StableGen, utilizing its ComfyUI backend, is built to work with SDXL-based checkpoints and the FLUX.1-dev checkpoint. A key distinction lies in ControlNet flexibility: StableGen allows the user to employ any number of active ControlNet units of supported types simultaneously (e.g., multiple depth units, or using only a Canny edge unit without requiring depth), offering more granular control over the conditioning signals compared to Diffused Texture Addon's tiered system. Furthermore, while both addons can use IPAdapter with external reference images for style guidance, StableGen is also designed to leverage IPAdapter internally as a *consistency* mechanism within its multi-view generation modes, by using the first-generated or most recently generated viewpoint image as an IPAdapter reference for subsequent views. Beyond these, StableGen offers further customization through its distinct generation modes with mode-specific settings, detailed control over the viewpoint blending process, and the ability to use viewpoint-specific prompts.

- **User Experience:** The Diffused Texture Addon integrates into Blender with its own set of UI panels for settings. Diffused Texture Addon's generation process isn't executed asynchronously, which can cause the Blender interface to freeze until completion. StableGen, by offloading the intensive generation tasks to an external ComfyUI backend, performs its operations asynchronously. This crucial difference ensures that Blender remains responsive, allowing the user to continue working or monitor progress without interruption. StableGen also provides a suite of utility tools, such as aids for camera setup (e.g., circular array generation), camera-specific prompt collection, and orbit GIF/MP4 export, to support the broader texturing workflow.
- **Development Context:** While the Diffused Texture Addon demonstrated promising multi-view principles and served as an early inspiration in exploring automated camera setups, StableGen was developed independently with a distinct architectural foundation and a broader design philosophy emphasizing flexible artist control over the entire multi-view texturing pipeline, from scene-wide application and manual camera control to detailed blending and asynchronous processing.

StableGen's Differentiating Factors Among AI Addons

Compared to the aforementioned AI-driven addons, StableGen carves out its niche primarily through the following differentiating factors:

- **Dedicated Multi-View Texturing Architecture:** StableGen's core design revolves around robust and adaptable strategies (Grid and Sequential modes with integrated inpainting and OSL-based blending) specifically for achieving cohesive textures across multiple viewpoints of complex 3D objects or entire scenes. This contrasts with more general-purpose AI image tools or addons with more limited or rigid multi-view implementations.
- **Flexible AI Guidance via ComfyUI Backend:** While not exposing raw ComfyUI graph editing, StableGen leverages the ComfyUI backend to execute its sophisticated, established workflows. This allows it to support specific advanced models (SDXL-based, FLUX.1-dev) and offers a deep customization of AI guidance which would be labor-intensive to include in a fully contained solution.
- **Scene-Wide Operation and Artist-Driven Viewpoint Control:** StableGen is designed to texture all meshes within a scene simultaneously and offers full manual placement and control over any number of camera viewpoints, granting artists precise command over how an object or scene is captured for texturing.
- **Asynchronous Operation and Blender Integration:** StableGen operates asynchronously, ensuring the Blender interface remains reasonably

responsive. It further integrates with Blender by using its native node system and OSL for texture projection and blending, making the setup transparent and potentially artist-modifiable.

- **Integrated Workflow Utilities:** StableGen includes practical tools such as aids for camera setup (e.g., circular array generation), management of viewpoint-specific prompts, and options for exporting orbit GIFs/MP4s, supporting a more complete texturing and review workflow.

6.4.3 Positioning StableGen in the Texturing Landscape

The comparative analysis highlights that StableGen carves out a distinct niche within the Blender texturing ecosystem. While manual texturing remains unparalleled for ultimate artistic control and precision, it is often time-consuming and skill-intensive. Existing AI addons like Dream Textures offer valuable tools for general AI image tasks and single-view texture projection, and the Diffused Texture Addon provides a more direct approach to AI texturing on single meshes with some multi-view capabilities, albeit with limitations in control and configuration.

