■ Training Code Export (Complete Code - No Truncation)

- [PY] config.py
- [PY] dataset.py
- [PY] evaluation\metrics.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] train.py
- [PY] training\losses.py
- [PY] training\trainer.py
- [PY] utils\visualization.py

■ File: config.py

==

```python
"""
Configuration settings for the Neural-Geometric 3D Model Generator
Enhanced with dynamic curriculum and adaptive training strategies
"""
from dataclasses import dataclass
from typing import Tuple, Dict, Any, Optional, List
import torch


@dataclass
class DataConfig:
    """Data-related configuration"""
    data_dir: str = "./data/floorplans"
    image_size: Tuple[int, int] = (256, 256)   # keep full resolution for accuracy
    voxel_size: int = 64
    batch_size: int = 4                        # balance speed & memory
    num_workers: int = 8                       # faster dataloader (tune per CPU)
    augment: bool = True


@dataclass
class ModelConfig:
    """Model architecture configuration optimized for high accuracy"""
    input_channels: int = 3
    num_classes: int = 5
    feature_dim: int = 512     # reduced from 768 ? faster while keeping strong accuracy
    num_attributes: int = 6
    voxel_size: int = 64
    max_polygons: int = 20     # enough for complex layouts
    max_points: int = 50       # good detail without huge cost
    dropout: float = 0.05
    use_attention: bool = True
    use_deep_supervision: bool = True

    # Auxiliary heads for novel training strategies
    use_latent_consistency: bool = True
    use_graph_constraints: bool = True
    latent_embedding_dim: int = 256


@dataclass
class CurriculumConfig:
    """Dynamic curriculum learning configuration"""
    # Adaptive stage transitioning
    use_dynamic_curriculum: bool = True
    stage_switch_patience: int = 5
    min_improvement_threshold: float = 0.001
    plateau_detection_window: int = 3

    # GradNorm / gradient tracking
    gradient_norm_window: int = 100

    # Objectives for multi-objective optimization
    objectives: Optional[List[str]] = None

    # Topology-aware scheduling
    topology_schedule: str = "progressive"   # "progressive", "linear_ramp", "exponential"
    topology_start_weight: float = 0.1
    topology_end_weight: float = 1.0
    topology_ramp_epochs: int = 20

    # config.py (snippet ? add into the existing config class/dict)
    # Mixed precision and training conveniences
    use_mixed_precision = True            # enable AMP
    cache_in_memory = False               # set True if host RAM can hold dataset
    accumulation_steps = 1                # effective batch size multiplier
    dvx_step_freq = 1                     # run DVX refinement every N steps (1 = every step)
    persistent_workers = True             # DataLoader persistent workers
    prefetch_factor = 4                   # DataLoader prefetch
    num_workers = 8                       # default num workers for DataLoader (tune by CPU)
    # Progressive resolution settings (example)
```

```
    voxel_size_stage = { "stage1": 32, "stage2": 32, "stage3": 64 }  # voxel sizes per stage
    image_size_stage = { "stage1": (128,128), "stage2": (192,192), "stage3": (256,256)}


    # Loss component scheduling
    loss_schedule: Dict[str, str] = None

    # Multi-objective optimization (GradNorm)
    use_gradnorm: bool = True
    gradnorm_alpha: float = 0.12
    gradnorm_update_freq: int = 5

    # Graph constraint scheduling
    graph_weight_schedule: str = "delayed_ramp"
    graph_start_epoch: int = 15
    graph_end_weight: float = 0.25

    def __post_init__(self):
        # Provide default loss schedule if not set
        if self.loss_schedule is None:
            self.loss_schedule = {
                "segmentation": "static",
                "dice": "static",
                "sdf": "early_decay",
                "attributes": "static",
                "polygon": "staged_ramp",
                "voxel": "late_ramp",
                "topology": "progressive",
                "latent_consistency": "mid_ramp",
                "graph": "delayed_ramp",
            }

        # Default objectives used by GradNorm / trainer monitoring
        if self.objectives is None:
            self.objectives = [
                "segmentation",
                "dice",
                "sdf",
                "attributes",
                "polygon",
                "voxel",
                "topology",
                "latent_consistency",
                "graph",
            ]


@dataclass
class TrainingConfig:
    """Training configuration with adaptive strategies"""
    device: str = "cuda" if torch.cuda.is_available() else "cpu"

    # Dynamic epoch limits (maxima; curriculum may switch earlier)
    max_stage1_epochs: int = 40
    max_stage2_epochs: int = 25
    max_stage3_epochs: int = 60

    # Minimum epochs per stage (avoid switching too early)
    min_stage1_epochs: int = 8
    min_stage2_epochs: int = 5
    min_stage3_epochs: int = 12

    # Learning rates (per stage)
    stage1_lr: float = 3e-4  # was 3e-4
    stage1_weight_decay: float = 1e-5

    stage2_lr: float = 1e-4  # was 1e-4
    stage2_weight_decay: float = 1e-5

    stage3_lr: float = 5e-5  # was 5e-5
    stage3_weight_decay: float = 1e-5

    # Advanced training techniques
```

```
    use_mixed_precision: bool = True
    use_cosine_restarts: bool = True
    warmup_epochs: int = 5
    grad_clip_norm: float = 1.0

    # Gradient monitoring for dynamic weighting
    track_gradient_norms: bool = True
    gradient_norm_window: int = 10  # rolling window for gradient tracking

    # Checkpointing
    checkpoint_freq: int = 1

    # Curriculum configuration
    curriculum: CurriculumConfig = None

    def __post_init__(self):
        if self.curriculum is None:
            self.curriculum = CurriculumConfig()


@dataclass
class LossConfig:
    """Loss function weights (will be dynamically adjusted during training)"""
    # Base weights (starting values)
    seg_weight: float = 1.0
    dice_weight: float = 1.0
    sdf_weight: float = 0.5
    attr_weight: float = 1.0
    polygon_weight: float = 1.0
    voxel_weight: float = 1.0
    topology_weight: float = 0.1  # start low, ramp up

    # New loss components
    latent_consistency_weight: float = 0.5
    graph_constraint_weight: float = 0.3

    # Dynamic weighting parameters
    enable_dynamic_weighting: bool = True
    weight_update_freq: int = 10
    weight_momentum: float = 0.9


@dataclass
class InferenceConfig:
    """Inference configuration"""
    model_path: str = "final_model.pth"
    test_images_dir: str = "./data/test_images"
    output_dir: str = "./outputs"
    export_intermediate: bool = True
    polygon_threshold: float = 0.5


# Curriculum stage transition logic
class StageTransitionCriteria:
    """Defines criteria for automatic stage transitions"""

    @staticmethod
    def should_transition_from_stage1(train_losses, val_losses, config: CurriculumConfig) -> bool:
        """Check if should transition from Stage 1 to Stage 2"""
        if len(val_losses) < config.plateau_detection_window:
            return False

        # Check for plateau in segmentation + dice losses
        recent_losses = val_losses[-config.plateau_detection_window:]
        if len(recent_losses) < 2:
            return False

        # Calculate improvement rate
        old_avg = sum(recent_losses[:len(recent_losses)//2]) / (len(recent_losses)//2)
        new_avg = sum(recent_losses[len(recent_losses)//2:]) / (len(recent_losses) -
+                 len(recent_losses)//2)

