# **■** Python Project Code Export

- [PY] class\_count.py
- [PY] clean\_pngs.py
- [PY] config.py
- [PY] convert\_polygons.py
- [PY] dataset.py
- [PY] demo.py
- [PY] evaluate.py
- [PY] evaluation\metrics.py
- [PY] infer.py
- [PY] inference\engine.py
- [PY] models\dvx.py
- [PY] models\encoder.py
- [PY] models\extrusion.py
- [PY] models\heads.py
- [PY] models\model.py
- [PY] param.py
- [PY] setup.py
- [PY] train.py
- [PY] training\losses.py
- [PY] training\trainer.py
- [PY] utils\visualization.py
- [PY] vortex.py

### **■** File: class\_count.py

```
-----
 1: import cv2, numpy as np, glob
 2: from collections import Counter
 3:
 4: all_classes = set()
 5: class_counts = Counter()
 6:
 7: # Go through all mask images
 8: for mask_file in glob.glob("./data/floorplans/train/*/mask.png"):
      mask = cv2.imread(mask_file, cv2.IMREAD_GRAYSCALE)
 9:
10:
      unique, counts = np.unique(mask, return_counts=True)
      for u, c in zip(unique, counts):
11:
12:
           class_counts[u] += c
13:
      all_classes.update(unique)
14:
15: # Total pixels
16: total_pixels = sum(class_counts.values())
17:
18: print("Classes found in dataset:", sorted(all_classes))
19: print("\nPixel distribution per class:")
20: for cls in sorted(class_counts.keys()):
21:
       percentage = (class_counts[cls] / total_pixels) * 100
22:
       print(f"Class {cls}: {class_counts[cls]} pixels ({percentage:.2f}%)")
```

-----

### ■ File: clean\_pngs.py

```
_____
 1: from PIL import Image
 2: import os
 3: import shutil
 4:
 5: data_dir = r"data/floorplans"
 6:
 7: def safe_clean_image(path):
        """Safely clean image by only removing ICC profiles, preserving all pixel data"""
 8:
 9:
10:
            # Create backup first (optional safety measure)
11:
           backup_path = path + ".backup"
12:
13:
            with Image.open(path) as img:
14:
                # Check if image is already clean
15:
                if 'icc_profile' not in img.info:
16:
                   print(f"Already clean: {path}")
17:
                    return True
18:
19:
                # Create backup
20:
                shutil.copy2(path, backup_path)
21:
22:
                # Method 1: Just strip ICC profile while preserving everything else
23:
               img_data = img.copy()
24:
25:
                # Remove only the problematic ICC profile
26:
                if 'icc_profile' in img_data.info:
27:
                   del img_data.info['icc_profile']
28:
29:
                \# Save with same format and quality, just without ICC profile
30:
                img_data.save(path, format="PNG", optimize=False) # No optimization to preserve exact pixels
31:
32:
                # Remove backup if successful
33:
                os.remove(backup_path)
34:
35:
            print(f"Cleaned ICC profile from: {path}")
36:
           return True
37:
38:
        except Exception as e:
39:
            # Restore backup if it exists
40:
           backup_path = path + ".backup"
41:
           if os.path.exists(backup path):
42:
                shutil.move(backup_path, path)
43:
                print(f"Restored backup for: {path}")
44:
```

```
print(f"Failed {path}: {e}")
 45:
 46:
             return False
 47:
 48: def verify_image_integrity(path):
         """Verify image can still be loaded properly after cleaning"""
 49:
 50:
 51:
             with Image.open(path) as img:
 52:
                 # Try to access pixel data to ensure image is valid
 53:
                 _ = img.size
 54:
                   = img.mode
                 # Try to load a small sample of pixel data
 55:
 56:
                 _{-} = img.getpixel((0, 0))
 57:
             return True
 58:
         except Exception as e:
 59:
            print(f"WARNING: Image integrity check failed for {path}: {e}")
 60:
             return False
 61:
 62: # Process only image.png files
 63: processed_files = []
 64: cleaned_count = 0
 65: failed_count = 0
 66: already_clean = 0
 67:
 68: print("Starting safe ICC profile removal for dataset...")
 69: print("This preserves all pixel data and only removes problematic metadata.")
 70:
 71: for root, _, files in os.walk(data_dir):
72:     for f in files:
             if f == "image.png": # Only process image.png files
 74:
                 path = os.path.join(root, f)
 75:
                 processed_files.append(path)
 76:
 77:
                 # Check if already clean
 78:
                 try:
 79:
                     with Image.open(path) as img:
 80:
                         if 'icc_profile' not in img.info:
 81:
                             already_clean += 1
 82:
                             continue
 83:
                 except:
 84:
                     pass
 85:
 86:
                 if safe_clean_image(path):
 87:
                     # Verify integrity after cleaning
                     if verify_image_integrity(path):
 88:
 89:
                         cleaned_count += 1
 90:
                     else:
 91:
                         failed_count += 1
 92:
                 else:
 93:
                     failed_count += 1
 94:
 95: print(f"\n" + "="*50)
 96: print(f"DATASET CLEANING SUMMARY")
 97: print(f"="*50)
 98: print(f"Total image.png files found: {len(processed_files)}")
 99: print(f"Already clean (no ICC profile): {already_clean}")
100: print(f"Successfully cleaned: {cleaned_count}")
101: print(f"Failed to clean: {failed_count}")
102: print(f"Total files processed: {already_clean + cleaned_count + failed_count}")
103:
104: if failed_count > 0:
        print(f"\nWARNING: {failed_count} files couldn't be cleaned.")
106:
         print(f"Check these files manually - they may be corrupted.")
107:
108: print(f"\nDataset should now be ready for training without libpng warnings!")
109:
110: # Optional: Test load a few random images to verify dataset integrity
111: print(f"\nTesting random samples for integrity...")
112: import random
113: test_files = random.sample(processed_files, min(5, len(processed_files)))
114: for test_path in test_files:
115:
       if verify_image_integrity(test_path):
          print(f"? {test_path}")
116:
117:
         else:
```

\_\_\_\_\_\_

### ■ File: config.py

```
______
 1: """
 2: Configuration settings for the Neural-Geometric 3D Model Generator
 3: Enhanced with dynamic curriculum and adaptive training strategies
 4: """
 5: from dataclasses import dataclass
 6: from typing import Tuple, Dict, Any, Optional, List
 7: import torch
 g :
 9:
10: @dataclass
11: class DataConfig:
 12:
        """Data-related configuration"""
       data_dir: str = "./data/floorplans"
13:
14:
      image_size: Tuple[int, int] = (256, 256)
                                                # keep full resolution for accuracy
15:
       voxel_size: int = 64
16:
       batch_size: int = 4
                                                  # balance speed & memory
17:
       num_workers: int = 8
                                                  # faster dataloader (tune per CPU)
18:
       augment: bool = True
19:
 20:
 21: @dataclass
 22: class ModelConfig:
       """Model architecture configuration optimized for high accuracy"""
 24:
       input_channels: int = 3
      num_classes: int = 5
 25:
 26:
        feature_dim: int = 512
                                 # reduced from 768 ? faster while keeping strong accuracy
 27:
       num_attributes: int = 6
 28:
       voxel_size: int = 64
       max_polygons: int = 20
                                  # enough for complex layouts
 29:
 30:
        max_points: int = 50
                                  # good detail without huge cost
 31:
        dropout: float = 0.05
 32:
       use_attention: bool = True
 33:
       use_deep_supervision: bool = True
 34:
 35:
        # Auxiliary heads for novel training strategies
36:
       use_latent_consistency: bool = True
 37:
        use_graph_constraints: bool = True
38:
        latent_embedding_dim: int = 256
 39:
 40:
 41: @dataclass
 42: class CurriculumConfig:
        """Dynamic curriculum learning configuration"""
 43:
 44:
        # Adaptive stage transitioning
 45:
       use_dynamic_curriculum: bool = True
 46:
       stage_switch_patience: int = 5
 47:
       min_improvement_threshold: float = 0.001
       plateau_detection_window: int = 3
 48:
 49:
        # GradNorm / gradient tracking
 50:
        gradient_norm_window: int = 100
 51:
 52:
 53:
        # Objectives for multi-objective optimization
 54:
        objectives: Optional[List[str]] = None
 55:
 56:
        # Topology-aware scheduling
 57:
        topology_schedule: str = "progressive" # "progressive", "linear_ramp", "exponential"
 58:
        topology_start_weight: float = 0.1
 59:
        topology_end_weight: float = 1.0
 60:
        topology_ramp_epochs: int = 20
 61:
        # config.py (snippet ? add into the existing config class/dict)
 62:
 63:
        # Mixed precision and training conveniences
 64:
        use_mixed_precision = True  # enable AMP
        cache_in_memory = False
 65:
                                             # set True if host RAM can hold dataset
 66:
        accumulation_steps = 1
                                             # effective batch size multiplier
 67:
                                             # run DVX refinement every N steps (1 = every step)
        dvx\_step\_freq = 1
```

```
68:
         persistent_workers = True
                                                # DataLoader persistent workers
 69:
        prefetch_factor = 4
                                                # DataLoader prefetch
 70:
                                                # default num workers for DataLoader (tune by CPU)
         num_workers = 8
 71:
         # Progressive resolution settings (example)
         voxel_size_stage = { "stage1": 32, "stage2": 32, "stage3": 64 } # voxel sizes per stage
 72:
         image_size_stage = { "stage1": (128,128), "stage2": (192,192), "stage3": (256,256)}
 73:
 74:
 75:
 76:
         # Loss component scheduling
 77:
         loss_schedule: Dict[str, str] = None
 78:
 79:
         # Multi-objective optimization (GradNorm)
 80:
         use_gradnorm: bool = True
 81:
         gradnorm_alpha: float = 0.12
 82:
         gradnorm_update_freq: int = 5
 83:
 84:
         # Graph constraint scheduling
 85:
         graph_weight_schedule: str = "delayed_ramp"
         graph_start_epoch: int = 15
 86:
 87:
         graph_end_weight: float = 0.5
 88:
 89:
         def __post_init__(self):
 90:
             # Provide default loss schedule if not set
 91:
             if self.loss_schedule is None:
                 self.loss_schedule = {
 92:
                     "segmentation": "static",
 93:
 94:
                     "dice": "static",
                     "sdf": "early_decay",
 95:
 96:
                     "attributes": "static",
 97:
                     "polygon": "staged_ramp",
 98:
                     "voxel": "late_ramp",
 99:
                     "topology": "progressive",
                     "latent_consistency": "mid_ramp",
100:
                     "graph": "delayed_ramp",
101:
                 }
102:
103:
             # Default objectives used by GradNorm / trainer monitoring
104:
105:
             if self.objectives is None:
106:
                 self.objectives = [
107:
                     "segmentation",
108:
                     "dice",
109:
                     "sdf",
110:
                     "attributes",
111:
                     "polygon",
112:
                     "voxel",
113:
                     "topology",
114:
                     "latent_consistency",
115:
                     "graph",
116:
                 1
117:
118:
119: @dataclass
120: class TrainingConfig:
121:
      """Training configuration with adaptive strategies"""
122:
         device: str = "cuda" if torch.cuda.is_available() else "cpu"
123:
         # Dynamic epoch limits (maxima; curriculum may switch earlier)
124:
125:
        max_stage1_epochs: int = 40
126:
        max_stage2_epochs: int = 25
         max_stage3_epochs: int = 60
127:
128:
129:
         # Minimum epochs per stage (avoid switching too early)
130:
         min_stage1_epochs: int = 8
131:
         min_stage2_epochs: int = 5
        min_stage3_epochs: int = 12
132:
133:
134:
         # Learning rates (per stage)
135:
         stage1_lr: float = 1e-5 # was 3e-4
136:
         stage1_weight_decay: float = 1e-5
137:
138:
         stage2_lr: float = 5e-6 \# was 1e-4
         stage2_weight_decay: float = 1e-5
139:
140:
```

```
stage3 lr: float = 1e-6 # was 5e-5
141:
         stage3_weight_decay: float = 1e-5
142:
143:
144:
         # Advanced training techniques
145:
        use_mixed_precision: bool = True
146:
        use_cosine_restarts: bool = True
147:
        warmup_epochs: int = 5
148:
        grad_clip_norm: float = 0.5
149:
150:
         # Gradient monitoring for dynamic weighting
151:
        track_gradient_norms: bool = True
152:
        gradient_norm_window: int = 10 # rolling window for gradient tracking
153:
154:
         # Checkpointing
155:
        checkpoint_freq: int = 1
156:
157:
         # Curriculum configuration
158:
        curriculum: CurriculumConfig = None
159:
160:
        def __post_init__(self):
161:
             if self.curriculum is None:
162:
                 self.curriculum = CurriculumConfig()
163:
164:
165: @dataclass
166: class LossConfig:
        """Loss function weights (will be dynamically adjusted during training)"""
168:
         # Base weights (starting values)
169:
       seg_weight: float = 1.0
170:
       dice_weight: float = 1.0
        sdf_weight: float = 0.5
171:
172:
        attr_weight: float = 1.0
        polygon_weight: float = 1.0
173:
        voxel_weight: float = 1.0
174:
        topology_weight: float = 0.1 # start low, ramp up
175:
176:
177:
         # New loss components
178:
        latent_consistency_weight: float = 0.5
179:
        graph_constraint_weight: float = 0.3
180:
181:
        # Dynamic weighting parameters
182:
        enable_dynamic_weighting: bool = True
        weight_update_freq: int = 10
184:
        weight_momentum: float = 0.9
185:
186:
187: @dataclass
188: class InferenceConfig:
        """Inference configuration"""
189:
190:
        model_path: str = "final_model.pth"
        test_images_dir: str = "./data/test_images"
191:
192:
       output_dir: str = "./outputs"
193:
        export_intermediate: bool = True
194:
        polygon_threshold: float = 0.5
195:
196:
197: # Curriculum stage transition logic
198: class StageTransitionCriteria:
199:
         """Defines criteria for automatic stage transitions"""
200:
201:
        @staticmethod
202:
         def should_transition_from_stagel(train_losses, val_losses, config: CurriculumConfig) -> bool:
203:
             """Check if should transition from Stage 1 to Stage 2""
204:
             if len(val_losses) < config.plateau_detection_window:</pre>
                 return False
205:
206:
207:
             # Check for plateau in segmentation + dice losses
208:
             recent_losses = val_losses[-config.plateau_detection_window:]
209:
             if len(recent_losses) < 2:</pre>
210:
                 return False
211:
212:
             # Calculate improvement rate
213:
             old_avg = sum(recent_losses[:len(recent_losses)//2]) / (len(recent_losses)//2)
```

```
214:
             new_avg = sum(recent_losses[len(recent_losses)//2:]) / (len(recent_losses) - len(recent_losses)//2)
215:
216:
             improvement_rate = (old_avg - new_avg) / (old_avg + 1e-8)
217:
             return improvement_rate < config.min_improvement_threshold</pre>
218:
219:
        @staticmethod
220:
         def should_transition_from_stage2(polygon_losses, config: CurriculumConfig) -> bool:
             """Check if should transition from Stage 2 to Stage 3"""
221:
222:
             if len(polygon_losses) < config.plateau_detection_window:</pre>
223:
                 return False
224:
225:
             # Check polygon loss plateau
             recent_losses = polygon_losses[-config.plateau_detection_window:]
226:
             if len(recent_losses) < 2:</pre>
227:
228:
                 return False
229:
230:
             old_avg = sum(recent_losses[:len(recent_losses)//2]) / (len(recent_losses)//2)
231:
             new_avg = sum(recent_losses[len(recent_losses)//2:]) / (len(recent_losses) - len(recent_losses)//2)
232:
233:
             improvement_rate = (old_avg - new_avg) / (old_avg + 1e-8)
234:
             return improvement_rate < config.min_improvement_threshold</pre>
235:
236:
237: # Default configurations (import these in your trainer)
238: DEFAULT_DATA_CONFIG = DataConfig()
239: DEFAULT_MODEL_CONFIG = ModelConfig()
240: DEFAULT_TRAINING_CONFIG = TrainingConfig()
241: DEFAULT_LOSS_CONFIG = LossConfig()
242: DEFAULT_INFERENCE_CONFIG = InferenceConfig()
```

### ■ File: convert\_polygons.py

```
______
 1: import json
 2: from pathlib import Path
 ვ∶
 4: def flatten_polygon_file(file_path: Path):
 5:
       with open(file_path, "r") as f:
 6:
            data = json.load(f)
 7:
 8:
       flat_list = []
 9:
10:
        # If file already flat, skip
 11:
        if isinstance(data, list):
 12:
            print(f"[SKIP] Already flat: {file_path}")
13:
            return
14:
        # Otherwise, flatten categories
15:
        for category in ["walls", "doors", "windows", "floors", "fixtures"]:
 16:
17:
            if category in data and isinstance(data[category], list):
18:
                for idx, poly in enumerate(data[category]):
19:
                   flat_list.append({
 20:
                        "id": idx,
 21:
                        "type": category[:-1], # "walls" -> "wall"
                        "points": poly.get("points", []),
 22:
                        "area": poly.get("area", None)
 23:
                    })
 24:
 25:
        # Save back in flat format
 26:
 27:
        with open(file_path, "w") as f:
 28:
            json.dump(flat_list, f, indent=2)
 29:
 30:
        print(f"[OK] Converted: {file_path}")
31:
 32:
33: def batch_convert(root_dir="data/floorplans"):
 34:
        root = Path(root_dir)
 35:
        for polygon_file in root.rglob("polygon.json"):
36:
            flatten_polygon_file(polygon_file)
 37:
 38:
 39: if __name__ == "__main__":
```

\_\_\_\_\_\_

```
40: # Change path if needed
41: batch_convert("data/floorplans")
```

\_\_\_\_\_

### ■ File: dataset.py

```
______
 1: """
 2: Dataset classes for the Neural-Geometric 3D Model Generator
 3: Enhanced with in-memory caching for faster training
 4: """
 5:
 6: import cv2
 7: import json
 8: import numpy as np
 9: import torch
10: from torch.utils.data import Dataset
 11: from pathlib import Path
12: from typing import Dict, List, Tuple, Optional, Union
13: import time
14:
15: from config import DEFAULT_DATA_CONFIG
16:
17:
18: class AdvancedFloorPlanDataset(Dataset):
19:
 20:
        Research-grade dataset with complete ground truth:
 21:
        - Floorplan image + segmentation mask
        - Attribute dictionary (geometric parameters)
 23:
        - Ground-truth mesh + voxelized occupancy
        - Polygon outlines for vectorization supervision
 24:
 25:
 26:
        Enhanced with optional in-memory caching for performance
 27:
 28:
 29:
        def __init__(
 30:
            self,
31:
            data_dir: str = None,
32:
            split: str = "train",
            image_size: Tuple[int, int] = None,
 33:
 34:
            voxel_size: int = None,
35:
            augment: bool = None,
 36:
            config=None,
       ):
 37:
 38:
            # Use config if provided, otherwise defaults from config.py
 39:
            if config is None:
40:
                config = DEFAULT_DATA_CONFIG
 41:
 42:
            self.data_dir = Path(data_dir or config.data_dir)
 43:
            self.split = split
44:
            self.image_size = image_size or config.image_size
 45:
            self.voxel_size = voxel_size or config.voxel_size
 46:
            self.augment = (
 47:
                augment if augment is not None else config.augment
 48:
            ) and split == "train"
 49:
 50:
            # Collect all samples that contain every required file
51:
            self.samples = self._find_complete_samples()
 52:
            print(f"Found {len(self.samples)} complete samples for {split}")
53:
 54:
            # NEW: In-memory caching for performance
 55:
            self.cache_in_memory = getattr(config, "cache_in_memory", False)
 56:
            self._cache = None
 57:
            if self.cache_in_memory and len(self.samples) > 0:
58:
 59:
                print(f"[DATA] Preloading {len(self.samples)} samples into RAM (cache_in_memory=True).")
                print("[DATA]] This may take significant memory but will speed up training...")
 60:
 61:
62:
                # Estimate memory usage
63:
                estimated_mb = self._estimate_memory_usage()
 64:
                print(f"[DATA] Estimated memory usage: {estimated_mb:.1f} MB")
 65:
 66:
                start_time = time.time()
```

```
67:
                 self. preload cache()
 68:
                 load_time = time.time() - start_time
 69:
                 print(f"[DATA] Cache preloading completed in {load_time:.2f}s")
 70:
 71:
         def _estimate_memory_usage(self):
 72:
             """Estimate memory usage for caching"""
 73:
             if not self.samples:
 74:
                 return 0.0
 75:
             H, W = self.image_size
 76:
 77:
             n_samples = len(self.samples)
 78:
 79:
             # Rough estimates in bytes
             image_bytes = H * W * 3 # RGB uint8
 80:
             mask_bytes = H * W # grayscale uint8
 81:
 82:
             voxel_bytes = self.voxel_size ** 3 * 4 # float32
 83:
             json_bytes = 1024 # rough estimate for params + polygons
 84:
             total_per_sample = image_bytes + mask_bytes + voxel_bytes + json_bytes
 85:
 86:
             total_mb = (total_per_sample * n_samples) / (1024 * 1024)
 87:
 88:
             return total mb
 89:
 90:
         def _preload_cache(self):
              """Preload all samples into memory"""
 91:
             self._cache = []
 92:
 93:
             for i, sample in enumerate(self.samples):
 94:
 95:
                 if i % 100 == 0:
 96:
                     print(f"[DATA] Loading sample {i+1}/{len(self.samples)}")
 97:
 98:
                 try:
                     # Load image
 99:
                     img = cv2.imread(str(sample["image"]))
100:
101:
                     if img is None:
102:
                         print(f"Warning: Could not load image {sample['image']}")
103:
                         continue
104:
                     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
105:
                     img = cv2.resize(img, self.image_size) # (W, H) format for cv2.resize
106:
107:
                     # Load mask
108:
                     mask = cv2.imread(str(sample["mask"]), cv2.IMREAD_GRAYSCALE)
109:
                     if mask is None:
110:
                         print(f"Warning: Could not load mask {sample['mask']}")
111:
                         continue
112:
                     mask = cv2.resize(mask, self.image_size, interpolation=cv2.INTER_NEAREST)
113:
114:
                     # Load voxel data
115:
                     try:
116:
                         voxel_data = np.load(sample["voxel"])
117:
                         vox = voxel_data["voxels"] # Keep as numpy array
118:
                     except Exception as e:
119:
                         print(f"Warning: Could not load voxel data {sample['voxel']}: {e}")
120:
                         # Create dummy voxel data
121:
                         vox = np.zeros((self.voxel_size, self.voxel_size, self.voxel_size), dtype=np.float32)
122:
123:
                     # Load parameters
124:
                     try:
125:
                         with open(sample["params"], "r") as f:
126:
                             params = json.load(f)
127:
                     except Exception as e:
128:
                         print(f"Warning: Could not load params {sample['params']}: {e}")
129:
                         params = self._get_default_attributes()
130:
                     # Load polygons
131:
132:
                     try:
                         with open(sample["polygon"], "r") as f:
133:
134:
                             polygons = json.load(f)
135:
                     except Exception as e:
136:
                         print(f"Warning: Could not load polygons {sample['polygon']}: {e}")
137:
                         polygons = {"walls": []}
138:
139:
                     self._cache.append({
```

```
"image": img,
140:
141:
                       "mask": mask,
142:
                       "vox": vox,
143:
                       "params": params,
                        "polygons": polygons,
144:
145:
                       "sample_id": sample["image"].parent.name,
146:
                   })
147:
148:
               except Exception as e:
                   print(f"Error loading sample \{i\}: \{e\}")
149:
150:
                   continue
151:
        def _get_default_attributes(self):
152:
153:
            """Return default attributes for missing param files"""
154:
            return {
155:
               "wall_height": 2.6,
               "wall_thickness": 0.15,
156:
157:
                "window_base_height": 0.7,
                "window_height": 0.95,
158:
159:
               "door_height": 2.6,
160:
               "pixel_scale": 0.02,
161:
            }
162:
        # -----
163:
        def _find_complete_samples(self):
164:
            """Locate samples that contain all the expected files."""
165:
166:
            samples = []
167:
            split_dir = self.data_dir / self.split
168:
169:
            if not split_dir.exists():
               print(f"Warning: Split directory {split_dir} does not exist")
170:
171:
                return samples
172:
            for sample_dir in split_dir.iterdir():
173:
174:
               if not sample_dir.is_dir():
175:
                   continue
176:
177:
               required_files = {
178:
                    "image": sample_dir / "image.png",
                    "mask": sample_dir / "mask.png",
179:
180:
                    "params": sample_dir / "params.json",
                    "mesh": sample_dir / "model.obj",
181:
182:
                    "voxel": sample_dir / "voxel_GT.npz",
183:
                    "polygon": sample_dir / "polygon.json",
184:
               }
185:
186:
                if all(f.exists() for f in required_files.values()):
187:
                   samples.append(required_files)
188:
189:
            return samples
190:
191:
        # -----
192:
        def __len__(self):
193:
           return len(self. cache) if self. cache is not None else len(self.samples)
194:
        # ------
195:
        def __getitem__(self, idx):
196:
197:
            # Use cached data if available
198:
            if self._cache is not None:
199:
               cached_sample = self._cache[idx]
200:
               image = cached_sample['image']
201:
               mask = cached_sample['mask']
202:
               vox = cached_sample['vox']
203:
               attributes = cached_sample['params']
               polygons_gt = cached_sample['polygons']
204:
205:
               sample_id = cached_sample['sample_id']
206:
            else:
207:
               # Fallback: load from disk on-the-fly
208:
               sample = self.samples[idx]
209:
210:
                # Load image and mask
                image = cv2.imread(str(sample["image"]))
211:
212:
                image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

```
213:
                 image = cv2.resize(image, self.image size)
214:
215:
                 mask = cv2.imread(str(sample["mask"]), cv2.IMREAD_GRAYSCALE)
216:
                mask = cv2.resize(mask, self.image_size, interpolation=cv2.INTER_NEAREST)
217:
218:
                 # Load attributes
219:
                 with open(sample["params"], "r") as f:
220:
                     attributes = json.load(f)
221:
                # Load voxel ground truth
222:
223:
                 voxel_data = np.load(sample["voxel"])
224:
                vox = voxel_data["voxels"]
225:
226:
                 # Load polygon ground truth
227:
                 with open(sample["polygon"], "r") as f:
228:
                     polygons_gt = json.load(f)
229:
230:
                 sample_id = sample["image"].parent.name
231:
232:
             # Normalize image to [0,1]
233:
             image = image.astype(np.float32) / 255.0
234:
235:
             # Clean mask (remove class 5 if present)
236:
             mask[mask == 5] = 0
237:
238:
             # Convert to tensors
239:
             image_tensor = torch.from_numpy(image).float().permute(2, 0, 1)
240:
             mask_tensor = torch.from_numpy(mask).long()
241:
             voxels_tensor = torch.from_numpy(vox.astype(np.float32))
242:
243:
            attr_tensor = self._process_attributes(attributes)
244:
            polygon_tensor = self._process_polygons(polygons_gt)
245:
246:
             # Apply augmentation if enabled
            if self.augment:
247:
248:
                 image_tensor, mask_tensor = self._augment(image_tensor, mask_tensor)
249:
250:
             # Add validation before returning
251:
            self._validate_sample_data(idx, image_tensor, mask_tensor, attr_tensor, voxels_tensor, polygon_tens
252:
253:
            return {
254:
                "image": image_tensor,
                 "mask": mask_tensor,
256:
                 "attributes": attr_tensor,
257:
                 "voxels_gt": voxels_tensor,
258:
                 "polygons_gt": polygon_tensor,
259:
                 "sample_id": sample_id,
260:
             }
261:
262:
         def _validate_sample_data(self, idx, image, mask, attributes, voxels, polygons):
263:
264:
             """Validate sample data for NaN/Inf values"""
265:
             tensors_to_check = [
266:
                ("image", image),
267:
                 ("mask", mask),
                ("attributes", attributes),
268:
269:
                ("voxels", voxels),
270:
                 ("polygons", polygons["polygons"])
271:
             ]
272:
273:
            corrupted_data = False
274:
275:
            for name, tensor in tensors_to_check:
276:
                 if torch.isnan(tensor).any():
277:
                     print(f"ERROR: \{name\} \ contains \ NaN \ values \ at \ sample \ \{idx\}")
278:
                     corrupted_data = True
279:
                 if torch.isinf(tensor).any():
280:
                     print(f"ERROR: {name} contains Inf values at sample {idx}")
281:
                     corrupted_data = True
282:
283:
             if corrupted_data:
284:
                 print(f"WARNING: Corrupted data detected in sample {idx}, replacing with safe fallback values")
285:
```

```
286:
                 # Replace corrupted tensors with safe fallback values
287:
                for name, tensor in tensors_to_check:
288:
                    if torch.isnan(tensor).any() or torch.isinf(tensor).any():
                         if name == "image":
289:
290:
                             # Replace with zeros (black image)
291:
                            image.data = torch.zeros_like(image)
292:
                         elif name == "mask":
293:
                            # Replace with zeros (background class)
294:
                            mask.data = torch.zeros_like(mask).long()
                         elif name == "attributes":
295:
                            # Replace with reasonable default values (0.5 normalized)
296:
297:
                            attributes.data = torch.ones_like(attributes) * 0.5
298:
                         elif name == "voxels":
299:
                             # Replace with empty voxel grid
                            voxels.data = torch.zeros_like(voxels)
300:
301:
                         elif name == "polygons":
302:
                            # Replace polygons with zeros
303:
                            polygons["polygons"].data = torch.zeros_like(polygons["polygons"])
304:
305:
306:
         def _process_attributes(self, attributes):
307:
             """Convert attribute dictionary to a normalized tensor."""
308:
             # Normalize common architectural parameters into [0,1]
309:
            attr list = [
                attributes.get("wall_height", 2.6) / 5.0,
310:
                attributes.get("wall_thickness", 0.15) / 0.5,
311:
312:
                attributes.get("window_base_height", 0.7) / 3.0,
313:
                attributes.get("window_height", 0.95) / 2.0,
314:
                attributes.get("door_height", 2.6) / 5.0,
315:
                attributes.get("pixel_scale", 0.01) / 0.02,
316:
            1
317:
318:
            # Ensure no NaN/Inf values in attribute processing
319:
            safe_attr_list = []
320:
            for val in attr_list:
321:
                if np.isnan(val) or np.isinf(val):
322:
                    safe_attr_list.append(0.5) # Default normalized value
323:
324:
                    safe_attr_list.append(max(0.0, min(1.0, val))) # Clamp to [0,1]
325:
326:
             return torch.tensor(safe_attr_list, dtype=torch.float32)
327:
328:
         # -----
329:
         def _process_polygons(self, polygons_gt):
330:
             ""Convert polygon ground truth into a fixed tensor representation.
331:
            Handles both formats:
332:
            1. Nested dict: { "walls": [...], "doors": [...], ... }
333:
             2. Flat list: [ {"type": "wall", "points": [...]}, ... ]
334:
335:
            max_polygons = 30  # number of polygons per sample
336:
            max points = 100
                                # max points per polygon
337:
338:
            processed = torch.zeros(max_polygons, max_points, 2)
339:
            valid_mask = torch.zeros(max_polygons, dtype=torch.bool)
340:
341:
            poly_idx = 0
342:
343:
            trv:
344:
                # --- Case 1: dict format ---
345:
                if isinstance(polygons_gt, dict):
346:
                    for class_name, polygon_list in polygons_gt.items():
347:
                        if not isinstance(polygon_list, list):
348:
                            continue
349:
                         for polygon in polygon_list:
                            if poly_idx >= max_polygons:
350:
351:
                                break
                            if "points" not in polygon:
352:
353:
                                continue
354:
355:
                            points = np.array(polygon["points"])
                            if len(points) > max_points:
356:
357:
                                # Subsample evenly if too many points
358:
                                indices = np.linspace(0, len(points) - 1, max_points, dtype=int)
```

```
359:
                                 points = points[indices]
360:
361:
                             # Check for NaN/Inf in points
362:
                             if np.any(np.isnan(points)) or np.any(np.isinf(points)):
363:
                                 print(f"Warning: Invalid polygon points detected, skipping polygon")
364:
                                 continue
365:
366:
                             # Normalize to [0,1] relative to image size
367:
                             points = points / np.array(self.image_size)
                             # Clamp to valid range
368:
                            points = np.clip(points, 0.0, 1.0)
369:
370:
371:
                            processed[poly_idx, : len(points)] = torch.from_numpy(points).float()
372:
                            valid_mask[poly_idx] = True
373:
                            poly_idx += 1
374:
                 # --- Case 2: list format ---
375:
376:
                 elif isinstance(polygons_gt, list):
377:
                    for polygon in polygons_gt:
378:
                         if poly_idx >= max_polygons:
379:
                            break
380:
                         if "points" not in polygon:
381:
                             continue
382:
383:
                         points = np.array(polygon["points"])
384:
                         if len(points) > max_points:
385:
                             indices = np.linspace(0, len(points) - 1, max_points, dtype=int)
386:
                            points = points[indices]
387:
388:
                         # Check for NaN/Inf in points
389:
                         if np.any(np.isnan(points)) or np.any(np.isinf(points)):
390:
                            print(f"Warning: Invalid polygon points detected, skipping polygon")
391:
                            continue
392:
393:
                         points = points / np.array(self.image_size)
394:
                        points = np.clip(points, 0.0, 1.0)
395:
396:
                        processed[poly_idx, : len(points)] = torch.from_numpy(points).float()
397:
                        valid_mask[poly_idx] = True
398:
                        poly_idx += 1
399:
400:
            except Exception as e:
                print(f"Warning: Error processing polygons: {e}")
402:
                 # Return safe empty polygon data
403:
                processed = torch.zeros(max_polygons, max_points, 2)
404:
                 valid_mask = torch.zeros(max_polygons, dtype=torch.bool)
405:
406:
             return {"polygons": processed, "valid_mask": valid_mask}
407:
408:
         # -----
        def _augment(self, image, mask):
409:
410:
             """Enhanced data augmentation with rotations, flips, and intensity changes."""
411:
             # Random rotation (multiples of 90° only for architectural data)
412:
            if torch.rand(1) < 0.5:</pre>
413:
                k = torch.randint(1, 4, (1,)).item()
                image = torch.rot90(image, k, dims=[1, 2])
414:
                mask = torch.rot90(mask, k, dims=[0, 1])
415:
416:
417:
             # Random horizontal flip
418:
             if torch.rand(1) < 0.5:
419:
                image = torch.flip(image, dims=[2])
420:
                mask = torch.flip(mask, dims=[1])
421:
422:
             # Random vertical flip
             if torch.rand(1) < 0.5:
423:
424:
                image = torch.flip(image, dims=[1])
425:
                mask = torch.flip(mask, dims=[0])
426:
427:
             # Slight brightness/contrast adjustment with safety checks
            if torch.rand(1) < 0.3:
428:
429:
                brightness = torch.rand(1) * 0.2 - 0.1 \# \pm 0.1
430:
                contrast = torch.rand(1) * 0.2 + 0.9 # 0.9-1.1
431:
                 image = torch.clamp(image * contrast + brightness, 0, 1)
```