StableGen aims to bridge the gap by providing a dedicated, powerful, and flexible AI-driven solution specifically for multi-view 3D texturing of entire scenes or complex objects. Its core strengths lie in its adaptable multi-view generation modes, the deep customization afforded by the ComfyUI backend, and its focus on providing artists with significant control over the texturing process, from viewpoint selection to detailed parameter tuning. While it introduces the requirement of managing a ComfyUI backend, this architecture is key to its power and adaptability. StableGen is positioned not as a replacement for manual techniques or simpler AI tools, but as a sophisticated addition to the artist's toolkit, particularly suited for rapid prototyping, look development, generating complex base textures, and exploring creative possibilities that would be difficult or overly time-consuming to achieve through other means. It endeavors to make advanced, multi-faceted AI texturing more accessible and integrated within the familiar Blender environment.

Chapter 7

Conclusion

Degree of Objectives Fulfillment

The research and development undertaken for this thesis were guided by specific objectives. The following outlines how each of these goals was addressed and achieved:

1. **To investigate contemporary latent diffusion texturing methodologies and existing Blender-integrated solutions.** This objective has been *fulfilled*. A comprehensive review of current AI texturing techniques, including diffusion models, GANs, NeRFs, and manifold-based methods, was conducted. This research informed the technical approach of StableGen and provided a basis for comparing it with existing Blender-based AI texturing addons.

2. **To formulate and prototype multiple strategies for achieving multi-view textural consistency and seamless integration.** This objective has been *fulfilled*. Several distinct strategies were formulated to address multi-view consistency and seamless texture integration, including the Grid Mode for simultaneous viewpoint generation and the Sequential (Inpainting) Mode for iterative refinement. Techniques for weighted blending of viewpoints using OSL and UV space inpainting for occlusion handling were also designed and prototyped.

3. **To implement the most viable methods within the functional StableGen plugin for Blender.** This objective has been *fulfilled*. The core strategies for multi-view texturing, alongside essential control mechanisms like ControlNet and IPAdapter, and utility features such as preset management and texture baking, were successfully implemented into the StableGen Blender plugin. This involved developing a user interface, managing backend communication with ComfyUI, and creating the necessary logic for texture projection and blending.

4. **To systematically evaluate the implemented texturing methods based on output quality, comparing them internally and against**

established solutions. This objective has been *fulfilled*. StableGen's implemented texturing methods were systematically evaluated through a perceptual quality survey with user participants and detailed qualitative visual analysis. These evaluations allowed for rigorous internal comparisons between StableGen's different operational modes and configurations. As an established baseline, manually created textures were also included in this evaluation framework for selected assets, providing a direct comparison of StableGen's output quality against a traditional texturing solution. The broader comparison of StableGen with other existing AI-based Blender solutions in terms of features and usability further contextualizes its relative standing.

5. **To compare the feature set and operational usability of StableGen with existing Blender-based AI driven texturing solutions.** This objective has been *fulfilled*. A dedicated comparative analysis was conducted, evaluating StableGen against traditional manual texturing workflows and two other Blender AI texturing addons (Dream Textures and Diffused Texture Addon). This comparison focused on their respective feature sets, approaches to multi-view texturing, control mechanisms, backend architectures, and overall operational usability, highlighting StableGen's distinct capabilities and positioning.

6. **To prepare and disseminate the StableGen plugin as an open-source project.** This objective has been *fulfilled*. The StableGen plugin has been developed with open-source dissemination as a key goal and is prepared for release. The source code is available on GitHub at `https://github.com/sakalond/StableGen` under a GPL-3.0 license. This thesis itself, particularly the System Design and Implementation chapters, serves as comprehensive initial documentation, which is complemented by user-focused guides within the repository to facilitate its adoption and further community development.

Overall, the principal aim of developing and evaluating StableGen as a functional and accessible tool for AI-driven 3D texturing within Blender has been successfully achieved. The plugin integrates modern diffusion model capabilities into a familiar 3D workflow, addressing key challenges in texturing and offering a novel contribution to the field.