        improvement_rate = (old_avg - new_avg) / (old_avg + 1e-8)
```

```
        return improvement_rate < config.min_improvement_threshold

    @staticmethod
    def should_transition_from_stage2(polygon_losses, config: CurriculumConfig) -> bool:
        """Check if should transition from Stage 2 to Stage 3"""
        if len(polygon_losses) < config.plateau_detection_window:
            return False

        # Check polygon loss plateau
        recent_losses = polygon_losses[-config.plateau_detection_window:]
        if len(recent_losses) < 2:
            return False

        old_avg = sum(recent_losses[:len(recent_losses)//2]) / (len(recent_losses)//2)
        new_avg = sum(recent_losses[len(recent_losses)//2:]) / (len(recent_losses) -
+                 len(recent_losses)//2)

        improvement_rate = (old_avg - new_avg) / (old_avg + 1e-8)
        return improvement_rate < config.min_improvement_threshold


# Default configurations (import these in your trainer)
DEFAULT_DATA_CONFIG = DataConfig()
DEFAULT_MODEL_CONFIG = ModelConfig()
DEFAULT_TRAINING_CONFIG = TrainingConfig()
DEFAULT_LOSS_CONFIG = LossConfig()
DEFAULT_INFERENCE_CONFIG = InferenceConfig()
```

■ File: dataset.py

==

```
"""
Dataset classes for the Neural-Geometric 3D Model Generator
Enhanced with in-memory caching for faster training
"""

import cv2
import json
import numpy as np
import torch
from torch.utils.data import Dataset
from pathlib import Path
from typing import Dict, List, Tuple, Optional, Union
import time

from config import DEFAULT_DATA_CONFIG


class AdvancedFloorPlanDataset(Dataset):
    """
    Research-grade dataset with complete ground truth:
    - Floorplan image + segmentation mask
    - Attribute dictionary (geometric parameters)
    - Ground-truth mesh + voxelized occupancy
    - Polygon outlines for vectorization supervision

    Enhanced with optional in-memory caching for performance
    """

    def __init__(
        self,
        data_dir: str = None,
        split: str = "train",
        image_size: Tuple[int, int] = None,
        voxel_size: int = None,
        augment: bool = None,
        config=None,
    ):
        # Use config if provided, otherwise defaults from config.py
        if config is None:
            config = DEFAULT_DATA_CONFIG

```

```python
        self.data_dir = Path(data_dir or config.data_dir)
        self.split = split
        self.image_size = image_size or config.image_size
        self.voxel_size = voxel_size or config.voxel_size
        self.augment = (
            augment if augment is not None else config.augment
        ) and split == "train"

        # Collect all samples that contain every required file
        self.samples = self._find_complete_samples()
        print(f"Found {len(self.samples)} complete samples for {split}")

        # NEW: In-memory caching for performance
        self.cache_in_memory = getattr(config, "cache_in_memory", False)
        self._cache = None

        if self.cache_in_memory and len(self.samples) > 0:
            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...")

            # Estimate memory usage
            estimated_mb = self._estimate_memory_usage()
            print(f"[DATA] Estimated memory usage: {estimated_mb:.1f} MB")

            start_time = time.time()
            self._preload_cache()
            load_time = time.time() - start_time
            print(f"[DATA] Cache preloading completed in {load_time:.2f}s")

    def _estimate_memory_usage(self):
        """Estimate memory usage for caching"""
        if not self.samples:
            return 0.0

        H, W = self.image_size
        n_samples = len(self.samples)

        # Rough estimates in bytes
        image_bytes = H * W * 3  # RGB uint8
        mask_bytes = H * W  # grayscale uint8
        voxel_bytes = self.voxel_size ** 3 * 4  # float32
        json_bytes = 1024  # rough estimate for params + polygons

        if self.voxel_size >= 128:
            voxel_gb = (voxel_bytes * n_samples) / (1024**3)
            print(f"[WARNING] Large voxel grid ({self.voxel_size}^3) may lead to high memory usage: {voxel_gb:.2f} GB just for voxels")
            print("[Warning] Consider reducing voxel_size or disabling cache_in_memory.")

        total_per_sample = image_bytes + mask_bytes + voxel_bytes + json_bytes
        total_mb = (total_per_sample * n_samples) / (1024 * 1024)

        return total_mb

    def _preload_cache(self):
        """Preload all samples into memory"""
        self._cache = []

        for i, sample in enumerate(self.samples):
            if i % 100 == 0:
                print(f"[DATA] Loading sample {i+1}/{len(self.samples)}")

            try:
                # Load image
                img = cv2.imread(str(sample["image"]))
                if img is None:
                    print(f"Warning: Could not load image {sample['image']}")
                    continue
                img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                img = cv2.resize(img, self.image_size)  # (W, H) format for cv2.resize

                # Load mask
                mask = cv2.imread(str(sample["mask"]), cv2.IMREAD_GRAYSCALE)
```

```
                if mask is None:
                    print(f"Warning: Could not load mask {sample['mask']}")
                    continue
                mask = cv2.resize(mask, self.image_size, interpolation=cv2.INTER_NEAREST)

                # Load voxel data
                try:
                    voxel_data = np.load(sample["voxel"])
                    vox = voxel_data["voxels"]  # Keep as numpy array
                except Exception as e:
                    print(f"Warning: Could not load voxel data {sample['voxel']}: {e}")
                    # Create dummy voxel data
                    vox = np.zeros((self.voxel_size, self.voxel_size, self.voxel_size),
                             dtype=np.float32)

                # Load parameters
                try:
                    with open(sample["params"], "r") as f:
                        params = json.load(f)
                except Exception as e:
                    print(f"Warning: Could not load params {sample['params']}: {e}")
                    params = self._get_default_attributes()

                # Load polygons
                try:
                    with open(sample["polygon"], "r") as f:
                        polygons = json.load(f)
                except Exception as e:
                    print(f"Warning: Could not load polygons {sample['polygon']}: {e}")
                    polygons = {"walls": []}

                self._cache.append({
                    "image": img,
                    "mask": mask,
                    "vox": vox,
                    "params": params,
                    "polygons": polygons,
                    "sample_id": sample["image"].parent.name,
                })

            except Exception as e:
                print(f"Error loading sample {i}: {e}")
                continue

    def _get_default_attributes(self):
        """Return default attributes for missing param files"""
        return {
            "wall_height": 2.6,
            "wall_thickness": 0.15,
            "window_base_height": 0.7,
            "window_height": 0.95,
            "door_height": 2.6,
            "pixel_scale": 0.02,
        }

    # ----------------------------------------------------------------------
    def _find_complete_samples(self):
        """Locate samples that contain all the expected files."""