```
432:
                # Additional safety check for augmented image
433:
434:
               if torch.isnan(image).any() or torch.isinf(image).any():
                   print("Warning: Augmentation produced invalid values, reverting to original")
435:
436:
                   # Revert to safe values
437:
                   image = torch.clamp(image, 0, 1)
438:
                   image = torch.where(torch.isnan(image) | torch.isinf(image),
439:
                                   torch.zeros_like(image), image)
440:
441:
           return image, mask
442:
443:
        # -----
444:
        def get_cache_info(self):
445:
            """Return information about caching status"""
446:
            return {
447:
               "cache_enabled": self.cache_in_memory,
               "cache_loaded": self._cache is not None,
448:
449:
                "cached_samples": len(self._cache) if self._cache else 0,
                "total_samples": len(self.samples),
450:
451:
               "estimated_memory_mb": self._estimate_memory_usage() if self.cache_in_memory else 0
452:
            }
453:
454:
        def disable_cache(self):
455:
            """Disable caching and free memory"""
456:
            if self._cache is not None:
               print(f"[DATA] Disabling cache and freeing memory for {len(self._cache)} samples")
457:
458:
               self._cache = None
459:
               self.cache_in_memory = False
460:
461:
       def enable_cache(self):
            """Enable caching if not already enabled"""
462:
463:
            if not self.cache_in_memory and self.samples:
               self.cache_in_memory = True
464:
465:
               print("[DATA] Enabling cache...")
466:
               self._preload_cache()
467:
468:
470: # Synthetic sample generator for testing without dataset
472: def create_synthetic_data_sample():
473:
        """Generate a synthetic floorplan with attributes, voxels, and polygons."""
474:
        image = np.ones((256, 256, 3), dtype=np.uint8) * 255
475:
        mask = np.zeros((256, 256), dtype=np.uint8)
476:
477:
        # Simple square room
478:
        room_points = np.array([[50, 50], [200, 50], [200, 200], [50, 200]])
479:
        cv2.fillPoly(mask, [room_points], 1) # Room = class 1
480:
       cv2.polylines(image, [room_points], True, (0, 0, 0), 3)
481:
482:
        # Add door
        cv2.rectangle(mask, (90, 50), (110, 70), 2) # Door = class 2
483:
484:
        cv2.rectangle(image, (90, 50), (110, 70), (255, 0, 0), -1)
485:
486:
        # Attributes
        attributes = {
487:
            "wall_height": 2.6,
488:
489:
           "wall_thickness": 0.15,
490:
            "window_base_height": 0.7,
491:
            "window_height": 0.95,
492:
            "door_height": 2.6,
493:
           "pixel_scale": 0.02,
494:
        }
495:
496:
        # Simple voxel GT
497:
        voxels = np.zeros((64, 64, 64), dtype=bool)
        voxels[:20, 10:50, 10:50] = True
498:
499:
500:
        # Polygon GT
501:
        polygons = {"walls": [{"points": room_points.tolist()}]}
502:
503:
        return image, mask, attributes, voxels, polygons
504:
```

```
505:
506: class SyntheticFloorPlanDataset(Dataset):
507:
508:
         Synthetic dataset for testing and development when real data is not available
509:
510:
511:
        def __init__(self, num_samples=1000, image_size=(256, 256), voxel_size=64):
512:
             self.num_samples = num_samples
             self.image_size = image_size
513:
514:
             self.voxel_size = voxel_size
515:
516:
        def __len__(self):
517:
             return self.num_samples
518:
519:
        def __getitem__(self, idx):
520:
             # Generate deterministic synthetic data based on index
521:
             np.random.seed(idx)
522:
            torch.manual_seed(idx)
523:
524:
            image, mask, attributes, voxels, polygons_gt = create_synthetic_data_sample()
525:
526:
             # Convert to tensors
527:
             image_tensor = torch.from_numpy(image.astype(np.float32) / 255.0).permute(2, 0, 1)
528:
            mask_tensor = torch.from_numpy(mask).long()
529:
            voxels_tensor = torch.from_numpy(voxels.astype(np.float32))
530:
531:
             # Process attributes and polygons using same methods as main dataset
532:
             dataset = AdvancedFloorPlanDataset.__new__(AdvancedFloorPlanDataset)
533:
            dataset.image_size = self.image_size
534:
535:
            attr_tensor = dataset._process_attributes(attributes)
536:
            polygon_tensor = dataset._process_polygons(polygons_gt)
537:
538:
            return {
                "image": image_tensor,
539:
540:
                 "mask": mask_tensor,
541:
                 "attributes": attr_tensor,
                 "voxels_gt": voxels_tensor,
542:
543:
                 "polygons_gt": polygon_tensor,
544:
                 "sample_id": f"synthetic_{idx:06d}",
545:
             }
```

#### ■ File: demo.py

```
______
 2: Demo script for the Neural-Geometric 3D Model Generator
 3: """
 4:
 5: import torch
 6: import cv2
 7: import numpy as np
 8: from pathlib import Path
10: from models.model import NeuralGeometric3DGenerator
11: from dataset import create_synthetic_data_sample
12: from utils.visualization import visualize_predictions, create_model_summary_report
13:
14:
15: def demo_pipeline():
16:
        """Demonstrate the complete pipeline with synthetic data"""
17:
        print("Neural-Geometric 3D Model Generator Demo")
 18:
        print("=" * 50)
19:
 20:
        # Create output directory
21:
        demo_dir = Path("./demo_outputs")
       demo_dir.mkdir(exist_ok=True)
 22:
 23:
 24:
       # Create synthetic sample
 25:
       print("Creating synthetic data sample...")
 26:
        image, mask, attributes, voxels, polygons = create_synthetic_data_sample()
 27:
```

```
28:
         # Save synthetic data
 29:
         cv2.imwrite(str(demo_dir / "demo_input.png"), image)
 30:
         cv2.imwrite(str(demo_dir / "demo_mask.png"), mask * 50)
 31:
 32:
         # Create model (random weights for demo)
 33:
        print("Initializing model...")
 34:
         model = NeuralGeometric3DGenerator()
 35:
        model.eval()
 36:
 37:
         # Convert to tensors
         image_tensor = torch.from_numpy(image / 255.0).float().permute(2, 0, 1).unsqueeze(0)
 38:
 39:
 40:
         # Create model summary
         create_model_summary_report(model, image_tensor, str(demo_dir / "model_summary.txt"))
41:
 42:
 43:
         # Forward pass
         print("Running forward pass...")
 44:
 45:
         with torch.no_grad():
            predictions = model(image_tensor)
 46:
 47:
 48:
        print("Forward pass completed")
 49:
        print(f"Segmentation shape: {predictions['segmentation'].shape}")
 50:
         print(f"Attributes shape: {predictions['attributes'].shape}")
        print(f"Polygons shape: {predictions['polygons'].shape}")
 51:
        print(f"Voxels shape: {predictions['voxels_pred'].shape}")
 52:
 53:
 54:
         # Extract and save results
         seg_pred = torch.argmax(predictions["segmentation"], dim=1).squeeze().numpy()
 55:
 56:
         attr_pred = predictions["attributes"].squeeze().numpy()
 57:
         cv2.imwrite(str(demo_dir / "demo_seg_pred.png"), seg_pred * 50)
 58:
 59:
 60:
         print(f"Predicted attributes: {attr_pred}")
 61:
         # Create visualization
 62:
 63:
        print("Creating visualizations...")
 64:
 65:
         # Create targets for visualization
 66:
         targets = {
 67:
             "mask": torch.from_numpy(mask).unsqueeze(0),
 68:
             "attributes": torch.from_numpy(np.array([
 69:
                 attributes["wall_height"] / 5.0,
 70:
                 attributes["wall_thickness"] / 0.5,
71:
                 attributes["window_base_height"] / 3.0,
 72:
                 attributes["window_height"] / 2.0,
                 attributes["door_height"] / 5.0,
 73:
 74:
                 attributes["pixel_scale"] / 0.02,
 75:
             ])).float().unsqueeze(0)
 76:
         }
 77:
 78:
         visualize_predictions(
 79:
            image_tensor,
 :08
             predictions,
 81:
             targets,
 82:
             save_path=str(demo_dir / "demo_predictions.png")
 83:
 84:
 85:
         print(f"Demo completed successfully! Results saved to {demo_dir}")
 86:
87:
 88: def demo_with_pretrained(model_path, input_image_path=None):
 89:
         """Demo with a pretrained model"""
 90:
         if not Path(model_path).exists():
 91:
             print(f"Model file {model_path} not found!")
92:
             return
 93:
         \verb|print(f"Running demo with pretrained model: {model_path}")|\\
 94:
 95:
96:
         # Load model
97:
        model = NeuralGeometric3DGenerator()
98:
         checkpoint = torch.load(model_path, map_location="cuda")
99:
         model.load_state_dict(checkpoint["model_state_dict"])
100:
         model.cuda()
```

```
101:
         model.eval()
102:
103:
         # Use provided image or create synthetic
104:
         if input_image_path and Path(input_image_path).exists():
105:
             image = cv2.imread(input_image_path)
106:
             image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
107:
             image = cv2.resize(image, (256, 256))
             image_tensor = torch.from_numpy(image / 255.0).float().permute(2, 0, 1).unsqueeze(0).cuda()
108:
109:
            print(f"Using input image: {input_image_path}")
         else:
110:
            print("Using synthetic data...")
             image, _, _, _ = create_synthetic_data_sample()
112:
113:
             image_tensor = torch.from_numpy(image / 255.0).float().permute(2, 0, 1).unsqueeze(0).cuda()
114:
115:
         # Run inference
116:
         with torch.no_grad():
117:
            predictions = model(image_tensor)
118:
         # Visualize results
119:
120:
        demo_dir = Path("./demo_outputs")
121:
        demo_dir.mkdir(exist_ok=True)
122:
123:
         visualize_predictions(
124:
            image tensor,
            predictions,
125:
126:
            save_path=str(demo_dir / "pretrained_demo.png")
127:
128:
129:
         print(f"Pretrained demo completed! Results saved to {demo_dir}")
130:
131:
132: if __name__ == "__main__":
133:
         import argparse
134:
        parser = argparse.ArgumentParser(description="Demo Neural-Geometric 3D Model Generator")
135:
136:
        parser.add_argument("--model_path", type=str, default=None,
137:
                           help="Path to pretrained model (optional)")
138:
        parser.add_argument("--input_image", type=str, default=None,
139:
                            help="Input image path (optional)")
140:
141:
         args = parser.parse_args()
142:
143:
       if args.model_path:
144:
            demo_with_pretrained(args.model_path, args.input_image)
145:
        else:
146:
             demo_pipeline()
```

# ■ File: evaluate.py

20: from evaluation.metrics import ModelEvaluator21: from inference.engine import ResearchInferenceEngine22: from utils.visualization import visualize\_predictions

```
______
 1: """
 2: evaluate.py
 3: Comprehensive evaluation CLI for Neural-Geometric 3D Model Generator.
 4:
 5: Usage examples:
 6: python evaluate.py --model_path checkpoints/final_model.pth --data_dir ./data/floorplans
 7:
     python evaluate.py --model_path checkpoints/final_model.pth --data_dir ./data/floorplans --visualize --sa
 8: """
 9:
10: import argparse
11: import json
12: from pathlib import Path
13: from typing import List, Dict
14:
15: import torch
16: from torch.utils.data import DataLoader
17:
18: # Project imports (match your repo layout)
19: from dataset import AdvancedFloorPlanDataset
```

```
23:
24:
25: def save_json(obj, path: Path):
26:
       path.parent.mkdir(parents=True, exist_ok=True)
        with open(path, "w") as f:
27:
28:
            json.dump(obj, f, indent=2)
29:
        print(f"[?] Saved JSON -> {path}")
30:
31:
32: def gather_per_sample_metrics(
       evaluator: ModelEvaluator,
34:
        dataset: AdvancedFloorPlanDataset,
35:
       device: str,
36:
       max_samples: int = None,
37: ) -> List[Dict]:
38:
39:
       Re-run evaluation loop sample-by-sample and collect per-sample metrics.
40:
        We use evaluator._evaluate_* helper methods (present in evaluation/metrics.py)
        so metrics match the overall evaluation.
41:
42:
43:
        loader = DataLoader(dataset, batch_size=1, shuffle=False)
44:
       per_sample = []
45:
        with torch.no_grad():
46:
            for idx, batch in enumerate(loader):
47:
48:
                if max_samples is not None and idx >= max_samples:
49:
                    break
50:
51:
                # Move tensors to device where applicable
52:
                batch_for_model = {}
53:
                for k, v in batch.items():
54:
                    if torch.is_tensor(v):
                        batch_for_model[k] = v.to(device)
55:
56:
                        batch_for_model[k] = v # dicts/strings stay as-is
57:
58:
59:
                # Forward
60:
                preds = evaluator.model(batch_for_model["image"])
61:
62:
                # segmentation
63:
                seg_res = evaluator._evaluate_segmentation(preds["segmentation"], batch_for_model["mask"])
64:
                # attributes
65:
                attr_res = evaluator._evaluate_attributes(preds["attributes"], batch_for_model["attributes"].to
66:
                # voxels
67:
                voxel_res = evaluator._evaluate_voxels(preds["voxels_pred"], batch_for_model["voxels_gt"].to(de
68:
                # polygons ? evaluator._evaluate_polygons expects format used in metrics.py
69:
                # batch["polygons_gt"] is a dict with "polygons" and "valid_mask"
70:
                poly_res = evaluator._evaluate_polygons(preds["polygons"], preds.get("polygon_validity", preds.
71:
72:
                sample_id = batch["sample_id"][0] if isinstance(batch["sample_id"], (list, tuple)) else batch["
73:
                sample metrics = {
                    "sample_id": str(sample_id),
74:
75:
                    "segmentation": seg_res,
76:
                    "attributes": attr_res,
77:
                    "voxels": voxel_res,
78:
                    "polygons": poly_res,
79:
                }
80:
                per_sample.append(sample_metrics)
81:
                if (idx + 1) % 10 == 0:
82:
83:
                    print(f"[INFO] Collected per-sample metrics for {idx+1}/{len(loader)} samples")
84:
85:
       return per_sample
86:
87:
88: def run_visualization_and_exports(
89:
        engine: ResearchInferenceEngine,
90:
        dataset: AdvancedFloorPlanDataset,
91:
        output_dir: Path,
92:
       device: str.
93:
       num_viz: int = 10,
94:
       max_export: int = 5,
95: ):
```

```
96:
  97:
               For the first `num_viz` samples, create visualizations using the model and optionally
  98:
               run deterministic 3D export to save intermediate results and a .obj.
  99:
100:
               output_dir.mkdir(parents=True, exist_ok=True)
101:
               loader = DataLoader(dataset, batch_size=1, shuffle=False)
102:
103:
               viz count = 0
104:
               export\_count = 0
105:
106:
               with torch.no_grad():
107:
                     for idx, batch in enumerate(loader):
                            sample_id = batch["sample_id"][0] if isinstance(batch["sample_id"], (list, tuple)) else batch["
108:
109:
                            sample_dir = Path(dataset.data_dir) / "test" / str(sample_id)
110:
111:
                            # Prepare tensors
                            image_tensor = batch["image"].to(device)
112:
113:
                            target_mask = batch["mask"].unsqueeze(0) if torch.is_tensor(batch["mask"]) else None
                            target_attrs = batch["attributes"].unsqueeze(0) if torch.is_tensor(batch["attributes"]) else No
114:
115:
116:
                            # Model predictions using engine.model (same underlying model)
117:
                            preds = engine.model(image_tensor)
118:
119:
                            # Visualization
120:
                            if viz_count < num_viz:</pre>
                                  vis_path = output_dir / f"viz_{sample_id}.png"
121:
122:
                                   try:
123:
                                         visualize_predictions(
124:
                                                image_tensor,
125:
                                                preds,
126:
                                                {"mask": target_mask, "attributes": target_attrs},
127:
                                                save_path=str(vis_path),
128:
129:
                                         print(f"[?] Saved visualization for sample {sample_id} -> {vis_path}")
130:
                                   except Exception as e:
131:
                                         print(f"[!] Visualization failed for {sample_id}: {e}")
132:
                                   viz_count += 1
133:
134:
                            # Export deterministic 3D (uses the image file path)
135:
                            if export_count < max_export:</pre>
136:
                                   image_file = sample_dir / "image.png"
137:
                                   out_obj = output_dir / f"{sample_id}_predicted_model.obj"
138:
                                   try:
139:
                                         success = engine.generate_3d_model(str(image_file), str(out_obj), export_intermediate=T
140:
                                         if success:
141:
                                               print(f"[?] Exported deterministic 3D model for {sample_id} -> {out_obj}")
142:
                                         else:
143:
                                               print(f"[!] 3D export returned False for {sample_id}")
144:
                                   except Exception as e:
                                         print(f"[!] 3D export failed for {sample_id}: {e}")
145:
146:
                                   export count += 1
147:
148:
                            if viz_count >= num_viz and export_count >= max_export:
149:
                                  break
150:
151:
152: def main():
153:
              parser = argparse.ArgumentParser(description="Evaluate Neural-Geometric 3D model")
              parser.add_argument("--model_path", "-m", required=True, help="Path to model checkpoint (checkpoint.pth
parser.add_argument("--data_dir", "-d", default="./data/floorplans", help="Dataset root with train/val/
154:
155:
156:
              parser.add_argument("--device", default=None, help="Device to use (cuda or cpu). Auto-detect if omitted
157:
              parser.add_argument("--visualize", action="store_true", help="Save visual comparison images (pred vs GT
               parser.add_argument("--save_outputs", action="store_true", help="Run deterministic 3D export for some s
158:
               parser.add_argument("--output_dir", default="./evaluation_outputs", help="Where to save reports/visuals
159:
              parser.add_argument("--num_viz", type=int, default=10, help="How many visualizations to produce (defaul
160:
161:
              parser.add_argument("--max_exports", type=int, default=3, help="How many deterministic 3D exports to ru
              parser.add_argument("--per_sample_json", action="store_true", help="Save per-sample metrics JSON (may be parser.add_argument("--per_sample_json", action="store_true", help="Save per-sample metrics JSON (may be parser.add_argument("--per_sample_json", action="store_true", help="Save per-sample metrics JSON (may be per-sample metrics 
162:
163:
              parser.add_argument("--limit_samples", type=int, default=None, help="If set, limit evaluation to first
164:
165:
              args = parser.parse_args()
166:
167:
               # Determine device
168:
               device = args.device or ("cuda" if torch.cuda.is_available() else "cpu")
```

```
print(f"[INFO] Using device: {device}")
169:
170:
171:
         model_path = Path(args.model_path)
172:
         if not model_path.exists():
             print(f"[ERROR] Model not found at: {model_path}")
173:
174:
             return
175:
176:
         # Load test dataset
         dataset = AdvancedFloorPlanDataset(data_dir=args.data_dir, split="test")
177:
178:
         if len(dataset) == 0:
179:
             print("[ERROR] No test samples found (dataset may be empty or data_dir incorrect).")
             return
180:
181:
182:
         # If user asked for a limited quick run, slice dataset.samples accordingly.
183:
         if args.limit samples is not None:
184:
             # Create a shallow copy dataset pointing to first N samples
185:
             dataset.samples = dataset.samples[: args.limit_samples]
186:
             print(f"[INFO] Limiting evaluation to first {len(dataset)} samples")
187:
188:
         # Create evaluator and run full evaluation
189:
         evaluator = ModelEvaluator(str(model_path), device=device)
190:
         summary = evaluator.evaluate_dataset(dataset)
191:
         evaluator.print_evaluation_results(summary)
192:
193:
         # Save summary JSON
194:
         out_dir = Path(args.output_dir)
195:
         out_dir.mkdir(parents=True, exist_ok=True)
196:
         save_json(summary, out_dir / f"{model_path.stem}_summary.json")
197:
198:
         # Optionally collect detailed per-sample metrics
199:
         if args.per_sample_json:
200:
             print("[INFO] Collecting per-sample metrics (this re-runs model inference sample-by-sample)...")
201:
             per_sample = gather_per_sample_metrics(evaluator, dataset, device, max_samples=None)
             save_json(per_sample, out_dir / f"{model_path.stem}_per_sample_metrics.json")
202:
203:
204:
         # Visualization and/or exports
205:
         if args.visualize or args.save_outputs:
206:
            print("[INFO] Initializing inference engine for visualizations/exports...")
207:
             engine = ResearchInferenceEngine(model_path=str(model_path), device=device)
208:
            run_visualization_and_exports(
209:
                 engine,
210:
                dataset,
211:
                out_dir,
212:
                 device,
213:
                 num viz=args.num viz,
214:
                 max_export=args.max_exports,
215:
             )
216:
217:
         print("[?] Evaluation finished.")
218:
219:
220: if __name__ == "__main__":
        main()
```

# ■ File: evaluation\metrics.py

```
______
 1: """
 2: Evaluation metrics and utilities for the Neural-Geometric 3D Model Generator
 3: """
 4:
 5: import torch
 6: import numpy as np
 7: from torch.utils.data import DataLoader
 9: from models.model import NeuralGeometric3DGenerator
10: from dataset import AdvancedFloorPlanDataset
11:
12:
13: def compute_iou(pred, target):
       """Compute IoU for segmentation"""
14:
15:
       intersection = (pred & target).float().sum()
```

```
union = (pred | target).float().sum()
16:
17:
        return (intersection / (union + 1e-6)).item()
18:
19:
20: def compute_3d_iou(pred, target):
21:
        """Compute 3D IoU for voxel grids"""
22:
        pred_bool = pred.bool()
23:
        target_bool = target.bool()
24:
25:
        intersection = (pred_bool & target_bool).float().sum()
        union = (pred_bool | target_bool).float().sum()
27:
28:
        return (intersection / (union + 1e-6)).item()
29:
30:
31: def compute_polygon_metrics(pred_polygons, gt_polygons, validity_pred, validity_gt):
        """Compute metrics for polygon prediction"""
32:
33:
        # Chamfer distance between polygon sets
34:
        valid_pred = pred_polygons[validity_pred > 0.5]
35:
        valid_gt = gt_polygons[validity_gt]
36:
37:
        if len(valid_pred) == 0 or len(valid_gt) == 0:
38:
            return {"chamfer_distance": float('inf'), "validity_accuracy": 0.0}
39:
        # Simplified chamfer distance computation
40:
        chamfer_dist = 0.0
41:
42:
        for pred_poly in valid_pred:
            min_dist = float('inf')
43:
44:
            for gt_poly in valid_gt:
45:
                dist = torch.norm(pred_poly - gt_poly, dim=-1).min().item()
46:
                min_dist = min(min_dist, dist)
47:
            chamfer_dist += min_dist
48:
        chamfer_dist /= len(valid_pred)
49:
50:
51:
        # Validity accuracy
52:
        validity_acc = ((validity_pred > 0.5) == validity_gt).float().mean().item()
53:
54:
       return {
            "chamfer_distance": chamfer_dist,
55:
56:
            "validity_accuracy": validity_acc
57:
        }
58:
59:
60: class ModelEvaluator:
61:
        """Comprehensive model evaluation"""
62:
63:
        def __init__(self, model_path, device="cuda"):
64:
            self.device = device
65:
            self.model = NeuralGeometric3DGenerator()
66:
67:
            # Load model
68:
            checkpoint = torch.load(model_path, map_location=device)
69:
            self.model.load_state_dict(checkpoint["model_state_dict"])
70:
            self.model.to(device)
71:
            self.model.eval()
72:
73:
            print(f"Loaded model from {model_path}")
74:
75:
        def evaluate_dataset(self, test_dataset):
76:
            """Comprehensive evaluation on test dataset"""
77:
            test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)
78:
79:
            # Metrics storage
            metrics = {
:08
81:
                "segmentation": {"ious": [], "class_ious": []},
                "attributes": {"maes": [], "mses": []},
82:
                "voxels": {"ious": [], "dice_scores": []},
83:
                "polygons": {"chamfer_distances": [], "validity_accs": []},
84:
85:
            }
86:
87:
            with torch.no_grad():
88:
                for batch_idx, batch in enumerate(test_loader):
```

```
batch = {k: v.to(self.device) if torch.is_tensor(v) else v
 89:
 90:
                             for k, v in batch.items()}
 91:
 92:
                     predictions = self.model(batch["image"])
 93:
 94:
                     # Evaluate segmentation
 95:
                     seg_metrics = self._evaluate_segmentation(
 96:
                         predictions["segmentation"], batch["mask"]
 97:
                     metrics["segmentation"]["ious"].append(seg_metrics["iou"])
 98:
 99:
                     metrics["segmentation"]["class_ious"].append(seg_metrics["class_ious"])
100:
101:
                     # Evaluate attributes
102:
                     attr_metrics = self._evaluate_attributes(
                         predictions["attributes"], batch["attributes"]
103:
104:
105:
                     metrics["attributes"]["maes"].append(attr_metrics["mae"])
106:
                     metrics["attributes"]["mses"].append(attr_metrics["mse"])
107:
108:
                     # Evaluate voxels
109:
                     voxel_metrics = self._evaluate_voxels(
110:
                         predictions["voxels_pred"], batch["voxels_gt"]
111:
112:
                     metrics["voxels"]["ious"].append(voxel_metrics["iou"])
                     metrics["voxels"]["dice_scores"].append(voxel_metrics["dice"])
113:
114:
115:
                     # Evaluate polygons
116:
                     poly_metrics = self._evaluate_polygons(
117:
                         predictions["polygons"],
118:
                         predictions["polygon_validity"],
119:
                         batch["polygons_gt"]
120:
                     metrics["polygons"]["chamfer_distances"].append(poly_metrics["chamfer_distance"])
121:
122:
                     metrics["polygons"]["validity_accs"].append(poly_metrics["validity_accuracy"])
123:
124:
                     if (batch_idx + 1) % 10 == 0:
125:
                         print(f"Evaluated {batch_idx + 1}/{len(test_loader)} samples")
126:
127:
             return self._compute_summary_metrics(metrics)
128:
129:
         def _evaluate_segmentation(self, pred_seg, target_mask):
              """Evaluate segmentation performance""
130:
131:
             seg_pred = torch.argmax(pred_seg, dim=1)
132:
133:
             # Overall IoU
134:
             overall_iou = compute_iou(seg_pred, target_mask)
135:
136:
             # Per-class IoU
137:
             num_classes = pred_seg.shape[1]
138:
             class_ious = []
139:
140:
            for c in range(num_classes):
141:
                 pred_c = (seg_pred == c)
142:
                 target_c = (target_mask == c)
143:
                 if target_c.sum() > 0: # Only compute if class exists in ground truth
144:
145:
                     iou_c = compute_iou(pred_c, target_c)
146:
                     class_ious.append(iou_c)
147:
148:
             return {
                 "iou": overall_iou,
149:
150:
                 "class_ious": class_ious
151:
             }
152:
153:
         def _evaluate_attributes(self, pred_attrs, target_attrs):
154:
             """Evaluate attribute prediction"""
155:
             mae = torch.mean(torch.abs(pred_attrs - target_attrs)).item()
156:
            mse = torch.mean((pred_attrs - target_attrs) ** 2).item()
157:
158:
             return {"mae": mae, "mse": mse}
159:
160:
         def _evaluate_voxels(self, pred_voxels, target_voxels):
161:
              """Evaluate 3D voxel prediction"""
```

```
162:
             pred_binary = (torch.sigmoid(pred_voxels) > 0.5).float()
             target_float = target_voxels.float()
163:
164:
165:
             # 3D IoU
166:
             iou_3d = compute_3d_iou(pred_binary, target_float)
167:
168:
             # 3D Dice score
169:
             intersection = (pred_binary * target_float).sum()
170:
             dice = (2 * intersection) / (pred_binary.sum() + target_float.sum() + 1e-6)
171:
172:
             return {
                 "iou": iou_3d,
173:
174:
                 "dice": dice.item()
175:
             }
176:
177:
         def _evaluate_polygons(self, pred_polygons, pred_validity, gt_polygons):
178:
              """Evaluate polygon prediction""'
179:
             return compute_polygon_metrics(
180:
                 pred_polygons[0],
181:
                 gt_polygons["polygons"][0],
182:
                 pred_validity[0],
183:
                 gt_polygons["valid_mask"][0]
184:
             )
185:
         def _compute_summary_metrics(self, metrics):
186:
187:
              """Compute summary statistics"""
188:
             summary = {}
189:
190:
             # Segmentation
191:
             summary["segmentation_mIoU"] = np.mean(metrics["segmentation"]["ious"])
192:
             summary["segmentation_std"] = np.std(metrics["segmentation"]["ious"])
193:
194:
             # Attributes
             summary["attribute_MAE"] = np.mean(metrics["attributes"]["maes"])
195:
             summary["attribute_MAE_std"] = np.std(metrics["attributes"]["maes"])
196:
197:
198:
             # Voxels
199:
             summary["voxel_mIoU"] = np.mean(metrics["voxels"]["ious"])
200:
             summary["voxel_mIoU_std"] = np.std(metrics["voxels"]["ious"])
201:
             summary["voxel_dice"] = np.mean(metrics["voxels"]["dice_scores"])
202:
             # Polygons
203:
204:
             valid_chamfer = [d for d in metrics["polygons"]["chamfer_distances"] if d != float('inf')]
205:
             if valid_chamfer:
206:
                 summary["polygon_chamfer"] = np.mean(valid_chamfer)
207:
                 summary["polygon_chamfer_std"] = np.std(valid_chamfer)
208:
             else:
209:
                 summary["polygon_chamfer"] = float('inf')
210:
                 summary["polygon_chamfer_std"] = 0.0
211:
             summary["polygon_validity_acc"] = np.mean(metrics["polygons"]["validity_accs"])
212:
213:
214:
             return summary
215:
216:
         def print_evaluation_results(self, summary):
            """Print formatted evaluation results""
217:
             print("=" * 60)
218:
            print("COMPREHENSIVE EVALUATION RESULTS")
219:
220:
            print("=" * 60)
221:
222:
             print(f"Segmentation mIoU: {summary['segmentation_mIoU']:.4f} ± {summary['segmentation_std']:.4f}")
223:
             print(f"Attribute MAE: {summary['attribute_MAE']:.4f} ± {summary['attribute_MAE_std']:.4f}")
224:
             print(f"Voxel 3D mIoU: {summary['voxel_mIoU']:.4f} ± {summary['voxel_mIoU_std']:.4f}")
225:
             print(f"Voxel Dice Score: {summary['voxel_dice']:.4f}")
226:
227:
             if summary['polygon_chamfer'] != float('inf'):
                 print(f"Polygon Chamfer Distance: {summary['polygon_chamfer']:.4f} ± {summary['polygon_chamfer_
228:
229:
             else:
230:
                 print("Polygon Chamfer Distance: No valid polygons")
231:
             print(f"Polygon Validity Accuracy: {summary['polygon_validity_acc']:.4f}")
232:
             print("=" * 60)
233:
234:
```

```
235:
236: def evaluate_model(model_path, data_dir="./data/floorplans"):
237: """Standalone evaluation function"""
        # Load test dataset
        test_dataset = AdvancedFloorPlanDataset(data_dir, split="test")
239:
240:
241:
       if len(test_dataset) == 0:
242:
           print("No test samples found!")
243:
            return None
244:
245:
       # Create evaluator
246:
        evaluator = ModelEvaluator(model_path)
247:
248:
        # Run evaluation
249:
        summary = evaluator.evaluate_dataset(test_dataset)
250:
251:
         # Print results
252:
        evaluator.print_evaluation_results(summary)
253:
       return summary
```

\_\_\_\_\_\_

### **■** File: infer.py

```
______
 1: """
 2: Main inference script for generating 3D models from 2D floorplans
 3: """
 4:
 5: import argparse
 6: from pathlib import Path
 7:
 8: from inference.engine import ResearchInferenceEngine
 9: from config import DEFAULT_INFERENCE_CONFIG
10:
11:
12: def main():
13:
      parser = argparse.ArgumentParser(description="Generate 3D models from 2D floorplans")
14:
        parser.add_argument("--model_path", type=str, default="final_model.pth",
15:
                          help="Path to trained model")
16:
       parser.add_argument("--input", type=str, required=True,
17:
                          help="Input image path or directory")
18:
       parser.add_argument("--output", type=str, required=True,
19:
                          help="Output path or directory")
 20:
       parser.add_argument("--device", type=str, default="cuda",
 21:
                          help="Inference device")
        parser.add_argument("--export_intermediate", action="store_true",
 22:
 23:
                          help="Export intermediate results")
       parser.add_argument("--polygon_threshold", type=float, default=0.5,
 24:
 25:
                          help="Threshold for polygon validity")
 26:
 27:
        args = parser.parse_args()
 28:
 29:
        # Initialize inference engine
 30:
        print(f"Initializing inference engine...")
        engine = ResearchInferenceEngine(
 31:
            model_path=args.model_path,
 32:
 33:
            device=args.device
 34:
        )
 35:
 36:
        input_path = Path(args.input)
 37:
        output_path = Path(args.output)
 38:
 39:
        if input_path.is_file():
           # Single image inference
 40:
 41:
            print(f"Processing single image: {input_path}")
42:
 43:
            if not output path.suffix:
                output_path = output_path / f"{input_path.stem}_model.obj"
 44:
45:
 46:
            success = engine.generate_3d_model(
 47:
                str(input path),
 48:
                str(output_path),
```

```
49:
                export_intermediate=args.export_intermediate
50:
51:
52:
           if success:
53:
               print(f"? Successfully generated: {output_path}")
54:
            else:
55:
                print(f"? Failed to generate model for: {input_path}")
56:
57:
       elif input_path.is_dir():
58:
            # Batch processing
59:
           print(f"Processing directory: {input_path}")
60:
61:
            # Find all image files
            image_extensions = {'.png', '.jpg', '.jpeg', '.bmp', '.tiff'}
62:
63:
            image files = [
64:
                f for f in input_path.iterdir()
65:
                if f.suffix.lower() in image_extensions
66:
67:
68:
            if not image_files:
69:
                print("No image files found in input directory!")
70:
71:
72:
           print(f"Found {len(image_files)} image files")
73:
74:
            # Create output directory
75:
           output_path.mkdir(exist_ok=True)
76:
77:
           # Process batch
78:
           results = engine.process_batch(image_files, output_path)
79:
80:
           # Print summary
           successful = sum(1 for r in results if r["success"])
81:
           print(f"\nBatch processing completed:")
82:
83:
           print(f"? Successful: {successful}/{len(results)}")
84:
           print(f"? Failed: {len(results) - successful}/{len(results)}")
85:
86:
           # List failed cases
87:
           failed_cases = [r for r in results if not r["success"]]
88:
           if failed_cases:
89:
                print("\nFailed cases:")
90:
                for case in failed_cases:
91:
                    error_msg = case.get("error", "Unknown error")
                    print(f" - {Path(case['input']).name}: {error_msg}")
92:
93:
94:
       else:
95:
           print(f"Error: Input path {input_path} does not exist!")
96:
97:
98: if __name__ == "__main__":
99:
        main()
```

\_\_\_\_\_

#### ■ File: inference\engine.py

```
______
 1: """
 2: Research-grade inference engine for 2D to 3D floorplan generation
 3: """
 4:
 5: import torch
 6: import cv2
 7: import numpy as np
 8: import json
 9: import trimesh
10: from pathlib import Path
11:
12: from models.model import NeuralGeometric3DGenerator
13: from config import DEFAULT_INFERENCE_CONFIG
14:
15:
16: class ResearchInferenceEngine:
17:
```

