Summary of 6 Papers:

**Paper 1: One Noise to Rule Them All: Learning a Unified Model of Spatially-Varying Noise Patterns**

**1. Paper name:**  
*One Noise to Rule Them All: Learning a Unified Model of Spatially-Varying Noise Patterns*

**2. Authors:**  
Arman Maesumi, Dylan Hu, Krishi Saripalli (Brown University), Vladimir G. Kim, Matthew Fisher (Adobe Research), Sören Pirk (Christian-Albrechts-University), Daniel Ritchie (Brown University)

**3. Summary:**  
This paper addresses the problem of generating and modeling **noise patterns** (used widely in graphics for textures, natural-looking surfaces, procedural generation). Traditionally, different types of noise functions (Perlin, Simplex, Worley, etc.) are hand-designed. This paper trains a **unified ML model** that can **learn and reproduce many kinds of noise patterns** with spatial variations. The goal is to replace multiple ad-hoc noise functions with one learnable model that works across different applications.

This paper is about **noise** – the random-looking patterns used in computer graphics to make things look natural (like wood grain, stone, clouds, fire, etc.).

👉 Normally, programmers have to **manually choose or design** a specific type of noise function (Perlin noise, Worley noise, etc.) for each effect.

👉 What this paper does is:

* Build **one single machine learning model** that can **imitate all these different noise patterns**.
* The model can also handle cases where the noise **changes across space** (so the texture isn’t the same everywhere).
* This means instead of using many different noise functions, artists/engineers could just use this **one learned noise generator**.

So in short:  
They replaced a **collection of hand-made noise algorithms** with a **single smart ML model** that can generate any of them.

**4. Background knowledge required:**

* Basics of **machine learning** (CNNs or implicit neural representations).
* Traditional noise functions (Perlin, Simplex, Worley…) are **hand-crafted formulas**.
* Instead, they **train a neural network** that learns from examples of these noise patterns.
* Once trained, the model can generate similar patterns (and even smoothly mix between different noise styles).
* Understanding of **noise functions** (Perlin noise, Worley noise, etc.) in computer graphics.
* General familiarity with texture generation in graphics.

**5. Pending future works (from paper & reasonable extensions):**

* Extending the model to handle **temporal noise** (animated noise over time).
* Integrating with **3D procedural generation** (not just 2D textures).
* Making the unified model more **efficient for real-time applications**.
* Exploring **controllability** (e.g., generating user-specified noise characteristics).

**6.  Temporal noise → Add *time* as an input so the noise changes over time → useful for animations like moving clouds or fire.**

** 3D procedural generation → Instead of just 2D textures, make the noise in 3D (like inside a volume) → for things such as smoke, marble, or fluids.**

** Efficiency → Make the model smaller and faster so it runs in real-time on a GPU (e.g., in games).**

** Controllability → Give users sliders (for roughness, frequency, etc.) so they can tweak the look of the noise instead of taking whatever the model outputs.**

**Paper 2: TexSliders: Diffusion-Based Texture Editing in CLIP Space**

**1. Paper name:**  
*TexSliders: Diffusion-Based Texture Editing in CLIP Space*

**2. Authors:**  
Min Jin Chong, Daichi Ito, Yuki Endo, Edgar Simo-Serra, Ziwei Liu, Michal Gharbi, Ramin Zabih

**3. Summary:**  
This paper is about **editing textures** (like wood, metal, fabric) in an **easy way**.

Normally, with AI texture tools, you have to **type text prompts** like *“make it shinier”* or *“more metallic.”* That’s not very precise.

👉 Instead, this paper makes a tool called **TexSliders**:

* You get **sliders** (like in Photoshop) for properties such as *shininess, roughness, metallic feel, etc.*
* When you move a slider, the AI (using **CLIP + diffusion models**) changes the texture in that direction.
* So you can interactively tweak textures without writing prompts or training models.

**In short:**  
It’s an **AI-powered texture editor with sliders** → making it easy and interactive for artists to adjust how a material looks.

**4. Background knowledge required:**

* Basics of **diffusion models** (how they generate images/textures).
* **CLIP embeddings** (how vision-language models represent attributes).

**About Diffusion models (how they generate images/textures)**

* Think of diffusion as **adding and then removing noise**.
* First, you take an image and slowly add random noise until it looks like static (TV snow).
* Then the AI learns to **reverse this process** → starting from random noise, it removes noise step by step until a clear image appears.
* That’s how Stable Diffusion, DALL·E, etc. make pictures from prompts.
* For **textures**, the process is the same but guided toward making surface patterns.

**CLIP embeddings (how vision-language models represent attributes)**

* CLIP is a model trained on **images + text captions**.
* It learns to put both pictures and words in the **same mathematical space** (embedding space).
* Example: The word “shiny” and a picture of shiny metal will end up close together in this space.
* So if you want to make a texture “shinier,” you move the texture’s embedding closer to the “shiny” direction.
* That’s how TexSliders knows which way to adjust the texture when you move a slider.