        samples = []
        split_dir = self.data_dir / self.split

        if not split_dir.exists():
            print(f"Warning: Split directory {split_dir} does not exist")
            return samples

        for sample_dir in split_dir.iterdir():
            if not sample_dir.is_dir():
                continue

            required_files = {
                "image": sample_dir / "image.png",
                "mask": sample_dir / "mask.png",
                "params": sample_dir / "params.json",
```

```python
                "mesh": sample_dir / "model.obj",
                "voxel": sample_dir / "voxel_GT.npz",
                "polygon": sample_dir / "polygon.json",
            }

            if all(f.exists() for f in required_files.values()):
                samples.append(required_files)

        return samples

    # ----------------------------------------------------------------------
    def __len__(self):
        return len(self._cache) if self._cache is not None else len(self.samples)

    # ----------------------------------------------------------------------
    def __getitem__(self, idx):
        # Use cached data if available
        if self._cache is not None:
            cached_sample = self._cache[idx]
            image = cached_sample['image']
            mask = cached_sample['mask']
            vox = cached_sample['vox']
            attributes = cached_sample['params']
            polygons_gt = cached_sample['polygons']
            sample_id = cached_sample['sample_id']
        else:
            # Fallback: load from disk on-the-fly
            sample = self.samples[idx]

            # Load image and mask
            image = cv2.imread(str(sample["image"]))
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            image = cv2.resize(image, self.image_size)

            mask = cv2.imread(str(sample["mask"]), cv2.IMREAD_GRAYSCALE)
            mask = cv2.resize(mask, self.image_size, interpolation=cv2.INTER_NEAREST)

            # Load attributes
            with open(sample["params"], "r") as f:
                attributes = json.load(f)

            # Load voxel ground truth
            voxel_data = np.load(sample["voxel"])
            vox = voxel_data["voxels"]

            # Load polygon ground truth
            with open(sample["polygon"], "r") as f:
                polygons_gt = json.load(f)

            sample_id = sample["image"].parent.name

        # Normalize image to [0,1]
        image = image.astype(np.float32) / 255.0

        # Clean mask (remove class 5 if present)
        if np.any(mask == 5):
            print(f"WARNING: Found class 5 in sample {idx}. Verify this class should be removed!")
        mask[mask == 5] = 0

        # Convert to tensors
        image_tensor = torch.from_numpy(image).float().permute(2, 0, 1)
        mask_tensor = torch.from_numpy(mask).long()
        voxels_tensor = torch.from_numpy(vox.astype(np.float32))

        attr_tensor = self._process_attributes(attributes)
        polygon_tensor = self._process_polygons(polygons_gt)

        # Apply augmentation if enabled
        if self.augment:
            image_tensor, mask_tensor = self._augment(image_tensor, mask_tensor)

        # Add validation before returning
        self._validate_sample_data(idx, image_tensor, mask_tensor, attr_tensor, voxels_tensor,
```

```
                 polygon_tensor)

        return {
            "image": image_tensor,
            "mask": mask_tensor,
            "attributes": attr_tensor,
            "voxels_gt": voxels_tensor,
            "polygons_gt": polygon_tensor,
            "sample_id": sample_id,
        }

    # ---------------------------------------------------------------------
    def _validate_sample_data(self, idx, image, mask, attributes, voxels, polygons):
        """Validate sample data for NaN/Inf values"""
        tensors_to_check = [
            ("image", image),
            ("mask", mask),
            ("attributes", attributes),
            ("voxels", voxels),
            ("polygons", polygons["polygons"])
        ]

        corrupted_data = False

        for name, tensor in tensors_to_check:
            if torch.isnan(tensor).any():
                print(f"ERROR: {name} contains NaN values at sample {idx}")
                corrupted_data = True
            if torch.isinf(tensor).any():
                print(f"ERROR: {name} contains Inf values at sample {idx}")
                corrupted_data = True

        if corrupted_data:
            print(f"WARNING: Corrupted data detected in sample {idx}, replacing with safe fallback values")

            # Replace corrupted tensors with safe fallback values
            for name, tensor in tensors_to_check:
                if torch.isnan(tensor).any() or torch.isinf(tensor).any():
                    if name == "image":
                        # Replace with zeros (black image)
                        image.data = torch.zeros_like(image)
                    elif name == "mask":
                        # Replace with zeros (background class)
                        mask.data = torch.zeros_like(mask).long()
                    elif name == "attributes":
                        # Replace with reasonable default values (0.5 normalized)
                        attributes.data = torch.ones_like(attributes) * 0.5
                    elif name == "voxels":
                        # Replace with empty voxel grid
                        voxels.data = torch.zeros_like(voxels)
                    elif name == "polygons":
                        # Replace polygons with zeros
                        polygons["polygons"].data = torch.zeros_like(polygons["polygons"])

    # ---------------------------------------------------------------------
    def _process_attributes(self, attributes):
        """Convert attribute dictionary to a normalized tensor."""
        # Normalize common architectural parameters into [0,1]
        attr_list = [
            attributes.get("wall_height", 2.6) / 5.0,
            attributes.get("wall_thickness", 0.15) / 0.5,
            attributes.get("window_base_height", 0.7) / 3.0,
            attributes.get("window_height", 0.95) / 2.0,
            attributes.get("door_height", 2.6) / 5.0,
            attributes.get("pixel_scale", 0.01) / 0.02,
        ]

        # Ensure no NaN/Inf values in attribute processing
        safe_attr_list = []
        for val in attr_list:
            if np.isnan(val) or np.isinf(val):
                safe_attr_list.append(0.5)  # Default normalized value
```

```
            else:
                safe_attr_list.append(max(0.0, min(1.0, val)))  # Clamp to [0,1]

        return torch.tensor(safe_attr_list, dtype=torch.float32)

    # ----------------------------------------------------------------------
    def _process_polygons(self, polygons_gt):
        """Convert polygon ground truth into a fixed tensor representation.