```
18:
        Complete inference system that converts 2D floorplans to 3D models
19:
        following the deterministic export pipeline
20:
21:
        def __init__(self, model_path=None, device="cuda", config=None):
22:
23:
            if config is None:
24:
                config = DEFAULT_INFERENCE_CONFIG
25:
            self.device = device
26:
27:
            self.config = config
            self.model = NeuralGeometric3DGenerator()
28:
29:
30:
            # Load trained model
31:
            model_path = model_path or config.model_path
            checkpoint = torch.load(model_path, map_location=device)
32:
33:
            self.model.load_state_dict(checkpoint["model_state_dict"])
34:
            self.model.to(device)
35:
            self.model.eval()
36:
37:
            print(f"Loaded trained model from {model_path}")
38:
39:
        def generate_3d_model(
40:
            self,
41:
            image path: str,
42:
            output_path: str,
43:
            export_intermediate: bool = None
44:
       ):
45:
46:
            Complete pipeline: Image -> Segmentation -> Polygons -> 3D Model
47:
48:
            export_intermediate = export_intermediate or self.config.export_intermediate
49:
50:
            # Load and preprocess image
51:
            image = self._load_image(image_path)
52:
53:
            with torch.no_grad():
54:
                # Neural network inference
55:
                predictions = self.model(image)
56:
57:
                # Extract predictions
58:
                segmentation = predictions["segmentation"]
59:
                attributes = predictions["attributes"]
60:
                polygons = predictions["polygons"]
                validity = predictions["polygon_validity"]
61:
62:
63:
                print("Neural network inference complete")
64:
65:
                # Convert to deterministic representations
66:
                mask_np = self._extract_mask(segmentation)
67:
                attributes_dict = self._extract_attributes(attributes)
68:
                polygons_list = self._extract_polygons(polygons, validity)
69:
70:
                print(f"Extracted: {len(polygons_list)} valid polygons")
71:
72:
                # Export intermediate results if requested
73:
                if export_intermediate:
74:
                    self._export_intermediates(
75:
                        mask_np, attributes_dict, polygons_list, Path(output_path).parent
76:
77:
78:
                # Generate 3D model using deterministic pipeline
79:
                success = self._generate_deterministic_3d(
:08
                    mask_np, attributes_dict, polygons_list, output_path
81:
82:
83:
                return success
84:
85:
        def _load_image(self, image_path):
             """Load and preprocess input image"""
86:
            image = cv2.imread(image_path)
87:
88:
            if image is None:
89:
                raise ValueError(f"Could not load image from {image_path}")
90:
```

```
91:
             image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
 92:
             image = cv2.resize(image, (256, 256))
             image = torch.from_numpy(image / 255.0).float()
 93:
 94:
             image = image.permute(2, 0, 1).unsqueeze(0)
 95:
             return image.to(self.device)
 96:
 97:
         def _extract_mask(self, segmentation):
 98:
              ""Convert soft segmentation to hard mask"""
 99:
             seg_pred = torch.argmax(segmentation, dim=1)
             mask_np = seg_pred.squeeze().cpu().numpy().astype(np.uint8)
100:
101:
            return mask_np
102:
103:
         def _extract_attributes(self, attributes):
104:
             """Convert normalized attributes back to physical values"""
105:
             attr_np = attributes.squeeze().cpu().numpy()
106:
             # Denormalize (reverse of normalization in dataset)
107:
108:
             attributes_dict = {
                "wall_height": float(attr_np[0] * 5.0),
109:
110:
                "wall_thickness": float(attr_np[1] * 0.5),
111:
                "window_base_height": float(attr_np[2] * 3.0),
112:
                 "window_height": float(attr_np[3] * 2.0),
                 "door_height": float(attr_np[4] * 5.0),
113:
114:
                 "pixel_scale": float(attr_np[5] * 0.02),
             }
115:
116:
117:
            return attributes_dict
118:
119:
        def _extract_polygons(self, polygons, validity, threshold=None):
120:
             """Extract valid polygons from network predictions"""
121:
             threshold = threshold or self.config.polygon_threshold
122:
            batch_size, num_polys, num_points, _ = polygons.shape
123:
124:
            polygons_list = []
125:
126:
            for poly_idx in range(num_polys):
                if validity[0, poly_idx] > threshold: # Only valid polygons
127:
128:
                    poly_points = polygons[0, poly_idx].cpu().numpy()
129:
130:
                     # Remove zero-padded points
131:
                     valid_points = poly_points[poly_points.sum(axis=1) > 0]
132:
133:
                     if len(valid_points) >= 3: # Minimum for a polygon
134:
                         # Convert to image coordinates (assuming 256x256 input)
135:
                         valid_points = valid_points * 256
136:
                         polygons_list.append(
137:
                             {
138:
                                 "points": valid_points.tolist(),
139:
                                 140:
                             }
                         )
141:
142:
143:
             return polygons_list
144:
145:
         def _export_intermediates(self, mask, attributes, polygons, output_dir):
             """Export intermediate results for debugging/analysis""
146:
             output_dir = Path(output_dir)
147:
148:
            output_dir.mkdir(exist_ok=True)
149:
150:
             # Export mask
151:
            cv2.imwrite(str(output_dir / "predicted_mask.png"), mask * 50)
152:
153:
             # Export attributes
154:
            with open(output_dir / "predicted_attributes.json", "w") as f:
                 json.dump(attributes, f, indent=2)
155:
156:
157:
             # Export polygons
158:
            with open(output_dir / "predicted_polygons.json", "w") as f:
159:
                 json.dump(polygons, f, indent=2)
160:
             # Visualize polygons on mask
161:
             vis\_img = np.zeros((256, 256, 3), dtype=np.uint8)
162:
             vis_img[:, :, 0] = mask * 50  # Background
163:
```

```
164:
165:
             for poly in polygons:
166:
                 points = np.array(poly["points"], dtype=np.int32)
167:
                 cv2.polylines(vis_img, [points], True, (0, 255, 0), 2)
168:
169:
             cv2.imwrite(str(output_dir / "polygon_visualization.png"), vis_img)
170:
171:
             print(f"Intermediate results exported to {output dir}")
172:
         def _generate_deterministic_3d(self, mask, attributes, polygons, output_path):
173:
             """Generate 3D model using deterministic geometric operations"""
174:
175:
176:
                 # Initialize mesh components
177:
                 vertices = []
178:
                 faces = []
179:
                 vertex_count = 0
180:
181:
                 # Extract geometric parameters
                 wall_height = attributes.get("wall_height", 2.6)
182:
183:
                 wall_thickness = attributes.get("wall_thickness", 0.15)
184:
                 pixel_scale = attributes.get("pixel_scale", 0.01)
185:
186:
                 print(
187:
                     f"Generating 3D model with wall_height={wall_height:.2f}m, thickness={wall_thickness:.2f}m"
188:
189:
190:
                 # Process each polygon (walls, rooms, etc.)
191:
                 for poly_idx, polygon in enumerate(polygons):
192:
                     poly_vertices, poly_faces = self._extrude_polygon_3d(
193:
                         polygon["points"],
194:
                         wall_height,
195:
                         wall_thickness,
                         pixel_scale,
196:
197:
                         vertex_count,
198:
                     )
199:
200:
                     vertices.extend(poly_vertices)
201:
                     faces.extend(poly_faces)
202:
                     vertex_count += len(poly_vertices)
203:
204:
                 # Add floor and ceiling
                 floor_verts, floor_faces = self._generate_floor_ceiling(
205:
                     mask, pixel_scale, wall_height, vertex_count
207:
208:
                 vertices.extend(floor_verts)
209:
                 faces.extend(floor_faces)
210:
211:
                 if len(vertices) == 0:
                     print("No geometry generated")
212:
213:
                     return False
214:
215:
216:
                 mesh = trimesh.Trimesh(vertices=np.array(vertices), faces=np.array(faces))
217:
218:
                 # Clean up mesh
219:
                 mesh.remove_duplicate_faces()
220:
                 mesh.remove_unreferenced_vertices()
221:
                mesh.fix_normals()
222:
223:
                 # Export
224:
                mesh.export(output_path)
225:
                 print(f"3D model exported to {output_path}")
226:
                 print(
227:
                     f"Mesh statistics: {len(mesh.vertices)} vertices, {len(mesh.faces)} faces"
228:
229:
230:
                 return True
231:
232:
             except Exception as e:
233:
                 print(f"Error generating 3D model: {str(e)}")
234:
235:
236:
         def _extrude_polygon_3d(self, points, height, thickness, scale, vertex_offset):
```

```
"""Extrude a 2D polygon to create 3D wall geometry"""
237:
238:
            vertices = []
239:
            faces = []
240:
             # Convert points to 3D coordinates
241:
242:
            points_3d = []
243:
             for point in points:
244:
                x = (point[0] - 128) * scale # Center and scale
245:
                 z = (128 - point[1]) * scale # Flip Y and scale
                 points_3d.append([x, 0, z])
246:
247:
248:
             # Create bottom vertices (y=0)
249:
             bottom_outer = points_3d
            bottom_inner = self._inset_polygon(points_3d, thickness)
250:
251:
252:
             # Create top vertices (y=height)
253:
             top_outer = [[p[0], height, p[2]] for p in bottom_outer]
254:
             top_inner = [[p[0], height, p[2]] for p in bottom_inner]
255:
256:
             # Combine all vertices
257:
             all_vertices = bottom_outer + bottom_inner + top_outer + top_inner
258:
            vertices.extend(all_vertices)
259:
260:
            n_points = len(points_3d)
261:
262:
             # Generate faces for walls
263:
             for i in range(n_points):
264:
                next_i = (i + 1) % n_points
266:
                 # Outer wall faces
267:
                 v1 = vertex_offset + i  # bottom outer
268:
                 v2 = vertex_offset + next_i # bottom outer next
                v3 = vertex_offset + 2 * n_points + next_i # top outer next
269:
                v4 = vertex_offset + 2 * n_points + i # top outer
270:
271:
272:
                faces.extend([[v1, v2, v3], [v1, v3, v4]])
273:
274:
                 # Inner wall faces (reverse winding)
275:
                 v1 = vertex_offset + n_points + i # bottom inner
276:
                 v2 = vertex_offset + n_points + next_i # bottom inner next
277:
                 v3 = vertex_offset + 3 * n_points + next_i # top inner next
                 v4 = vertex_offset + 3 * n_points + i # top inner
278:
279:
280:
                faces.extend([[v1, v3, v2], [v1, v4, v3]])
281:
282:
             # Top cap (between outer and inner)
283:
             for i in range(n_points):
284:
                next_i = (i + 1) % n_points
285:
286:
                 v1 = vertex_offset + 2 * n_points + i # top outer
                v2 = vertex_offset + 2 * n_points + next_i # top outer next
287:
                 v3 = vertex_offset + 3 * n_points + next_i # top inner next
289:
                 v4 = vertex_offset + 3 * n_points + i # top inner
290:
291:
                 faces.extend([[v1, v2, v3], [v1, v3, v4]])
292:
             # Bottom cap (between outer and inner)
293:
294:
             for i in range(n_points):
295:
                next_i = (i + 1) % n_points
296:
297:
                 v1 = vertex_offset + i # bottom outer
298:
                 v2 = vertex_offset + next_i # bottom outer next
299:
                 v3 = vertex_offset + n_points + next_i # bottom inner next
300:
                 v4 = vertex_offset + n_points + i # bottom inner
301:
302:
                 faces.extend([[v1, v3, v2], [v1, v4, v3]])
303:
304:
            return vertices, faces
305:
        def _inset_polygon(self, points, inset_distance):
306:
307:
             """Create inset polygon for wall thickness"""
308:
             if len(points) < 3:
309:
                 return points
```

```
310:
311:
             # Simple inset by moving each point inward along angle bisector
312:
             inset_points = []
313:
             n = len(points)
314:
315:
            for i in range(n):
316:
                 prev_i = (i - 1) % n
317:
                 next_i = (i + 1) % n
318:
319:
                p_prev = np.array(points[prev_i])
320:
                p_curr = np.array(points[i])
321:
                p_next = np.array(points[next_i])
322:
323:
                # Vectors to adjacent points
324:
                v1 = p_curr - p_prev
325:
                 v2 = p_next - p_curr
326:
327:
                 # Normalize vectors (in XZ plane, ignore Y)
328:
                 v1\_norm = np.array([v1[0], 0, v1[2]])
                 v2\_norm = np.array([v2[0], 0, v2[2]])
330:
331:
                 v1_len = np.linalg.norm(v1_norm)
332:
                 v2_len = np.linalg.norm(v2_norm)
333:
334:
                if v1_len > 1e-6:
335:
                    v1_norm /= v1_len
336:
                 if v2_len > 1e-6:
337:
                     v2_norm /= v2_len
338:
339:
                 # Angle bisector
340:
                 bisector = v1_norm + v2_norm
341:
                 bisector_len = np.linalg.norm(bisector)
342:
                if bisector_len > 1e-6:
343:
                     bisector /= bisector_len
344:
345:
346:
                     # Move point inward
                     inset_point = p_curr - bisector * inset_distance
347:
348:
                     inset_points.append([inset_point[0], inset_point[1], inset_point[2]])
349:
                 else:
350:
                     inset_points.append(points[i])
351:
352:
            return inset_points
353:
354:
        def _generate_floor_ceiling(self, mask, scale, wall_height, vertex_offset):
355:
             """Generate floor and ceiling geometry from segmentation mask"""
356:
             vertices = []
357:
            faces = []
358:
359:
             # Find floor regions (assuming class 0 = floor/room)
360:
             floor_mask = (mask == 0).astype(np.uint8)
361:
362:
             # Find contours
             contours, _ = cv2.findContours(
363:
364:
                 floor_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
365:
366:
367:
             for contour in contours:
368:
                 if cv2.contourArea(contour) < 100: # Skip small regions</pre>
369:
                     continue
370:
371:
                 # Simplify contour
372:
                 epsilon = 0.02 * cv2.arcLength(contour, True)
373:
                 approx = cv2.approxPolyDP(contour, epsilon, True)
374:
375:
                if len(approx) < 3:</pre>
376:
                     continue
377:
378:
                 # Convert to 3D coordinates
379:
                 floor_points = []
380:
                 for point in approx.reshape(-1, 2):
381:
                    x = (point[0] - 128) * scale
382:
                     z = (128 - point[1]) * scale
```

```
383:
                      floor_points.append([x, 0, z]) # Floor at y=0
384:
385:
                 ceiling_points = []
386:
                 for point in approx.reshape(-1, 2):
387:
                     x = (point[0] - 128) * scale
                      z = (128 - point[1]) * scale
388:
389:
                      ceiling_points.append([x, wall_height, z]) # Ceiling at y=wall_height
390:
391:
                 # Add vertices
                 n_points = len(floor_points)
392:
393:
                 vertices.extend(floor_points)
394:
                 vertices.extend(ceiling_points)
395:
                 # Triangulate floor
396:
                 if n_points >= 3:
397:
398:
                      for i in range(1, n_points - 1):
399:
                         faces.append(
400:
                              [vertex_offset, vertex_offset + i + 1, vertex_offset + i]
401:
402:
403:
                      # Triangulate ceiling (reverse winding)
404:
                      for i in range(1, n_points - 1):
405:
                          faces.append(
406:
                              Γ
407:
                                  vertex_offset + n_points,
408:
                                  vertex_offset + n_points + i,
409:
                                  vertex_offset + n_points + i + 1,
410:
411:
412:
413:
                 vertex_offset += 2 * n_points
414:
415:
             return vertices, faces
416:
417:
         def process_batch(self, image_paths, output_dir):
418:
              """Process multiple images in batch"""
419:
             output_dir = Path(output_dir)
420:
             output_dir.mkdir(exist_ok=True)
421:
422:
             results = []
423:
424:
             for img_path in image_paths:
425:
                 img_path = Path(img_path)
426:
                 print(f"Processing: {img_path.name}")
427:
428:
                 output_path = output_dir / f"{img_path.stem}_model.obj"
429:
430:
                 try:
431:
                      success = self.generate_3d_model(
432:
                          str(img_path), str(output_path), export_intermediate=True
433:
434:
435:
                     results.append({
436:
                          "input": str(img_path),
437:
                          "output": str(output_path),
                          "success": success
438:
439:
                     })
440:
441:
                      if success:
                          print(f"? Generated: {output_path}")
442:
443:
444:
                          print(f"? Failed: {img_path.name}")
445:
446:
                 except Exception as e:
447:
                     print(f"? \ Error \ processing \ \{img\_path.name\} \hbox{: } \{str(e)\}")
448:
                     results.append({
449:
                          "input": str(img_path),
                          "output": str(output_path),
450:
                          "success": False,
451:
452:
                          "error": str(e)
453:
                      })
454:
455:
             return results
```

### ■ File: models\dvx.py

```
1: """
2: Robust Differentiable Vectorization (DVX) module.
3:
4: Improvements vs naive DVX:
5: - Projects backbone feature maps to `feature_dim` if channels don't match via 1x1 conv.
6: - Multi-step iterative refinement (improves final polygon accuracy).
     Safe guards for shapes, device handling, and grid-sampling.
8: - Returns init_polygons, final polygons, per-step displacements, and validity scores.
9:
10: Usage:
11: - features: dict of feature maps (e.g. "p2", "p4"), each tensor (B, C, H, W).
12: - segmentation: (B, 1, H_img, W_img) or similar ? only used for optional initialization logic.
13: """
14:
15: from typing import Dict, Any, Optional, Tuple
16: import torch
17: import torch.nn as nn
18: import torch.nn.functional as F
19:
20:
21: class DifferentiableVectorization(nn.Module):
22:
      def init (
           self,
23:
24:
           max_polygons: int = 20,
25:
           max_points: int = 50,
           feature_dim: int = 256,
26:
27:
           displacement_scale: float = 0.12,
28:
           num_refinement_steps: int = 3,
29:
           align_corners: bool = False,
30:
           padding_mode: str = "border", # options for grid_sample
           use_proj_conv: bool = True,
31:
32:
33:
34:
           Args:
35:
                max_polygons: maximum polygons to predict per image
36:
                max_points: number of control points per polygon
37:
               feature_dim: number of channels the DVX expects (will project backbone features to this)
38:
                displacement_scale: multiplier for predicted displacement (tanh output)
39:
                num\_refinement\_steps: how many iterative refinement steps to apply (>=1)
40:
               align_corners: align_corners for F.grid_sample
41:
                padding_mode: padding_mode for F.grid_sample
42:
                use_proj_conv: whether to use 1x1 conv to project backbone features to feature_dim (recommended
43:
44:
            super().__init_
45:
           assert max_points > 2, "max_points must be > 2"
46:
           assert num_refinement_steps >= 1
47:
48:
           self.max_polygons = int(max_polygons)
           self.max_points = int(max_points)
49:
50:
           self.feature_dim = int(feature_dim)
51:
           self.displacement_scale = float(displacement_scale)
52:
            self.num_refinement_steps = int(num_refinement_steps)
53:
            self.align_corners = bool(align_corners)
           self.padding_mode = padding_mode
54:
           self.use_proj_conv = bool(use_proj_conv)
55:
56:
57:
            # init_net: from pooled p4 -> flattened -> produce normalized coords in [0,1]
58:
            # AdaptiveAvgPool2d(8) -> (B, C, 8, 8) -> flatten -> Linear(C*8*8 -> hidden)
59:
           hidden = max(512, feature_dim * 2)
60:
            self.init_pool = nn.AdaptiveAvgPool2d(8)
61:
62:
            # we'll create a projector conv for p4/p2 channels if necessary at runtime
            # but also create an MLP init_net that assumes feature_dim channels after pooling
63:
64:
            self.init_mlp = nn.Sequential(
65:
                nn.Flatten(),
66:
                nn.Linear(self.feature_dim * 8 * 8, hidden),
67:
                nn.ReLU(inplace=True),
68:
               nn.Linear(hidden, 1024),
               nn.ReLU(inplace=True),
69:
70:
               nn.Linear(1024, self.max_polygons * self.max_points * 2),
71:
                nn.Sigmoid(),
```

```
72:
 73:
 74:
             # refinement network: maps (feature_dim + 2) -> displacement in [-1,1]
 75:
             self.refine_net = nn.Sequential(
 76:
                 nn.Linear(self.feature_dim + 2, 256),
 77:
                 nn.ReLU(inplace=True),
 78:
                 nn.Linear(256, 128),
 79:
                 nn.ReLU(inplace=True),
 :08
                 nn.Linear(128, 2),
 81:
                 nn.Tanh(),
 82:
 83:
 84:
             # validity net (reads flattened coords only)
 25:
             self.validity_net = nn.Sequential(
 86:
                 nn.Linear(self.max_points * 2, 128),
 87:
                 nn.ReLU(inplace=True),
 88:
                 nn.Linear(128, 1),
 89:
                 nn.Sigmoid(),
 90:
             )
 91:
 92:
             # projector convs (create lazily when first seen a feature channel mismatch)
 93:
             # stored per-key: e.g., self._proj_convs['p2'] = nn.Conv2d(in_ch, feature_dim, 1)
 94:
             self._proj_convs = nn.ModuleDict()
 95:
             self._proj_created = set()
 96:
 97:
         def _ensure_projector(self, key: str, in_channels: int):
 98:
             Ensure a 1xl conv exists that projects `in_channels` -> self.feature_dim for feature map `key`.
 99:
100:
101:
             if not self.use_proj_conv:
102:
                 return None
103:
             if key in self._proj_created:
104:
                 return self._proj_convs[key]
105:
             if in_channels != self.feature_dim:
106:
107:
                 conv = nn.Conv2d(in_channels, self.feature_dim, kernel_size=1, stride=1, padding=0)
108:
                 # initialize conv: kaiming
109:
                 nn.init.kaiming_normal_(conv.weight, a=0.2)
110:
                 if conv.bias is not None:
111:
                     nn.init.zeros_(conv.bias)
112:
                 self._proj_convs[key] = conv
113:
             else:
114:
                 # identity mapping using 1x1 conv with weights = identity-like is tricky
115:
                 # Instead simply keep no conv; we'll pass feature as-is
116:
                 self._proj_convs[key] = nn.Identity()
117:
             self._proj_created.add(key)
             return self._proj_convs[key]
118:
119:
120:
         def _project_feature(self, key: str, feat: torch.Tensor) -> torch.Tensor:
121:
122:
             Project or verify feature map to have self.feature_dim channels.
123:
             If projector conv wasn't present and channels == feature_dim, returns feat unchanged.
124:
125:
             in_ch = feat.shape[1]
126:
             proj = self._ensure_projector(key, in_ch)
             if proj is None:
127:
                 # projection not desired; assert channels match
128:
129:
                 if in_ch != self.feature_dim:
130:
                     raise RuntimeError(
                         f"Feature '{key}' channels ({in_ch}) != feature_dim ({self.feature_dim}) "
131:
132:
                         "and projection disabled."
133:
                     )
134:
                 return feat
135:
             # if proj is Identity, apply it still (fast path)
136:
             return proj(feat)
137:
         def forward(
138:
139:
             self,
140:
             features: Dict[str, torch.Tensor],
141:
             segmentation: Optional[torch.Tensor] = None,
142:
            return_all_steps: bool = False,
143:
         ) -> Dict[str, Any]:
144:
             ....
```

```
features: dict with keys like "p2", "p4" containing tensors (B, C, H, W)
145:
             segmentation: optional (B, 1, H_img, W_img) or similar (not strictly required)
146:
147:
             return_all_steps: if True returns per-step intermediate polygons & displacements
148:
149:
             # pick features for init and refinement
            p4 = features.get("p4", None)
150:
151:
            p2 = features.get("p2", None)
152:
153:
             if p4 is None and p2 is None:
                 raise ValueError("At least one of 'p4' or 'p2' must be present in features.")
154:
155:
156:
             # prefer p4 for init; fallback to p2 if not present
157:
             init_feat = p4 if p4 is not None else p2
158:
             refine_feat = p2 if p2 is not None else p4
159:
160:
            B = init_feat.shape[0]
161:
162:
             # Project features to feature_dim (if needed)
163:
             init_feat = self._project_feature("p4_init", init_feat)
164:
            refine_feat = self._project_feature("p2_refine", refine_feat)
165:
166:
             # -- Initialize polygons --
167:
             # Pool then MLP; ensure init_mlp expects feature_dim channels
168:
             pooled = self.init_pool(init_feat) # [B, C', 8, 8]
169:
             if pooled.shape[1] != self.feature_dim:
170:
                 # If the projector returned Identity but pooled channels mismatch, try to apply a runtime proje
171:
                 pooled = self._project_feature("p4_init_postpool", pooled)
172:
173:
             init_logits = self.init_mlp(pooled) # [B, max_polygons * max_points * 2]
174:
             init_polygons = init_logits.view(B, self.max_polygons, self.max_points, 2) # normalized [0,1]
175:
176:
             # Iterative refinement
             polygons = init_polygons.clone()
177:
178:
             per_step_displacements = []
179:
             for step in range(self.num_refinement_steps):
180:
                 # sample features at the polygon control-point locations
181:
                 displ = self._single_refine_step(polygons, refine_feat)
182:
                 per_step_displacements.append(displ)
183:
                 polygons = torch.clamp(polygons + displ, 0.0, 1.0)
184:
185:
             # final validity
            validity = self._predict_validity(polygons)
186:
187:
188:
             out: Dict[str, Any] = {
189:
                 "polygons": polygons, # [B, P, N, 2]
190:
                 "validity": validity,
                                        # [B, P]
191:
                 "init_polygons": init_polygons,
192:
                 "refinement_displacements": per_step_displacements, # list of [B, P, N, 2]
193:
             }
194:
195:
             if return_all_steps:
196:
                 out["all_step_polygons"] = [
197:
                     torch.clamp(init_polygons + sum(per_step_displacements[:i + 1]), 0.0, 1.0)
198:
                     for i in range(len(per_step_displacements))
199:
200:
201:
             return out
202:
203:
         def _single_refine_step(self, polygons: torch.Tensor, feature_map: torch.Tensor) -> torch.Tensor:
204:
205:
             One refinement step: sample features at polygon points, predict displacement (scaled), return displ
206:
             polygons: [B, P, N, 2] in [0,1]
             feature_map: [B, C, H, W] with C == feature_dim (or projected)
207:
208:
             returns displacement: [B, P, N, 2] in [-displacement_scale, displacement_scale]
209:
210:
            B, P, N, _ = polygons.shape
211:
             # flatten pts to sample
            coords = polygons.view(B, -1, 2) \# [B, P*N, 2], coords in [0,1]
212:
             grid = coords * 2.0 - 1.0 \# to [-1,1]
213:
214:
            # grid_sample expects (B, H_out, W_out, 2); use W_out=1
215:
            grid_sample = grid.view(B, -1, 1, 2)
216:
            sampled = F.grid_sample(
217:
                 feature_map,
```

```
218:
                grid_sample,
219:
                mode="bilinear",
220:
                padding_mode=self.padding_mode,
221:
                align_corners=self.align_corners,
            ) # [B, C, P*N, 1]
222:
223:
            sampled = sampled.squeeze(-1).permute(0, 2, 1).contiguous() # [B, P*N, C]
224:
225:
            # combine sampled features and coords (coords in [0,1])
             input_feats = torch.cat([sampled, coords], dim=-1) # [B, P*N, C+2]
226:
227:
             # predict displacements in [-1,1] via tanh on last layer
228:
            disp = self.refine_net(input_feats) # [B, P*N, 2], values ~[-1,1]
229:
            disp = disp.view(B, P, N, 2)
230:
            disp = disp * self.displacement_scale # scale
231:
            return disp
232:
233:
        def _predict_validity(self, polygons: torch.Tensor) -> torch.Tensor:
234:
            B, P, N, \_ = polygons.shape
235:
             if N != self.max_points:
                # If someone truncated or padded points, adapt: flatten to last dim whatever it is
236:
237:
                poly_flat = polygons.view(B * P, -1)
238:
            else:
239:
                poly_flat = polygons.view(B * P, -1)
240:
            validity = self.validity_net(poly_flat) # [B*P, 1]
            validity = validity.view(B, P)
241:
242:
            return validity
243:
245: # ------ quick unit test / smoke test ------
246: def _smoke_test():
247:
       torch.manual_seed(0)
248:
        B = 2
249:
        C1 = 384 # different from feature_dim to test projector conv
        C2 = 128
250:
251:
        H2, W2 = 64, 64
252:
        H4, W4 = 16, 16
253:
254:
        # create dummy backbone features with different channels
255:
       p2 = torch.randn(B, C1, H2, W2)
256:
        p4 = torch.randn(B, C2, H4, W4)
        seg = torch.rand(B, 1, H2 * 4, W2 * 4) # just a placeholder
257:
258:
259:
        dvx = DifferentiableVectorization(
260:
         max_polygons=4,
261:
            max_points=16,
262:
            feature_dim=256,
263:
            displacement_scale=0.08,
264:
           num_refinement_steps=3,
265:
           align_corners=False,
           padding_mode="border",
266:
267:
            use_proj_conv=True,
268:
269:
270:
         # ensure module moves projector convs to device when dvx.to(device) called
271:
        dvx = dvx.eval() # inference mode ok
272:
         # Forward pass
        \verb"out = dvx(\{"p2": p2, "p4": p4\}, seg, return_all_steps=True)"
273:
        print("polygons shape:", out["polygons"].shape) # expected [B, P, N, 2]
274:
        print("validity shape:", out["validity"].shape) # expected [B, P]
275:
276:
        print("init shape:", out["init_polygons"].shape)
        print("refinement steps:", len(out["refinement_displacements"]))
277:
278:
        # check ranges
279:
        assert out["polygons"].min().item() >= 0.0 - 1e-6
280:
        assert out["polygons"].max().item() <= 1.0 + 1e-6</pre>
281:
        print("smoke test passed")
282:
284: if __name__ == "__main__":
       _smoke_test()
285:
```

------

### ■ File: models\encoder.py

```
-----
 1: """
 2: Encoder architecture for multi-scale feature extraction
 3: """
 4:
 5: import torch
 6: import torch.nn as nn
 7: import torch.nn.functional as F
 8:
 9:
10: class ResidualBlock(nn.Module):
11:
        """Basic residual block for the encoder"""
12:
13:
       def __init__(self, in_channels, out_channels, stride=1):
14:
            super().__init__()
15:
16:
           self.conv1 = nn.Conv2d(in_channels, out_channels, 3, stride, 1, bias=False)
17:
           self.bn1 = nn.BatchNorm2d(out_channels)
           self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False)
19:
           self.bn2 = nn.BatchNorm2d(out_channels)
20:
21:
           self.shortcut = nn.Sequential()
22:
           if stride != 1 or in_channels != out_channels:
23:
                self.shortcut = nn.Sequential(
                   nn.Conv2d(in_channels, out_channels, 1, stride, bias=False),
24:
25:
                    nn.BatchNorm2d(out_channels),
26:
                )
27:
28:
       def forward(self, x):
           out = F.relu(self.bn1(self.conv1(x)))
29:
30:
            out = self.bn2(self.conv2(out))
           out += self.shortcut(x)
31:
32:
           return F.relu(out)
33:
35: class MultiScaleEncoder(nn.Module):
37:
        Advanced encoder with skip connections and multi-scale feature extraction
38:
       Based on ResNet architecture with Feature Pyramid Network (FPN)
39:
40:
       def __init__(self, input_channels=3, feature_dim=512):
42:
           super().__init__()
43:
44:
            # Stem
45:
           self.stem = nn.Seguential(
46:
               nn.Conv2d(input_channels, 64, 7, 2, 3, bias=False),
47:
               nn.BatchNorm2d(64),
48:
               nn.ReLU(inplace=True),
49:
               nn.MaxPool2d(3, 2, 1),
50:
           )
51:
52:
            # ResNet blocks
53:
            self.layer1 = self._make_layer(64, 64, 2, stride=1) # 64x64
           self.layer2 = self._make_layer(64, 128, 2, stride=2) # 32x32
54:
           self.layer3 = self._make_layer(128, 256, 2, stride=2) # 16x16
55:
56:
           self.layer4 = self._make_layer(256, 512, 2, stride=2) # 8x8
57:
           # FPN lateral connections
58:
59:
           self.lateral4 = nn.Conv2d(512, feature_dim, 1)
60:
           self.lateral3 = nn.Conv2d(256, feature_dim, 1)
           self.lateral2 = nn.Conv2d(128, feature_dim, 1)
61:
62:
            self.lateral1 = nn.Conv2d(64, feature_dim, 1)
63:
64:
            # FPN output layers
65:
           self.smooth4 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
66:
           self.smooth3 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
67:
           self.smooth2 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
68:
           self.smooth1 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
69:
70:
            # Global context
71:
            self.global_pool = nn.AdaptiveAvgPool2d(1)
```

```
self.global_fc = nn.Sequential(
 72:
 73:
               nn.Linear(512, feature_dim),
 74:
                nn.ReLU(),
 75:
                nn.Linear(feature_dim, feature_dim)
 76:
            )
 77:
 78:
        def _make_layer(self, in_channels, out_channels, blocks, stride=1):
 79:
            lavers = []
 :08
            layers.append(ResidualBlock(in_channels, out_channels, stride))
            for _ in range(1, blocks):
 81:
                layers.append(ResidualBlock(out_channels, out_channels))
 82:
 83:
            return nn.Sequential(*layers)
 84:
       def forward(self, x):
 85:
 86:
            # Bottom-up pathway
 87:
            x = self.stem(x) # 64x64
 88:
 89:
            c1 = self.layer1(x) # 64x64
           c2 = self.layer2(c1) # 32x32
 90:
 91:
           c3 = self.layer3(c2) # 16x16
 92:
           c4 = self.layer4(c3) # 8x8
 93:
 94:
           # Global context
 95:
           global_feat = self.global_pool(c4).flatten(1)
           global_feat = self.global_fc(global_feat)
 96:
 97:
 98:
            # Top-down pathway (FPN)
 99:
           p4 = self.lateral4(c4)
           p3 = self.lateral3(c3) + F.interpolate(p4, scale_factor=2)
100:
101:
           p2 = self.lateral2(c2) + F.interpolate(p3, scale_factor=2)
102:
           p1 = self.lateral1(c1) + F.interpolate(p2, scale_factor=2)
103:
104:
           # Smooth
          p4 = self.smooth4(p4)
105:
           p3 = self.smooth3(p3)
106:
107:
           p2 = self.smooth2(p2)
           p1 = self.smooth1(p1)
108:
109:
110:
           return {
               "p1": p1, # 64x64
111:
112:
                "p2": p2, # 32x32
                "p3": p3, # 16x16
113:
114:
                "p4": p4, # 8x8
                "global": global_feat,
115:
116:
            }
______
```

■ File: models\extrusion.py

```
_______
 2: Vectorized Differentiable 3D extrusion module for converting polygons to 3D occupancy
 3: Optimized version with GPU-accelerated vectorized operations
 4: """
 5:
 6: import torch
 7: import torch.nn as nn
 8: import torch.nn.functional as F
 9: import math
10: import logging
11:
12:
13: # -----
14: # Logging and sanitization helper
15: # -----
16: logger = logging.getLogger(__name__)
17: if not logger.handlers:
18:
      handler = logging.StreamHandler()
      handler.setFormatter(logging.Formatter("%(asctime)s | %(levelname)s | %(message)s"))
19:
20:
     logger.addHandler(handler)
21: logger.setLevel(logging.INFO)
22:
23:
```