Bibliography

1. ARGONIUS. *Subway Station Entrance* [BlendSwap]. 2017. [visited on 2025-05-09]. Available from: `https://www.blendswap.com/blend/19305`.
2. BLENDERALLDAY. *Cycles Fast Food Double Cheeseburger* [BlendSwap]. 2013. [visited on 2025-05-09]. Available from: `https://www.blendswap.com/blend/9443`.
3. BOKHOVKIN, Alexey; TULSIANI, Shubham; DAI, Angela. *Mesh2Tex: Generating Mesh Textures from Image Queries* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2304.05868`. arXiv:2304.05868.
4. BYTEDANCE. *SDXL-Lightning* [Hugging Face Model Repository]. 2024. [visited on 2025-05-09]. Available from: `https://huggingface.co/ByteDance/SDXL-Lightning`. Model for accelerated SDXL generation.
5. CANNY, John. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* [online]. 1986, vol. PAMI-8, no. 6, pp. 679–698 [visited on 2025-05-09]. ISSN 0162-8828. Available from DOI: `10.1109/TPAMI.1986.4767851`.
6. CEYLAN, Duygu; DESCHAINTRE, Valentin; GROUEIX, Thibault; MARTIN, Rosalie; HUANG, Chun-Hao; ROUFFET, Romain; KIM, Vladimir; LASSAGNE, Gaëtan. *MatAtlas: Text-driven Consistent Geometry Texturing and Material Assignment* [online]. arXiv, 2024 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2404.02899`. arXiv:2404.02899.
7. CHEN, Dave Zhenyu; SIDDIQUI, Yawar; LEE, Hsin-Ying; TULYAKOV, Sergey; NIESSNER, Matthias. *Text2Tex: Text-driven Texture Synthesis via Diffusion Models* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2303.11396`. arXiv:2303.11396.
8. COMFYANONYMOUS. *ComfyUI* [online]. 2023. [visited on 2025-05-16]. Available from: `https://github.com/comfyanonymous/ComfyUI`. A node-based graphical user interface and backend for advanced diffusion model workflows.

9. DAREN. *Breanna Older* [BlendSwap]. 2014. [visited on 2025-05-09]. Available from: https://www.blendswap.com/blend/12450.

10. GOGH, Vincent van. *The Starry Night (from Google Art Project)* [online]. Wikimedia Commons / Google Art Project, 1889-06. [visited on 2025-05-15]. Available from: https://commons.wikimedia.org/wiki/File:Van_Gogh_-_Starry_Night_-_Google_Art_Project.jpg. Digital image of Vincent van Gogh's 1889 oil on canvas painting. The original artwork is in the collection of the Museum of Modern Art, New York. This digital version is stated to be from the Google Art Project and is in the Public Domain.

11. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep learning*. Cambridge, Massachusetts: The MIT Press, 2016. Adaptive computation and machine learning. ISBN 9780262035613.

12. GOODFELLOW, Ian J.; POUGET-ABADIE, Jean; MIRZA, Mehdi; XU, Bing; WARDE-FARLEY, David; OZAIR, Sherjil; COURVILLE, Aaron; BENGIO, Yoshua. *Generative Adversarial Networks* [online]. arXiv, 2014 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.1406.2661. arXiv:1406.2661.

13. HASECKE, Frederik. *Diffused Texture Addon: Generate Diffuse Textures on Meshes directly in Blender 3D with Stable Diffusion* [online]. 2024. Version 0.0.4 [visited on 2025-05-16]. Available from: https://github.com/FrederikHasecke/diffused-texture-addon. Blender addon for AI-powered texture generation directly on 3D meshes using Stable Diffusion.

14. HO, Jonathan; JAIN, Ajay; ABBEEL, Pieter. *Denoising Diffusion Probabilistic Models* [online]. arXiv, 2020 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2006.11239. arXiv:2006.11239.

15. HU, Minghui; ZHENG, Jianbin; LIU, Daqing; ZHENG, Chuanxia; WANG, Chaoyue; TAO, Dacheng; CHAM, Tat-Jen. *Cocktail: Mixing Multi-Modality Controls for Text-Conditional Image Generation* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2306.00964. arXiv:2306.00964.

16. HUANG, Yi-Hua; CAO, Yan-Pei; LAI, Yu-Kun; SHAN, Ying; GAO, Lin. *NeRF-Texture: Synthesizing Neural Radiance Field Textures* [online]. arXiv, 2024 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2412.10004. arXiv:2412.10004.

17. JORDAN, M. I.; MITCHELL, T. M. Machine learning: Trends, perspectives, and prospects. *Science* [online]. 2015, vol. 349, no. 6245, pp. 255–260 [visited on 2025-05-09]. ISSN 0036-8075, ISSN 1095-9203. Available from DOI: 10.1126/science.aaa8415.