        Handles both formats:
        1. Nested dict: { "walls": [...], "doors": [...], ... }
        2. Flat list:   [ {"type": "wall", "points": [...]}, ... ]
        """
        max_polygons = 30   # number of polygons per sample
        max_points = 100     # max points per polygon

        processed = torch.zeros(max_polygons, max_points, 2)
        valid_mask = torch.zeros(max_polygons, dtype=torch.bool)

        poly_idx = 0

        try:
            # --- Case 1: dict format ---
            if isinstance(polygons_gt, dict):
                for class_name, polygon_list in polygons_gt.items():
                    if not isinstance(polygon_list, list):
                        continue
                    for polygon in polygon_list:
                        if poly_idx >= max_polygons:
                            break
                        if "points" not in polygon:
                            continue

                        points = np.array(polygon["points"])
                        if len(points) > max_points:
                            # Subsample evenly if too many points
                            indices = np.linspace(0, len(points) - 1, max_points, dtype=int)
                            points = points[indices]

                        # Check for NaN/Inf in points
                        if np.any(np.isnan(points)) or np.any(np.isinf(points)):
                            print(f"Warning: Invalid polygon points detected, skipping polygon")
                            continue

                        # Normalize to [0,1] relative to image size
                        points = points / np.array(self.image_size)
                        # Clamp to valid range
                        points = np.clip(points, 0.0, 1.0)

                        processed[poly_idx, : len(points)] = torch.from_numpy(points).float()
                        valid_mask[poly_idx] = True
                        poly_idx += 1

            # --- Case 2: list format ---
            elif isinstance(polygons_gt, list):
                for polygon in polygons_gt:
                    if poly_idx >= max_polygons:
                        break
                    if "points" not in polygon:
                        continue

                    points = np.array(polygon["points"])
                    if len(points) > max_points:
                        indices = np.linspace(0, len(points) - 1, max_points, dtype=int)
                        points = points[indices]

                    # Check for NaN/Inf in points
                    if np.any(np.isnan(points)) or np.any(np.isinf(points)):
                        print(f"Warning: Invalid polygon points detected, skipping polygon")
                        continue

                    points = points / np.array(self.image_size)
                    points = np.clip(points, 0.0, 1.0)

```

```
                    processed[poly_idx, : len(points)] = torch.from_numpy(points).float()
                    valid_mask[poly_idx] = True
                    poly_idx += 1

        except Exception as e:
            print(f"Warning: Error processing polygons: {e}")
            # Return safe empty polygon data
            processed = torch.zeros(max_polygons, max_points, 2)
            valid_mask = torch.zeros(max_polygons, dtype=torch.bool)

        return {"polygons": processed, "valid_mask": valid_mask}

    # ----------------------------------------------------------------------
    def _augment(self, image, mask):
        """Enhanced data augmentation with rotations, flips, and intensity changes."""
        # Random rotation (multiples of 90° only for architectural data)
        if torch.rand(1) < 0.5:
            k = torch.randint(1, 4, (1,)).item()
            image = torch.rot90(image, k, dims=[1, 2])
            mask = torch.rot90(mask, k, dims=[0, 1])

        # Random horizontal flip
        if torch.rand(1) < 0.5:
            image = torch.flip(image, dims=[2])
            mask = torch.flip(mask, dims=[1])

        # Random vertical flip
        if torch.rand(1) < 0.5:
            image = torch.flip(image, dims=[1])
            mask = torch.flip(mask, dims=[0])

        # Slight brightness/contrast adjustment with safety checks
        if torch.rand(1) < 0.3:
            brightness = torch.rand(1) * 0.2 - 0.1  # ±0.1
            contrast = torch.rand(1) * 0.2 + 0.9     # 0.9-1.1
            image = torch.clamp(image * contrast + brightness, 0, 1)

            # Additional safety check for augmented image
            if torch.isnan(image).any() or torch.isinf(image).any():
                print("Warning: Augmentation produced invalid values, reverting to original")
                # Revert to safe values
                image = torch.clamp(image, 0, 1)
                image = torch.where(torch.isnan(image) | torch.isinf(image),
                                  torch.zeros_like(image), image)

        return image, mask

    # ----------------------------------------------------------------------
    def get_cache_info(self):
        """Return information about caching status"""
        return {
            "cache_enabled": self.cache_in_memory,
            "cache_loaded": self._cache is not None,
            "cached_samples": len(self._cache) if self._cache else 0,
            "total_samples": len(self.samples),
            "estimated_memory_mb": self._estimate_memory_usage() if self.cache_in_memory else 0
        }

    def disable_cache(self):
        """Disable caching and free memory"""
        if self._cache is not None:
            print(f"[DATA] Disabling cache and freeing memory for {len(self._cache)} samples")
            self._cache = None
            self.cache_in_memory = False

    def enable_cache(self):
        """Enable caching if not already enabled"""
        if not self.cache_in_memory and self.samples:
            self.cache_in_memory = True
            print("[DATA] Enabling cache...")
            self._preload_cache()
```

```python
# ======================================================================
# Synthetic sample generator for testing without dataset
# ======================================================================
def create_synthetic_data_sample():
    """Generate a synthetic floorplan with attributes, voxels, and polygons."""
    image = np.ones((256, 256, 3), dtype=np.uint8) * 255
    mask = np.zeros((256, 256), dtype=np.uint8)

    # Simple square room
    room_points = np.array([[50, 50], [200, 50], [200, 200], [50, 200]])
    cv2.fillPoly(mask, [room_points], 1)  # Room = class 1
    cv2.polylines(image, [room_points], True, (0, 0, 0), 3)

    # Add door
    cv2.rectangle(mask, (90, 50), (110, 70), 2)  # Door = class 2
    cv2.rectangle(image, (90, 50), (110, 70), (255, 0, 0), -1)

    # Attributes
    attributes = {
        "wall_height": 2.6,
        "wall_thickness": 0.15,
        "window_base_height": 0.7,
        "window_height": 0.95,
        "door_height": 2.6,
        "pixel_scale": 0.02,
    }

    # Simple voxel GT
    voxels = np.zeros((64, 64, 64), dtype=bool)
    voxels[:20, 10:50, 10:50] = True

    # Polygon GT
    polygons = {"walls": [{"points": room_points.tolist()}]}

    return image, mask, attributes, voxels, polygons


class SyntheticFloorPlanDataset(Dataset):
    """
    Synthetic dataset for testing and development when real data is not available
    """

    def __init__(self, num_samples=1000, image_size=(256, 256), voxel_size=64):
        self.num_samples = num_samples
        self.image_size = image_size
        self.voxel_size = voxel_size

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        # Generate deterministic synthetic data based on index
        np.random.seed(idx)
        torch.manual_seed(idx)

        image, mask, attributes, voxels, polygons_gt = create_synthetic_data_sample()

        # Convert to tensors
        image_tensor = torch.from_numpy(image.astype(np.float32) / 255.0).permute(2, 0, 1)
        mask_tensor = torch.from_numpy(mask).long()
        voxels_tensor = torch.from_numpy(voxels.astype(np.float32))

        # Process attributes and polygons using same methods as main dataset
        dataset = AdvancedFloorPlanDataset.__new__(AdvancedFloorPlanDataset)
        dataset.image_size = self.image_size

        attr_tensor = dataset._process_attributes(attributes)
        polygon_tensor = dataset._process_polygons(polygons_gt)

        return {
            "image": image_tensor,
            "mask": mask_tensor,
            "attributes": attr_tensor,
```

```
            "voxels_gt": voxels_tensor,
            "polygons_gt": polygon_tensor,
            "sample_id": f"synthetic_{idx:06d}",
        }
```

■ File: evaluation\metrics.py

==

```
"""
Evaluation metrics and utilities for the Neural-Geometric 3D Model Generator