```
24: def _sanitize_normalized_height(value, sample_id=None, default=0.6):
25:
26:
       Ensure normalized height value is finite and in [0,1].
27:
       Returns a float in [0,1].
28:
29:
       Arqs:
30:
           value: torch scalar tensor or float
31:
           sample_id: optional identifier for logging (string or int)
32:
           default: fallback normalized height
33:
34:
       try:
35:
           if isinstance(value, torch.Tensor):
36:
              raw = float(value.item())
37:
           else:
38:
              raw = float(value)
39:
      except Exception:
40:
          raw = float("nan")
41:
       # Build label for logging
42:
43:
       sid = f"[sample={sample_id}]" if sample_id is not None else ""
44:
45:
       # Check finite
46:
       if not math.isfinite(raw):
47:
           logger.warning(f"{sid} Invalid wall height value (not finite): {raw}; using default {default}")
48:
           raw = default
49:
50:
       # Clamp to [0,1]
       if raw < 0.0 or raw > 1.0:
51:
           logger.warning(f"{sid} Wall height normalized {raw} out of [0,1]; clamping.")
52:
53:
           raw = max(0.0, min(1.0, raw))
54:
55:
       return raw
56:
57:
58: def _sanitize_tensor(tensor, default_value=0.0, name="tensor"):
59:
60:
       Sanitize an entire tensor by replacing NaN/Inf values with default.
61:
62:
       Args:
63:
           tensor: Input tensor
64:
           default_value: Value to replace invalid entries with
           name: Name for logging
65:
66:
67:
      Returns:
68:
          Sanitized tensor
69:
70:
       if tensor.numel() == 0:
71:
           return tensor
72:
73:
       # Check for any invalid values
74:
       invalid_mask = ~torch.isfinite(tensor)
       num_invalid = invalid_mask.sum().item()
75:
76:
77:
       if num invalid > 0:
78:
           logger.warning(f"Found {num_invalid} invalid values in {name}, replacing with {default_value}")
           tensor = tensor.clone()
79:
:08
           tensor[invalid_mask] = default_value
81:
82:
       return tensor
83:
84:
85: # -----
86: # Main extrusion module
87: # -----
88: class DifferentiableExtrusion(nn.Module):
89:
       . . .
90:
       Vectorized Differentiable 3D extrusion module
       Converts polygons + attributes to soft 3D occupancy grids
91:
92:
93:
94:
       def __init__(self, voxel_size: int = 64):
95:
           super().__init__()
96:
           self.voxel_size = int(voxel_size)
```

```
97:
             self.register buffer(" coords", None)
 98:
 99:
         def ensure coords(self, device):
              """Initialize or update coordinate grid if needed"""
100:
             if (self._coords is None or
101:
102:
                 self._coords.device != device or
103:
                 self._coords.shape[0] != (self.voxel_size * self.voxel_size)):
104:
105:
                 H = W = self.voxel_size
                 y, x = torch.meshgrid(
106:
107:
                     torch.arange(H, device=device),
108:
                     torch.arange(W, device=device),
109:
                     indexing="ij"
110:
111:
                 coords = torch.stack([x.flatten().float(), y.flatten().float()], dim=1) # [H*W, 2]
112:
                 coords = coords / float(self.voxel_size - 1)
113:
                 self.register_buffer("_coords", coords)
114:
115:
         def polygon_sdf(self, polygon_xy):
116:
117:
             Compute signed distance field for a polygon using vectorized operations.
118:
119:
             device = polygon_xy.device
120:
            self._ensure_coords(device)
121:
             pts = self._coords # [M, 2]
122:
             P = polygon_xy.shape[0]
123:
             if P < 2:
124:
125:
                 return torch.full((pts.shape[0],), 1.0, device=device)
126:
127:
             # Sanitize polygon coordinates
128:
             polygon_xy = _sanitize_tensor(polygon_xy, default_value=0.0, name="polygon_xy")
129:
130:
             v0 = polygon_xy.unsqueeze(1)
             v1 = {\tt torch.roll(polygon\_xy, shifts=-1, dims=0).unsqueeze(1)}
131:
132:
             pts_exp = pts.unsqueeze(0)
133:
134:
             e = v1 - v0
135:
             v = pts_exp - v0
             e_norm_sq = (e**2).sum(dim=2, keepdim=True) + 1e-8
136:
137:
             t = (v * e).sum(dim=2, keepdim=True) / e_norm_sq
138:
             t_{clamped} = t.clamp(0.0, 1.0)
139:
140:
             proj = v0 + t_clamped * e
141:
             diff = pts_exp - proj
142:
             dists = torch.norm(diff, dim=2)
143:
144:
             # Sanitize distances before min operation
             dists = _sanitize_tensor(dists, default_value=1.0, name="distances")
145:
             min_dist_per_point, _ = dists.min(dim=0)
146:
147:
148:
            x_pts = pts[:, 0].unsqueeze(0)
149:
             y_pts = pts[:, 1].unsqueeze(0)
             x0, y0 = v0[..., 0], v0[..., 1]
150:
151:
             x1, y1 = v1[..., 0], v1[..., 1]
152:
             y_{crosses} = ((y0 \le y_{pts}) & (y1 > y_{pts})) | ((y1 \le y_{pts}) & (y0 > y_{pts}))
153:
154:
             inter_x = x0 + (x1 - x0) * ((y_pts - y0) / (y1 - y0 + 1e-8))
155:
             crossings = (inter_x > x_pts) & y_crosses
156:
             crossing_count = crossings.sum(dim=0)
157:
             inside = (crossing_count % 2 == 1)
158:
159:
             sdf = min_dist_per_point.clone()
160:
             sdf[inside] = -sdf[inside]
161:
162:
             # Final sanitization of SDF output
163:
             sdf = _sanitize_tensor(sdf, default_value=1.0, name="sdf")
164:
             return sdf
165:
166:
        def forward(self, polygons, attributes, validity_scores, sample_ids=None):
167:
168:
             Convert polygons to 3D voxel occupancy.
169:
             sample_ids: optional list/array of identifiers (e.g., filenames or dataset indices)
```

```
170:
171:
            device = polygons.device
172:
            B, N, P, \_ = polygons.shape
            D = H = W = self.voxel_size
173:
174:
175:
            # Sanitize input tensors
176:
            polygons = _sanitize_tensor(polygons, default_value=0.0, name="input_polygons")
            attributes = _sanitize_tensor(attributes, default_value=0.6, name="input_attributes")
177:
            validity_scores = _sanitize_tensor(validity_scores, default_value=0.0, name="input_validity_scores"
178:
179:
180:
            voxels = torch.zeros((B, D, H, W), device=device)
181:
182:
            for b in range(B):
                # pick identifier if available
183:
184:
                sid = sample_ids[b] if sample_ids is not None else b
185:
186:
                # Sanitize height with logging
187:
                wall_height_normalized = attributes[b, 0]
                sanitized_norm = _sanitize_normalized_height(
188:
                    wall_height_normalized, sample_id=sid, default=0.6
190:
                )
191:
192:
                wall_height_m = sanitized_norm * 5.0
                height_frac = wall_height_m / 5.0
193:
                height_voxels = int(round(height_frac * D))
194:
195:
                height_voxels = max(1, min(D, height_voxels))
196:
                # Process each polygon for this batch
197:
198:
                validity_mask = validity_scores[b] >= 0.5
199:
                if not validity_mask.any():
200:
                    continue
201:
202:
               combined_mask = torch.zeros((H, W), device=device)
203:
                sharpness = 100.0
204:
205:
                for n in range(N):
206:
                    if not validity_mask[n]:
207:
                        continue
208:
209:
                    polygon = polygons[b, n] # [P, 2]
210:
211:
                    # Filter out zero-padded vertices
212:
                    vertex_mask = (polygon.sum(dim=1) != 0.0)
213:
                    if vertex_mask.sum().item() < 3:</pre>
214:
                        continue
215:
216:
                    valid_polygon = polygon[vertex_mask]
217:
                    # Compute SDF for this polygon
218:
219:
                    sdf = self.polygon_sdf(valid_polygon)
                    mask = torch.sigmoid(-sdf * sharpness)
220:
221:
                    mask_2d = mask.view(H, W)
222:
223:
                    # Sanitize mask before combining
224:
                    \label{eq:mask2d} mask\_2d = \_sanitize\_tensor(mask\_2d, default\_value=0.0, name=f"mask\_2d\_b\{b\}\_n\{n\}")
225:
                    combined_mask = torch.maximum(combined_mask, mask_2d)
226:
227:
                # Create 3D mask by extruding to the computed height
228:
                mask_3d = combined_mask.unsqueeze(0).expand(height_voxels, -1, -1)
229:
230:
                # Sanitize final mask before assignment
231:
                mask_3d = _sanitize_tensor(mask_3d, default_value=0.0, name=f"final_mask_3d_b{b}")
232:
                voxels[b, :height_voxels] = mask_3d
233:
            # Final sanitization of output
234:
235:
            voxels = _sanitize_tensor(voxels, default_value=0.0, name="output_voxels")
236:
            return voxels
237:
238:
239: # -----
240: # Fast extrusion module
241: # -----
242: class DifferentiableExtrusionFast(nn.Module):
```

```
243:
244:
        Optimized version that batches polygon processing.
245:
246:
        def __init__(self, voxel_size: int = 64):
247:
248:
             super().__init__()
249:
             self.voxel_size = int(voxel_size)
250:
             self.register_buffer("_coords", None)
251:
252:
        def ensure coords(self, device):
             if (self._coords is None or
253:
254:
                 self._coords.device != device or
255:
                 self._coords.shape[0] != (self.voxel_size * self.voxel_size)):
256:
257:
                 H = W = self.voxel size
258:
                 y, x = torch.meshgrid(
                     torch.arange(H, device=device),
259:
260:
                     torch.arange(W, device=device),
                     indexing="ij"
261:
262:
                 )
263:
                 coords = torch.stack([x.flatten().float(), y.flatten().float()], dim=1)
264:
                 coords = coords / float(self.voxel_size - 1)
265:
                 self.register_buffer("_coords", coords)
266:
267:
         def batch_polygon_sdf(self, polygons_batch, validity_mask):
268:
             device = polygons_batch.device
269:
             self._ensure_coords(device)
270:
271:
            N, P, _ = polygons_batch.shape
272:
             M = self._coords.shape[0]
273:
             sdfs = torch.full((N, M), 1.0, device=device)
274:
275:
             valid_indices = torch.where(validity_mask)[0]
             if len(valid_indices) == 0:
276:
277:
                return sdfs
278:
279:
             valid_polygons = polygons_batch[valid_indices]
             for i, poly_idx in enumerate(valid_indices):
280:
281:
                 poly = valid_polygons[i]
                 vertex_mask = (poly.sum(dim=1) != 0.0)
282:
283:
                 if vertex_mask.sum().item() >= 3:
284:
                     valid_poly = poly[vertex_mask]
285:
                     sdf = self.polygon_sdf(valid_poly)
286:
                     sdfs[poly_idx] = sdf
287:
288:
             return sdfs
289:
290:
        def polygon_sdf(self, polygon_xy):
291:
            device = polygon_xy.device
292:
             self._ensure_coords(device)
293:
            pts = self._coords
            P = polygon_xy.shape[0]
295:
296:
            if P < 2:
297:
                 return torch.full((pts.shape[0],), 1.0, device=device)
298:
299:
             # Sanitize polygon coordinates
            polygon_xy = _sanitize_tensor(polygon_xy, default_value=0.0, name="polygon_xy_fast")
300:
301:
302:
             v0 = polygon_xy.unsqueeze(1)
303:
             v1 = torch.roll(polygon_xy, shifts=-1, dims=0).unsqueeze(1)
304:
            pts_exp = pts.unsqueeze(0)
305:
306:
             e = v1 - v0
307:
             v = pts_exp - v0
308:
             e_norm_sq = (e^**2).sum(dim=2, keepdim=True) + 1e-8
309:
            t = (v * e).sum(dim=2, keepdim=True) / e_norm_sq
310:
            t_{clamped} = t.clamp(0.0, 1.0)
311:
312:
            proj = v0 + t_clamped * e
313:
            diff = pts_exp - proj
314:
            dists = torch.norm(diff, dim=2)
315:
```

```
316:
            # Sanitize distances before min operation
            dists = _sanitize_tensor(dists, default_value=1.0, name="distances_fast")
317:
318:
            min_dist_per_point, _ = dists.min(dim=0)
319:
320:
            x_pts = pts[:, 0].unsqueeze(0)
321:
            y_pts = pts[:, 1].unsqueeze(0)
322:
            x0, y0 = v0[..., 0], v0[..., 1]
323:
            x1, y1 = v1[..., 0], v1[..., 1]
324:
            y\_crosses = ((y0 \le y\_pts) & (y1 > y\_pts)) | ((y1 \le y\_pts) & (y0 > y\_pts))
325:
            inter_x = x0 + (x1 - x0) * ((y_pts - y0) / (y1 - y0 + 1e-8))
326:
327:
            crossings = (inter_x > x_pts) & y_crosses
328:
            crossing_count = crossings.sum(dim=0)
329:
            inside = (crossing_count % 2 == 1)
330:
331:
            sdf = min_dist_per_point.clone()
332:
            sdf[inside] = -sdf[inside]
333:
            # Final sanitization of SDF output
334:
335:
            sdf = _sanitize_tensor(sdf, default_value=1.0, name="sdf_fast")
336:
            return sdf
337:
338:
        def forward(self, polygons: torch.Tensor, attributes: torch.Tensor, validity_scores: torch.Tensor) -> t
339:
            device = polygons.device
340:
            B, N, P, _ = polygons.shape
            D = H = W = self.voxel_size
341:
342:
343:
            # Sanitize input tensors
344:
            polygons = _sanitize_tensor(polygons, default_value=0.0, name="input_polygons_fast")
345:
            attributes = _sanitize_tensor(attributes, default_value=0.6, name="input_attributes_fast")
346:
            validity_scores = _sanitize_tensor(validity_scores, default_value=0.0, name="input_validity_scores_
347:
            voxels = torch.zeros((B, D, H, W), device=device)
348:
349:
350:
            for b in range(B):
351:
                validity_mask = validity_scores[b] >= 0.5
352:
                if not validity_mask.any():
353:
                    continue
354:
355:
                sdfs = self.batch_polygon_sdf(polygons[b], validity_mask)
356:
357:
                # Sanitize SDFs before sigmoid
358:
                sdfs = _sanitize_tensor(sdfs, default_value=1.0, name=f"batch_sdfs_b{b}")
359:
360:
                sharpness = 100.0
361:
                masks = torch.sigmoid(-sdfs * sharpness)
362:
                masks_2d = masks.view(N, H, W)
363:
                # Sanitize masks
364:
                masks_2d = _sanitize_tensor(masks_2d, default_value=0.0, name=f"masks_2d_b{b}")
365:
366:
367:
                # Sanitize height
368:
                wall_height_normalized = attributes[b, 0]
                sanitized_norm = _sanitize_normalized_height(wall_height_normalized, sample_id=b, default=0.6)
369:
370:
                wall_height_m = sanitized_norm * 5.0
371:
                height_frac = wall_height_m / 5.0
                height_voxels = int(round(height_frac * D))
372:
373:
                height_voxels = max(1, min(D, height_voxels))
374:
375:
                combined_mask = torch.zeros((H, W), device=device)
376:
                for n in range(N):
377:
                    if validity_mask[n]:
378:
                        combined_mask = torch.maximum(combined_mask, masks_2d[n])
379:
380:
                mask_3d = combined_mask.unsqueeze(0).expand(height_voxels, -1, -1)
381:
382:
                # Sanitize final mask before assignment
                383:
                voxels[b, :height_voxels] = mask_3d
384:
385:
            # Final sanitization of output
386:
387:
            voxels = _sanitize_tensor(voxels, default_value=0.0, name="output_voxels_fast")
388:
            return voxels
```

# ■ File: models\heads.py

```
-----
 1: """
 2: Multi-task prediction heads for the Neural-Geometric 3D Model Generator
 3: """
 4:
 5: import torch
 6: import torch.nn as nn
 7: import torch.nn.functional as F
 8:
 9:
10: class SegmentationHead(nn.Module):
11:
        """Semantic segmentation head with multi-scale fusion"""
12:
13:
       def __init__(self, feature_dim=512, num_classes=5, dropout=0.1):
14:
            super().__init__()
15:
16:
            # Multi-scale fusion
17:
           self.fusion = nn.Sequential(
18:
               nn.Conv2d(feature_dim * 4, feature_dim, 3, 1, 1),
19:
                nn.BatchNorm2d(feature_dim),
20:
                nn.ReLU(),
21:
                nn.Dropout2d(dropout),
22:
           )
23:
           # Segmentation decoder
24:
25:
           self.decoder = nn.Sequential(
               nn.Conv2d(feature_dim, feature_dim // 2, 3, 1, 1),
26:
27:
               nn.BatchNorm2d(feature_dim // 2),
28:
               nn.ReLU(),
29:
               nn.Conv2d(feature_dim // 2, feature_dim // 4, 3, 1, 1),
30:
               nn.BatchNorm2d(feature_dim // 4),
31:
               nn.ReLU().
               nn.Conv2d(feature_dim // 4, num_classes, 1),
32:
            )
33:
34:
        def forward(self, features):
35:
36:
            # Fuse multi-scale features
37:
            p1, p2, p3, p4 = features["p1"], features["p2"], features["p3"], features["p4"]
38:
39:
            # Upsample all to p1 resolution
40:
           p2_up = F.interpolate(
41:
               p2, size=p1.shape[-2:], mode="bilinear", align_corners=False
42:
            )
43:
           p3_up = F.interpolate(
44:
               p3, size=p1.shape[-2:], mode="bilinear", align_corners=False
45:
46:
           p4_up = F.interpolate(
47:
               p4, size=p1.shape[-2:], mode="bilinear", align_corners=False
48:
49:
50:
           fused = torch.cat([p1, p2_up, p3_up, p4_up], dim=1)
51:
           fused = self.fusion(fused)
52:
53:
           # Final segmentation
54:
           seq = self.decoder(fused)
           return F.interpolate(seg, scale_factor=4, mode="bilinear", align_corners=False)
55:
56:
58: class AttributeHead(nn.Module):
59:
        """Attribute regression head for geometric parameters"""
60:
        def __init__(self, feature_dim=512, num_attributes=6, dropout=0.2):
61:
62:
            super().__init__()
63:
64:
           self.regressor = nn.Sequential(
65:
               nn.Linear(feature_dim, feature_dim),
66:
               nn.ReLU(),
67:
               nn.Dropout(dropout),
68:
               nn.Linear(feature_dim, feature_dim // 2),
69:
               nn.ReLU(),
70:
               nn.Dropout(dropout),
71:
                nn.Linear(feature_dim // 2, num_attributes),
```

```
72:
                nn.Sigmoid(), # Output in [0,1] range
 73:
 74:
 75:
         def forward(self, global_features):
 76:
            return self.regressor(global_features)
 77:
 78:
 79: class SDFHead(nn.Module):
 :08
         """Signed Distance Field prediction for sharp boundaries"""
 81:
        def __init__(self, feature_dim=512, dropout=0.1):
 83:
            super().__init__()
 85:
            self.sdf_decoder = nn.Sequential(
 86:
                nn.Conv2d(feature_dim, feature_dim // 2, 3, 1, 1),
 87:
                nn.BatchNorm2d(feature_dim // 2),
 88:
                nn.ReLU(),
 89:
                nn.Dropout2d(dropout),
                nn.Conv2d(feature_dim // 2, feature_dim // 4, 3, 1, 1),
 90:
 91:
                nn.BatchNorm2d(feature_dim // 4),
 92:
                nn.ReLU(),
 93:
                nn.Conv2d(feature_dim // 4, 1, 1),
 94:
                nn.Tanh(), # SDF in [-1, 1]
 95:
             )
 96:
       def forward(self, features):
 97:
 98:
            # Use highest resolution features
 99:
            p1 = features["p1"]
           sdf = self.sdf_decoder(p1)
101:
            return F.interpolate(sdf, scale_factor=4, mode="bilinear", align_corners=False)
```

\_\_\_\_\_\_

# ■ File: models\model.py

```
______
 1: """
 2: Advanced loss functions for multi-task training with dynamic weighting
 3: Enhanced with cross-modal consistency, graph constraints, and GradNorm
 4: Modified to support conditional geometric losses via run_full_geometric flag
 5: """
 6: import torch
 7: import torch.nn as nn
 8: import torch.nn.functional as F
 9: from .encoder import MultiScaleEncoder
 10: from .heads import SegmentationHead, AttributeHead, SDFHead
 11: from .dvx import DifferentiableVectorization
12: from .extrusion import DifferentiableExtrusion
13:
14:
 15: class L2Normalize(nn.Module):
       """L2 normalization layer"""
16:
17:
18:
      def __init__(self, dim=1):
19:
           super().__init__()
 20:
            self.dim = dim
 21:
       def forward(self, x):
 22:
 23:
           return F.normalize(x, p=2, dim=self.dim)
 24:
 25:
 26: class LatentEmbeddingHead(nn.Module):
 27:
        """Auxiliary head for cross-modal latent consistency"""
 28:
 29:
        def __init__(self, feature_dim: int, embedding_dim: int = 256):
           super().__init__()
 30:
           self.embedding_dim = embedding_dim
32:
 33:
           # 2D embedding path
 34:
            self.embedding_2d = nn.Sequential(
35:
               nn.AdaptiveAvgPool2d((1, 1)),
 36:
               nn.Flatten(),
 37:
               nn.Linear(feature_dim, embedding_dim * 2),
 38:
               nn.ReLU(),
```

```
39:
                 nn.Dropout(0.1),
 40:
                 nn.Linear(embedding_dim * 2, embedding_dim),
 41:
                 L2Normalize(dim=1), # L2 normalize for cosine similarity
 42:
             )
 43:
 44:
            # 3D embedding path (from voxel features)
 45:
            self.embedding_3d = nn.Sequential(
                nn.AdaptiveAvgPool3d((1, 1, 1)),
 46:
 47:
                 nn.Flatten(),
 48:
                 nn.Linear(feature_dim, embedding_dim * 2),
 49:
                nn.ReLU(),
 50:
                 nn.Dropout(0.1),
                 nn.Linear(embedding_dim * 2, embedding_dim),
 51:
 52:
                 L2Normalize(dim=1),
 53:
             )
 54:
 55:
         def forward(
 56:
            self, features_2d: torch.Tensor, features_3d: torch.Tensor = None
 57:
         ) -> tuple:
 59:
             Generate 2D and 3D embeddings for consistency loss
 60:
 61:
             Args:
                features_2d: [B, C, H, W] - 2D feature maps
 62:
                 features_3d: [B, C, D, H, W] - 3D feature maps (optional)
 63:
 64:
 65:
             Returns:
 66:
               tuple: (embedding_2d, embedding_3d)
 67:
 68:
             # 2D embedding
 69:
             emb_2d = self.embedding_2d(features_2d)
 70:
 71:
             # 3D embedding (if available, otherwise use 2D features reshaped)
             if features_3d is not None:
 72:
                emb_3d = self.embedding_3d(features_3d)
 73:
 74:
             else:
 75:
                 # Create pseudo-3D from 2D features
 76:
                 B, C, H, W = features_2d.shape
 77:
                 pseudo_3d = features_2d.unsqueeze(2).expand(
 78:
                    B, C, 4, H, W
 79:
                 ) # Duplicate across depth
 80:
                 emb_3d = self.embedding_3d(pseudo_3d)
 82:
            return emb_2d, emb_3d
 83:
 84:
 85: class GraphStructureHead(nn.Module):
         """Head for predicting graph structure (room connectivity)"""
 87:
 88:
         def __init__(self, feature_dim: int, max_rooms: int = 10):
 89:
             super().__init__()
 90:
            self.max_rooms = max_rooms
 91:
 92:
             # Room detection branch
 93:
             self.room_detector = nn.Sequential(
                nn.Conv2d(feature_dim, feature_dim // 2, 3, padding=1),
 94:
 95:
 96:
                 nn.Conv2d(feature_dim // 2, max_rooms, 3, padding=1),
 97:
                 nn.Sigmoid(), # Room probability maps
 98:
 99:
100:
             # Room feature extractor
101:
            self.room_features = nn.Sequential(
102:
                 nn.AdaptiveAvgPool2d((8, 8)), # Pool to fixed size
                 nn.Flatten(),
103:
104:
                nn.Linear(feature_dim * 64, 256),
105:
                nn.ReLU(),
                 nn.Linear(256, 128), # Room feature vectors
106:
107:
             )
108:
109:
             # Adjacency predictor
110:
            self.adjacency_net = nn.Sequential(
111:
                 nn.Linear(128 * 2, 64), # Pairwise room features
```

```
112:
                nn.ReLU().
113:
                nn.Linear(64, 32),
114:
                nn.ReLU(),
                 nn.Linear(32, 1),
115:
                 nn.Sigmoid(), # Adjacency probability
116:
117:
118:
119:
        def forward(self, features: torch.Tensor) -> dict:
120:
            Predict room graph structure
121:
122:
123:
             Arqs:
                features: [B, C, H, W] - Feature maps
124:
125:
126:
            Returns:
127:
                dict with 'room_maps', 'room_features', 'adjacency_matrix'
128:
129:
            B = features.shape[0]
130:
131:
            # Detect room probability maps
132:
            room_maps = self.room_detector(features) # [B, max_rooms, H, W]
133:
134:
             # Extract room features
135:
            room_feats = self.room_features(features) # [B, 128]
136:
137:
             # Create adjacency matrix for all room pairs
138:
            adjacency_matrices = []
139:
140:
            for b in range(B):
141:
                # Get room features for this batch item
142:
                 feat_b = room_feats[b : b + 1] # [1, 128]
143:
144:
                 # Create pairwise combinations
145:
                adj_matrix = torch.zeros(
                     (self.max_rooms, self.max_rooms), device=features.device
146:
147:
148:
149:
                for i in range(self.max_rooms):
150:
                     for j in range(i + 1, self.max_rooms):
151:
                         # Concatenate features for room pair
152:
                         pair_feat = torch.cat([feat_b, feat_b], dim=1) # [1, 256]
153:
154:
                         # Predict adjacency
155:
                         adj_prob = self.adjacency_net(pair_feat) # [1, 1]
156:
157:
                         # Fill symmetric matrix
158:
                         adj_matrix[i, j] = adj_prob.squeeze()
159:
                         adj_matrix[j, i] = adj_prob.squeeze()
160:
161:
                 adjacency_matrices.append(adj_matrix)
162:
163:
            return {
164:
                 "room_maps": room_maps,
165:
                 "room_features": room_feats,
166:
                 "adjacency_matrices": torch.stack(adjacency_matrices),
             }
167:
168:
169:
170: class NeuralGeometric3DGenerator(nn.Module):
171:
172:
        Enhanced neural-geometric system with auxiliary heads for novel training strategies:
173:
        - Cross-modal latent consistency
174:
         - Graph structure prediction
175:
        - Multi-view embeddings for dynamic curriculum
        - Conditional geometric computation via run_full_geometric flag
176:
177:
178:
179:
        def __init__(
180:
             self,
181:
            input channels=3,
182:
           num_classes=5,
183:
            feature_dim=512,
184:
            num_attributes=6,
```

```
185:
            voxel size=64.
186:
            max_polygons=20,
187:
            max_points=50,
188:
            use_latent_consistency=True,
189:
            use_graph_constraints=True,
190:
            latent embedding dim=256,
       ):
191:
192:
            super().__init__()
193:
194:
             # Store configuration
195:
            self.use_latent_consistency = use_latent_consistency
196:
             self.use_graph_constraints = use_graph_constraints
197:
            self.feature_dim = feature_dim
198:
199:
            # Core components
200:
             self.encoder = MultiScaleEncoder(input_channels, feature_dim)
201:
             self.seg_head = SegmentationHead(feature_dim, num_classes)
202:
             self.attr_head = AttributeHead(feature_dim, num_attributes)
             self.sdf_head = SDFHead(feature_dim)
203:
204:
             self.dvx = DifferentiableVectorization(max_polygons, max_points, feature_dim)
205:
             self.extrusion = DifferentiableExtrusion(voxel_size)
206:
207:
             # NEW: Auxiliary heads for novel training strategies
208:
             if use_latent_consistency:
209:
                 self.latent_head = LatentEmbeddingHead(feature_dim, latent_embedding_dim)
210:
211:
             if use_graph_constraints:
212:
                 self.graph_head = GraphStructureHead(feature_dim)
213:
214:
             # Enhanced feature processing for multi-stage training
215:
            self.feature_enhancer = nn.Sequential(
216:
                 nn.Conv2d(feature_dim, feature_dim, 3, padding=1),
217:
                nn.GroupNorm(32, feature_dim),
218:
                nn.ReLU(),
                nn.Conv2d(feature_dim, feature_dim, 3, padding=1),
219:
220:
                 nn.GroupNorm(32, feature_dim),
221:
             )
222:
223:
             # lazy-created 3d voxel processor will be attached on first use
224:
             self._voxel_processor = None
225:
226:
         def _select_spatial_feature(self, features):
227:
228:
             Given encoder output (dict or tensor), select a spatial 4-D feature map
229:
             Prefer high-resolution feature maps (p1) and avoid selecting 'global' vector.
230:
231:
             # If encoder returned a tensor already, make sure it's 4D
232:
             if not isinstance(features, dict):
                if features.dim() == 4:
233:
234:
                     return features
235:
                 else:
236:
                     raise ValueError(
237:
                         f"Encoder returned a tensor with shape {tuple(features.shape)}; "
238:
                         "expected a 4D feature map [B, C, H, W]."
239:
240:
             # Encoder returned dict: prefer pl,p2,p3,p4,high_res,out,main but NEVER 'global'
241:
242:
             preferred_keys = ["p1", "p2", "p3", "p4", "high_res", "out", "main"]
243:
             for k in preferred_keys:
244:
                 if k in features:
245:
                     candidate = features[k]
246:
                     if isinstance(candidate, torch.Tensor) and candidate.dim() == 4:
247:
                         return candidate
248:
249:
             # As a last resort, scan dict values for the first 4D tensor that isn't 'global'
250:
             for k, v in features.items():
251:
                 if k == "global":
252:
                     continue
253:
                 if isinstance(v, torch.Tensor) and v.dim() == 4:
254:
                     return v
255:
             # If nothing found, raise informative error rather than silently picking wrong shape
256:
257:
             raise RuntimeError(
```

```
258:
                 "No spatial 4D feature map found in encoder output. Encoder returned keys: "
259:
                 f"{list(features.keys())}. Ensure encoder provides at least one [B,C,H,W] tensor "
260:
                 "under keys like 'p1','p2','p3','p4','out', or 'high_res'."
261:
262:
263:
         def forward(self, image, run_full_geometric=True, return_aux=True):
264:
             Enhanced forward pass with auxiliary outputs and conditional geometric computation
265:
266:
267:
             Arqs:
268:
                 image: [B, C, H, W] input images
269:
                 run_full_geometric: Whether to run heavy DVX and extrusion computations
270:
                 return_aux: Whether to compute auxiliary outputs
271:
272:
             Returns:
273:
                 dict with predictions, conditionally including geometric outputs
274:
275:
             # Multi-scale feature extraction
276:
             features = self.encoder(image)
277:
278:
             # Enhance features
279:
             spatial_feat = self._select_spatial_feature(features)
280:
             enhanced_features = self.feature_enhancer(spatial_feat)
281:
             # keep structured features dict for heads that expect multi-scale inputs
282:
283:
             if isinstance(features, dict):
284:
                 features["enhanced"] = enhanced_features
285:
                 main_features = enhanced_features
286:
             else:
287:
                 features = {"main": enhanced_features, "enhanced": enhanced_features}
288:
                 main_features = enhanced_features
289:
290:
             # Core predictions (always computed - these are fast)
291:
             segmentation = self.seg_head(features)
292:
             attributes = self.attr_head(
293:
                 features.get("global")
                 if isinstance(features, dict) and "global" in features
294:
295:
                 else main_features.mean(dim=[2, 3])
296:
             )
297:
             sdf = self.sdf_head(features)
298:
299:
             # Base outputs
300:
             outputs = {
301:
                 "segmentation": segmentation,
302:
                 "attributes": attributes,
303:
                 "sdf": sdf,
304:
                 "features": features,
305:
             }
306:
307:
             # Conditional geometric computation (heavy operations)
308:
             if run full geometric:
309:
                 # DVX polygon fitting
310:
                 dvx_output = self.dvx(features, segmentation)
311:
                 polygons = dvx_output.get("polygons", None)
312:
                 validity = dvx_output.get("validity", None)
313:
                 # 3D extrusion (defensive: ensure inputs exist)
314:
315:
                 try:
316:
                     voxels_pred = self.extrusion(polygons, attributes, validity)
317:
                 except Exception as e:
318:
                     # Log or print a helpful message for debugging; avoid crashing training
319:
                     # (Replace print with logger if you have one)
320:
                     print(f"[Warning] extrusion failed: {e}")
321:
                     voxels_pred = None
322:
323:
                 # Add geometric outputs
324:
                 outputs.update({
                     "polygons": polygons,
325:
326:
                      "polygon_validity": validity,
327:
                     "voxels_pred": voxels_pred,
328:
                 })
329:
330:
                 # NEW: Auxiliary outputs for novel training strategies (only when geometric is enabled)
```

```
331:
                 if return aux:
332:
                     # Cross-modal consistency embeddings
333:
                     if self.use_latent_consistency:
334:
                         if voxels_pred is not None:
                             voxel_features = self._create_3d_features_from_voxels(voxels_pred)
335:
336:
                             latent_2d, latent_3d = self.latent_head(main_features, voxel_features)
                         else:
337:
                             # Fall back to pseudo-3D built from 2D features if voxels not available
338:
339:
                             latent_2d, latent_3d = self.latent_head(main_features, None)
340:
                         outputs["latent_2d_embedding"] = latent_2d
                         outputs["latent_3d_embedding"] = latent_3d
341:
342:
             else:
343:
                 # Geometric path explicitly skipped for this stage
344:
                 outputs.update({
                     "polygons": None,
345:
                     "polygon_validity": None,
346:
347:
                     "voxels_pred": None,
348:
                 })
349:
350:
                 # Still compute some auxiliary outputs that don't depend on geometry
351:
                 if return_aux and self.use_latent_consistency:
352:
                     # Use pseudo-3D features for 2D-only consistency inside latent head
353:
                     latent_2d, latent_3d = self.latent_head(main_features, None)
                     outputs["latent_2d_embedding"] = latent_2d
354:
355:
                     outputs["latent_3d_embedding"] = latent_3d
356:
357:
             # Graph structure predictions (independent of geometric computation)
358:
             if return_aux and self.use_graph_constraints:
359:
                 graph_output = self.graph_head(main_features)
360:
                 outputs.update(graph_output)
361:
362:
             return outputs
363:
364:
         def get_latent_embeddings(self, image):
365:
366:
             Convenience method to get just the latent embeddings
367:
             Used by trainer for consistency loss
368:
369:
             if not self.use_latent_consistency:
370:
                return None, None
371:
372:
             with torch.no_grad():
373:
                 features = self.encoder(image)
374:
                 spatial_feat = self._select_spatial_feature(features)
375:
                 main_features = self.feature_enhancer(spatial_feat)
376:
377:
                 # Quick forward to get segmentation/attributes
378:
                 segmentation = self.seg_head(features)
379:
                 attributes = self.attr_head(
380:
                     features.get("global")
                     if isinstance(features, dict) and "global" in features
381:
382:
                     else main_features.mean(dim=[2, 3])
383:
384:
385:
                 # Attempt DVX + extrusion, but be defensive (may be expensive)
386:
                 dvx_output = self.dvx(features, segmentation)
387:
                 polygons = dvx_output.get("polygons", None)
388:
                 validity = dvx_output.get("validity", None)
389:
390:
391:
                     voxels_pred = self.extrusion(polygons, attributes, validity)
392:
                 except Exception as e:
393:
                     print(f"[Warning] get_latent_embeddings: extrusion failed: {e}")
394:
                     voxels_pred = None
395:
396:
                 # If voxels not available, latent_head will fall back to pseudo-3D
397:
                 if voxels_pred is not None:
398:
                     voxel_features = self._create_3d_features_from_voxels(voxels_pred)
399:
                 else:
400:
                     voxel features = None
401:
402:
             return self.latent head(main features, voxel features)
403:
```