**5. Pending future works:**

* Expanding the system to support **3D textures/materials** (not just 2D).
* Improving **real-time editing** (diffusion models are slow).
* Allowing **multi-attribute editing** beyond linear sliders (hierarchical or nonlinear controls).
* Exploring integration with **game engines or 3D modeling tools**.

**6. How these future works might be implemented:**

* **3D textures:** Extend the model to 3D voxel grids or neural material representations.
* **Real-time editing:** Use diffusion model acceleration (latent diffusion, distillation, or caching).
* **Multi-attribute editing:** Replace sliders with a learned interface (e.g., embedding manipulations guided by a small neural network).
* **Game engine integration:** Wrap the pipeline in plugins (Unity/Unreal) for artist-friendly deployment.

**Paper 3: Deep Hybrid Camera Deblurring for Smartphone Cameras**

**1. Paper name:**  
*Deep Hybrid Camera Deblurring for Smartphone Cameras*

**2. Authors:**  
Jaesung Rim (POSTECH), Junyong Lee (Samsung AI Center), Heemin Yang, Sunghyun Cho (POSTECH)

**3. Summary:**  
Smartphone photos often suffer from **motion blur** due to long exposure times in low light. The paper proposes a **hybrid camera system**:

* One sensor takes **short-exposure images** (sharp but noisy).
* Another sensor takes **long-exposure images** (less noisy but blurry).  
  A **deep neural network** then fuses them to generate a **clean, sharp final image**.  
  The contribution: a dataset + a novel network that learns how to combine these complementary images effectively.

**4. Background knowledge required:**

* Basics of **image formation** (exposure, blur, noise).
* **CNNs / image-to-image translation** (deep learning for vision).
* Knowledge of **image restoration tasks** (denoising, deblurring).

**5. Pending future works:**

* Extend to **video deblurring** (not just single images).
* Improve performance in **extremely low light** conditions.
* Reduce **computational cost** for real-time smartphone usage.
* Adapt to **monocular (single camera) systems**, since not all phones have dual sensors.

**6. How future works might be implemented:**

* **Video deblurring:** Use recurrent networks or transformers that model temporal consistency.
* **Extreme low light:** Combine with physics-based priors (noise models) + pre-trained denoising modules.
* **Efficiency:** Use lightweight networks (MobileNet, pruning, quantization).
* **Monocular systems:** Simulate long/short exposure pairs using synthetic augmentation + train networks to infer missing info.

**📄 Paper 4: N-BVH: Neural Ray Queries with Bounding Volume Hierarchies**

**1. Paper name:**  
*N-BVH: Neural Ray Queries with Bounding Volume Hierarchies*

**2. Authors:**  
Boyang Deng (Adobe Research), Graham Fyffe, Jonathan Ragan-Kelley, Gordon Wetzstein (Stanford), Ravi Ramamoorthi (UC San Diego), et al.

**3. Summary:**  
In ray tracing, **Bounding Volume Hierarchies (BVHs)** are essential for efficient intersection queries (deciding which object a ray hits first). Traditional BVHs are hand-optimized data structures.  
This paper introduces **N-BVH**, where a **neural network augments BVHs** to accelerate queries and improve memory usage. Essentially, the BVH is still there, but a neural model predicts traversal or intersection outcomes more efficiently.

**4. Background knowledge required:**

* **Ray tracing basics** (rays, intersections, acceleration structures).
* **Bounding Volume Hierarchies (BVH)** (tree-based acceleration).
* **Neural networks** (basic supervised learning).

**5. Pending future works:**

* Extending neural acceleration to other **spatial data structures** (kd-trees, octrees).
* Optimizing **training overhead** so it’s practical for dynamic scenes.
* Balancing **robustness vs. speed** (avoiding neural prediction errors).
* GPU hardware integration for **real-time ray tracing engines**.
* **🔹 Comparison with BVH**

| **Structure** | **How it splits** | **Shape of regions** | **Typical use** |
| --- | --- | --- | --- |
| **BVH** | Groups objects into bounding boxes | Arbitrary boxes (not aligned) | Ray tracing (most common) |
| **kd-tree** | Splits space with planes | Half-spaces (axis-aligned) | Ray tracing, nearest neighbor |
| **Octree** | Splits space into 8 cubes | Uniform cubes | Voxels, point clouds, collision |

**6. How future works might be implemented:**

* **Other structures:** Train neural models to mimic query traversal in kd-trees/octrees.
* **Dynamic scenes:** Incremental training or online learning so the neural model adapts as objects move.
* **Robustness:** Hybrid approach — fallback to classic traversal if neural prediction confidence is low.
* **GPU integration:** Fuse neural inference into shader pipelines (Tensor cores for ray queries).