"""

import torch
import numpy as np
from torch.utils.data import DataLoader

from models.model import NeuralGeometric3DGenerator
from dataset import AdvancedFloorPlanDataset


def compute_iou(pred, target):
    """Compute IoU for segmentation"""
    intersection = (pred & target).float().sum()
    union = (pred | target).float().sum()
    return (intersection / (union + 1e-6)).item()


def compute_3d_iou(pred, target):
    """Compute 3D IoU for voxel grids"""
    pred_bool = pred.bool()
    target_bool = target.bool()

    intersection = (pred_bool & target_bool).float().sum()
    union = (pred_bool | target_bool).float().sum()

    return (intersection / (union + 1e-6)).item()


def compute_polygon_metrics(pred_polygons, gt_polygons, validity_pred, validity_gt):
    """Compute metrics for polygon prediction"""
    # Chamfer distance between polygon sets
    valid_pred = pred_polygons[validity_pred > 0.5]
    valid_gt = gt_polygons[validity_gt]

    if len(valid_pred) == 0 or len(valid_gt) == 0:
        return {"chamfer_distance": float('inf'), "validity_accuracy": 0.0}

    # Simplified chamfer distance computation
    chamfer_dist = 0.0
    for pred_poly in valid_pred:
        min_dist = float('inf')
        for gt_poly in valid_gt:
            dist = torch.norm(pred_poly - gt_poly, dim=-1).min().item()
            min_dist = min(min_dist, dist)
        chamfer_dist += min_dist

    chamfer_dist /= len(valid_pred)

    # Validity accuracy
    validity_acc = ((validity_pred > 0.5) == validity_gt).float().mean().item()

    return {
        "chamfer_distance": chamfer_dist,
        "validity_accuracy": validity_acc
    }


class ModelEvaluator:
    """Comprehensive model evaluation"""

    def __init__(self, model_path, device="cuda"):
        self.device = device
```

```
        self.model = NeuralGeometric3DGenerator()

        # Load model
        checkpoint = torch.load(model_path, map_location=device)
        self.model.load_state_dict(checkpoint["model_state_dict"])
        self.model.to(device)
        self.model.eval()

        print(f"Loaded model from {model_path}")

    def evaluate_dataset(self, test_dataset):
        """Comprehensive evaluation on test dataset"""
        test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)

        # Metrics storage
        metrics = {
            "segmentation": {"ious": [], "class_ious": []},
            "attributes": {"maes": [], "mses": []},
            "voxels": {"ious": [], "dice_scores": []},
            "polygons": {"chamfer_distances": [], "validity_accs": []},
        }

        with torch.no_grad():
            for batch_idx, batch in enumerate(test_loader):
                batch = {k: v.to(self.device) if torch.is_tensor(v) else v
                        for k, v in batch.items()}

                predictions = self.model(batch["image"])

                # Evaluate segmentation
                seg_metrics = self._evaluate_segmentation(
                    predictions["segmentation"], batch["mask"]
                )
                metrics["segmentation"]["ious"].append(seg_metrics["iou"])
                metrics["segmentation"]["class_ious"].append(seg_metrics["class_ious"])

                # Evaluate attributes
                attr_metrics = self._evaluate_attributes(
                    predictions["attributes"], batch["attributes"]
                )
                metrics["attributes"]["maes"].append(attr_metrics["mae"])
                metrics["attributes"]["mses"].append(attr_metrics["mse"])

                # Evaluate voxels
                voxel_metrics = self._evaluate_voxels(
                    predictions["voxels_pred"], batch["voxels_gt"]
                )
                metrics["voxels"]["ious"].append(voxel_metrics["iou"])
                metrics["voxels"]["dice_scores"].append(voxel_metrics["dice"])

                # Evaluate polygons
                poly_metrics = self._evaluate_polygons(
                    predictions["polygons"],
                    predictions["polygon_validity"],
                    batch["polygons_gt"]
                )
                metrics["polygons"]["chamfer_distances"].append(poly_metrics["chamfer_distance"])
                metrics["polygons"]["validity_accs"].append(poly_metrics["validity_accuracy"])

                if (batch_idx + 1) % 10 == 0:
                    print(f"Evaluated {batch_idx + 1}/{len(test_loader)} samples")

        return self._compute_summary_metrics(metrics)

    def _evaluate_segmentation(self, pred_seg, target_mask):
        """Evaluate segmentation performance"""
        seg_pred = torch.argmax(pred_seg, dim=1)

        # Overall IoU
        overall_iou = compute_iou(seg_pred, target_mask)

        # Per-class IoU
        num_classes = pred_seg.shape[1]
```

```
        class_ious = []

        for c in range(num_classes):
            pred_c = (seg_pred == c)
            target_c = (target_mask == c)

            if target_c.sum() > 0:  # Only compute if class exists in ground truth
                iou_c = compute_iou(pred_c, target_c)
                class_ious.append(iou_c)

        return {
            "iou": overall_iou,
            "class_ious": class_ious
        }

    def _evaluate_attributes(self, pred_attrs, target_attrs):
        """Evaluate attribute prediction"""
        mae = torch.mean(torch.abs(pred_attrs - target_attrs)).item()
        mse = torch.mean((pred_attrs - target_attrs) ** 2).item()

        return {"mae": mae, "mse": mse}

    def _evaluate_voxels(self, pred_voxels, target_voxels):
        """Evaluate 3D voxel prediction"""
        pred_binary = (torch.sigmoid(pred_voxels) > 0.5).float()
        target_float = target_voxels.float()

        # 3D IoU
        iou_3d = compute_3d_iou(pred_binary, target_float)

        # 3D Dice score
        intersection = (pred_binary * target_float).sum()
        dice = (2 * intersection) / (pred_binary.sum() + target_float.sum() + 1e-6)

        return {
            "iou": iou_3d,
            "dice": dice.item()
        }

    def _evaluate_polygons(self, pred_polygons, pred_validity, gt_polygons):
        """Evaluate polygon prediction"""
        return compute_polygon_metrics(
            pred_polygons[0],
            gt_polygons["polygons"][0],
            pred_validity[0],
            gt_polygons["valid_mask"][0]
        )

    def _compute_summary_metrics(self, metrics):
        """Compute summary statistics"""
        summary = {}

        # Segmentation
        summary["segmentation_mIoU"] = np.mean(metrics["segmentation"]["ious"])
        summary["segmentation_std"] = np.std(metrics["segmentation"]["ious"])

        # Attributes
        summary["attribute_MAE"] = np.mean(metrics["attributes"]["maes"])
        summary["attribute_MAE_std"] = np.std(metrics["attributes"]["maes"])

        # Voxels
        summary["voxel_mIoU"] = np.mean(metrics["voxels"]["ious"])
        summary["voxel_mIoU_std"] = np.std(metrics["voxels"]["ious"])
        summary["voxel_dice"] = np.mean(metrics["voxels"]["dice_scores"])

        # Polygons
        valid_chamfer = [d for d in metrics["polygons"]["chamfer_distances"] if d != float('inf')]
        if valid_chamfer:
            summary["polygon_chamfer"] = np.mean(valid_chamfer)
            summary["polygon_chamfer_std"] = np.std(valid_chamfer)
        else:
            summary["polygon_chamfer"] = float('inf')
            summary["polygon_chamfer_std"] = 0.0
```

```
        summary["polygon_validity_acc"] = np.mean(metrics["polygons"]["validity_accs"])

        return summary

    def print_evaluation_results(self, summary):
        """Print formatted evaluation results"""
        print("=" * 60)
        print("COMPREHENSIVE EVALUATION RESULTS")
        print("=" * 60)

        print(f"Segmentation mIoU: {summary['segmentation_mIoU']:.4f} ±
+                {summary['segmentation_std']:.4f}")
        print(f"Attribute MAE: {summary['attribute_MAE']:.4f} ± {summary['attribute_MAE_std']:.4f}")
        print(f"Voxel 3D mIoU: {summary['voxel_mIoU']:.4f} ± {summary['voxel_mIoU_std']:.4f}")
        print(f"Voxel Dice Score: {summary['voxel_dice']:.4f}")

        if summary['polygon_chamfer'] != float('inf'):
            print(f"Polygon Chamfer Distance: {summary['polygon_chamfer']:.4f} ±
+                    {summary['polygon_chamfer_std']:.4f}")
        else:
            print("Polygon Chamfer Distance: No valid polygons")

        print(f"Polygon Validity Accuracy: {summary['polygon_validity_acc']:.4f}")
        print("=" * 60)


def evaluate_model(model_path, data_dir="./data/floorplans"):
    """Standalone evaluation function"""
    # Load test dataset
    test_dataset = AdvancedFloorPlanDataset(data_dir, split="test")

    if len(test_dataset) == 0:
        print("No test samples found!")