```
404:
         def _create_3d_features_from_voxels(self, voxels):
405:
406:
             Create 3D feature representation from voxel predictions
407:
408:
             Args:
409:
                voxels: [B, D, H, W] voxel predictions
410:
411:
             Returns:
             [B, C, D, H, W] 3D features
412:
413:
             # Defensive check
414:
415:
             if voxels is None:
416:
                 raise ValueError(
417:
                     "Received voxels=None in _create_3d_features_from_voxels(). "
                     "This indicates that the geometric pipeline was skipped or extrusion failed. "
418:
419:
                     "Call this method only when voxels are available, or use latent_head(..., None) to "
420:
                     "compute pseudo-3D features from 2D."
421:
                 )
422:
423:
             # Ensure expected shape
424:
             if voxels.dim() != 4:
425:
                 raise ValueError(f"voxels must have shape [B,D,H,W], got {tuple(voxels.shape)}")
426:
427:
             B, D, H, W = voxels.shape
428:
429:
             # Expand voxels to have feature channels
430:
             # Simple approach: repeat voxel values across feature dimension
431:
             rep_ch = max(1, self.feature_dim // 4)
432:
             voxel_features = voxels.unsqueeze(1).expand(B, rep_ch, D, H, W).contiguous()
433:
             # Add some learned 3D processing
434:
435:
             if self._voxel_processor is None:
436:
                 # Build with correct device
437:
                 device = voxels.device
                 self._voxel_processor = nn.Sequential(
438:
439:
                     nn.Conv3d(rep_ch, max(rep_ch, self.feature_dim // 2), 3, padding=1),
440:
441:
                     nn.Conv3d(max(rep_ch, self.feature_dim // 2), self.feature_dim, 3, padding=1),
442:
                 ).to(device)
443:
444:
             return self._voxel_processor(voxel_features)
445:
446:
         def get_stage_parameters(self, stage: int):
447:
448:
             Get parameters for specific training stage
449:
             Useful for stage-specific optimization
450:
451:
            if stage == 1:
                 # Stage 1: 2D components only
452:
453:
                 params = []
                 params.extend(list(self.encoder.parameters()))
454:
455:
                params.extend(list(self.seg_head.parameters()))
456:
                params.extend(list(self.attr_head.parameters()))
457:
                params.extend(list(self.sdf_head.parameters()))
458:
                 params.extend(list(self.feature_enhancer.parameters()))
459:
460:
                 if self.use_latent_consistency:
461:
                     params.extend(list(self.latent_head.parameters()))
462:
463:
             elif stage == 2:
464:
                 # Stage 2: DVX components
465:
                 params = list(self.dvx.parameters())
466:
467:
             else: # stage == 3
468:
                 # Stage 3: All parameters
469:
                 params = list(self.parameters())
470:
471:
             return params
472:
473:
        def freeze_stage_parameters(self, stages_to_freeze: list):
474:
475:
             Freeze parameters for specific stages
476:
```

```
477:
            Args:
478:
               stages_to_freeze: List of stage numbers to freeze
479:
480:
             for stage in stages_to_freeze:
481:
                 stage_params = self.get_stage_parameters(stage)
482:
                 for param in stage_params:
483:
                     param.requires_grad = False
484:
485:
         def unfreeze_stage_parameters(self, stages_to_unfreeze: list):
486:
487:
            Unfreeze parameters for specific stages
488:
489:
            Arqs:
490:
                stages_to_unfreeze: List of stage numbers to unfreeze
491:
492:
             for stage in stages_to_unfreeze:
493:
                 stage_params = self.get_stage_parameters(stage)
494:
                 for param in stage_params:
495:
                     param.requires_grad = True
496:
497:
        def get_curriculum_metrics(self):
498:
499:
             Get metrics useful for curriculum learning decisions
500:
501:
            metrics = {}
502:
503:
             # Parameter counts per stage
            for stage in [1, 2, 3]:
504:
505:
                 stage_params = self.get_stage_parameters(stage)
506:
                 metrics[f"stage_{stage}_params"] = sum(p.numel() for p in stage_params)
507:
508:
            # Feature dimensions
            metrics["feature_dim"] = self.feature_dim
509:
            metrics["has_latent_consistency"] = self.use_latent_consistency
510:
511:
            metrics["has_graph_constraints"] = self.use_graph_constraints
512:
513:
            return metrics
```

\_\_\_\_\_

## **■** File: param.py

```
1: import json
 2: import numpy as np
 3: from pathlib import Path
 4:
 5: # Adjust this to your dataset path
 6: data_root = Path("./data/floorplans")
 7:
 8: # Expected attributes with their default values
 9: expected_keys = {
10: "wall_height": 2.6,
       "wall_thickness": 0.15,
11:
12:
       "window_base_height": 0.7,
13:
       "window_height": 0.95,
       "door_height": 2.6,
14:
15:
       "pixel_scale": 0.01
16: }
17:
18: def is_valid_number(value):
19: if value is None:
20:
           return False
21:
      if isinstance(value, (int, float)):
22:
           return np.isfinite(value)
23:
       return False
24:
25: def check_params_file(params_file):
26:
       invalid_entries = []
27:
        try:
           with open(params_file, "r") as f:
28:
29:
               params = json.load(f)
       except Exception as e:
30:
31:
           invalid_entries.append(f"Could not load JSON: {e}")
```

```
32:
           return invalid entries
33:
34:
       for key in expected_keys.keys():
35:
           val = params.get(key)
36:
            if val is None:
37:
               invalid_entries.append(f"missing '{key}'")
38:
            elif not is_valid_number(val):
39:
               invalid_entries.append(f"{key}={val} (invalid)")
40:
       return invalid_entries
41:
42: def check_split(split="train"):
       split_dir = data_root / split
43:
        total_files = 0
44:
       good_files = 0
45:
46:
       bad files = 0
47:
48:
       print(f"\nChecking split: {split}")
49:
       if not split_dir.exists():
50:
           print(f"Warning: {split_dir} does not exist")
51:
           return
52:
53:
        # Recursively find all params.json files
54:
        for params_file in split_dir.rglob("params.json"):
55:
            total files += 1
56:
           invalid_entries = check_params_file(params_file)
57:
58:
           if invalid_entries:
59:
                print(f"[BAD] {params_file}")
60:
                for entry in invalid_entries:
61:
                   print(f"
                              - {entry}")
62:
                bad_files += 1
63:
           else:
64:
                good_files += 1
65:
      print(f"\nSummary for split: {split}")
66:
67:
       print(f"Total files checked: {total_files}")
       print(f"Good files: {good_files}")
68:
       print(f"Bad files: {bad_files}")
69:
70:
71: def main():
      for split in ["train", "val", "test"]:
72:
73:
           check_split(split)
74:
75: if __name__ == "__main__":
76:
      main()
```

## ■ File: setup.py

```
1: """
 2: Setup script for the Neural-Geometric 3D Model Generator project
 3: """
 4:
 5: from pathlib import Path
 6: import os
 7:
 8: def create_project_structure():
 9:
       """Create the complete project directory structure"""
10:
11:
       # Define directory structure
12:
       directories = [
13:
          "models",
14:
           "training",
           "inference",
15:
16:
           "evaluation",
17:
           "utils",
18:
           "data/floorplans/train",
           "data/floorplans/val",
19:
20:
           "data/floorplans/test",
21:
           "data/test_images",
22:
           "checkpoints",
23:
           "outputs",
```

```
24:
            "demo_outputs",
25:
            "evaluation_results",
26:
            "logs"
27:
       ]
28:
29:
       # Create directories
30:
      for directory in directories:
31:
          Path(directory).mkdir(parents=True, exist_ok=True)
32:
           print(f"Created directory: {directory}")
33:
34:
      # Create __init__.py files
35:
      init_files = [
36:
            "models/__init__.py",
37:
            "training/__init__.py",
           "inference/__init__.py",
38:
39:
           "evaluation/__init__.py",
           "utils/__init__.py"
40:
      ]
41:
42:
      init_content = {
        "models/__init__.py": '''""
44:
45: Model components for Neural-Geometric 3D Model Generator
46: """
47:
48: from .encoder import MultiScaleEncoder, ResidualBlock
49: from .heads import SegmentationHead, AttributeHead, SDFHead
50: from .dvx import DifferentiableVectorization
51: from .extrusion import DifferentiableExtrusion
52: from .model import NeuralGeometric3DGenerator
53:
54: __all__ = [
55:
       'MultiScaleEncoder',
       'ResidualBlock',
56:
57:
       'SegmentationHead',
58:
       'AttributeHead',
59:
       'SDFHead',
       'DifferentiableVectorization',
60:
    'DifferentiableExtrusion',
'NeuralGeometric3DGenerator'
61:
62:
63: ]''',
64:
           "training/__init__.py": '''""
65:
66: Training components for Neural-Geometric 3D Model Generator
67: """
69: from .losses import ResearchGradeLoss
70: from .trainer import MultiStageTrainer
71:
72: __all__ = [
73:
       'ResearchGradeLoss',
74:
       'MultiStageTrainer'
75: ]''',
76:
           "inference/__init__.py": '''""
77:
78: Inference components for Neural-Geometric 3D Model Generator
79: """
:08
81: from .engine import ResearchInferenceEngine
82:
83: __all__ = [
84: 'ResearchInferenceEngine'
85: ]''',
86:
           "evaluation/__init__.py": '''"""
87:
88: Evaluation components for Neural-Geometric 3D Model Generator
90:
91: from .metrics import ModelEvaluator, evaluate_model, compute_iou, compute_3d_iou
92:
93: __all__ = [
94: 'ModelEvaluator',
95:
       'evaluate_model',
96:
       'compute_iou',
```

```
97:
        'compute_3d_iou'
 98: ]''',
 99:
100:
             "utils/__init__.py": '''""
101: Utility functions for Neural-Geometric 3D Model Generator
103:
104: from .visualization import (
      plot_training_history,
105:
106:
        visualize_predictions,
       visualize_polygons,
108:
       save_model_outputs,
       create_comparison_grid,
109:
110:
        create_3d_visualization
111: )
112:
113: __all__ = [
114:
      'plot_training_history',
        'visualize_predictions',
115:
       'visualize_polygons',
        'save_model_outputs',
117:
118:
         'create_comparison_grid',
119:
        'create_3d_visualization'
120: ]'''
121:
       }
122:
123:
       for file_path, content in init_content.items():
124:
           with open(file_path, 'w') as f:
                f.write(content)
126:
           print(f"Created: {file_path}")
127:
128:
129: def create_sample_config():
        """Create a sample configuration file for easy customization"""
130:
131:
       sample_config = '''""
132:
133: Sample configuration for quick customization
134: Copy this to config_custom.py and modify as needed
135: """
136:
137: from config import *
138:
139: # Custom configuration example
140: CUSTOM_DATA_CONFIG = DataConfig(
141:
     data_dir="./my_data/floorplans",
142:
        batch_size=16, # Larger batch if you have more GPU memory
        num_workers=8, # More workers if you have more CPU cores
143:
144: )
145:
146: CUSTOM_TRAINING_CONFIG = TrainingConfig(
147: stagel_epochs=30, # More epochs for better 2D learning
       stage2_epochs=20, # More DVX training
149:
       stage3_epochs=50, # Longer end-to-end training
150:
        stage1_lr=2e-4,
                           # Higher learning rate
151: )
152:
153: CUSTOM_MODEL_CONFIG = ModelConfig(
154: feature_dim=768, # Larger model
        voxel_size=128,  # Higher resolution 3D
max_polygons=30,  # More polygons
155:
156:
157: )
158: '''
159:
160:
        with open("config_custom_example.py", "w") as f:
          f.write(sample_config)
161:
        print("Created: config_custom_example.py")
163:
164:
165: def create_gitignore():
         """Create .gitignore file"""
166:
167:
       gitignore_content = '''# Python
168:
169: __pycache__/
```

```
170: *.py[cod]
171: *$py.class
172: *.so
173: .Python
174: build/
175: develop-eggs/
176: dist/
177: downloads/
178: eggs/
179: .eggs/
180: lib/
181: lib64/
182: parts/
183: sdist/
184: var/
185: wheels/
186: *.egg-info/
187: .installed.cfg
188: *.egg
189:
190: # PyTorch
191: *.pth
192: *.pt
193:
194: # Data
195: data/
196: datasets/
197: *.npz
198: *.obj
199: *.off
200: *.ply
201:
202: # Outputs
203: outputs/
204: results/
205: checkpoints/
206: logs/
207: demo_outputs/
208: evaluation_results/
209: training_progress/
210:
211: # IDE
212: .vscode/
213: .idea/
214: *.swp
215: *.swo
216:
217: # OS
218: .DS_Store
219: Thumbs.db
220:
221: # Jupyter
222: .ipynb_checkpoints
223:
224: # Environment
225: .env
226: .venv
227: env/
228: venv/
229:
230: # Images and videos
231: *.png
232: *.jpg
233: *.jpeg
234: *.gif
235: *.mp4
236: *.avi
237:
238: # Except sample images
239: !sample_images/
240: !docs/images/
241: '''
242:
```

```
with open(".gitignore", "w") as f:
243:
            f.write(gitignore_content)
244:
245:
        print("Created: .gitignore")
246:
247:
248: def main():
249:
        """Main setup function"""
250:
       print("Setting up Neural-Geometric 3D Model Generator project...")
        print("=" * 60)
251:
252:
253:
        # Create directory structure
254:
        create_project_structure()
255:
        print()
256:
257:
        # Create sample config
258:
        create_sample_config()
       print()
259:
260:
261:
        # Create gitignore
262:
        create_gitignore()
263:
       print()
264:
265:
        print("Project setup completed!")
        print("=" * 60)
266:
        print("IMPORTANT ACCURACY EXPECTATIONS:")
267:
        print("- 90%+ accuracy in 2D-to-3D generation is extremely challenging")
268:
269:
        print("- Actual accuracy depends heavily on:")
        print(" * Dataset quality and size (need 10K+ samples)")
270:
        print(" * Ground truth accuracy")
271:
272:
       print(" * Problem complexity (simple vs complex floorplans)")
       print(" * Evaluation metrics used")
273:
274:
        print("- Realistic expectations:")
275:
        print(" * Segmentation: 75-85% mIoU with good data")
        print(" * 3D reconstruction: 60-75% IoU for architectural scenes")
276:
       print(" * Polygon fitting: 70-80% accuracy")
277:
278:
        print("=" * 60)
        print("\nNext steps:")
279:
280:
        print("1. Install dependencies: pip install -r requirements.txt")
281:
       print("2. Prepare high-quality dataset (critical for accuracy)")
       print("3. Run demo: python demo.py")
282:
283:
        print("4. Start training: python train.py")
284:
286: if __name__ == "__main__":
287:
       main()
```

## ■ File: train.py

```
______
 2: Enhanced training script for the Neural-Geometric 3D Model Generator
 3: Implements novel training strategies: dynamic curriculum, adaptive weighting, cross-modal consistency
 4: ""'
 5:
 6: import argparse
 7: import torch
 8: from torch.utils.data import DataLoader
 9: from pathlib import Path
10: import json
11: import matplotlib.pyplot as plt
12: import numpy as np
13:
 14: from dataset import AdvancedFloorPlanDataset
15: from models.model import NeuralGeometric3DGenerator
16: from training.trainer import AdaptiveMultiStageTrainer, MultiStageTrainer
17: from utils.visualization import plot_training_history, plot_curriculum_analysis
 18: from config import (
19:
       DEFAULT_DATA_CONFIG,
20:
       DEFAULT_MODEL_CONFIG,
 21:
      DEFAULT_TRAINING_CONFIG,
 22:
       DEFAULT_LOSS_CONFIG,
 23:
       TrainingConfig,
```

```
24:
        CurriculumConfig
25: )
26:
27:
28: def create_enhanced_config(args):
29:
        """Create enhanced training configuration with novel strategies"""
30:
        config = TrainingConfig()
31:
32:
        # Basic settings
        config.device = args.device or ("cuda" if torch.cuda.is_available() else "cpu")
33:
34:
35:
        # Dynamic curriculum settings
36:
        if args.dynamic_curriculum:
37:
            config.curriculum = CurriculumConfig()
38:
           config.curriculum.use_dynamic_curriculum = True
39:
            config.curriculum.stage_switch_patience = args.patience
40:
           config.curriculum.min_improvement_threshold = args.min_improvement
41:
            # Adjust epoch limits for dynamic training
42:
43:
            config.max_stagel_epochs = args.max_stagel_epochs
44:
            config.max_stage2_epochs = args.max_stage2_epochs
           config.max_stage3_epochs = args.max_stage3_epochs
45:
46:
47:
           print("Dynamic curriculum learning enabled")
           print(f" Stage switch patience: {config.curriculum.stage_switch_patience}")
48:
           print(f" Min improvement threshold: {config.curriculum.min_improvement_threshold}")
49:
50:
        else:
51:
            # Disable dynamic curriculum for traditional training
52:
            config.curriculum.use_dynamic_curriculum = False
53:
            print("Using traditional fixed-epoch training")
54:
55:
        # GradNorm dynamic weighting
56:
        if args.gradnorm:
57:
            config.curriculum.use_gradnorm = True
            config.curriculum.gradnorm_alpha = args.gradnorm_alpha
58:
59:
            config.curriculum.gradnorm_update_freq = args.gradnorm_freq
60:
            print(f"GradNorm dynamic weighting enabled (alpha={args.gradnorm_alpha})")
61:
62:
        # Topology-aware scheduling
       if args.topology_schedule != "static":
63:
64:
            config.curriculum.topology_schedule = args.topology_schedule
65:
            config.curriculum.topology_start_weight = args.topology_start
           config.curriculum.topology_end_weight = args.topology_end
67:
           print(f"Topology-aware scheduling: {args.topology_schedule}")
68:
           print(f" Weights: {args.topology_start} -> {args.topology_end}")
69:
70:
        return config
71:
72:
73: def create_enhanced_model(args):
74:
        """Create enhanced model with auxiliary heads"""
75:
       model = NeuralGeometric3DGenerator(
76:
            input_channels=args.input_channels,
77:
            num classes=args.num classes,
78:
            feature_dim=args.feature_dim,
79:
           num_attributes=args.num_attributes,
:08
           voxel_size=args.voxel_size,
81:
           max_polygons=args.max_polygons,
82:
           max_points=args.max_points,
83:
           use_latent_consistency=args.latent_consistency,
84:
           use graph constraints=args.graph constraints.
85:
           latent_embedding_dim=args.embedding_dim
86:
       )
87:
       print(f"Enhanced model created:")
88:
89:
       print(f" Feature dim: {args.feature_dim}")
       print(f" Latent consistency: {args.latent_consistency}")
90:
91:
       print(f" Graph constraints: {args.graph_constraints}")
92:
93:
        # Print parameter counts
94:
        total_params = sum(p.numel() for p in model.parameters())
95:
        trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
96:
        print(f" Total parameters: {total_params:,}")
```

```
97:
         print(f" Trainable parameters: {trainable_params:,}")
 98:
 99:
         return model
100:
101:
102: def visualize_training_results(history, output_dir):
         """Create comprehensive training visualizations"""
103:
         output_dir = Path(output_dir)
104:
         output_dir.mkdir(exist_ok=True)
105:
106:
107:
         # Traditional loss curves
108:
         plot_training_history(history, save_path=str(output_dir / "training_history.png"))
109:
110:
         # Novel curriculum analysis plots
111:
         if "stage_transitions" in history and history["stage_transitions"]:
112:
             plot_curriculum_analysis(history, save_path=str(output_dir / "curriculum_analysis.png"))
113:
114:
         # Dynamic weight evolution
          \  \, \text{if "dynamic\_weights" in history and history["dynamic\_weights"]:} \\
115:
116:
             plt.figure(figsize=(12, 8))
117:
118:
             # Extract weight evolution data
119:
             epochs = [entry["epoch"] for entry in history["dynamic_weights"]]
120:
             weight_names = list(history["dynamic_weights"][0]["weights"].keys())
121:
122:
             for weight_name in weight_names:
123:
                 weights = [entry["weights"].get(weight_name, 0) for entry in history["dynamic_weights"]]
                 if any(w > 0.001 \text{ for } w \text{ in weights}): # Only plot significant weights
124:
125:
                     plt.plot(epochs, weights, label=weight_name, linewidth=2)
126:
127:
             plt.xlabel("Global Epoch")
128:
             plt.ylabel("Loss Weight")
129:
             plt.title("Dynamic Loss Weight Evolution")
130:
            plt.legend()
             plt.grid(True, alpha=0.3)
131:
132:
             plt.tight layout()
122:
             plt.savefig(output_dir / "weight_evolution.png", dpi=300)
134:
             plt.close()
135:
136:
         # Component loss breakdown
137:
         fig, axes = plt.subplots(1, 3, figsize=(18, 5))
138:
         stage_names = ["stage1", "stage2", "stage3"]
139:
140:
         for idx, stage_name in enumerate(stage_names):
141:
             if stage_name in history and "component_losses" in history[stage_name]:
142:
                 component_data = history[stage_name]["component_losses"]
143:
                 if component data:
144:
                      # Get component names from first entry
145:
                     component_names = list(component_data[0].keys())
146:
147:
                     for comp_name in component_names:
148:
                         if comp_name in ['seg', 'dice', 'polygon', 'voxel', 'topology',
149:
                                         'latent_consistency', 'graph']:
150:
                              values = [entry.get(comp_name, 0) for entry in component_data]
151:
                              if any(v > 0.001 \text{ for } v \text{ in values}): # Only plot significant losses
                                  axes[idx].plot(values, label=comp_name, linewidth=2)
152:
153:
154:
                     axes[idx].set_title(f"{stage_name.upper()} Component Losses")
155:
                     axes[idx].set_xlabel("Epoch")
                     axes[idx].set_ylabel("Loss Value")
156:
157:
                     axes[idx].legend()
158:
                     axes[idx].grid(True, alpha=0.3)
159:
160:
         plt.tight_layout()
         plt.savefig(output_dir / "component_losses.png", dpi=300)
161:
162:
         plt.close()
163:
164:
         print(f"Training visualizations saved to {output_dir}")
165:
166:
167: def save_training_summary(history, config, output_dir):
168:
         """Save comprehensive training summary"""
169:
         output_dir = Path(output_dir)
```

```
170:
171:
         summary = {
172:
             "training_config": {
                 "dynamic_curriculum": config.curriculum.use_dynamic_curriculum,
173:
174:
                 "gradnorm_enabled": config.curriculum.use_gradnorm,
                 "topology_schedule": config.curriculum.topology_schedule,
175:
176:
                 "max_epochs": [config.max_stage1_epochs, config.max_stage2_epochs, config.max_stage3_epochs]
177:
             },
178:
             "training results": {},
             "novel_strategies_summary": {}
179:
180:
181:
182:
         # Training results
183:
         for stage_name, data in history.items():
184:
             if isinstance(data, dict) and "val loss" in data and data["val loss"]:
185:
                 summary["training_results"][stage_name] = {
                     "final_val_loss": data["val_loss"][-1],
186:
187:
                      "best_val_loss": min(data["val_loss"]),
                      "epochs_trained": len(data["val_loss"])
188:
189:
                 }
190:
191:
         # Novel strategies summary
192:
         if "stage_transitions" in history:
193:
             summary["novel_strategies_summary"]["adaptive_transitions"] = len(history["stage_transitions"])
194:
195:
         if "dynamic_weights" in history:
196:
             summary["novel_strategies_summary"]["weight_updates"] = len(history["dynamic_weights"])
197:
198:
         if "curriculum_events" in history:
199:
             summary["novel_strategies_summary"]["curriculum_events"] = len(history["curriculum_events"])
200:
201:
         # Save as JSON
         with open(output_dir / "training_summary.json", 'w') as f:
202:
203:
             json.dump(summary, f, indent=2)
204:
205:
         print(f"Training summary saved to {output_dir / 'training_summary.json'}")
206:
207:
208: def main():
209:
        parser = argparse.ArgumentParser(description="Enhanced Neural-Geometric 3D Model Generator Training")
210:
211:
         # Basic arguments
212:
         parser.add_argument("--data_dir", type=str, default="./data/floorplans",
213:
                            help="Path to dataset directory")
214:
         parser.add_argument("--batch_size", type=int, default=2, help="Batch size")
215:
         parser.add_argument("--num_workers", type=int, default=4, help="Number of data workers")
        parser.add_argument("--device", type=str, default=None, help="Training device")
216:
217:
        parser.add_argument("--resume", type=str, default=None, help="Resume from checkpoint")
        parser.add_argument("--output_dir", type=str, default="./checkpoints",
218:
219:
                            help="Output directory for checkpoints")
220:
221:
         # Training mode selection
222:
         parser.add_argument("--training_mode", type=str, choices=["traditional", "adaptive"],
223:
                            default="adaptive", help="Training mode (traditional fixed epochs vs adaptive)")
224:
         parser.add_argument("--stage", type=str, choices=["1", "2", "3", "all"], default="all",
                            help="Training stage to run (only for traditional mode)")
225:
226:
227:
         # Novel training strategies
228:
         parser.add_argument("--dynamic-curriculum", action="store_true", default=True,
229:
                            help="Enable adaptive stage transitioning")
230:
         parser.add_argument("--patience", type=int, default=5,
231:
                            help="Epochs without improvement before stage transition")
232:
         parser.add_argument("--min-improvement", type=float, default=0.001,
233:
                            help="Minimum relative improvement threshold")
234:
235:
         parser.add_argument("--gradnorm", action="store_true", default=True,
236:
                            help="Enable GradNorm dynamic loss weighting")
237:
         parser.add_argument("--gradnorm-alpha", type=float, default=0.12,
238:
                            help="GradNorm restoring force parameter")
         parser.add_argument("--gradnorm-freq", type=int, default=5,
239:
240:
                            help="GradNorm update frequency (batches)")
241:
242:
         parser.add_argument("--topology-schedule", type=str,
```

```
243:
                            choices=["static", "progressive", "linear_ramp"],
                            default="progressive", help="Topology loss scheduling strategy")
244:
245:
         parser.add_argument("--topology-start", type=float, default=0.1,
                            help="Starting weight for topology loss")
247:
         parser.add_argument("--topology-end", type=float, default=1.0,
248:
                            help="Ending weight for topology loss")
249:
250:
         # Model enhancements
         parser.add_argument("--latent-consistency", action="store_true", default=True,
251:
                           help="Enable cross-modal latent consistency loss")
252:
        parser.add_argument("--graph-constraints", action="store_true", default=True,
253:
254:
                            help="Enable graph-based topology constraints")
255:
        parser.add_argument("--embedding-dim", type=int, default=256,
256:
                            help="Latent embedding dimension")
257:
258:
         # Model architecture
259:
        parser.add_argument("--input_channels", type=int, default=3, help="Input image channels")
260:
         parser.add_argument("--num_classes", type=int, default=5, help="Number of segmentation classes")
         \verb|parser.add_argument("--feature_dim", type=int, default=768, help="Feature dimension")| \\
261:
262:
        parser.add_argument("--num_attributes", type=int, default=6, help="Number of attribute predictions")
        parser.add_argument("--voxel_size", type=int, default=64, help="3D voxel grid size")
263:
264:
        parser.add_argument("--max_polygons", type=int, default=30, help="Maximum number of polygons")
265:
        parser.add_argument("--max_points", type=int, default=100, help="Maximum points per polygon")
266:
267:
         # Dynamic epoch limits
         \verb|parser.add_argument("--max-stagel-epochs", type=int, default=50, help="Max epochs for Stage 1")| \\
268:
269:
         \verb|parser.add_argument("--max-stage2-epochs", type=int, default=35, help="Max epochs for Stage 2")| \\
         parser.add_argument("--max-stage3-epochs", type=int, default=100, help="Max epochs for Stage 3")
270:
271:
272:
        parser.add_argument("--persistent_workers",action="store_true",default=False,help="Keep DataLoader work
273:
274:
         parser.add_argument("--prefetch_factor",type=int,default=2,help="Number of batches preloaded by each wo
275:
276:
277:
        args = parser.parse_args()
278:
279:
         # Setup device
280:
        device = args.device or ("cuda" if torch.cuda.is_available() else "cpu")
281:
        print(f"Using device: {device}")
282:
283:
         import torch.backends.cudnn as cudnn
        if device == "cuda":
284:
285:
            cudnn.benchmark = True
286:
287:
         # Create output directory
288:
         output_dir = Path(args.output_dir)
289:
        output_dir.mkdir(exist_ok=True)
290:
291:
         # Create enhanced configuration
         config = create_enhanced_config(args)
292:
293:
294:
        print("\n" + "="*80)
295:
        print("NEURAL-GEOMETRIC 3D MODEL GENERATOR - ENHANCED TRAINING")
296:
        print("="*80)
297:
        print("Novel Training Strategies Enabled:")
298:
        if config.curriculum.use_dynamic_curriculum:
299:
            print("? Adaptive Stage Transitioning (Dynamic Curriculum)")
300:
        if config.curriculum.use_gradnorm:
301:
            print("? Multi-objective Optimization with GradNorm")
302:
         if config.curriculum.topology_schedule != "static":
303:
            print("? Topology-aware Loss Scheduling")
304:
        if args.latent_consistency:
305:
            print("? Cross-modal Latent Consistency Learning")
306:
         if args.graph_constraints:
307:
            print("? Graph-based Topology Constraints")
308:
        print("="*80)
309:
310:
        # Create datasets
311:
         print("\nLoading datasets...")
312:
         train dataset = AdvancedFloorPlanDataset(
313:
             args.data_dir, split="train", augment=True
314:
315:
         val_dataset = AdvancedFloorPlanDataset(
```

```
316:
             args.data_dir, split="val", augment=False
317:
318:
319:
         print(f"Train samples: {len(train_dataset)}")
        print(f"Validation samples: {len(val_dataset)}")
320:
321:
322:
         if len(train_dataset) == 0:
323:
             print("Error: No training samples found!")
324:
             return
325:
326:
         # Create data loaders
327:
         train loader = DataLoader(
328:
             train_dataset,
329:
             batch_size=args.batch_size,
330:
            shuffle=True.
331:
            num_workers=args.num_workers,
332:
             pin_memory=True,
333:
             drop_last=True,
334:
             persistent_workers=args.persistent_workers if args.num_workers > 0 else False,
335:
             prefetch_factor=args.prefetch_factor if args.num_workers > 0 else None
336:
        )
337:
338:
         val_loader = DataLoader(
339:
             val dataset.
340:
             batch_size=max(1, args.batch_size),
341:
             shuffle=False,
342:
             num_workers=max(1, args.num_workers // 2),
343:
             pin_memory=True,
344:
            drop_last=False,
345:
             persistent_workers=args.persistent_workers if args.num_workers > 0 else False,
346:
             prefetch_factor=args.prefetch_factor if args.num_workers > 0 else None
347:
348:
349:
         # Create enhanced model
350:
         print("\nInitializing enhanced model...")
351:
        model = create_enhanced_model(args)
352:
353:
         # Create appropriate trainer
354:
        if args.training_mode == "adaptive":
355:
            print("\nUsing Adaptive Multi-Stage Trainer with Novel Strategies")
356:
             trainer = AdaptiveMultiStageTrainer(
357:
                model=model,
358:
                 train_loader=train_loader,
359:
                 val_loader=val_loader,
360:
                 device=device,
361:
                 config=config
362:
             )
363:
         else:
            print("\nUsing Traditional Multi-Stage Trainer")
364:
365:
             trainer = MultiStageTrainer(
366:
                model=model,
367:
                 train_loader=train_loader,
368:
                 val_loader=val_loader,
369:
                 device=device,
370:
                 config=config
371:
             )
372:
373:
         # Resume from checkpoint if specified
374:
         if args.resume:
375:
             print(f"Resuming from checkpoint: {args.resume}")
376:
             trainer.load_checkpoint(args.resume)
377:
378:
         # Run training
379:
         if args.training_mode == "adaptive" or args.stage == "all":
380:
             \verb|print("\nStarting adaptive multi-stage training with novel strategies...")|\\
381:
             history = trainer.train_all_stages()
382:
         else:
383:
             # Traditional single-stage training
384:
             stage_num = int(args.stage)
385:
             print(f"Training Stage {stage_num} only...")
386:
            if stage_num == 1:
387:
                 trainer.train_stage1()
388:
             elif stage_num == 2:
```

```
389:
                trainer.train stage2()
390:
            elif stage_num == 3:
391:
                trainer.train_stage3()
392:
            history = trainer.history
393:
394:
        # Save final model
395:
        final_model_path = output_dir / "final_enhanced_model.pth"
396:
        if hasattr(trainer, '_save_checkpoint'):
397:
             trainer._save_checkpoint(str(final_model_path))
398:
        print(f"Final model saved to: {final_model_path}")
399:
400:
        # Create comprehensive visualizations
401:
        print("\nGenerating training analysis...")
402:
        visualize_training_results(history, output_dir)
403:
404:
         # Save training summary
405:
        save_training_summary(history, config, output_dir)
406:
407:
        print(f"\n? Enhanced training completed successfully!")
408:
        print(f"? Results saved to: {output_dir}")
409:
       print("\nNovel contributions implemented:")
       print("- Dynamic curriculum learning with adaptive stage transitions")
410:
411:
        print("- Multi-objective optimization with gradient-based reweighting")
        print("- Topology-aware progressive constraint injection")
412:
       print("- Cross-modal latent consistency learning")
413:
414:
        print("- Graph-based architectural constraint learning")
415:
416:
417: if __name__ == "__main__":
418:
       main()
```

\_\_\_\_\_\_

# ■ File: training\losses.py

```
______
 1: """
 2: Advanced loss functions for multi-task training with dynamic weighting
 3: Enhanced with cross-modal consistency, graph constraints, and GradNorm
 4: Modified to support conditional geometric losses via run_full_geometric flag
 5: """
 6:
 7: import torch
 8: import torch.nn as nn
 9: import torch.nn.functional as F
 10: import cv2
 11: import numpy as np
12: from typing import Dict, Optional, Tuple, List
13: import networkx as nx
14:
15:
16: class DynamicLossWeighter:
        """Implements GradNorm and other dynamic weighting strategies"""
18:
       def __init__(self, loss_names: List[str], alpha: float = 0.12, device: str = 'cuda'):
19:
 20:
            self.loss_names = loss_names
 21:
           self.alpha = alpha
           self.device = device
 22:
 23:
 24:
            # Initialize weights
            self.weights = {name: 1.0 for name in loss_names} # FIX: keep floats, easier logging
 25:
 26:
            self.initial_task_losses = None
 27:
            self.running_mean_losses = {name: 0.0 for name in loss_names}
            self.update_count = 0
 28:
 29:
 30:
        def update_weights(self, task_losses: Dict[str, torch.Tensor],
 31:
                         shared_parameters, update_freq: int = 10):
            """Update loss weights using GradNorm algorithm with enhanced stability"""
32:
 33:
            if self.update_count % update_freq != 0:
 34:
                self.update_count += 1
 35:
                return self.weights
 36:
 37:
            # Store initial losses on first update
 38:
            if self.initial_task_losses is None:
```