**📄 Paper 5: Stochastic Computation of Barycentric Coordinates**

**1. Paper name:**  
*Stochastic Computation of Barycentric Coordinates*

**2. Authors:**  
Zhiqin Chen (Simon Fraser University), David I. Levin (University of Toronto), Alec Jacobson (University of Toronto), et al.

**3. Summary:**  
Barycentric coordinates are widely used in graphics (e.g., interpolation inside polygons/meshes). Computing them exactly for complex shapes is **hard**.  
This paper introduces a **stochastic (randomized) method** that estimates barycentric coordinates via **Monte Carlo sampling**. Instead of computing closed-form solutions, they approximate them probabilistically, making it scalable to complex geometries.

**1. What are barycentric coordinates?**

Imagine you have a **triangle** with points A, B, C.  
If you pick any point **P** inside that triangle, you can describe it as a “mix” of A, B, and C.

For example:

* P = 0.2·A + 0.5·B + 0.3·C
* The numbers (0.2, 0.5, 0.3) are the **barycentric coordinates**.
* They always add up to 1.

📌 Why useful in graphics?

* To interpolate colors, textures, lighting smoothly across triangles in a 3D model.
* Every pixel on a 3D surface uses these coordinates behind the scenes.

**2. The problem**

For **simple shapes (like triangles)** → barycentric coordinates are easy to compute (just solve a formula).  
But for **complicated polygons, meshes, or 3D volumes** → exact computation becomes very hard, or impossible in closed form.

**3. The paper’s idea**

Instead of trying to compute barycentric coordinates with exact math, the authors say:

👉 “Let’s use **random sampling (Monte Carlo)** to estimate them.”

* You randomly throw points inside the polygon/mesh.
* You measure how much influence each vertex has over those random samples.
* Over many samples, you approximate the barycentric coordinates for any point.

This is **probabilistic** but very scalable.

**4. Why it’s good**

* Works for **very complex shapes** (where formulas don’t exist).
* Can handle high-dimensional problems.
* Monte Carlo sampling gets more accurate if you take more samples.

✅ **In simple words**:  
Normally, barycentric coordinates are like a recipe (“this point is 20% A, 50% B, 30% C”).  
For triangles, the recipe is easy.  
For messy shapes, the recipe is super hard.

The paper says: instead of exact cooking instructions, just **taste (sample) many times randomly** → and from that, you figure out the approximate recipe

**4. Background knowledge required:**

* **Geometry basics** (polygons, interpolation, barycentric coordinates).
* **Monte Carlo methods** (random sampling, probability estimation).
* Some linear algebra.

**5. Pending future works:**

* Extend to **higher dimensions** (4D+ geometry, volumetric meshes).
* Improve **variance reduction** (more accurate estimates with fewer samples).
* GPU acceleration for **real-time barycentric queries**.
* Integration into **physics simulations** or **real-time rendering** pipelines.

**6. How future works might be implemented:**

* **Higher dimensions:** Adapt sampling strategy to n-simplices.
* **Variance reduction:** Use stratified or importance sampling to converge faster.
* **GPU acceleration:** Implement stochastic estimation kernels directly on GPU.
* **Physics/rendering integration:** Replace exact barycentric computations in simulation/rendering with stochastic approximations for speed.

**📄 Paper 6: Navigation-Driven Approximate Convex Decomposition**

**1. Paper name:**  
*Navigation-Driven Approximate Convex Decomposition*

**2. Authors:**  
Brett Allen, Jose Díaz, Niloy J. Mitra, and collaborators.

**3. Summary:**  
Convex decomposition is about breaking a complex 3D model into convex parts, which are easier for collision detection and path planning.  
This paper proposes a **navigation-driven approach**: instead of decomposing purely by geometry, the decomposition is optimized for **navigation efficiency** (e.g., shortest paths, movement costs). It blends computational geometry with graph-based navigation analysis.

**4. Background knowledge required:**

* **Convex geometry basics** (polygons, convexity).
* **Graph algorithms** (shortest paths, traversal).
* **Pathfinding concepts** (navigation meshes).

**5. Pending future works:**

* Extend to **dynamic environments** (objects moving, obstacles appearing/disappearing).
* Improve scalability for **large, complex scenes** (games, robotics).
* Combine with **learning-based navigation** (hybrid geometry + ML).
* Generalize beyond navigation (e.g., physics simulations, visibility queries).

**6. How future works might be implemented:**

* **Dynamic environments:** Use incremental decomposition algorithms that update convex regions on-the-fly.
* **Scalability:** Parallel decomposition on GPUs, hierarchical methods.
* **Hybrid navigation + ML:** Train models to suggest decomposition strategies guided by navigation data.
* **Generalization:** Extend cost functions (not just navigation distance, but also visibility/coverage).