        return None

    # Create evaluator
    evaluator = ModelEvaluator(model_path)

    # Run evaluation
    summary = evaluator.evaluate_dataset(test_dataset)

    # Print results
    evaluator.print_evaluation_results(summary)

    return summary
```

■ File: inference\engine.py

==

```
"""
Research-grade inference engine for 2D to 3D floorplan generation
"""

import torch
import cv2
import numpy as np
import json
import trimesh
from pathlib import Path

from models.model import NeuralGeometric3DGenerator
from config import DEFAULT_INFERENCE_CONFIG


class ResearchInferenceEngine:
    """
    Complete inference system that converts 2D floorplans to 3D models
    following the deterministic export pipeline
    """

    def __init__(self, model_path=None, device="cuda", config=None):
```

```
        if config is None:
            config = DEFAULT_INFERENCE_CONFIG

        self.device = device
        self.config = config
        self.model = NeuralGeometric3DGenerator()

        # Load trained model
        model_path = model_path or config.model_path
        checkpoint = torch.load(model_path, map_location=device)
        self.model.load_state_dict(checkpoint["model_state_dict"])
        self.model.to(device)
        self.model.eval()

        print(f"Loaded trained model from {model_path}")

    def generate_3d_model(
        self,
        image_path: str,
        output_path: str,
        export_intermediate: bool = None
    ):
        """
        Complete pipeline: Image -> Segmentation -> Polygons -> 3D Model
        """
        export_intermediate = export_intermediate or self.config.export_intermediate

        # Load and preprocess image
        image = self._load_image(image_path)

        with torch.no_grad():
            # Neural network inference
            predictions = self.model(image)

            # Extract predictions
            segmentation = predictions["segmentation"]
            attributes = predictions["attributes"]
            polygons = predictions["polygons"]
            validity = predictions["polygon_validity"]

            print("Neural network inference complete")

            # Convert to deterministic representations
            mask_np = self._extract_mask(segmentation)
            attributes_dict = self._extract_attributes(attributes)
            polygons_list = self._extract_polygons(polygons, validity)

            print(f"Extracted: {len(polygons_list)} valid polygons")

            # Export intermediate results if requested
            if export_intermediate:
                self._export_intermediates(
                    mask_np, attributes_dict, polygons_list, Path(output_path).parent
                )

            # Generate 3D model using deterministic pipeline
            success = self._generate_deterministic_3d(
                mask_np, attributes_dict, polygons_list, output_path
            )

            return success

    def _load_image(self, image_path):
        """Load and preprocess input image"""
        image = cv2.imread(image_path)
        if image is None:
            raise ValueError(f"Could not load image from {image_path}")

        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = cv2.resize(image, (256, 256))
        image = torch.from_numpy(image / 255.0).float()
        image = image.permute(2, 0, 1).unsqueeze(0)
        return image.to(self.device)
```

```python

    def _extract_mask(self, segmentation):
        """Convert soft segmentation to hard mask"""
        seg_pred = torch.argmax(segmentation, dim=1)
        mask_np = seg_pred.squeeze().cpu().numpy().astype(np.uint8)
        return mask_np

    def _extract_attributes(self, attributes):
        """Convert normalized attributes back to physical values"""
        attr_np = attributes.squeeze().cpu().numpy()

        # Denormalize (reverse of normalization in dataset)
        attributes_dict = {
            "wall_height": float(attr_np[0] * 5.0),
            "wall_thickness": float(attr_np[1] * 0.5),
            "window_base_height": float(attr_np[2] * 3.0),
            "window_height": float(attr_np[3] * 2.0),
            "door_height": float(attr_np[4] * 5.0),
            "pixel_scale": float(attr_np[5] * 0.02),
        }

        return attributes_dict

    def _extract_polygons(self, polygons, validity, threshold=None):
        """Extract valid polygons from network predictions"""
        threshold = threshold or self.config.polygon_threshold
        batch_size, num_polys, num_points, _ = polygons.shape

        polygons_list = []

        for poly_idx in range(num_polys):
            if validity[0, poly_idx] > threshold:  # Only valid polygons
                poly_points = polygons[0, poly_idx].cpu().numpy()

                # Remove zero-padded points
                valid_points = poly_points[poly_points.sum(axis=1) > 0]

                if len(valid_points) >= 3:  # Minimum for a polygon
                    # Convert to image coordinates (assuming 256x256 input)
                    valid_points = valid_points * 256
                    polygons_list.append(
                        {
                            "points": valid_points.tolist(),
                            "class": "wall",  # Simplified - in practice classify polygon type
                        }
                    )

        return polygons_list

    def _export_intermediates(self, mask, attributes, polygons, output_dir):
        """Export intermediate results for debugging/analysis"""
        output_dir = Path(output_dir)
        output_dir.mkdir(exist_ok=True)

        # Export mask
        cv2.imwrite(str(output_dir / "predicted_mask.png"), mask * 50)

        # Export attributes
        with open(output_dir / "predicted_attributes.json", "w") as f:
            json.dump(attributes, f, indent=2)

        # Export polygons
        with open(output_dir / "predicted_polygons.json", "w") as f:
            json.dump(polygons, f, indent=2)

        # Visualize polygons on mask
        vis_img = np.zeros((256, 256, 3), dtype=np.uint8)
        vis_img[:, :, 0] = mask * 50  # Background

        for poly in polygons:
            points = np.array(poly["points"], dtype=np.int32)
            cv2.polylines(vis_img, [points], True, (0, 255, 0), 2)

```

```
        cv2.imwrite(str(output_dir / "polygon_visualization.png"), vis_img)

        print(f"Intermediate results exported to {output_dir}")

    def _generate_deterministic_3d(self, mask, attributes, polygons, output_path):
        """Generate 3D model using deterministic geometric operations"""
        try:
            # Initialize mesh components
            vertices = []
            faces = []
            vertex_count = 0

            # Extract geometric parameters
            wall_height = attributes.get("wall_height", 2.6)
            wall_thickness = attributes.get("wall_thickness", 0.15)
            pixel_scale = attributes.get("pixel_scale", 0.01)

            print(
                f"Generating 3D model with wall_height={wall_height:.2f}m,
+                         thickness={wall_thickness:.2f}m"
            )