```
39:
                 self.initial_task_losses = {name: loss.item() for name, loss in task_losses.items()}
 40:
 41:
             # Update running mean
 42:
             for name, loss in task_losses.items():
                 self.running_mean_losses[name] = (0.9 * self.running_mean_losses[name] +
 43:
 44:
                                                 0.1 * loss.item())
 45:
 46:
             # Calculate relative decrease rates
 47:
             loss_ratios = {}
             for name in self.loss_names:
 48:
 49:
                 if name in task_losses:
 50:
                     current_loss = self.running_mean_losses[name]
 51:
                     initial_loss = self.initial_task_losses[name]
 52:
                     loss_ratios[name] = current_loss / (initial_loss + 1e-8)
 53:
 54:
             # Calculate average relative decrease
             if not loss_ratios:
 55:
 56:
                 self.update_count += 1
 57:
                 return self.weights
 58:
             avg_loss_ratio = sum(loss_ratios.values()) / len(loss_ratios)
 59:
 60:
             # Calculate gradient norms with enhanced safety checks
 61:
             grad_norms = {}
 62:
             for name in self.loss_names:
                 if name in task_losses:
 63:
                     # Check if loss is finite before gradient computation
 64:
 65:
                     if not torch.isfinite(task_losses[name]):
                         print(f"Warning: Non-finite loss for {name}, skipping gradient computation")
 66:
                         continue
 67:
 68:
 69:
                     trv:
 70:
                         grads = torch.autograd.grad(
 71:
                             task_losses[name], shared_parameters,
 72:
                             retain_graph=True, create_graph=False, allow_unused=True
 73:
 74:
                         grad_norm = 0.0
 75:
                          for grad in grads:
 76:
                              if grad is not None and torch.isfinite(grad).all():
 77:
                                  grad_norm += grad.norm().item() ** 2
 78:
                              elif grad is not None:
 79:
                                  print(f"Warning: Non-finite gradient detected for {name}")
 80:
                                  grad_norm = 0.0
 81:
                                  break
 82:
 83:
                          if grad_norm > 0:
 84:
                             grad_norms[name] = grad_norm ** 0.5
 85:
                     except Exception as e:
 86:
                         print(f"Warning: Gradient computation failed for {name}: {e}")
 87:
                         continue
 88:
 89:
             if not grad_norms:
 90:
                 self.update_count += 1
 91:
                 return self.weights
 92:
 93:
             avg_grad_norm = sum(grad_norms.values()) / len(grad_norms)
 94:
 95:
             # Enhanced safety parameters
 96:
             epsilon = 1e-6 # Increased minimum threshold
 97:
             max_target_grad = 10.0 # Reduced maximum allowed gradient
 98:
             max_weight_update = 2.0 # Reduced maximum weight change
 99:
100:
             for name in self.loss_names:
101:
                 if name in grad_norms and name in loss_ratios:
102:
                     target_grad = avg_grad_norm * (loss_ratios[name] ** self.alpha)
103:
                     target_grad = min(target_grad, max_target_grad)
104:
105:
                     safe_grad_norm = max(grad_norms[name], epsilon)
106:
                     if safe_grad_norm == epsilon:
107:
                          print(f"? Small gradient norm for {name}, clamped to {epsilon}")
108:
109:
                     weight_update = target_grad / safe_grad_norm
110:
                     weight_update = min(weight_update, max_weight_update)
111:
                     if weight_update == max_weight_update:
```

```
print(f"? Weight update for {name} clamped to {max_weight_update}")
112:
113:
114:
                     # More conservative momentum update
                     new_w = 0.95 * self.weights[name] + 0.05 * float(weight_update) # Slower adaptation
115:
116:
                     self.weights[name] = float(np.clip(new_w, 0.01, 3.0)) # Tighter bounds
117:
118:
             self.update_count += 1
119:
             return self.weights
120:
121: class GraphTopologyExtractor:
         """Extracts graph structure from segmentation for topology constraints"""
123:
124:
         @staticmethod
125:
         def extract_room_graph(segmentation: torch.Tensor) -> Dict[str, torch.Tensor]:
126:
             """Extract room connectivity graph from segmentation mask""
127:
             B, C, H, W = segmentation.shape
128:
             device = segmentation.device
129:
             # Get room predictions (assume classes: 0=bg, 1=wall, 2=door, 3=window, 4=room)
130:
131:
             room_probs = F.softmax(segmentation, dim=1)
132:
             room_mask = room_probs[:, 4] if C > 4 else torch.zeros((B, H, W), device=device)
133:
             wall_mask = room_probs[:, 1] if C > 1 else torch.zeros((B, H, W), device=device)
134:
135:
             # Simple connectivity: rooms connected if they share wall boundary
136:
             adjacency_matrices = []
137:
             room_features = []
138:
139:
             for b in range(B):
140:
                 room_b = room_mask[b].detach().cpu().numpy()
141:
                 wall_b = wall_mask[b].detach().cpu().numpy()
142:
143:
                 # Find connected components (rooms)
144:
                 try:
145:
                     from scipy import ndimage
                     labeled_rooms, num_rooms = ndimage.label(room_b > 0.5)
146:
147:
148:
                     # Create adjacency matrix
149:
                     adj_matrix = np.zeros((max(num_rooms, 1), max(num_rooms, 1)))
150:
                     room_centroids = []
151:
152:
                     for i in range(1, num_rooms + 1):
153:
                         room_i_mask = (labeled_rooms == i)
154:
                         if np.sum(room_i_mask) > 0:
155:
                             centroid = ndimage.center_of_mass(room_i_mask)
156:
                             room_centroids.append(centroid)
157:
158:
                             # Check connectivity to other rooms through walls
159:
                             for j in range(i + 1, num_rooms + 1):
160:
                                 room_j_mask = (labeled_rooms == j)
161:
                                 if np.sum(room_j_mask) > 0:
162:
                                     # Check if rooms are connected via wall adjacency
163:
                                     connectivity = GraphTopologyExtractor._check_room_connectivity(
164:
                                         room_i_mask, room_j_mask, wall_b
165:
166:
                                      adj_matrix[i-1, j-1] = connectivity
167:
                                     adj_matrix[j-1, i-1] = connectivity
168:
169:
                     # Convert to tensor
170:
                     adj_tensor = torch.from_numpy(adj_matrix).float().to(device)
171:
                     centroids_tensor = torch.from_numpy(np.array(room_centroids) if room_centroids else np.zero
172:
173:
                 except ImportError:
174:
                     # Fallback if scipy not available
175:
                     adj_tensor = torch.zeros((1, 1), device=device)
                     centroids_tensor = torch.zeros((0, 2), device=device)
176:
177:
178:
                 adjacency_matrices.append(adj_tensor)
                 {\tt room\_features.append(centroids\_tensor)}
179:
180:
181:
             return {
                 "adjacency_matrices": adjacency_matrices,
182:
                 "room_features": room_features
183:
184:
```

```
185:
186:
        @staticmethod
187:
       def _check_room_connectivity(room1_mask, room2_mask, wal1_mask):
              """Check if two rooms are connected through walls"""
189:
             try:
190:
                 from scipy.ndimage import binary_dilation
191:
192:
                 # Dilate room masks to check wall adjacency
                 dilated1 = binary_dilation(room1_mask, iterations=2)
193:
194:
                 dilated2 = binary_dilation(room2_mask, iterations=2)
195:
196:
                 # Check overlap through wall areas
197:
                 wall_overlap = (dilated1 & dilated2) & (wall_mask > 0.3)
198:
                return float(np.sum(wall_overlap) > 0)
199:
            except ImportError:
200:
                # Simple distance-based fallback
201:
                return 0.0
202:
203:
204: class ResearchGradeLoss(nn.Module):
205:
206:
        Multi-task loss combining:
207:
         - Traditional losses (segmentation, SDF, attributes, polygons, voxels, topology)
208:
        - NEW: Cross-modal latent consistency
         - NEW: Graph-based topology constraints
209:
210:
         - NEW: Dynamic loss weighting via GradNorm
211:
         - NEW: Conditional geometric losses based on run_full_geometric flag
212:
213:
214:
        def __init__(
215:
            self,
216:
             seg_weight: float = 1.0,
217:
            dice weight: float = 1.0,
            sdf_weight: float = 0.5,
218:
            attr_weight: float = 1.0,
219:
220:
            polygon_weight: float = 1.0,
221:
             voxel_weight: float = 1.0,
222:
            topology_weight: float = 0.5,
223:
            latent_consistency_weight: float = 0.5,
           graph_constraint_weight: float = 0.3,
224:
225:
            enable_dynamic_weighting: bool = True,
            gradnorm_alpha: float = 0.12,
226:
227:
            device: str = 'cuda',
228:
            weight_update_freq: int = 10,
229:
            weight_momentum: float = 0.9,
       ):
230:
231:
            super().__init__()
232:
233:
             # Store initial weights
234:
             self.initial_weights = {
                 'seg': float(seg_weight),
235:
236:
                 'dice': float(dice_weight),
237:
                 'sdf': float(sdf_weight),
238:
                 'attr': float(attr_weight),
239:
                 'polygon': float(polygon_weight),
                 'voxel': float(voxel_weight),
240:
                 'topology': float(topology_weight),
241:
                 'latent_consistency': float(latent_consistency_weight),
242:
243:
                 'graph': float(graph_constraint_weight)
             }
244:
245:
246:
             # Current weights (will be dynamically updated)
247:
            self.weights = self.initial_weights.copy()
248:
249:
             # Core losses
250:
            self.ce_loss = nn.CrossEntropyLoss()
251:
            self.mse_loss = nn.MSELoss()
252:
             self.l1_loss = nn.L1Loss()
253:
             self.cosine_loss = nn.CosineEmbeddingLoss()
254:
255:
             # Dynamic weighting
             self.enable_dynamic_weighting = enable_dynamic_weighting
256:
257:
             if enable_dynamic_weighting:
```

```
258:
                 self.loss weighter = DynamicLossWeighter(
259:
                     list(self.initial_weights.keys()), alpha=gradnorm_alpha, device=device,
260:
                 )
261:
                 self.loss_weighter.update_freq = weight_update_freq
262:
                 self.loss_weighter.momentum = weight_momentum
263:
264:
             self.device = device
265:
         def update_loss_weights(self, new_weights: Dict[str, float]):
266:
              ""Update loss weights (called by trainer for curriculum scheduling)"""
267:
268:
             for key, value in new_weights.items():
269:
                 if key in self.weights:
270:
                     self.weights[key] = float(value)
271:
272:
         def forward(self, predictions: dict, targets: dict, shared_parameters=None, run_full_geometric=True):
273:
274:
             Compute multi-task loss with conditional geometric computation and dynamic weighting.
275:
276:
             Args:
277:
                 predictions: Model predictions dict
278:
                 targets: Ground truth targets dict
279:
                 shared_parameters: Model parameters for GradNorm (optional)
280:
                 run_full_geometric: Whether geometric losses should be computed
281:
282:
             Returns:
283:
                tuple: (total_loss, individual_losses_dict)
284:
285:
             # Input validation - check for NaN/Inf values
286:
             for name, tensor in predictions.items():
287:
                 if torch.is_tensor(tensor) and (torch.isnan(tensor).any() or torch.isinf(tensor).any()):
288:
                     print(f"WARNING: NaN/Inf in predictions[{name}] - zeroing out")
289:
                     predictions[name] = torch.zeros_like(tensor)
290:
291:
             for name, tensor in targets.items():
                  if \ torch.is\_tensor(tensor) \ and \ (torch.isnan(tensor).any() \ or \ torch.isinf(tensor).any()); \\
292:
293:
                     print(f"WARNING: NaN/Inf in targets[{name}] - zeroing out")
294:
                     targets[name] = torch.zeros_like(tensor)
295:
296:
             device = self._get_device_from_inputs(predictions, targets)
297:
             losses = {}
298:
             total_loss = torch.tensor(0.0, device=device)
299:
300:
             # ---- 1) Core losses (always computed - lightweight) ----
301:
             if "segmentation" in predictions and "mask" in targets:
302:
                 seg_pred = predictions["segmentation"]
303:
                 seg_target = targets["mask"].long()
304:
305:
                 ce_loss = self.ce_loss(seg_pred, seg_target)
306:
                 losses["ce"] = ce_loss
307:
                 losses["seg"] = ce_loss # alias for dynamic weighting
308:
309:
                 dice_loss = self._dice_loss(seg_pred, seg_target)
310:
                 losses["dice"] = dice_loss
311:
312:
             if "sdf" in predictions and "mask" in targets:
313:
                 sdf_pred = predictions["sdf"]
                 sdf_pred = torch.clamp(sdf_pred, -1.0, 1.0) # FIX: prevent huge values
314:
315:
                 sdf_target = self._mask_to_sdf(targets["mask"])
316:
                 sdf_target = sdf_target.to(sdf_pred.device).type_as(sdf_pred)
                 losses["sdf"] = self.mse_loss(sdf_pred, sdf_target)
317:
318:
319:
             if "attributes" in predictions and "attributes" in targets:
320:
                 pred_attr = predictions["attributes"].float()
321:
                 tgt_attr = targets["attributes"].float().to(pred_attr.device)
                 losses["attr"] = self.l1_loss(pred_attr, tgt_attr)
322:
323:
324:
             # ---- 2) Conditional geometric losses (heavy operations) ----
325:
             if run full geometric:
326:
                 # Polygon loss (only if model produced polygons)
                 if ("polygons" in predictions and predictions["polygons"] is not None and
327:
328:
                     "polygons_gt" in targets):
329:
                     losses["polygon"] = self._polygon_loss(predictions, targets["polygons_gt"])
330:
                 else:
```

```
331:
                     # Zero loss if polygons not available
332:
                     losses["polygon"] = torch.tensor(0.0, device=device)
333:
334:
                 # Voxel loss (only if model produced voxels)
                 if ("voxels_pred" in predictions and predictions["voxels_pred"] is not None and
335:
336:
                      "voxels_gt" in targets):
337:
                     pred_vox = predictions["voxels_pred"].float()
338:
                     tgt_vox = targets["voxels_gt"].float().to(pred_vox.device)
339:
                     losses["voxel"] = self._voxel_iou_loss(pred_vox, tgt_vox)
340:
                 else:
341:
                     # Zero loss if voxels not available
                     losses["voxel"] = torch.tensor(0.0, device=device)
342:
343:
344:
                 # Cross-modal latent consistency (only if embeddings available)
345:
                 if ("latent 2d embedding" in predictions and "latent 3d embedding" in predictions and
346:
                     predictions["latent_2d_embedding"] is not None and predictions["latent_3d_embedding"] is no
347:
                     consistency_loss = self._latent_consistency_loss(
348:
                         predictions["latent_2d_embedding"],
                         predictions["latent_3d_embedding"]
349:
350:
351:
                     losses["latent_consistency"] = consistency_loss
352:
                 else:
353:
                     losses["latent_consistency"] = torch.tensor(0.0, device=device)
354:
             else:
                 # When geometric computation is skipped, use zero losses
355:
356:
                 losses["polygon"] = torch.tensor(0.0, device=device)
357:
                 losses["voxel"] = torch.tensor(0.0, device=device)
358:
                 losses["latent_consistency"] = torch.tensor(0.0, device=device)
359:
360:
             # ---- 3) Independent auxiliary losses (always computed if enabled) ----
361:
             # Traditional topology loss
362:
             if "segmentation" in predictions:
                 losses["topology"] = self._topology_loss(predictions["segmentation"])
363:
364:
365:
             # Graph-based topology constraints
366:
             if "segmentation" in predictions:
367:
                 graph_loss = self._graph_topology_loss(predictions["segmentation"])
368:
                 losses["graph"] = graph_loss
369:
370:
             # ---- 4) Apply weighting ----
371:
             if self.enable_dynamic_weighting and shared_parameters is not None:
372:
                 \ensuremath{\sharp} Only include differentiable losses for GradNorm
373:
                 task_losses = {
374:
                     name: loss for name, loss in losses.items()
375:
                     if name in self.weights and isinstance(loss, torch.Tensor) and loss.requires_grad
376:
377:
378:
                 dynamic_weights = self.loss_weighter.update_weights(task_losses, shared_parameters)
379:
380:
                 # Apply weights (dynamic for diff losses, static for non-diff losses)
                 for name, loss in losses.items():
381:
382:
                     if name in self.weights:
383:
                         if name in dynamic_weights:
384:
                             weight = dynamic_weights[name]
385:
                         else:
386:
                             weight = self.weights[name]
387:
                         total_loss = total_loss + weight * loss
388:
             else:
389:
                 # Static weights
                 for name, loss in losses.items():
390:
391:
                     if name in self.weights:
392:
                         total_loss = total_loss + self.weights[name] * loss
393:
394:
             # Final NaN/Inf guard
395:
             for k, v in list(losses.items()):
396:
                 if torch.isnan(v).any() or torch.isinf(v).any():
397:
                     print(f"[Warning] {k} loss is NaN/Inf ? zeroed out")
398:
                     losses[k] = torch.tensor(0.0, device=device)
399:
400:
             losses["total"] = total_loss
401:
             return total loss, losses
402:
403:
         def __call__(self, predictions: dict, targets: dict, shared_parameters=None, run_full_geometric=True):
```

```
404:
405:
             Convenience method for trainer compatibility
406:
407:
408:
                predictions: Model predictions dict
409:
                 targets: Ground truth targets dict
410:
                 shared_parameters: Model parameters for GradNorm (optional)
411:
                 run_full_geometric: Whether to compute geometric losses
412:
413:
             Returns:
414:
                tuple: (total_loss, individual_losses_dict)
415:
416:
             return self.forward(predictions, targets, shared_parameters, run_full_geometric)
417:
418:
        def _get_device_from_inputs(self, predictions, targets):
419:
             """Helper to determine device from inputs"""
             for pred_dict in [predictions, targets]:
420:
421:
                 for value in pred_dict.values():
422:
                     if torch.is_tensor(value):
423:
                         return value.device
424:
             return self.device
425:
426:
         # ---- NEW: Cross-modal latent consistency loss ----
427:
         def _latent_consistency_loss(self, embedding_2d: torch.Tensor, embedding_3d: torch.Tensor) -> torch.Ten
428:
             Ensure 2D floorplan embeddings match 3D voxelized structure embeddings
429:
430:
             embedding_2d: [B, D] - 2D floorplan embeddings
             embedding_3d: [B, D] - 3D structure embeddings
431:
432:
433:
             if embedding_2d.shape != embedding_3d.shape:
434:
                 # Project to same dimension if needed
435:
                 min_dim = min(embedding_2d.shape[-1], embedding_3d.shape[-1])
436:
                 embedding_2d = embedding_2d[..., :min_dim]
437:
                 embedding_3d = embedding_3d[..., :min_dim]
438:
439:
             # Cosine similarity loss (maximize similarity)
440:
             target = torch.ones(embedding_2d.shape[0], device=embedding_2d.device)
441:
             cosine_loss = self.cosine_loss(embedding_2d, embedding_3d, target)
442:
443:
             # L2 consistency loss
444:
             12_loss = F.mse_loss(embedding_2d, embedding_3d)
445:
446:
             return 0.7 * cosine_loss + 0.3 * 12_loss
447:
448:
         # ---- NEW: Graph-based topology constraints ----
449:
         def _graph_topology_loss(self, segmentation_logits: torch.Tensor) -> torch.Tensor:
450:
451:
             Graph-based topology constraints on room connectivity
452:
             segmentation_logits: [B, C, H, W]
453:
454:
             try:
455:
                 # Extract graph structure
456:
                 graph_data = GraphTopologyExtractor.extract_room_graph(segmentation_logits)
457:
                 device = segmentation_logits.device
458:
                 total_graph_loss = torch.tensor(0.0, device=device)
459:
460:
                 batch_size = segmentation_logits.shape[0]
461:
462:
                 for b in range(batch_size):
463:
                     if b < len(graph_data["adjacency_matrices"]):</pre>
                         adj_matrix = graph_data["adjacency_matrices"][b]
464:
465:
                         if adj_matrix.numel() == 0:
466:
                             continue
467:
468:
                         # Connectivity constraint: encourage reasonable connectivity
469:
                         # Penalize isolated rooms (degree 0) and over-connected rooms
470:
                         degrees = adj_matrix.sum(dim=1)
471:
472:
                         # Isolation penalty (rooms should have at least 1 connection)
473:
                         isolation_penalty = torch.exp(-degrees).mean()
474:
475:
                         # Over-connection penalty (rooms shouldn't connect to everything)
476:
                         max_reasonable_connections = min(4, adj_matrix.shape[0] - 1)
```

```
477:
                                         over_connection_penalty = F.relu(degrees - max_reasonable_connections).mean()
478:
479:
                                         # Graph smoothness (connected rooms should have similar features)
480:
                                         if b < len(graph_data["room_features"]) and graph_data["room_features"][b].numel() > 0:
481:
                                               room_features = graph_data["room_features"][b]
482:
                                               if room_features.shape[0] > 1:
483:
                                                      feature_distances = torch.cdist(room_features, room_features)
484:
                                                      # Weight by adjacency - connected rooms should be similar
485:
                                                      smoothness_loss = (adj_matrix * feature_distances).sum() / (adj_matrix.sum() +
486:
                                                else:
487:
                                                      smoothness_loss = torch.tensor(0.0, device=device)
488:
                                         else:
489:
                                               smoothness_loss = torch.tensor(0.0, device=device)
490:
491:
                                         batch_graph_loss = (0.4 * isolation_penalty +
492:
                                                                      0.3 * over_connection_penalty +
                                                                      0.3 * smoothness_loss)
493:
494:
                                         total_graph_loss = total_graph_loss + batch_graph_loss
495:
496:
                           return total_graph_loss / batch_size
497:
498:
                    except Exception as e:
499:
                            # Fallback to zero loss if graph extraction fails
500:
                           return torch.tensor(0.0, device=segmentation_logits.device)
501:
502:
              # ---- Existing helper methods (preserved) ----
503:
              def _dice_loss(self, pred: torch.Tensor, target: torch.Tensor, smooth: float = 1e-6) -> torch.Tensor:
                      """Dice loss implementation"""
504:
505:
                     pred_soft = F.softmax(pred, dim=1)
506:
                     B = pred_soft.shape[0]
507:
                     C = pred_soft.shape[1]
508:
509:
                     dice losses = []
510:
                     for c in range(C):
                           pred_c = pred_soft[:, c, :, :]
511:
512:
                           target_c = (target == c).float().to(pred_c.device)
                           intersection = (pred_c * target_c).view(B, -1).sum(dim=1)
513:
                           union = pred_c.view(B, -1).sum(dim=1) + target_c.view(B, -1).sum(dim=1)
514:
515:
                           dice = (2.0 * intersection + smooth) / (union + smooth)
516:
                           dice_losses.append((1.0 - dice).mean())
517:
518:
                     return torch.stack(dice_losses).mean()
519:
520:
              def _mask_to_sdf(self, mask: torch.Tensor) -> torch.Tensor:
521:
                      """Convert mask to SDF with performance warning"""
522:
                     device = mask.device if torch.is_tensor(mask) else None
523:
                     if not torch.is tensor(mask):
524:
                           mask = torch.tensor(mask, device=device)
525:
526:
                     B, H, W = mask.shape
                     sdf = torch.zeros((B, 1, H, W), dtype=torch.float32, device=device)
527:
528:
529:
                     # FIX: Add performance warning for CV2 bottleneck
530:
                     if B > 8: # Warn for large batches
531:
                           \verb|print(f"[Performance Warning]| SDF conversion with batch_size={B} uses CPU cv2 - consider GPU improved the conversion of the conversio
532:
533:
                     for b in range(B):
534:
                           mask_np = mask[b].detach().cpu().numpy().astype(np.uint8) # FIX: explicit detach
535:
                            try:
536:
                                  dist_inside = cv2.distanceTransform((mask_np > 0).astype(np.uint8), cv2.DIST_L2, 5)
537:
                                  dist_outside = cv2.distanceTransform((mask_np == 0).astype(np.uint8), cv2.DIST_L2, 5)
538:
                                  sdf_np = dist_inside.astype(np.float32) - dist_outside.astype(np.float32)
539:
                                  sdf_np = np.tanh(sdf_np / 10.0).astype(np.float32)
540:
                                  sdf[b, 0] = torch.from_numpy(sdf_np)
541:
                            except Exception:
542:
                                  # Fallback if cv2 fails
543:
                                  sdf[b, 0] = torch.zeros_like(mask[b].float())
544:
545:
                     return sdf
546:
547:
              def _polygon_loss(self, predictions: dict, targets: dict) -> torch.Tensor:
548:
                       ""Polygon/DVX loss (preserved from original)"""
549:
                     pred_polys = predictions.get("polygons")
```

```
550:
             tqt polys = targets.get("polygons")
551:
             valid_mask = targets.get("valid_mask")
552:
553:
             if pred_polys is None or tgt_polys is None:
554:
                 return torch.tensor(0.0, device=pred_polys.device if pred_polys is not None else self.device)
555:
556:
             pred_polys = pred_polys.float()
557:
             tgt_polys = tgt_polys.float().to(pred_polys.device)
558:
559:
             point_loss = self.mse_loss(pred_polys, tgt_polys)
560:
561:
             pred_valid = predictions.get("polygon_validity")
562:
             if pred_valid is None or valid_mask is None:
563:
                 validity_loss = torch.tensor(0.0, device=pred_polys.device)
564:
             else:
565:
                 pred_valid = pred_valid.float().to(pred_polys.device)
566:
                 valid_mask_f = valid_mask.float().to(pred_polys.device)
567:
                 validity_loss = self.mse_loss(pred_valid, valid_mask_f)
568:
569:
             smoothness_loss = self._polygon_smoothness(pred_polys)
570:
             rect_loss = self._rectilinearity_loss(pred_polys)
571:
572:
             return point_loss + 0.1 * validity_loss + 0.05 * smoothness_loss + 0.1 * rect_loss
573:
574:
         def _polygon_smoothness(self, polygons: torch.Tensor) -> torch.Tensor:
575:
              """Polygon smoothness loss (preserved)"""
576:
             if polygons is None or polygons.numel() == 0:
577:
                 return torch.tensor(0.0, device=polygons.device if polygons is not None else self.device)
578:
579:
             p1 = polygons
580:
             p2 = torch.roll(polygons, -1, dims=2)
581:
             p3 = torch.roll(polygons, -2, dims=2)
             curvature = torch.norm(p1 - 2.0 * p2 + p3, dim=-1)
582:
583:
             return curvature.mean()
584:
585:
         def _rectilinearity_loss(self, polygons: torch.Tensor) -> torch.Tensor:
586:
              """Encourage axis-aligned structure (preserved)"""
587:
             if polygons is None or polygons.numel() == 0:
588:
                 return torch.tensor(0.0, device=polygons.device if polygons is not None else self.device)
589:
590:
             edges = torch.roll(polygons, -1, dims=2) - polygons
591:
             edge_norms = torch.norm(edges, dim=-1, keepdim=True)
592:
             edges_normalized = edges / (edge_norms + 1e-6)
593:
594:
             edge1 = edges normalized
595:
             edge2 = torch.roll(edges_normalized, -1, dims=2)
596:
597:
             cos_angles = (edge1 * edge2).sum(dim=-1)
             cos2 = cos_angles ** 2
598:
599:
             perp_penalty = cos2
             parallel\_penalty = (cos2 - 1.0) ** 2
600:
601:
             angle_penalty = torch.minimum(perp_penalty, parallel_penalty)
602:
             return angle_penalty.mean()
603:
604:
         def _voxel_iou_loss(self, pred_voxels: torch.Tensor, target_voxels: torch.Tensor) -> torch.Tensor:
             """3D voxel IoU loss (preserved)"""
605:
             pred_prob = torch.sigmoid(torch.clamp(pred_voxels, -10.0, 10.0)) # FIX: safe sigmoid range
606:
607:
             target = target_voxels.float().to(pred_prob.device)
608:
609:
             intersection = (pred_prob * target).view(pred_prob.shape[0], -1).sum(dim=1)
610:
             union = (pred_prob.view(pred_prob.shape[0], -1).sum(dim=1) +
611:
                     target.view(target.shape[0], -1).sum(dim=1) - intersection)
612:
613:
             iou = (intersection + 1e-6) / (union + 1e-6)
             return (1.0 - iou).mean()
614:
615:
616:
         def _topology_loss(self, segmentation_logits: torch.Tensor) -> torch.Tensor:
617:
              """Traditional topology loss (preserved)"""
618:
             seg_soft = F.softmax(segmentation_logits, dim=1)
619:
             C = seq soft.shape[1]
620:
             device = seg soft.device
621:
622:
             walls = seg_soft[:, 1] if C > 1 else torch.zeros_like(seg_soft[:, 0])
```

```
doors = seq soft[:, 2] if C > 2 else torch.zeros like(walls)
623:
             windows = seg_soft[:, 3] if C > 3 else torch.zeros_like(walls)
624:
625:
626:
             door_wall_overlap = doors * walls
627:
             window_wall_overlap = windows * walls
628:
629:
             door_penalty = torch.maximum(doors - door_wall_overlap, torch.zeros_like(doors))
630:
             window_penalty = torch.maximum(windows - window_wall_overlap, torch.zeros_like(windows))
631:
632:
             connectivity_loss = self._connectivity_loss(walls)
633:
634:
             return door_penalty.mean() + window_penalty.mean() + 0.1 * connectivity_loss
635:
636:
         def _connectivity_loss(self, wall_prob: torch.Tensor) -> torch.Tensor:
637:
             """Connectivity loss for walls (preserved)"""
638:
             if wall_prob is None or wall_prob.numel() == 0:
639:
                return torch.tensor(0.0, device=wall_prob.device if wall_prob is not None else self.device)
640:
             \texttt{kernel = torch.ones((1, 1, 3, 3), device=wall\_prob.device, dtype=wall\_prob.dtype) / 9.0}
641:
642:
             neighbors = F.conv2d(wall_prob.unsqueeze(1), kernel, padding=1).squeeze(1)
643:
644:
             isolation_penalty = wall_prob * torch.exp(-neighbors)
645:
             return isolation_penalty.mean()
646:
647:
648: class LossScheduler:
649:
         """Manages curriculum-based loss weight scheduling"""
650:
651:
        def __init__(self, config):
652:
             self.config = config
653:
             self.loss_schedules = config.loss_schedule
654:
655:
         def get_scheduled_weights(self, current_stage: int, current_epoch: int,
656:
                                 stage_epoch: int, total_stage_epochs: int,
657:
                                 base_weights: Dict[str, float]) -> Dict[str, float]:
658:
659:
             Calculate loss weights based on curriculum schedule
660:
661:
             Arqs:
662:
                current_stage: Current training stage (1, 2, 3)
663:
                 current_epoch: Global epoch count
664:
                 stage_epoch: Epoch within current stage
665:
                 total_stage_epochs: Total epochs planned for current stage
666:
                 base_weights: Base weight configuration
667:
668:
             scheduled_weights = base_weights.copy()
669:
670:
             for loss_name, schedule_type in self.loss_schedules.items():
671:
                if loss_name not in scheduled_weights:
672:
                     continue
673:
674:
                 base_weight = scheduled_weights[loss_name]
675:
676:
                 if schedule_type == "static":
677:
                     # Keep original weight
                     continue
678:
679:
680:
                 elif schedule_type == "progressive":
681:
                     # Gradually increase throughout training
                     if loss_name == "topology":
682:
683:
                         start_weight = self.config.topology_start_weight
684:
                         end_weight = self.config.topology_end_weight
685:
                         ramp_epochs = self.config.topology_ramp_epochs
686:
                         progress = min(current_epoch / ramp_epochs, 1.0)
687:
                         scheduled_weights[loss_name] = start_weight + progress * (end_weight - start_weight)
688:
689:
                 elif schedule_type == "linear_ramp":
690:
                     # Linear increase within current stage
691:
                     progress = stage_epoch / max(total_stage_epochs, 1)
692:
                     scheduled_weights[loss_name] = base_weight * progress
693:
694:
                 elif schedule_type == "exponential":
695:
                     # Exponential increase
```

```
progress = stage_epoch / max(total_stage_epochs, 1)
696:
697:
                     scheduled_weights[loss_name] = base_weight * (progress ** 2)
698:
699:
                 elif schedule_type == "early_decay":
700:
                     # Decay after Stage 1 (for SDF loss)
701:
                     if current_stage > 1:
702:
                         scheduled_weights[loss_name] = base_weight * 0.3
703:
704:
                 elif schedule_type == "staged_ramp":
705:
                     # Ramp up during specific stage (polygon in Stage 2)
706:
                     if current_stage == 2:
707:
                         progress = stage_epoch / max(total_stage_epochs, 1)
708:
                         scheduled_weights[loss_name] = base_weight * progress
709:
                     elif current_stage < 2:</pre>
710:
                         scheduled_weights[loss_name] = 0.0
711:
712:
                 elif schedule_type == "late_ramp":
713:
                     # Ramp up in Stage 3 (voxel loss)
714:
                     if current_stage == 3:
715:
                         progress = stage_epoch / max(total_stage_epochs, 1)
716:
                         scheduled_weights[loss_name] = base_weight * progress
717:
                     elif current_stage < 3:</pre>
718:
                         scheduled_weights[loss_name] = 0.0
719:
720:
                 elif schedule_type == "mid_ramp":
721:
                     # Activate mid-training (latent consistency)
722:
                     if current_stage >= 2:
723:
                         if current_stage == 2:
724:
                             progress = min(stage_epoch / (total_stage_epochs * 0.5), 1.0)
725:
                             scheduled_weights[loss_name] = base_weight * progress
726:
                         else: # Stage 3
727:
                             scheduled_weights[loss_name] = base_weight
728:
                     else:
                         scheduled_weights[loss_name] = 0.0
729:
730:
731:
                 elif schedule_type == "delayed_ramp":
732:
                     # FIX: gentler ramp for graph constraints
733:
                     if current_epoch >= self.config.graph_start_epoch:
734:
                         epochs_since_start = current_epoch - self.config.graph_start_epoch
                         ramp_duration = 50 # FIX: slower ramp (was 20)
735:
736:
                         progress = min(epochs_since_start / ramp_duration, 1.0)
737:
                         scheduled_weights[loss_name] = self.config.graph_end_weight * progress
738:
739:
                         scheduled_weights[loss_name] = 0.0
740:
741:
             return scheduled_weights
```

\_\_\_\_\_

#### ■ File: training\trainer.py

```
_______
 1: """
 2: Dynamic Multi-stage training system with adaptive curriculum learning
 3: Implements novel training strategies: dynamic stage transitions, topology-aware scheduling,
 4: multi-objective optimization, and cross-modal consistency learning
 5: """
 6:
 7: import torch
 8: import torch.nn.utils
 9:
10: # training/trainer.py - Fixed AMP imports
11: from torch.amp import autocast, GradScaler
12: import time
 13: import numpy as np
14: import random
15: from pathlib import Path
16: from tqdm import tqdm
 17: from typing import Dict, List, Optional, Tuple
18: from collections import deque
 20: from .losses import ResearchGradeLoss, LossScheduler
 21: from config import DEFAULT_TRAINING_CONFIG, DEFAULT_LOSS_CONFIG, StageTransitionCriteria
 22:
```

```
23:
24: class CurriculumState:
25:
        """Tracks curriculum learning state and metrics"""
26:
        def __init__(self, config):
27:
28:
            self.config = config
29:
30:
            # Loss history for plateau detection
            self.loss_history = {
31:
                "stage1": deque(maxlen=config.plateau_detection_window * 2),
32:
                "stage2": deque(maxlen=config.plateau_detection_window * 2),
33:
                "stage3": deque(maxlen=config.plateau_detection_window * 2),
34:
35:
36:
37:
            # Component loss tracking
38:
            self.component_losses = {
                "segmentation": deque(maxlen=20),
39:
40:
                "dice": deque(maxlen=20),
                "polygon": deque(maxlen=20),
41:
42:
                "voxel": deque(maxlen=20),
43:
                "topology": deque(maxlen=20),
44:
                "latent_consistency": deque(maxlen=20),
45:
                "graph": deque(maxlen=20),
46:
            }
47:
            # Gradient magnitude tracking for dynamic weighting
48:
49:
            self.gradient_norms = {
50:
                name: deque(maxlen=config.gradient_norm_window)
51:
                for name in self.component_losses.keys()
52:
            }
53:
54:
            # Stage transition tracking
55:
            self.epochs_without_improvement = 0
            self.best_val_loss = float("inf")
56:
57:
            self.stage_transition_epochs = []
58:
59:
            # Dynamic weights history
60:
            self.weight_history = []
61:
62:
        def update_loss_history(self, stage: str, val_loss: float):
63:
            """Update validation loss history for plateau detection"""
64:
            if stage in self.loss history:
65:
                self.loss_history[stage].append(val_loss)
66:
67:
            # Update improvement tracking
68:
            if val_loss < self.best_val_loss:</pre>
69:
                self.best_val_loss = val_loss
70:
                self.epochs_without_improvement = 0
71:
            else:
72:
                self.epochs_without_improvement += 1
73:
74:
        def update_component_losses(self, loss_components: Dict[str, float]):
75:
            """Update individual loss component history"""
76:
            for name, loss_val in loss_components.items():
77:
                if name in self.component_losses:
78:
                    self.component_losses[name].append(loss_val)
79:
80:
        def should_transition(self, current_stage: int) -> bool:
81:
            """Check if should transition to next stage"""
82:
            if current_stage == 1:
83:
                val_losses = list(self.loss_history["stage1"])
84:
                return StageTransitionCriteria.should_transition_from_stage1(
85:
                    [], val_losses, self.config
86:
87:
            elif current_stage == 2:
88:
                polygon_losses = list(self.component_losses["polygon"])
                return StageTransitionCriteria.should_transition_from_stage2(
89:
                    polygon_losses, self.config
90:
91:
92:
93:
            return False
94:
95:
```