18. KATRI, Carson et al. *Dream Textures: Stable Diffusion built-in to Blender* [online]. 2022. Version 0.4.1 [visited on 2025-05-16]. Available from: https://github.com/carson-katri/dream-textures. Blender addon for AI-powered texture and image generation.

19. KINGMA, Diederik P.; WELLING, Max. *Auto-Encoding Variational Bayes* [online]. arXiv, 2022 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.1312.6114. arXiv:1312.6114.

20. KYNE, Daniel. *Pair Rank (Scoring, Examples, and Other FAQs)* [OpinionX Knowledge Base]. 2025. [visited on 2025-05-09]. Available from: https://knowledge.opinionx.co/en/articles/5866366-pair-rank-scoring-examples-and-other-faqs.

21. LE, Cindy; HETANG, Congrui; LIN, Chendi; CAO, Ang; HE, Yihui. *EucliDreamer: Fast and High-Quality Texturing for 3D Models with Stable Diffusion Depth* [online]. arXiv, 2024 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2311.15573. arXiv:2311.15573.

22. LEVIN, Eran; FRIED, Ohad. *Differential Diffusion: Giving Each Pixel Its Strength* [online]. arXiv, 2024 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2306.00950. arXiv:2306.00950.

23. LI, Ming; YANG, Taojiannan; KUANG, Huafeng; WU, Jie; WANG, Zhaoning; XIAO, Xuefeng; CHEN, Chen. *ControlNet++: Improving Conditional Controls with Efficient Consistency Feedback* [online]. arXiv, 2024 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2404.07987. arXiv:2404.07987.

24. NG, Andrew; JORDAN, Michael. On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. In: DIETTERICH, T.; BECKER, S.; GHAHRAMANI, Z. (eds.). *Advances in Neural Information Processing Systems*. MIT Press, 2001, vol. 14. Available from DOI: 10.7551/mitpress/1120.001.0001.

25. NICHOL, Alex; DHARIWAL, Prafulla. *Improved Denoising Diffusion Probabilistic Models* [online]. arXiv, 2021 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2102.09672. arXiv:2102.09672.

26. OPINIONX. *OpinionX - Customer Ranking Stack* [https://www.opinionx.co/]. OpinionX, 2025. [visited on 2025-05-09]. Available from: https://www.opinionx.co/. Homepage of the survey platform used.

27. RICHARDSON, Elad; METZER, Gal; ALALUF, Yuval; GIRYES, Raja; COHEN-OR, Daniel. *TEXTure: Text-Guided Texturing of 3D Shapes* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2302.01721. arXiv:2302.01721.

28. ROMBACH, Robin; BLATTMANN, Andreas; LORENZ, Dominik; ESSER, Patrick; OMMER, Björn. *High-Resolution Image Synthesis with Latent Diffusion Models* [online]. arXiv, 2022 [visited on 2025-05-09]. Available from DOI: 10.48550/arXiv.2112.10752. arXiv:2112.10752.

29. SATURN CLOUD. *Training Stability in GANs* [online]. Saturn Cloud, 2023-07. [visited on 2025-05-16]. Available from: https://saturncloud.io/glossary/training-stability-in-gans/. Glossary entry defining training stability in Generative Adversarial Networks.

30. SG161222. *RealVisXL V5.0* [Hugging Face Model Repository]. 2024. [visited on 2025-05-09]. Available from: `https://huggingface.co/SG161222/RealVisXL_V5.0`. SDXL Checkpoint.

31. SIDDIQUI, Yawar; THIES, Justus; MA, Fangchang; SHAN, Qi; NIESSNER, Matthias; DAI, Angela. *Texturify: Generating Textures on 3D Shape Surfaces* [online]. 2022. [visited on 2025-05-09]. Available from: `https://arxiv.org/abs/2204.02411v1`.

32. SOBOLIČOVÁ, Veronika. *Model starořecké obce v rámci projektu Politeia 21*. Prague, 2025. Bachelor Thesis. Faculty of Information Technology, Czech Technical University in Prague. Supervisor: Ing. Radek Richtr, Ph.D. Language of work: Slovak. English translation of title: Model of an Ancient Greek Settlement within the Politeia 21 Project. The work is part of the Politeia 21 project.