            # Process each polygon (walls, rooms, etc.)
            for poly_idx, polygon in enumerate(polygons):
                poly_vertices, poly_faces = self._extrude_polygon_3d(
                    polygon["points"],
                    wall_height,
                    wall_thickness,
                    pixel_scale,
                    vertex_count,
                )

                vertices.extend(poly_vertices)
                faces.extend(poly_faces)
                vertex_count += len(poly_vertices)

            # Add floor and ceiling
            floor_verts, floor_faces = self._generate_floor_ceiling(
                mask, pixel_scale, wall_height, vertex_count
            )
            vertices.extend(floor_verts)
            faces.extend(floor_faces)

            if len(vertices) == 0:
                print("No geometry generated")
                return False

            # Create mesh
            mesh = trimesh.Trimesh(vertices=np.array(vertices), faces=np.array(faces))

            # Clean up mesh
            mesh.remove_duplicate_faces()
            mesh.remove_unreferenced_vertices()
            mesh.fix_normals()

            # Export
            mesh.export(output_path)
            print(f"3D model exported to {output_path}")
            print(
                f"Mesh statistics: {len(mesh.vertices)} vertices, {len(mesh.faces)} faces"
            )

            return True

        except Exception as e:
            print(f"Error generating 3D model: {str(e)}")
            return False

    def _extrude_polygon_3d(self, points, height, thickness, scale, vertex_offset):
        """Extrude a 2D polygon to create 3D wall geometry"""
        vertices = []
        faces = []
```

```
        # Convert points to 3D coordinates
        points_3d = []
        for point in points:
            x = (point[0] - 128) * scale  # Center and scale
            z = (128 - point[1]) * scale  # Flip Y and scale
            points_3d.append([x, 0, z])

        # Create bottom vertices (y=0)
        bottom_outer = points_3d
        bottom_inner = self._inset_polygon(points_3d, thickness)

        # Create top vertices (y=height)
        top_outer = [[p[0], height, p[2]] for p in bottom_outer]
        top_inner = [[p[0], height, p[2]] for p in bottom_inner]

        # Combine all vertices
        all_vertices = bottom_outer + bottom_inner + top_outer + top_inner
        vertices.extend(all_vertices)

        n_points = len(points_3d)

        # Generate faces for walls
        for i in range(n_points):
            next_i = (i + 1) % n_points

            # Outer wall faces
            v1 = vertex_offset + i  # bottom outer
            v2 = vertex_offset + next_i  # bottom outer next
            v3 = vertex_offset + 2 * n_points + next_i  # top outer next
            v4 = vertex_offset + 2 * n_points + i  # top outer

            faces.extend([[v1, v2, v3], [v1, v3, v4]])

            # Inner wall faces (reverse winding)
            v1 = vertex_offset + n_points + i  # bottom inner
            v2 = vertex_offset + n_points + next_i  # bottom inner next
            v3 = vertex_offset + 3 * n_points + next_i  # top inner next
            v4 = vertex_offset + 3 * n_points + i  # top inner

            faces.extend([[v1, v3, v2], [v1, v4, v3]])

        # Top cap (between outer and inner)
        for i in range(n_points):
            next_i = (i + 1) % n_points

            v1 = vertex_offset + 2 * n_points + i  # top outer
            v2 = vertex_offset + 2 * n_points + next_i  # top outer next
            v3 = vertex_offset + 3 * n_points + next_i  # top inner next
            v4 = vertex_offset + 3 * n_points + i  # top inner

            faces.extend([[v1, v2, v3], [v1, v3, v4]])

        # Bottom cap (between outer and inner)
        for i in range(n_points):
            next_i = (i + 1) % n_points

            v1 = vertex_offset + i  # bottom outer
            v2 = vertex_offset + next_i  # bottom outer next
            v3 = vertex_offset + n_points + next_i  # bottom inner next
            v4 = vertex_offset + n_points + i  # bottom inner

            faces.extend([[v1, v3, v2], [v1, v4, v3]])

        return vertices, faces

    def _inset_polygon(self, points, inset_distance):
        """Create inset polygon for wall thickness"""
        if len(points) < 3:
            return points

        # Simple inset by moving each point inward along angle bisector
        inset_points = []
        n = len(points)
```

```
        for i in range(n):
            prev_i = (i - 1) % n
            next_i = (i + 1) % n

            p_prev = np.array(points[prev_i])
            p_curr = np.array(points[i])
            p_next = np.array(points[next_i])

            # Vectors to adjacent points
            v1 = p_curr - p_prev
            v2 = p_next - p_curr

            # Normalize vectors (in XZ plane, ignore Y)
            v1_norm = np.array([v1[0], 0, v1[2]])
            v2_norm = np.array([v2[0], 0, v2[2]])

            v1_len = np.linalg.norm(v1_norm)
            v2_len = np.linalg.norm(v2_norm)

            if v1_len > 1e-6:
                v1_norm /= v1_len
            if v2_len > 1e-6:
                v2_norm /= v2_len

            # Angle bisector
            bisector = v1_norm + v2_norm
            bisector_len = np.linalg.norm(bisector)

            if bisector_len > 1e-6:
                bisector /= bisector_len

                # Move point inward
                inset_point = p_curr - bisector * inset_distance
                inset_points.append([inset_point[0], inset_point[1], inset_point[2]])
            else:
                inset_points.append(points[i])

        return inset_points

    def _generate_floor_ceiling(self, mask, scale, wall_height, vertex_offset):
        """Generate floor and ceiling geometry from segmentation mask"""
        vertices = []
        faces = []

        # Find floor regions (assuming class 0 = floor/room)
        floor_mask = (mask == 0).astype(np.uint8)

        # Find contours
        contours, _ = cv2.findContours(
            floor_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
        )

        for contour in contours:
            if cv2.contourArea(contour) < 100:  # Skip small regions
                continue

            # Simplify contour
            epsilon = 0.02 * cv2.arcLength(contour, True)
            approx = cv2.approxPolyDP(contour, epsilon, True)

            if len(approx) < 3:
                continue

            # Convert to 3D coordinates
            floor_points = []
            for point in approx.reshape(-1, 2):
                x = (point[0] - 128) * scale
                z = (128 - point[1]) * scale
                floor_points.append([x, 0, z])  # Floor at y=0

            ceiling_points = []
            for point in approx.reshape(-1, 2):
```

```
                x = (point[0] - 128) * scale