```
96: class AdaptiveMultiStageTrainer:
 97:
 98:
        Advanced multi-stage trainer with dynamic curriculum learning:
 99:
         - Adaptive stage transitioning based on performance plateaus
        - Topology-aware loss scheduling
100:
101:
        - Multi-objective optimization with dynamic weighting
102:
        - Cross-modal latent consistency learning
103:
         - Graph-based topology constraints
104:
105:
106:
         # Class constant for rolling checkpoint path
107:
        ROLLING_CHECKPOINT = "latest_checkpoint.pth"
108:
109:
         def __init__(self, model, train_loader, val_loader, device=None, config=None):
110:
             if config is None:
111:
                 config = DEFAULT_TRAINING_CONFIG
112:
113:
             self.model = model.to(device or config.device)
             self.train_loader = train_loader
114:
115:
             self.val_loader = val_loader
116:
             self.device = device or config.device
117:
            self.config = config
118:
119:
             # Initialize curriculum state
             self.curriculum_state = CurriculumState(config.curriculum)
120:
             self.loss_scheduler = LossScheduler(config.curriculum)
121:
122:
123:
             # Training state tracking for resume functionality
124:
            self.current_stage = 1
125:
            self.current_epoch = 0
126:
            self.global_epoch = 0
            self.stage_epoch = 0
127:
128:
            self.stage_start_time = None
129:
            self.epoch_times = []
130:
131:
             # Add AMP and optimization settings - Updated for new PyTorch API
132:
             self.use_amp = getattr(config, "use_mixed_precision", True)
             self.scaler = GradScaler("cuda", enabled=self.use_amp)
133:
134:
             self.accumulation_steps = getattr(config, "accumulation_steps", 1)
             self.dvx_step_freq = getattr(config, "dvx_step_freq", 1)
135:
136:
             self.voxel_size_stage = getattr(config, "voxel_size_stage", None)
             self.image_size_stage = getattr(config, "image_size_stage", None)
137:
138:
            self.\_step = 0
139:
140:
             # Enhanced optimizers with better hyperparameters
141:
             self.optimizer_2d = torch.optim.AdamW(
142:
                list(self.model.encoder.parameters())
143:
                 + list(self.model.seg_head.parameters())
144:
                 + list(self.model.attr_head.parameters())
                 + list(self.model.sdf_head.parameters()),
145:
146:
                 lr=config.stage1_lr,
147:
                 weight_decay=config.stagel_weight_decay,
148:
                 betas=(0.9, 0.999),
149:
             )
150:
            self.optimizer_dvx = torch.optim.AdamW(
151:
152:
                 self.model.dvx.parameters(),
153:
                 lr=config.stage2_lr,
154:
                 weight_decay=config.stage2_weight_decay,
155:
                 betas=(0.9, 0.999),
156:
             )
157:
158:
            self.optimizer_full = torch.optim.AdamW(
159:
                 self.model.parameters(),
160:
                 lr=config.stage3_lr,
161:
                 weight_decay=config.stage3_weight_decay,
162:
                 betas=(0.9, 0.999),
163:
             )
164:
165:
             # Enhanced learning rate schedulers
             if config.use_cosine_restarts:
166:
167:
                 self.scheduler_2d = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
168:
                     self.optimizer_2d, T_0=20, T_mult=1
```

```
169:
170:
                 self.scheduler_dvx = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
171:
                     self.optimizer_dvx, T_0=15, T_mult=1
172:
                 self.scheduler_full = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
173:
174:
                     self.optimizer_full, T_0=30, T_mult=1
175:
                 )
176:
            else:
                 self.scheduler_2d = torch.optim.lr_scheduler.CosineAnnealingLR(
177:
                     self.optimizer_2d, T_max=config.max_stage1_epochs
178:
179:
180:
                 self.scheduler_dvx = torch.optim.lr_scheduler.CosineAnnealingLR(
181:
                     self.optimizer_dvx, T_max=config.max_stage2_epochs
182:
                 self.scheduler_full = torch.optim.lr_scheduler.CosineAnnealingLR(
183:
184:
                     self.optimizer_full, T_max=config.max_stage3_epochs
185:
186:
             # Enhanced loss function with dynamic weighting
187:
188:
             base_loss_kwargs = {
189:
                 k: v
190:
                 for k, v in DEFAULT_LOSS_CONFIG.__dict__.items()
191:
                 if k != "enable_dynamic_weighting"
192:
             }
            self.loss_fn = ResearchGradeLoss(
193:
194:
                 **base_loss_kwargs,
195:
                 enable_dynamic_weighting=bool(config.curriculum.use_gradnorm),
196:
                 gradnorm_alpha=float(config.curriculum.gradnorm_alpha),
197:
                 device=self.device,
198:
             )
199:
200:
             self.history = {
                 "stagel": {"train_loss": [], "val_loss": [], "component_losses": []},
201:
                 "stage2": {"train_loss": [], "val_loss": [], "component_losses": []},
202:
                 "stage3": {"train_loss": [], "val_loss": [], "component_losses": []},
203:
204:
                 "stage_transitions": [],
205:
                 "dynamic_weights": [],
206:
                 "curriculum_events": [],
207:
             }
208:
209:
         def _get_eta_string(self, epoch, total_epochs):
              """Calculate and format ETA string"""
210:
211:
             if len(self.epoch_times) == 0:
212:
                return "ETA: calculating..."
213:
214:
             avg_epoch_time = sum(self.epoch_times) / len(self.epoch_times)
215:
             remaining_epochs = total_epochs - epoch - 1
216:
             eta_seconds = avg_epoch_time * remaining_epochs
217:
218:
            if eta_seconds < 60:
                return f"ETA: {int(eta_seconds)}s"
219:
220:
             elif eta_seconds < 3600:</pre>
221:
                return f"ETA: {int(eta_seconds // 60)}m {int(eta_seconds % 60)}s"
222:
            else:
223:
                 hours = int(eta_seconds // 3600)
                 minutes = int((eta_seconds % 3600) // 60)
224:
                 return f"ETA: {hours}h {minutes}m"
225:
226:
         def _get_shared_parameters(self):
227:
              ""Get shared parameters for GradNorm weighting"""
228:
229:
             # Return encoder parameters as shared across tasks
230:
             return list(self.model.encoder.parameters())
231:
232:
         def _update_loss_weights_for_curriculum(
             self, current_stage: int, stage_epoch: int, total_stage_epochs: int
233:
234:
             """Update loss weights based on curriculum schedule"""
235:
236:
             base_weights = {
237:
                 "seg": self.loss_fn.initial_weights["seg"],
                 "dice": self.loss_fn.initial_weights["dice"],
238:
239:
                 "sdf": self.loss_fn.initial_weights["sdf"],
                 "attr": self.loss_fn.initial_weights["attr"],
240:
241:
                 "polygon": self.loss_fn.initial_weights["polygon"],
```

```
242:
                 "voxel": self.loss_fn.initial_weights["voxel"],
243:
                 "topology": self.loss_fn.initial_weights["topology"],
244:
                 "latent_consistency": self.loss_fn.initial_weights["latent_consistency"],
                 "graph": self.loss_fn.initial_weights["graph"],
245:
             }
246:
247:
248:
            scheduled_weights = self.loss_scheduler.get_scheduled_weights(
249:
                 current_stage,
250:
                 self.global_epoch,
251:
                 stage_epoch,
252:
                 total_stage_epochs,
253:
                 base_weights,
254:
255:
256:
            self.loss_fn.update_loss_weights(scheduled_weights)
257:
258:
             # Log weight changes
259:
             self.history["dynamic_weights"].append(
260:
                {
261:
                     "epoch": self.global_epoch,
262:
                     "stage": current_stage,
263:
                     "weights": scheduled_weights.copy(),
264:
                 }
265:
266:
267:
         def _train_epoch(self, mode="stage1"):
268:
              """Enhanced training epoch with AMP, gradient accumulation, and DVX gating"""
269:
             self.model.train()
270:
            total_loss = 0
271:
            component_loss_sums = {}
272:
273:
             # Select appropriate optimizer based on mode
274:
            if mode == "stage1":
275:
                 optimizer = self.optimizer_2d
             elif mode == "stage2":
276:
                optimizer = self.optimizer_dvx
278:
             else: # stage3
                optimizer = self.optimizer_full
279:
280:
281:
             # Progress bar for training batches
282:
             train_pbar = tqdm(
283:
                 self.train_loader, desc=f"Training {mode.upper()}", leave=False, ncols=120
284:
             )
285:
286:
             batch_count = 0
287:
             epoch_start_time = time.time()
288:
289:
             for batch_idx, batch in enumerate(train_pbar):
290:
                 self._step += 1
291:
                 batch = {
                     k: v.to(self.device, non_blocking=True) if torch.is_tensor(v) else v
292:
293:
                     for k, v in batch.items()
294:
295:
296:
                 # Gate heavy DVX/extrusion: only run full forward every dvx_step_freq steps
297:
                 run_full_geometric = (self.dvx_step_freq <= 1) or (</pre>
298:
                     (self._step % self.dvx_step_freq) == 0
299:
300:
                 # First-batch profiling (optional timing helper)
301:
302:
                 if batch_idx == 0 and self.global_epoch == 0:
303:
                     torch.cuda.synchronize()
304:
                     t0 = time.time()
305:
                     with autocast("cuda", enabled=self.use_amp):
                         out = self.model(batch["image"], run_full_geometric=True)
306:
307:
                         # Prepare targets for loss computation
308:
                         targets = self._prepare_targets(batch, mode)
309:
                         shared_params = (
310:
                             self._get_shared_parameters()
                             if self.config.curriculum.use_gradnorm
311:
312:
                             else None
313:
314:
                         1, _ = self.loss_fn(
```

```
315:
                              out, targets, shared params, run full geometric=True
316:
                         )
317:
                     torch.cuda.synchronize()
                     print(f"First-batch forward+loss time: {time.time() - t0:.3f}s")
318:
319:
320:
                 with autocast("cuda", enabled=self.use_amp):
321:
                     # Forward pass with geometric gating
322:
                     predictions = self.model(
323:
                         batch["image"], run_full_geometric=run_full_geometric
324:
325:
326:
                     # Add latent embeddings if model supports it
                     if hasattr(self.model, "get_latent_embeddings"):
327:
                         latent_2d, latent_3d = self.model.get_latent_embeddings(
328:
329:
                             batch["image"]
330:
                         predictions["latent_2d_embedding"] = latent_2d
331:
332:
                         predictions["latent_3d_embedding"] = latent_3d
333:
334:
                     # Prepare targets based on training mode
335:
                     targets = self._prepare_targets(batch, mode)
336:
337:
                     # Get shared parameters for dynamic weighting
338:
                     shared params = (
339:
                         self._get_shared_parameters()
340:
                         if self.config.curriculum.use_gradnorm
341:
                         else None
342:
343:
344:
                     # Compute loss with dynamic weighting and geometric gating
345:
                     loss, loss_components = self.loss_fn(
346:
                         predictions,
347:
                         targets,
348:
                         shared_params,
349:
                         run_full_geometric=run_full_geometric,
350:
351:
352:
                     # Scale loss for gradient accumulation
353:
                     loss = loss / self.accumulation_steps
354:
355:
                 # Scale and backward pass
356:
                 self.scaler.scale(loss).backward()
357:
                 # Gradient accumulation step
358:
359:
                 if ((batch_idx + 1) % self.accumulation_steps) == 0:
360:
                     # Unscale and clip gradients
361:
                     self.scaler.unscale_(optimizer)
362:
363:
                     # Apply gradient clipping
364:
                     torch.nn.utils.clip_grad_norm_(
365:
                         self.model.parameters(), self.config.grad_clip_norm
366:
367:
368:
                     # Optimizer step with scaler
369:
                     self.scaler.step(optimizer)
370:
                     self.scaler.update()
371:
                     optimizer.zero_grad()
372:
373:
                 current_loss = loss.item() * self.accumulation_steps
374:
                 total_loss += current_loss
375:
376:
                 # Track component losses
377:
                 for name, component_loss in loss_components.items():
378:
                     if name != "total":
379:
                         loss_val = (
380:
                             component_loss.item()
381:
                             if torch.is_tensor(component_loss)
                             else component_loss
382:
383:
384:
                         if name not in component loss sums:
385:
                             component_loss_sums[name] = 0
386:
                         component_loss_sums[name] += loss_val
387:
```

```
388:
                 batch count += 1
389:
390:
                 # Occasional lightweight logging
391:
                 if (batch_idx + 1) % 50 == 0:
                     elapsed = time.time() - epoch_start_time
392:
393:
                     avg_time_per_batch = elapsed / (batch_idx + 1)
394:
                     current_weights = {
395:
                         k: f''(v:.3f)'' for k, v in self.loss_fn.weights.items() if v > 0.001
396:
397:
                     print(
                          f"[Epoch {self.global_epoch}] Batch {batch_idx+1}/{len(self.train_loader)} | "
398:
399:
                          f"avg batch {avg_time_per_batch:.3f}s | loss {total_loss/batch_count:.4f}"
400:
401:
402:
                 # Update progress bar
403:
                 current_weights = {
                     k: f"{v:.3f}" for k, v in self.loss_fn.weights.items() if v > 0.001
404:
405:
406:
                 train_pbar.set_postfix(
407:
                     {
408:
                          "loss": f"{current_loss:.4f}",
409:
                          "weights": str(current_weights)[:50] + "..."
410:
                          if len(str(current_weights)) > 50
411:
                          else str(current_weights),
412:
                     }
413:
                 )
414:
415:
             # Final epoch timing
416:
             epoch_time = time.time() - epoch_start_time
417:
             avg_loss = total_loss / batch_count
418:
             print(
419:
                 f"Epoch {self.global_epoch} finished in {epoch_time/60:.2f} min. avg loss: {avg_loss:.4f}"
420:
421:
422:
             # Average component losses
423:
             avg_component_losses = {
424:
                 name: loss_sum / batch_count
425:
                 for name, loss_sum in component_loss_sums.items()
426:
             }
427:
428:
             return avg_loss, avg_component_losses
429:
430:
         def _prepare_targets(self, batch, mode):
              """Prepare targets based on training mode"""
431:
432:
             if mode == "stage1":
433:
                 return {"mask": batch["mask"], "attributes": batch["attributes"]}
434:
             elif mode == "stage2":
435:
                 return {
436:
                      "polygons_gt": {
437:
                          "polygons": batch["polygons_gt"]["polygons"].to(self.device),
                          "valid_mask": batch["polygons_gt"]["valid_mask"].to(self.device),
438:
439:
                     }
440:
                 }
441:
             else: # stage3
442:
                 return {
                      "mask": batch["mask"],
443:
                      "attributes": batch["attributes"],
444:
445:
                      "voxels_gt": batch["voxels_gt"],
                      "polygons_gt": {
446:
                          "polygons": batch["polygons_gt"]["polygons"].to(self.device),
447:
448:
                          "valid_mask": batch["polygons_gt"]["valid_mask"].to(self.device),
449:
                     },
450:
                 }
451:
         def _validate(self, mode="stage1"):
452:
453:
              """Enhanced validation with detailed metrics and AMP support"""
454:
             self.model.eval()
455:
             total_loss = 0
456:
             component_loss_sums = {}
457:
             val_pbar = tqdm(
458:
                 self.val_loader, desc=f"Validating {mode.upper()}", leave=False, ncols=120
459:
460:
```

```
461:
462:
            batch_count = 0
463:
             with torch.no_grad():
464:
                 for batch in val_pbar:
465:
                     batch = {
466:
                         k: v.to(self.device, non_blocking=True) if torch.is_tensor(v) else v
467:
                         for k, v in batch.items()
468:
469:
                     with autocast("cuda", enabled=self.use_amp):
470:
                         # Always run full geometric computation during validation
471:
472:
                         predictions = self.model(batch["image"], run_full_geometric=True)
473:
474:
                         # Add latent embeddings if available
475:
                         if hasattr(self.model, "get_latent_embeddings"):
476:
                             latent_2d, latent_3d = self.model.get_latent_embeddings(
477:
                                 batch["image"]
478:
                             predictions["latent_2d_embedding"] = latent_2d
479:
480:
                             predictions["latent_3d_embedding"] = latent_3d
481:
482:
                         targets = self._prepare_targets(batch, mode)
483:
484:
                         loss, loss_components = self.loss_fn(
485:
                             predictions, targets, run_full_geometric=True
486:
                         )
487:
488:
                     current_loss = loss.item()
489:
                     total_loss += current_loss
490:
491:
                     # Track component losses
492:
                     for name, component_loss in loss_components.items():
493:
                         if name != "total":
                             loss_val = (
494:
                                 component_loss.item()
495:
496:
                                 if torch.is_tensor(component_loss)
497:
                                 else component_loss
498:
499:
                             if name not in component_loss_sums:
500:
                                 component_loss_sums[name] = 0
501:
                             component_loss_sums[name] += loss_val
502:
503:
                     batch count += 1
                     val_pbar.set_postfix({"loss": f"{current_loss:.4f}"})
504:
505:
506:
             avg_component_losses = {
507:
                 name: loss_sum / batch_count
508:
                 for name, loss_sum in component_loss_sums.items()
509:
             }
510:
             return total_loss / batch_count, avg_component_losses
511:
512:
513:
         def train_stage_adaptive(self, stage: int, max_epochs: int, min_epochs: int):
514:
515:
             Train a stage with adaptive termination based on curriculum learning
516:
517:
            Arqs:
518:
                stage: Stage number (1, 2, 3)
519:
                 max_epochs: Maximum epochs for this stage
520:
                 min_epochs: Minimum epochs before considering transition
521:
             print("=" * 60)
522:
523:
             print(f"STAGE {stage}: Adaptive Training with Dynamic Curriculum")
524:
             print("=" * 60)
525:
526:
             self.current_stage = stage
527:
            self.stage_start_time = time.time()
528:
529:
             # Only reset if not resuming
530:
             if not hasattr(self, "epoch_times") or self.epoch_times is None:
531:
                 self.epoch_times = []
532:
533:
             start_epoch = int(self.stage_epoch or 0)
```

```
534:
535:
             # Set parameter gradients for current stage
536:
             self._configure_stage_parameters(stage)
537:
             mode_name = f"stage{stage}"
538:
539:
540:
             for epoch in range(start_epoch, max_epochs):
541:
                 epoch_start_time = time.time()
542:
                 self.stage_epoch = epoch
                 self.global_epoch += 1
543:
544:
545:
                 # Update loss weights based on curriculum
546:
                 self._update_loss_weights_for_curriculum(stage, epoch, max_epochs)
547:
548:
                 print(
549:
                     f"\nStage {stage} - Epoch {epoch+1}/{max_epochs} (Global: {self.global_epoch})"
550:
                 )
551:
                 # Training and validation
552:
553:
                 train_loss, train_components = self._train_epoch(mode_name)
554:
                 val_loss, val_components = self._validate(mode_name)
555:
556:
                 # Record epoch time
557:
                 epoch_time = time.time() - epoch_start_time
                 self.epoch_times.append(epoch_time)
558:
559:
560:
                 if len(self.epoch_times) > 10:
561:
                     self.epoch_times.pop(0)
562:
563:
                 # Update curriculum state
564:
                 self.curriculum_state.update_loss_history(mode_name, val_loss)
565:
                 self.curriculum_state.update_component_losses(val_components)
566:
                 # Store training history
567:
                 self.history[mode_name]["train_loss"].append(train_loss)
568:
569:
                 self.history[mode_name]["val_loss"].append(val_loss)
570:
                 self.history[mode_name]["component_losses"].append(val_components)
571:
572:
                 # Update learning rate
573:
                 if stage == 1:
574:
                     self.scheduler_2d.step()
575:
                 elif stage == 2:
                    self.scheduler_dvx.step()
577:
                 else:
578:
                     self.scheduler_full.step()
579:
580:
                 # Display comprehensive results
581:
                 self._display_epoch_results(
582:
                     epoch,
583:
                     max_epochs,
584:
                     train loss,
585:
                     val_loss,
586:
                     train_components,
587:
                     val components,
588:
                     epoch_time,
589:
590:
                 \ensuremath{\sharp} Check for adaptive stage transition
591:
592:
                 if epoch >= min_epochs:
                     should_transition = self.curriculum_state.should_transition(stage)
593:
594:
                     if should_transition:
595:
596:
                             f"\n? ADAPTIVE TRANSITION: Stage {stage} converged after {epoch+1} epochs"
597:
598:
                         print(
599:
                                  Detected performance plateau - transitioning to next stage"
600:
601:
602:
                          self.history["stage_transitions"].append(
603:
                             {
604:
                                  "from_stage": stage,
605:
                                  "epoch": epoch + 1,
606:
                                  "global_epoch": self.global_epoch,
```

```
607:
                                   "reason": "performance_plateau",
608:
                               }
609:
                          )
610:
                          self.history["curriculum_events"].append(
611:
612:
                               {
613:
                                   "type": "stage_transition",
                                   "stage": stage,
614:
615:
                                   "epoch": self.global_epoch,
                                   "details": f"Converged after {epoch+1} epochs",
616:
617:
618:
619:
                          break
620:
621:
                  # Save rolling checkpoint
622:
                  if (epoch + 1) % self.config.checkpoint_freq == 0:
                      self._save_rolling_checkpoint()
623:
624:
              print(f"\nStage {stage} completed after {epoch+1} epochs")
625:
626:
627:
         def _configure_stage_parameters(self, stage: int):
628:
               ""Configure which parameters require gradients for each stage"""
629:
              # First freeze everything
630:
              for param in self.model.parameters():
631:
                  param.requires_grad = False
632:
633:
              if stage == 1:
                  # Stage 1: Segmentation + Attributes (2D only)
634:
635:
                  for param in self.model.encoder.parameters():
636:
                      param.requires_grad = True
637:
                  for param in self.model.seg_head.parameters():
638:
                      param.requires_grad = True
639:
                  for param in self.model.attr_head.parameters():
640:
                     param.requires_grad = True
641:
                  for param in self.model.sdf_head.parameters():
642:
                      param.requires_grad = True
643:
644:
             elif stage == 2:
645:
                  # Stage 2: DVX training (polygon fitting) - keep encoder frozen initially
646:
                  for param in self.model.dvx.parameters():
647:
                      param.requires_grad = True
648:
                  # Optionally unfreeze encoder in later epochs
649:
                  if self.stage_epoch > 10:
650:
                      for param in self.model.encoder.parameters():
651:
                          param.requires_grad = True
652:
653:
             else: # stage == 3
654:
                  # Stage 3: End-to-end fine-tuning (all parameters)
655:
                  for param in self.model.parameters():
656:
                      param.requires_grad = True
657:
658:
         def _display_epoch_results(
659:
             self,
660:
             epoch: int,
661:
             total_epochs: int,
             train_loss: float,
662:
             val_loss: float,
663:
664:
             train_components: Dict,
665:
             val_components: Dict,
666:
             epoch_time: float,
667:
668:
             """Display comprehensive epoch results with curriculum information"""
669:
             eta_str = self._get_eta_string(epoch, total_epochs)
670:
             \label{eq:print}  \texttt{print}(\texttt{f"Train Loss: } \{\texttt{train\_loss:.4f}\}, \ \texttt{Val Loss: } \{\texttt{val\_loss:.4f}\}")
671:
672:
             print(f"Epoch time: {epoch_time:.1f}s, {eta_str}")
673:
674:
              # Show significant component losses
675:
              significant_components = {
676:
                 k: v
677:
                  for k, v in val_components.items()
678:
                  if v > 0.01
679:
                  and k
```

```
680:
                 in [
681:
                      "sea",
682:
                     "dice",
683:
                      "polygon",
684:
                      "voxel",
685:
                      "topology",
686:
                      "latent_consistency",
687:
                      "graph",
688:
                 ]
             }
689:
             if significant_components:
690:
691:
                 comp_str = ", ".join(
692:
                     [f"{k}: {v:.3f}" for k, v in significant_components.items()]
693:
694:
                 print(f"Components: {comp_str}")
695:
696:
             # Show current loss weights for active components
697:
             active\_weights = \{k: v for k, v in self.loss\_fn.weights.items() if v > 0.001\}
698:
             if active_weights:
699:
                 weight_str = ", ".join([f"{k}: {v:.3f}" for k, v in active_weights.items()])
700:
                 print(f"Weights: {weight_str}")
701:
702:
             # Show curriculum status
703:
             plateau_epochs = self.curriculum_state.epochs_without_improvement
704:
             if plateau epochs > 0:
705:
                 print(f"Plateau: {plateau_epochs} epochs without improvement")
706:
707:
         def _save_rolling_checkpoint(self):
708:
             """Enhanced checkpoint saving with curriculum state, RNG state, and scaler state"""
709:
             checkpoint = {
                 "model_state_dict": self.model.state_dict(),
710:
711:
                 "optimizer_2d_state_dict": self.optimizer_2d.state_dict(),
                 "optimizer_dvx_state_dict": self.optimizer_dvx.state_dict(),
712:
                 "optimizer_full_state_dict": self.optimizer_full.state_dict(),
713:
                 "scheduler_2d_state_dict": self.scheduler_2d.state_dict(),
714:
715:
                 "scheduler_dvx_state_dict": self.scheduler_dvx.state_dict(),
                 "scheduler_full_state_dict": self.scheduler_full.state_dict(),
716:
717:
                 "scaler_state_dict": self.scaler.state_dict(),  # Add AMP scaler state
718:
                 "loss_fn_state": {
                      "weights": self.loss_fn.weights,
719:
720:
                      "initial_weights": self.loss_fn.initial_weights,
721:
722:
                 "history": self.history,
723:
                 "config": self.config,
724:
                 "current_stage": self.current_stage,
725:
                 "current_epoch": self.current_epoch,
726:
                 "global_epoch": self.global_epoch,
727:
                 "stage_epoch": self.stage_epoch,
                 "epoch_times": self.epoch_times,
728:
729:
                 "step_counter": self._step,  # Save step counter for DVX gating
                 "curriculum_state": {
730:
731:
                      "loss_history": dict(self.curriculum_state.loss_history),
732:
                      "component_losses": dict(self.curriculum_state.component_losses),
733:
                      "epochs_without_improvement": self.curriculum_state.epochs_without_improvement,
734:
                      "best_val_loss": self.curriculum_state.best_val_loss,
                      "stage\_transition\_epochs": self.curriculum\_state.stage\_transition\_epochs,\\
735:
                 },
736:
                 "rng_state": {
737:
738:
                      "torch": torch.get_rng_state(),
                      "cuda": torch.cuda.get_rng_state_all()
739:
740:
                     if torch.cuda.is_available()
741:
                     else None,
742:
                      "numpy": np.random.get_state(),
                      "python": random.getstate(),
743:
                 },
744:
745:
             }
746:
747:
             checkpoint_path = self.ROLLING_CHECKPOINT
748:
             torch.save(checkpoint, checkpoint_path)
749:
             print(f"Rolling checkpoint saved: {checkpoint_path}")
750:
751:
         def load checkpoint(self, filename):
752:
              """Enhanced checkpoint loading with curriculum state restoration and device handling"""
```

```
753:
             checkpoint = torch.load(filename, map location=self.device)
754:
755:
             self.model.load_state_dict(checkpoint["model_state_dict"])
756:
             self.optimizer_2d.load_state_dict(checkpoint["optimizer_2d_state_dict"])
757:
             self.optimizer_dvx.load_state_dict(checkpoint["optimizer_dvx_state_dict"])
758:
             self.optimizer_full.load_state_dict(checkpoint["optimizer_full_state_dict"])
759:
760:
             # Load scaler state for AMP
             if "scaler_state_dict" in checkpoint:
761:
                 self.scaler.load_state_dict(checkpoint["scaler_state_dict"])
762:
763:
764:
             # Safer scheduler loading
765:
             for sched_key, sched_obj in [
                 ("scheduler_2d_state_dict", self.scheduler_2d),
766:
                 ("scheduler_dvx_state_dict", self.scheduler_dvx),
767:
768:
                 ("scheduler_full_state_dict", self.scheduler_full),
             ]:
769:
770:
                 if sched_key in checkpoint:
771:
                     sched_obj.load_state_dict(checkpoint[sched_key])
772:
773:
             # Load loss weights with proper device handling
774:
             if "loss_fn_state" in checkpoint:
775:
                 loaded_weights = checkpoint["loss_fn_state"]["weights"]
776:
                 if isinstance(loaded weights, dict):
777:
                     self.loss fn.weights = {
778:
                         k: (v.to(self.device) if torch.is_tensor(v) else v)
779:
                         for k, v in loaded_weights.items()
                     }
780:
781:
                 else:
782:
                     self.loss_fn.weights = loaded_weights
783:
                 self.loss_fn.initial_weights = checkpoint["loss_fn_state"][
784:
                      "initial_weights"
785:
786:
             if "history" in checkpoint:
787:
788:
                 self.history = checkpoint["history"]
789:
790:
             # Restore training state
791:
             if "current_stage" in checkpoint:
792:
                 self.current_stage = checkpoint["current_stage"]
793:
             if "current_epoch" in checkpoint:
794:
                 self.current_epoch = checkpoint["current_epoch"]
795:
             if "global_epoch" in checkpoint:
796:
                 self.global_epoch = checkpoint["global_epoch"]
797:
             if "stage_epoch" in checkpoint:
798:
                 self.stage_epoch = checkpoint["stage_epoch"]
799:
             if "epoch_times" in checkpoint:
800:
                 self.epoch_times = checkpoint["epoch_times"]
             if "step_counter" in checkpoint:
801:
802:
                 self._step = checkpoint["step_counter"]
803:
804:
             # Restore curriculum state
805:
             if "curriculum_state" in checkpoint:
806:
                 cs = checkpoint["curriculum_state"]
807:
                 for key, history in cs["loss_history"].items():
                     self.curriculum_state.loss_history[key] = deque(
808:
809:
                         history, maxlen=self.config.curriculum.plateau_detection_window * 2
810:
811:
                 for key, history in cs["component_losses"].items():
812:
                     self.curriculum_state.component_losses[key] = deque(history, maxlen=20)
813:
                 self.curriculum_state.epochs_without_improvement = cs.get(
814:
                     "epochs_without_improvement", 0
815:
816:
                 self.curriculum_state.best_val_loss = cs.get("best_val_loss", float("inf"))
                 self.curriculum_state.stage_transition_epochs = cs.get(
817:
818:
                     "stage_transition_epochs", []
819:
                 )
820:
821:
             # Restore RNG states
822:
             if "rng_state" in checkpoint:
823:
                 rs = checkpoint["rng_state"]
824:
825:
                 # --- Torch RNG (CPU) ---
```

```
826:
                 trv:
827:
                      torch_state = rs.get("torch", None)
828:
                     if torch_state is not None:
829:
                          # If it's already a torch tensor with uint8 dtype, use directly
                          if torch.is_tensor(torch_state) and torch_state.dtype == torch.uint8:
830:
831:
                              torch.set_rng_state(torch_state)
                          else:
832:
833:
                              # Convert lists / numpy arrays / other tensors to uint8 torch tensor
                              torch.set_rng_state(torch.tensor(torch_state, dtype=torch.uint8))
834:
835:
                 except Exception as e:
                     print(f"Warning: could not restore torch RNG state (\{e\}), skipping.")
836:
837:
838:
                 # --- CUDA RNG (all devices) ---
839:
840:
                     cuda state = rs.get("cuda", None)
841:
                      if cuda_state is not None and torch.cuda.is_available():
842:
                          # cuda_state might be a list of states (one per device)
843:
                          cuda_tensors = []
                          for s in cuda_state:
844:
845:
                              if torch.is_tensor(s) and s.dtype == torch.uint8:
846:
                                  cuda_tensors.append(s)
847:
848:
                                  cuda_tensors.append(torch.tensor(s, dtype=torch.uint8))
849:
                          torch.cuda.set_rng_state_all(cuda_tensors)
850:
                     print(f"Warning: could not restore CUDA RNG state ({e}), skipping.")
851:
852:
                 # --- numpy RNG ---
853:
854:
                 try:
855:
                      if "numpy" in rs and rs["numpy"] is not None:
                         np.random.set_state(rs["numpy"])
856:
857:
                 except Exception as e:
                     \label{eq:print}  \text{print}(\texttt{f"Warning: could not restore numpy RNG state }(\{\texttt{e}\})\text{, skipping."})
858:
859:
                 # --- python random RNG ---
860:
861:
                 try:
862:
                     if "python" in rs and rs["python"] is not None:
863:
                          random.setstate(rs["python"])
864:
                 except Exception as e:
865:
                     print(f"Warning: could not restore python RNG state ({e}), skipping.")
866:
867:
             # Restore DataLoader sampler states if available
             if "dataloader_state" in checkpoint:
869:
                 dl_state = checkpoint["dataloader_state"]
870:
                 if dl_state["train_sampler_state"] is not None and hasattr(
871:
                      self.train_loader.sampler, "__dict__"
872:
                 ):
873:
                      try:
874:
                          self.train_loader.sampler.__dict__.update(
875:
                              dl_state["train_sampler_state"]
876:
877:
                      except Exception:
878:
                         print("Warning: Could not restore train_loader sampler state")
879:
                 if dl_state["val_sampler_state"] is not None and hasattr(
880:
                      self.val_loader.sampler, "__dict__'
881:
                 ):
882:
                      try:
883:
                          self.val_loader.sampler.__dict__.update(
884:
                              dl_state["val_sampler_state"]
885:
886:
                      except Exception:
887:
                          print("Warning: Could not restore val_loader sampler state")
888:
889:
             print(f"Checkpoint loaded: {filename}")
890:
             print(
891:
                 f"Resuming from Stage {self.current_stage}, Global Epoch {self.global_epoch}"
892:
             )
893:
             print(
                 f"Curriculum state restored with {self.curriculum_state.epochs_without_improvement} epochs with
894:
895:
896:
897:
         def train_all_stages(self):
898:
```