33. SOHL-DICKSTEIN, Jascha; WEISS, Eric A.; MAHESWARANATHAN, Niru; GANGULI, Surya. *Deep Unsupervised Learning using Nonequilibrium Thermodynamics* [online]. arXiv, 2015 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.1503.03585`. arXiv:1503.03585.

34. SONG, Jiaming; MENG, Chenlin; ERMON, Stefano. *Denoising Diffusion Implicit Models* [online]. arXiv, 2022 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2010.02502`. arXiv:2010.02502.

35. THECALI. *Pontiac GTO 67* [BlendSwap]. 2014. [visited on 2025-05-09]. Available from: `https://www.blendswap.com/blend/13575`.

36. UCUPUMAR. *Brown* [BlendSwap]. 2015. [visited on 2025-05-09]. Available from: `https://www.blendswap.com/blend/15262`.

37. YE, Hu; ZHANG, Jun; LIU, Sibo; HAN, Xiao; YANG, Wei. *IP-Adapter: Text Compatible Image Prompt Adapter for Text-to-Image Diffusion Models* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2308.06721`. arXiv:2308.06721.

38. ZENG, Xianfang; CHEN, Xin; QI, Zhongqi; LIU, Wen; ZHAO, Zibo; WANG, Zhibin; FU, Bin; LIU, Yong; YU, Gang. *Paint3D: Paint Anything 3D with Lighting-Less Texture Diffusion Models* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2312.13913`. arXiv:2312.13913.

39. ZHANG, Lvmin; RAO, Anyi; AGRAWALA, Maneesh. *Adding Conditional Control to Text-to-Image Diffusion Models* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2302.05543`. arXiv:2302.05543.

40. ZHAO, Shihao; CHEN, Dongdong; CHEN, Yen-Chun; BAO, Jianmin; HAO, Shaozhe; YUAN, Lu; WONG, Kwan-Yee K. *Uni-ControlNet: All-in-One Control to Text-to-Image Diffusion Models* [online]. arXiv, 2023 [visited on 2025-05-09]. Available from DOI: `10.48550/arXiv.2305.16322`. arXiv:2305.16322.

Appendices

Appendix A

List of Abbreviations

AI Artificial Intelligence

API Application Programming Interface

BibTeX Bibliography TeX (Bibliographic tool for LaTeX)

bpy Blender Python (Blender's Python API)

CC0 Creative Commons Zero (Public Domain Dedication)

CFG Classifier-Free Guidance (Scale for diffusion models)

CLIP Contrastive Language-Image Pre-training

ComfyUI A powerful and modular GUI for Stable Diffusion

ControlNet A neural network structure to control diffusion models

CTU Czech Technical University in Prague

FLUX A new generation diffusion model architecture (e.g., FLUX.1-dev)

FOV Field of View

GAN Generative Adversarial Network

GIF Graphics Interchange Format

GPL GNU General Public License

GPU Graphics Processing Unit

HDRI High Dynamic Range Imaging

HTTP Hypertext Transfer Protocol

IPAdapter Image Prompt Adapter

JSON JavaScript Object Notation

KOS Study Information System at CTU

LDM Latent Diffusion Model

LoRA Low-Rank Adaptation

MP4 MPEG-4 Part 14 (Digital multimedia container format)

NeRF Neural Radiance Field

N-Panel Properties panel in Blender (often toggled with 'N' key)

OSL Open Shading Language

PBR Physically Based Rendering

PDF Portable Document Format

PIL Python Imaging Library (Pillow fork)

PNG Portable Network Graphics

ReLU Rectified Linear Unit

RGB Red, Green, Blue (Color model)

SD Stable Diffusion

SDS Score Distillation Sampling

SDXL Stable Diffusion XL

VAE Variational Autoencoder

ViT Vision Transformer

UV (Mapping) A process of projecting a 2D image to a 3D model's surface

Appendix B

Contents of the Attached Data Storage

The digital archive submitted to KOS alongside this thesis contains the following materials, organized to provide comprehensive access to the research outputs and the developed software:

- `readme.txt` .... Overview of the archive contents and usage instructions
- `src/` ..... Source code for the project and thesis LaTeX
  - `impl/` ..... Complete Python source code for the Blender plugin
  - `thesis/` ..... Complete LaTeX source files for this thesis
- `text/` ..... Contains the final thesis text
  - `thesis.pdf` ..... The final thesis text in PDF format