```
899:
             Run complete adaptive multi-stage training pipeline
900:
901:
             This is the main entry point that orchestrates the dynamic curriculum learning
902:
             if Path(self.ROLLING_CHECKPOINT).exists():
903:
904:
                 print(f"Found existing checkpoint: {self.ROLLING_CHECKPOINT}")
905:
                 print("Resuming adaptive training from checkpoint...")
906:
                 self.load_checkpoint(self.ROLLING_CHECKPOINT)
907:
                 \verb|print("Starting fresh adaptive training pipeline...")|\\
908:
909:
                 self.current_stage = 1
910:
                 self.current_epoch = 0
911:
                 self.global_epoch = 0
912:
913:
             print("\n" + "=" * 80)
914:
             print("ADAPTIVE MULTI-STAGE TRAINING WITH DYNAMIC CURRICULUM")
915:
             print("Novel Training Strategies:")
916:
             print("? Adaptive Stage Transitioning (Dynamic Curriculum)")
917:
             print("? Topology-aware Loss Scheduling")
918:
             print("? Multi-objective Optimization with Dynamic Weighting")
919:
             print("? Cross-modal Latent Consistency Learning")
920:
             print("? Graph-based Topology Constraints")
921:
             print("=" * 80)
922:
923:
             # Stage 1: Adaptive 2D training
924:
             if self.current_stage <= 1:</pre>
925:
                 print("\n? STAGE 1: Adaptive 2D Segmentation + Attributes Training")
926:
                 self.train_stage_adaptive(
927:
                     stage=1,
928:
                     max_epochs=self.config.max_stage1_epochs,
929:
                     min_epochs=self.config.min_stage1_epochs,
930:
                 self.current_stage = 2
931:
932:
                 self.stage\_epoch = 0
                 print("\nStage 1 completed. Transitioning to Stage 2...")
933:
934:
935:
             # Stage 2: Adaptive DVX training
936:
             if self.current_stage <= 2:</pre>
937:
                 print("\n? STAGE 2: Adaptive DVX Polygon Fitting Training")
938:
                 self.train_stage_adaptive(
939:
                     stage=2,
940:
                     max_epochs=self.config.max_stage2_epochs,
941:
                     min_epochs=self.config.min_stage2_epochs,
942:
                 )
943:
                 self.current stage = 3
944:
                 self.stage\_epoch = 0
945:
                 print("\nStage 2 completed. Transitioning to Stage 3...")
946:
947:
             # Stage 3: Adaptive end-to-end fine-tuning
948:
             if self.current_stage <= 3:
949:
                 print("\n? STAGE 3: Adaptive End-to-End Fine-tuning with Full Loss Suite")
950:
                 self.train_stage_adaptive(
951:
                     stage=3,
952:
                     max_epochs=self.config.max_stage3_epochs,
953:
                     min_epochs=self.config.min_stage3_epochs,
954:
                 print("\nStage 3 completed!")
955:
956:
             print("\n" + "=" * 80)
957:
             print("? ALL ADAPTIVE TRAINING STAGES COMPLETED!")
958:
             print("=" * 80)
959:
960:
961:
             # Generate training report
962:
             self._generate_training_report()
963:
964:
             # Save final model
             self._save_checkpoint("final_adaptive_model.pth")
965:
966:
967:
             # Clean up rolling checkpoint
968:
             if Path(self.ROLLING_CHECKPOINT).exists():
969:
                 Path(self.ROLLING_CHECKPOINT).unlink()
970:
                 print(f"Cleaned up rolling checkpoint: {self.ROLLING_CHECKPOINT}")
971:
```

```
972:
             return self.history
973:
974:
         def _generate_training_report(self):
975:
              """Generate comprehensive training report with curriculum insights"""
             print("\n" + "=" * 60)
976:
977:
             print("ADAPTIVE TRAINING REPORT")
978:
             print("=" * 60)
979:
980:
             # Stage transition summary
981:
             if self.history["stage_transitions"]:
982:
                 print("\n? Stage Transitions:")
                 for transition in self.history["stage_transitions"]:
983:
984:
                         f" ? Stage {transition['from_stage']} ? {transition['from_stage']+1}: "
985:
986:
                         f"Epoch {transition['epoch']} (Global: {transition['global_epoch']})"
987:
                     print(f"
988:
                                 Reason: {transition['reason']}")
989:
990:
             # Dynamic weight evolution
991:
             if self.history["dynamic_weights"]:
992:
                 print(
993:
                     f"\n?? Dynamic Weight Updates: {len(self.history['dynamic_weights'])} updates"
994:
995:
                 final_weights = self.history["dynamic_weights"][-1]["weights"]
                 print(" Final loss weights:")
996:
                 for name, weight in final_weights.items():
997:
998:
                     if weight > 0.001:
                         print(f"
999:
                                      {name}: {weight:.3f}")
1000:
1001:
              # Curriculum events
1002:
              if self.history["curriculum_events"]:
1003:
                  print(
1004:
                      f"\n? Curriculum Events: {len(self.history['curriculum_events'])} events"
1005:
1006:
                  for event in self.history["curriculum_events"][-5:]: # Show last 5 events
1007:
                      print(
1008:
                          f"
                              ? {event['type']} at global epoch {event['epoch']}: {event['details']}"
1009:
1010:
1011:
              # Performance summary
1012:
              print("\n? Final Performance:")
1013:
              for stage_name, data in self.history.items():
1014:
                  if isinstance(data, dict) and "val_loss" in data and data["val_loss"]:
1015:
                      final_loss = data["val_loss"][-1]
1016:
                      best_loss = min(data["val_loss"])
1017:
                      print(
                              ? {stage_name.upper()}: Final={final_loss:.4f}, Best={best_loss:.4f}"
1018:
                          f"
1019:
1020:
1021:
              print("\n? Training completed with novel adaptive curriculum strategies!")
1022:
              print("=" * 60)
1023:
1024:
          def _save_checkpoint(self, filename):
1025:
               """Save final training checkpoint"""
1026:
              checkpoint = {
                  "model_state_dict": self.model.state_dict(),
1027:
1028:
                  "optimizer_2d_state_dict": self.optimizer_2d.state_dict(),
                  "optimizer_dvx_state_dict": self.optimizer_dvx.state_dict(),
1029:
1030:
                  "optimizer_full_state_dict": self.optimizer_full.state_dict(),
                  "scheduler_2d_state_dict": self.scheduler_2d.state_dict(),
1031:
                  "scheduler_dvx_state_dict": self.scheduler_dvx.state_dict(),
1032:
1033:
                  "scheduler_full_state_dict": self.scheduler_full.state_dict(),
                  "scaler_state_dict": self.scaler.state_dict(),
1034:
1035:
                  "loss_fn_state": {
                      "weights": self.loss_fn.weights,
1036:
1037:
                      "initial_weights": self.loss_fn.initial_weights,
1038:
                  },
1039:
                  "history": self.history,
1040:
                  "config": self.config,
1041:
                  "final_stage": self.current_stage,
1042:
                  "total_epochs": self.global_epoch,
1043:
                  "training_complete": True,
1044:
                  "curriculum_summary": {
```

```
"stage_transitions": len(self.history["stage_transitions"]),
1045:
1046:
                      "weight_updates": len(self.history["dynamic_weights"]),
1047:
                      "curriculum_events": len(self.history["curriculum_events"]),
1048:
                  },
1049:
1050:
              torch.save(checkpoint, filename)
1051:
             print(f"Final model saved: {filename}")
1052:
1053:
1054: # Legacy compatibility class
1055: class MultiStageTrainer(AdaptiveMultiStageTrainer):
1056:
1057:
          Legacy wrapper for backward compatibility
1058:
          Redirects to the new adaptive trainer
1059:
1060:
         def __init__(self, *args, **kwargs):
1061:
1062:
              super().__init__(*args, **kwargs)
1063:
              print("Note: Using enhanced AdaptiveMultiStageTrainer with dynamic curriculum")
1064:
1065:
         def train_stage1(self, epochs=None):
1066:
               """Legacy method - redirects to adaptive training"""
1067:
              max_epochs = epochs or self.config.max_stagel_epochs
1068:
              min_epochs = self.config.min_stage1_epochs
1069:
              return self.train_stage_adaptive(1, max_epochs, min_epochs)
1070:
1071:
          def train_stage2(self, epochs=None):
              """Legacy method - redirects to adaptive training"""
1072:
1073:
              max_epochs = epochs or self.config.max_stage2_epochs
1074:
              min_epochs = self.config.min_stage2_epochs
1075:
              return self.train_stage_adaptive(2, max_epochs, min_epochs)
1076:
1077:
          def train_stage3(self, epochs=None):
              """Legacy method - redirects to adaptive training"""
1078:
1079:
              max_epochs = epochs or self.config.max_stage3_epochs
1080:
              min_epochs = self.config.min_stage3_epochs
1081:
              return self.train_stage_adaptive(3, max_epochs, min_epochs)
```

-----

## ■ File: utils\visualization.py

31:

```
______
 1: """
 2: Visualization and utility functions
 3: """
 4:
 5: import matplotlib.pyplot as plt
 6: import numpy as np
 7: import cv2
 8: import torch
 9: from pathlib import Path
 10: from evaluation.metrics import compute_iou
11:
12:
13: def plot_training_history(history, save_path="training_history.png"):
        """Plot training curves for all stages"""
14:
        fig, axes = plt.subplots(1, 3, figsize=(15, 5))
15:
16:
 17:
        for idx, (stage, data) in enumerate(history.items()):
            if isinstance(data, dict) and "train_loss" in data and data["train_loss"]: # Only plot if stage wa
18:
19:
               axes[idx].plot(data["train_loss"], label="Train", linewidth=2)
 20:
               axes[idx].plot(data["val_loss"], label="Validation", linewidth=2)
               axes[idx].set_title(f"{stage.upper()} Training")
 21:
 22:
                axes[idx].set_xlabel("Epoch")
               axes[idx].set_ylabel("Loss")
 23:
 24:
               axes[idx].legend()
 25:
               axes[idx].grid(True, alpha=0.3)
 26:
 27:
        plt.tight_layout()
        plt.savefig(save_path, dpi=300, bbox_inches="tight")
 28:
 29:
        plt.show()
 30:
```

```
32: def plot_curriculum_analysis(history, save_path="curriculum_analysis.png"):
         """Plot curriculum learning analysis including stage transitions and adaptive behavior"""
 34:
         fig, axes = plt.subplots(2, 2, figsize=(15, 10))
 35:
 36:
         # Plot 1: Stage transition timeline
 37:
         if "stage_transitions" in history and history["stage_transitions"]:
 38:
             transitions = history["stage_transitions"]
 39:
             # Extract transition epochs and reasons
 40:
             transition_epochs = [t["epoch"] for t in transitions]
 41:
             transition_stages = [t["from_stage"] + " ? " + t["to_stage"] for t in transitions]
 42:
             transition_reasons = [t.get("reason", "threshold") for t in transitions]
 43:
 44:
 45:
             # Create timeline
 46:
             y_positions = range(len(transition_epochs))
 47:
             colors = ['red' if 'patience' in reason else 'green' for reason in transition_reasons]
 48:
 49:
             axes[0, 0].barh(y_positions, transition_epochs, color=colors, alpha=0.7)
 50:
             axes[0, 0].set_yticks(y_positions)
 51:
             axes[0, 0].set_yticklabels(transition_stages)
 52:
             axes[0, 0].set_xlabel("Epoch")
             axes[0, 0].set_title("Stage Transition Timeline")
 53:
 54:
             axes[0, 0].grid(True, alpha=0.3)
 55:
             # Add legend
 56:
 57:
             axes[0, 0].legend(['Patience-based', 'Threshold-based'], loc='lower right')
 58:
         else:
             axes[0, 0].text(0.5, 0.5, "No stage transitions recorded",
 59:
                            ha='center', va='center', transform=axes[0, 0].transAxes)
 60:
 61:
             axes[0, 0].set_title("Stage Transition Timeline")
 62:
 63:
         # Plot 2: Loss component evolution
         if "dynamic_weights" in history and history["dynamic_weights"]:
 64:
 65:
             weight_data = history["dynamic_weights"]
             epochs = [entry["epoch"] for entry in weight_data]
 66:
 67:
 68:
             # Plot each loss component weight
 69:
             weight_names = list(weight_data[0]["weights"].keys()) if weight_data else []
 70:
             for weight_name in weight_names[:5]: # Limit to top 5 for readability
 71:
                 weights = [entry["weights"].get(weight_name, 0) for entry in weight_data]
 72:
                 if any(w > 0.001 \text{ for } w \text{ in weights}): # Only plot significant weights
 73:
                     axes[0, 1].plot(epochs, weights, label=weight_name, linewidth=2, marker='o', markersize=3)
 74:
             axes[0, 1].set_xlabel("Global Epoch")
 75:
             axes[0, 1].set_ylabel("Loss Weight")
 76:
 77:
             axes[0, 1].set_title("Dynamic Loss Weight Evolution")
 78:
             axes[0, 1].legend()
 79:
             axes[0, 1].grid(True, alpha=0.3)
 80:
         else:
 81:
             axes[0, 1].text(0.5, 0.5, "No dynamic weights recorded",
 82:
                            ha='center', va='center', transform=axes[0, 1].transAxes)
 83:
             axes[0, 1].set_title("Dynamic Loss Weight Evolution")
 84:
 85:
         # Plot 3: Curriculum progress indicators
 86:
         if "curriculum_events" in history and history["curriculum_events"]:
 87:
             events = history["curriculum_events"]
 88:
             event_types = {}
 89:
 90:
             for event in events:
                 event_type = event.get("type", "unknown")
 91:
 92:
                 if event_type not in event_types:
 93:
                     event_types[event_type] = []
 94:
                 event_types[event_type].append(event["epoch"])
 95:
             # Plot event timeline
 96:
 97:
             y_offset = 0
 98:
             for event_type, epochs in event_types.items():
 99:
                 axes[1, 0].scatter(epochs, [y_offset] * len(epochs),
100:
                                    label=event_type, s=50, alpha=0.7)
101:
                 y_offset += 1
102:
103:
             axes[1, 0].set_xlabel("Epoch")
104:
             axes[1, 0].set_ylabel("Event Type")
```

```
105:
             axes[1, 0].set_title("Curriculum Learning Events")
             axes[1, 0].legend()
106:
107:
             axes[1, 0].grid(True, alpha=0.3)
108:
         else:
             axes[1, 0].text(0.5, 0.5, "No curriculum events recorded",
109:
110:
                            ha='center', va='center', transform=axes[1, 0].transAxes)
111:
             axes[1, 0].set_title("Curriculum Learning Events")
112:
         # Plot 4: Stage performance comparison
113:
         stage_names = ["stage1", "stage2", "stage3"]
114:
115:
         stage_performance = {}
116:
117:
         for stage_name in stage_names:
112:
             if stage_name in history and isinstance(history[stage_name], dict):
                 stage_data = history[stage_name]
119:
120:
                 if "val_loss" in stage_data and stage_data["val_loss"]:
                     stage_performance[stage_name] = {
121:
122:
                         "final_loss": stage_data["val_loss"][-1],
                         "best_loss": min(stage_data["val_loss"]),
123:
124:
                         "epochs": len(stage_data["val_loss"])
                     }
125:
126:
127:
         if stage_performance:
128:
             stages = list(stage_performance.keys())
129:
             final_losses = [stage_performance[s]["final_loss"] for s in stages]
130:
             best_losses = [stage_performance[s]["best_loss"] for s in stages]
131:
132:
             x = np.arange(len(stages))
133:
             width = 0.35
134:
135:
             axes[1, 1].bar(x - width/2, final_losses, width, label='Final Loss', alpha=0.8)
136:
             axes[1, 1].bar(x + width/2, best_losses, width, label='Best Loss', alpha=0.8)
137:
             axes[1, 1].set_xlabel("Training Stage")
138:
139:
             axes[1, 1].set_ylabel("Validation Loss")
140:
             axes[1, 1].set_title("Stage Performance Comparison")
141:
             axes[1, 1].set_xticks(x)
142:
             axes[1, 1].set_xticklabels([s.upper() for s in stages])
143:
             axes[1, 1].legend()
144:
             axes[1, 1].grid(True, alpha=0.3)
145:
146:
             # Add epoch count annotations
147:
             for i, stage in enumerate(stages):
                 epochs = stage_performance[stage]["epochs"]
148:
149:
                 axes[1, 1].text(i, max(final_losses) * 0.9, f'{epochs} epochs',
150:
                                ha='center', va='bottom', fontsize=9)
151:
         else:
152:
             axes[1, 1].text(0.5, 0.5, "No stage performance data",
153:
                            ha='center', va='center', transform=axes[1, 1].transAxes)
154:
             axes[1, 1].set_title("Stage Performance Comparison")
155:
156:
        plt.tight_layout()
157:
        plt.savefig(save_path, dpi=300, bbox_inches="tight")
158:
        plt.close()
159:
160:
         print(f"Curriculum analysis saved to {save_path}")
161:
162:
163: def visualize_predictions(image, predictions, targets=None, save_path=None):
164:
         """Visualize model predictions"""
165:
         fig, axes = plt.subplots(2, 3, figsize=(15, 10))
166:
167:
         # Original image
168:
         if len(image.shape) == 4:
169:
             img_np = image[0].permute(1, 2, 0).cpu().numpy()
170:
171:
             img_np = image.permute(1, 2, 0).cpu().numpy()
172:
173:
         axes[0, 0].imshow(img_np)
174:
         axes[0, 0].set_title("Input Image")
175:
         axes[0, 0].axis('off')
176:
177:
         # Predicted segmentation
```

```
if "segmentation" in predictions:
178:
179:
             seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0].cpu().numpy()
180:
             axes[0, 1].imshow(seg_pred, cmap='tab10')
181:
             axes[0, 1].set_title("Predicted Segmentation")
182:
             axes[0, 1].axis('off')
183:
184:
         # Ground truth segmentation (if available)
         if targets and "mask" in targets:
185:
             gt_mask = targets["mask"][0].cpu().numpy()
186:
187:
             axes[0, 2].imshow(gt_mask, cmap='tab10')
             axes[0, 2].set_title("Ground Truth Segmentation")
188:
189:
             axes[0, 2].axis('off')
190:
191:
         # SDF prediction
192:
         if "sdf" in predictions:
193:
             sdf_pred = predictions["sdf"][0, 0].cpu().numpy()
194:
             im = axes[1, 0].imshow(sdf_pred, cmap='RdBu', vmin=-1, vmax=1)
195:
             axes[1, 0].set_title("Predicted SDF")
             axes[1, 0].axis('off')
196:
197:
             plt.colorbar(im, ax=axes[1, 0])
198:
199:
         # Polygon visualization
200:
         if "polygons" in predictions:
201:
             poly_vis = visualize_polygons(
202:
                predictions["polygons"][0],
203:
                 predictions["polygon_validity"][0],
204:
                 image_size=(256, 256)
205:
206:
             axes[1, 1].imshow(poly_vis)
207:
             axes[1, 1].set_title("Predicted Polygons")
208:
             axes[1, 1].axis('off')
209:
         # 3D voxel slice
210:
         if "voxels_pred" in predictions:
211:
            voxels = torch.sigmoid(predictions["voxels_pred"][0]).cpu().numpy()
212:
213:
             # Show middle slice
214:
             mid_slice = voxels[voxels.shape[0]//2]
215:
            axes[1, 2].imshow(mid_slice, cmap='viridis')
216:
            axes[1, 2].set_title("3D Voxels (Mid Slice)")
217:
             axes[1, 2].axis('off')
218:
219:
        plt.tight_layout()
220:
221:
       if save_path:
222:
            plt.savefig(save_path, dpi=300, bbox_inches="tight")
223:
224:
        plt.show()
225:
226:
227: def visualize_polygons(polygons, validity, image_size=(256, 256), threshold=0.5):
         """Visualize predicted polygons"""
228:
229:
         vis_img = np.zeros((*image_size, 3), dtype=np.uint8)
230:
231:
         for poly_idx, (polygon, valid_score) in enumerate(zip(polygons, validity)):
232:
             if valid_score > threshold:
                 # Convert to image coordinates
233:
                 points = polygon.cpu().numpy() * np.array(image_size)
234:
235:
236:
                 # Remove zero-padded points
237:
                 valid_points = points[points.sum(axis=1) > 0]
238:
239:
                 if len(valid_points) >= 3:
240:
                     points_int = valid_points.astype(np.int32)
241:
                     # Different colors for different polygons
242:
243:
                     color = plt.cm.tab10(poly_idx % 10)[:3]
244:
                     color = tuple(int(c * 255) for c in color)
245:
246:
                     cv2.polylines(vis_img, [points_int], True, color, 2)
247:
248:
                     # Add polygon index
249:
                     center = points_int.mean(axis=0).astype(int)
250:
                     cv2.putText(vis_img, str(poly_idx), tuple(center),
```

```
251:
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1)
252:
253:
         return vis ima
254:
255:
256: def save_model_outputs(predictions, output_dir, sample_id):
257:
         """Save all model outputs for detailed analysis"""
258:
         output_dir = Path(output_dir)
259:
         output_dir.mkdir(exist_ok=True)
260:
261:
         sample_dir = output_dir / sample_id
262:
         sample_dir.mkdir(exist_ok=True)
263:
264:
         # Save segmentation
265:
         if "segmentation" in predictions:
266:
             seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0].cpu().numpy()
267:
             cv2.imwrite(str(sample_dir / "segmentation.png"), seg_pred * 50)
268:
         # Save SDF
269:
270:
        if "sdf" in predictions:
271:
             sdf_pred = predictions["sdf"][0, 0].cpu().numpy()
272:
             sdf_normalized = ((sdf_pred + 1) * 127.5).astype(np.uint8)
273:
             cv2.imwrite(str(sample_dir / "sdf.png"), sdf_normalized)
274:
275:
         # Save attributes
         if "attributes" in predictions:
276:
277:
             attrs = predictions["attributes"][0].cpu().numpy()
             np.save(sample_dir / "attributes.npy", attrs)
278:
279:
280:
         # Save polygons
281:
         if "polygons" in predictions:
282:
             polygons = predictions["polygons"][0].cpu().numpy()
283:
             validity = predictions["polygon_validity"][0].cpu().numpy()
284:
285:
             np.save(sample_dir / "polygons.npy", polygons)
286:
             np.save(sample_dir / "polygon_validity.npy", validity)
287:
288:
         # Save voxels
289:
        if "voxels_pred" in predictions:
290:
             voxels = torch.sigmoid(predictions["voxels_pred"][0]).cpu().numpy()
291:
             np.save(sample_dir / "voxels.npy", voxels)
292:
294: def create_comparison_grid(input_images, predictions, targets=None, num_samples=4):
295:
         """Create a comparison grid showing inputs, predictions, and targets"""
296:
         fig, axes = plt.subplots(num_samples, 4, figsize=(16, 4 * num_samples))
297:
298:
         for i in range(min(num_samples, len(input_images))):
299:
             # Input image
300:
             img = input_images[i].permute(1, 2, 0).cpu().numpy()
301:
             axes[i, 0].imshow(img)
302:
            axes[i, 0].set_title(f"Sample {i+1}: Input")
303:
            axes[i, 0].axis('off')
304:
305:
             # Predicted segmentation
             seg_pred = torch.argmax(predictions["segmentation"][i], dim=0).cpu().numpy()
306:
307:
             axes[i, 1].imshow(seg_pred, cmap='tab10')
308:
            axes[i, 1].set_title("Predicted Seg")
309:
            axes[i, 1].axis('off')
310:
311:
             # Ground truth segmentation (if available)
312:
             if targets and "mask" in targets:
313:
                 gt_mask = targets["mask"][i].cpu().numpy()
314:
                 axes[i, 2].imshow(gt_mask, cmap='tab10')
                 axes[i, 2].set_title("GT Segmentation")
315:
316:
                 axes[i, 2].text(0.5, 0.5, "No GT", ha='center', va='center',
317:
318:
                                transform=axes[i, 2].transAxes)
319:
                 axes[i, 2].set_title("GT Segmentation")
320:
            axes[i, 2].axis('off')
321:
322:
             # Polygon overlay
323:
             poly_vis = visualize_polygons(
```

```
324:
                 predictions["polygons"][i],
325:
                 predictions["polygon_validity"][i]
326:
             )
327:
             axes[i, 3].imshow(poly_vis)
             axes[i, 3].set_title("Predicted Polygons")
328:
329:
             axes[i, 3].axis('off')
330:
331:
         plt.tight_layout()
332:
         return fig
333:
334:
335: def analyze_failure_cases(predictions, targets, threshold_iou=0.5):
336:
         """Analyze failure cases for debugging""
337:
         failure_indices = []
338:
339:
         for i, (pred_seg, gt_mask) in enumerate(zip(predictions["segmentation"], targets["mask"])):
340:
             seg_pred = torch.argmax(pred_seg, dim=0)
341:
             iou = compute_iou(seg_pred, gt_mask)
342:
343:
             if iou < threshold_iou:</pre>
344:
                 failure_indices.append({
345:
                      "index": i,
346:
                      "iou": iou,
347:
                      "pred_classes": torch.unique(seg_pred).tolist(),
                      "gt_classes": torch.unique(gt_mask).tolist()
348:
                 })
349:
350:
         return failure_indices
351:
352:
353:
354: class ProgressiveVisualization:
355:
         """Track and visualize training progress"""
356:
         def __init__(self, save_dir="./training_progress"):
357:
358:
             self.save_dir = Path(save_dir)
359:
             self.save_dir.mkdir(exist_ok=True)
360:
361:
         def log_epoch_results(self, epoch, stage, predictions, targets, sample_image):
362:
             """Log results for a specific epoch"""
             epoch_dir = self.save_dir / f"{stage}_epoch_{epoch}"
363:
364:
             epoch_dir.mkdir(exist_ok=True)
365:
             # Save prediction visualization
367:
             fig = plt.figure(figsize=(12, 8))
368:
             visualize_predictions(sample_image, predictions, targets)
369:
             plt.savefig(epoch_dir / "predictions.png", dpi=150, bbox_inches="tight")
370:
             plt.close()
371:
372:
             # Save individual outputs
373:
             save_model_outputs(predictions, epoch_dir, "sample")
374:
375:
         def create_training_animation(self, stage, metric_name="total_loss"):
376:
             """Create animated GIF showing training progress"""
377:
             # This would create an animation of training progress
378:
             # Implementation depends on having saved epoch results
379:
             pass
380:
381:
382: def compute_architectural_metrics(predictions, image_size=(256, 256)):
         """Compute architecture-specific metrics""
383:
384:
         metrics = {}
385:
386:
         if "segmentation" in predictions:
387:
             seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0]
388:
389:
             # Room count
390:
             room_mask = (seg_pred == 0).cpu().numpy().astype(np.uint8)
             contours, _ = cv2.findContours(room_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
391:
392:
             room_count = len([c for c in contours if cv2.contourArea(c) > 100])
             metrics["room_count"] = room_count
393:
394:
395:
             # Wall connectivity
396:
             wall_mask = (seg_pred == 1).cpu().numpy().astype(np.uint8)
```

```
397:
             wall_components = cv2.connectedComponents(wall_mask)[0] - 1 # Subtract background
398:
             metrics["wall_components"] = max(0, wall_components)
399:
400:
             # Door and window counts
             door_pixels = (seg_pred == 2).sum().item()
401:
402:
             window_pixels = (seg_pred == 3).sum().item()
403:
             metrics["door_pixels"] = door_pixels
404:
             metrics["window_pixels"] = window_pixels
405:
        if "polygons" in predictions:
406:
407:
            validity = predictions["polygon_validity"][0]
408:
             valid_polygons = (validity > 0.5).sum().item()
409:
            metrics["valid_polygon_count"] = valid_polygons
410:
411:
            # Average polygon area
412:
            polygons = predictions["polygons"][0]
413:
             areas = []
414:
            for poly_idx, (polygon, valid) in enumerate(zip(polygons, validity)):
415:
                if valid > 0.5:
416:
                     # Compute polygon area using shoelace formula
417:
                     points = polygon.cpu().numpy() * np.array(image_size)
418:
                     valid_points = points[points.sum(axis=1) > 0]
419:
                     if len(valid_points) >= 3:
420:
                         area = compute_polygon_area(valid_points)
421:
                         areas.append(area)
422:
423:
             metrics["avg_polygon_area"] = np.mean(areas) if areas else 0.0
424:
425:
        return metrics
426:
427:
428: def compute_polygon_area(points):
         """Compute polygon area using shoelace formula"""
429:
        if len(points) < 3:
430:
431:
            return 0.0
432:
433:
        x = points[:, 0]
434:
       y = points[:, 1]
435:
436:
        # Shoelace formula
437:
        area = 0.5 * abs(sum(x[i] * y[i+1] - x[i+1] * y[i] for i in range(-1, len(x)-1)))
438:
        return area
439:
440:
441: def create_model_summary_report(model, sample_input, save_path="model_summary.txt"):
442:
         """Create detailed model summary report""
443:
        with open(save_path, "w") as f:
444:
            f.write("Neural-Geometric 3D Model Generator - Model Summary\n")
            f.write("=" * 60 + "\n\n")
445:
446:
447:
             # Model architecture
448:
             f.write("MODEL ARCHITECTURE:\n")
449:
             f.write("-" * 20 + "\n")
450:
451:
             total_params = sum(p.numel() for p in model.parameters())
             trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
452:
453:
454:
            f.write(f"Total parameters: {total_params:,}\n")
455:
             f.write(f"Trainable parameters: {trainable_params:,}\n")
456:
             f.write(f"Model size: {total_params * 4 / 1024 / 1024:.2f} MB\n\n")
457:
458:
             # Component breakdown
459:
             f.write("COMPONENT PARAMETERS:\n")
460:
             f.write("-" * 25 + "\n")
461:
462:
             encoder_params = sum(p.numel() for p in model.encoder.parameters())
463:
             seg_params = sum(p.numel() for p in model.seg_head.parameters())
464:
             attr_params = sum(p.numel() for p in model.attr_head.parameters())
465:
             sdf_params = sum(p.numel() for p in model.sdf_head.parameters())
            dvx_params = sum(p.numel() for p in model.dvx.parameters())
466:
467:
             ext_params = sum(p.numel() for p in model.extrusion.parameters())
468:
469:
             f.write(f"Encoder: {encoder_params:,} ({encoder_params*total_params*100:.1f}%)\n")
```

```
470:
            471:
            f.write(f"Attribute Head: {attr_params:,} ({attr_params/total_params*100:.1f}%)\n")
472:
            f.write(f"SDF Head: {sdf_params:,} ({sdf_params/total_params*100:.1f}%)\n")
473:
            f.write(f"DVX Module: {dvx_params:,} ({dvx_params/total_params*100:.1f}%)\n")
            f.write(f"Extrusion Module: {ext_params:,} ({ext_params/total_params*100:.1f}%)\n\n")
474:
475:
476:
            # Forward pass analysis
477:
            f.write("FORWARD PASS ANALYSIS:\n")
478:
            f.write("-" * 25 + "\n")
479:
480:
            model.eval()
481:
            with torch.no_grad():
482:
                predictions = model(sample_input)
483:
484:
                for key, value in predictions.items():
485:
                    if torch.is_tensor(value):
                        f.write(f"{key}: {list(value.shape)} - {value.dtype}\n")
486:
487:
                        f.write(f"\{key\}: \{type(value)\}\n")
488:
489:
490:
        print(f"Model summary saved to {save_path}")
491:
492:
493: def debug_gradient_flow(model, loss):
        """Debug gradient flow through the model"""
494:
495:
        print("Gradient Flow Analysis:")
496:
        print("-" * 30)
497:
498:
        total_norm = 0
499:
       component_norms = {}
500:
501:
        for name, param in model.named_parameters():
502:
            if param.grad is not None:
503:
                param_norm = param.grad.norm().item()
504:
                total_norm += param_norm ** 2
505:
506:
                # Group by component
507:
                component = name.split('.')[0]
508:
                if component not in component_norms:
509:
                    component_norms[component] = 0
510:
                component_norms[component] += param_norm ** 2
511:
512:
       total_norm = total_norm ** 0.5
513:
514:
        print(f"Total gradient norm: {total_norm:.4f}")
515:
        print("Component gradient norms:")
516:
517:
        for component, norm in component_norms.items():
518:
            norm = norm ** 0.5
519:
            print(f" {component}: {norm:.4f} ({norm/total_norm*100:.1f}%)")
520:
522: def create_3d_visualization(voxels, output_path="3d_preview.png"):
523:
         """Create 3D visualization of voxel prediction""
524:
        try:
525:
            import matplotlib.pyplot as plt
            from mpl_toolkits.mplot3d import Axes3D
526:
527:
528:
            # Convert to binary
            if isinstance(voxels, torch.Tensor):
529:
530:
                voxels = voxels.cpu().numpy()
531:
532:
            binary_voxels = voxels > 0.5
533:
534:
            # Get occupied voxel coordinates
535:
            occupied = np.where(binary_voxels)
536:
537:
            if len(occupied[0]) == 0:
538:
                print("No occupied voxels to visualize")
539:
                return
540:
541:
            # Create 3D plot
542:
            fig = plt.figure(figsize=(10, 8))
```

```
543:
            ax = fig.add_subplot(111, projection='3d')
544:
545:
             # Plot occupied voxels
            ax.scatter(occupied[0], occupied[1], occupied[2],
546:
547:
                      c=occupied[2], cmap='viridis', s=1, alpha=0.6)
548:
549:
            ax.set_xlabel('X')
550:
            ax.set_ylabel('Y')
551:
            ax.set_zlabel('Z')
            ax.set_title('3D Voxel Occupancy')
552:
553:
554:
            plt.savefig(output_path, dpi=150, bbox_inches="tight")
555:
            plt.close()
556:
557:
            print(f"3D visualization saved to {output_path}")
558:
559:
       except ImportError:
560:
            print("3D visualization requires matplotlib with 3D support")
```

\_\_\_\_\_

## **■** File: vortex.py

```
______
 1: import cv2
 2: import numpy as np
 3: from pathlib import Path
 4: import json
 5:
 6: # Base dataset path
 7: data_root = Path("./data/floorplans")
 8:
 9: def is_valid_mask(mask_file):
10: m = cv2.imread(str(mask_file), 0)
 11:
       return m is not None and np.sum(m) > 0
12:
13: def is_valid_voxel(voxel_file):
14:
       try:
15:
           data = np.load(str(voxel_file))
16:
           key = "voxels" # Use the correct key in your npz files
17:
           if key not in data.files:
18:
               print(f"?? Key '{key}' not found in {voxel_file}, available keys: {data.files}")
19:
               return False
 20:
            v = data[key]
21:
           return not np.isnan(v).any()
 22:
       except Exception as e:
 23:
           print(f"?? Error reading {voxel_file}: {e}")
24:
           return False
 25:
 26: def is_valid_polygon(polygon_file):
 27:
      try:
28:
           with open(polygon_file, "r") as f:
               json.load(f)
30:
           return True
 31:
       except Exception as e:
 32:
           print(f"?? Invalid polygon {polygon_file}: {e}")
33:
           return False
 34:
 35: def check_split(split="train"):
 36:
        split_dir = data_root / split
        mask_files = list(split_dir.rglob("mask.png"))
 37:
 38:
       voxel_files = list(split_dir.rglob("voxel_GT.npz"))
 39:
       polygon_files = list(split_dir.rglob("polygon.json"))
 40:
 41:
        print(f"\n? Checking split: {split}")
 42:
 43:
        # Check mask files
 44:
        total_masks = len(mask_files)
 45:
        bad_masks = 0
46:
        for f in mask_files:
47:
          if not is_valid_mask(f):
 48:
               bad_masks += 1
 49:
                print(f"?? Invalid mask: {f}")
 50:
```

```
51:
       # Check voxel files
52:
       total_voxels = len(voxel_files)
53:
      bad_voxels = 0
54:
       for f in voxel_files:
55:
            if not is_valid_voxel(f):
                bad_voxels += 1
56:
                 print(f"?? Invalid voxel: {f}")
57:
58:
59:
        # Check polygon files
        total_polygons = len(polygon_files)
60:
61:
       bad_polygons = 0
62:
      for f in polygon_files:
63:
            if not is_valid_polygon(f):
64:
                bad_polygons += 1
65:
       # Summary
66:
67:
       print(f"\n? Summary for split: {split}")
       print(f"Total mask files checked: {total_masks}, Invalid: {bad_masks}")
print(f"Total voxel files checked: {total_voxels}, Invalid: {bad_voxels}")
68:
69:
70:
       print(f"Total polygon files checked: {total_polygons}, Invalid: {bad_polygons}")
71:
72: def main():
73:
      for split in ["train", "val", "test"]:
74:
           check_split(split)
75:
76: if __name__ == "__main__":
77:
      main()
```

\_\_\_\_\_