                z = (128 - point[1]) * scale
                ceiling_points.append([x, wall_height, z])  # Ceiling at y=wall_height

            # Add vertices
            n_points = len(floor_points)
            vertices.extend(floor_points)
            vertices.extend(ceiling_points)

            # Triangulate floor
            if n_points >= 3:
                for i in range(1, n_points - 1):
                    faces.append(
                        [vertex_offset, vertex_offset + i + 1, vertex_offset + i]
                    )

                # Triangulate ceiling (reverse winding)
                for i in range(1, n_points - 1):
                    faces.append(
                        [
                            vertex_offset + n_points,
                            vertex_offset + n_points + i,
                            vertex_offset + n_points + i + 1,
                        ]
                    )

            vertex_offset += 2 * n_points

        return vertices, faces

    def process_batch(self, image_paths, output_dir):
        """Process multiple images in batch"""
        output_dir = Path(output_dir)
        output_dir.mkdir(exist_ok=True)

        results = []

        for img_path in image_paths:
            img_path = Path(img_path)
            print(f"Processing: {img_path.name}")

            output_path = output_dir / f"{img_path.stem}_model.obj"

            try:
                success = self.generate_3d_model(
                    str(img_path), str(output_path), export_intermediate=True
                )

                results.append({
                    "input": str(img_path),
                    "output": str(output_path),
                    "success": success
                })

                if success:
                    print(f"? Generated: {output_path}")
                else:
                    print(f"? Failed: {img_path.name}")

            except Exception as e:
                print(f"? Error processing {img_path.name}: {str(e)}")
                results.append({
                    "input": str(img_path),
                    "output": str(output_path),
                    "success": False,
                    "error": str(e)
                })

        return results
```

■ File: models\dvx.py

==

```
"""
Robust Differentiable Vectorization (DVX) module.

Improvements vs naive DVX:
- Projects backbone feature maps to `feature_dim` if channels don't match via 1x1 conv.
- Multi-step iterative refinement (improves final polygon accuracy).
- Safe guards for shapes, device handling, and grid-sampling.
- Returns init_polygons, final polygons, per-step displacements, and validity scores.

Usage:
- features: dict of feature maps (e.g. "p2", "p4"), each tensor (B, C, H, W).
- segmentation: (B, 1, H_img, W_img) or similar ? only used for optional initialization logic.
"""

from typing import Dict, Any, Optional, Tuple
import torch
import torch.nn as nn
import torch.nn.functional as F


class DifferentiableVectorization(nn.Module):
    def __init__(
        self,
        max_polygons: int = 20,
        max_points: int = 50,
        feature_dim: int = 256,
        displacement_scale: float = 0.12,
        num_refinement_steps: int = 3,
        align_corners: bool = False,
        padding_mode: str = "border",  # options for grid_sample
        use_proj_conv: bool = True,
    ):
        """
        Args:
            max_polygons: maximum polygons to predict per image
            max_points: number of control points per polygon
            feature_dim: number of channels the DVX expects (will project backbone features to this)
            displacement_scale: multiplier for predicted displacement (tanh output)
            num_refinement_steps: how many iterative refinement steps to apply (>=1)
            align_corners: align_corners for F.grid_sample
            padding_mode: padding_mode for F.grid_sample
            use_proj_conv: whether to use 1x1 conv to project backbone features to feature_dim
+                     (recommended)
        """
        super().__init__()
        assert max_points > 2, "max_points must be > 2"
        assert num_refinement_steps >= 1

        self.max_polygons = int(max_polygons)
        self.max_points = int(max_points)
        self.feature_dim = int(feature_dim)
        self.displacement_scale = float(displacement_scale)
        self.num_refinement_steps = int(num_refinement_steps)
        self.align_corners = bool(align_corners)
        self.padding_mode = padding_mode
        self.use_proj_conv = bool(use_proj_conv)

        # init_net: from pooled p4 -> flattened -> produce normalized coords in [0,1]
        # AdaptiveAvgPool2d(8) -> (B, C, 8, 8) -> flatten -> Linear(C*8*8 -> hidden)
        hidden = max(512, feature_dim * 2)
        self.init_pool = nn.AdaptiveAvgPool2d(8)

        # 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
        self.init_mlp = nn.Sequential(
            nn.Flatten(),
            nn.Linear(self.feature_dim * 8 * 8, hidden),
            nn.ReLU(inplace=True),
            nn.Linear(hidden, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, self.max_polygons * self.max_points * 2),
```

```
            nn.Sigmoid(),
        )

        # refinement network: maps (feature_dim + 2) -> displacement in [-1,1]
        self.refine_net = nn.Sequential(
            nn.Linear(self.feature_dim + 2, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 2),
            nn.Tanh(),
        )

        # validity net (reads flattened coords only)
        self.validity_net = nn.Sequential(
            nn.Linear(self.max_points * 2, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 1),
            nn.Sigmoid(),
        )

        # projector convs (create lazily when first seen a feature channel mismatch)
        # stored per-key: e.g., self._proj_convs['p2'] = nn.Conv2d(in_ch, feature_dim, 1)
        self._proj_convs = nn.ModuleDict()
        self._proj_created = set()

    def _ensure_projector(self, key: str, in_channels: int):
        """
        Ensure a 1x1 conv exists that projects `in_channels` -> self.feature_dim for feature map `key`.
        """
        if not self.use_proj_conv:
            return None
        if key in self._proj_created:
            return self._proj_convs[key]

        if in_channels != self.feature_dim:
            conv = nn.Conv2d(in_channels, self.feature_dim, kernel_size=1, stride=1, padding=0)
            # initialize conv: kaiming
            nn.init.kaiming_normal_(conv.weight, a=0.2)
            if conv.bias is not None:
                nn.init.zeros_(conv.bias)
            self._proj_convs[key] = conv
        else:
            # identity mapping using 1x1 conv with weights = identity-like is tricky
            # Instead simply keep no conv; we'll pass feature as-is
            self._proj_convs[key] = nn.Identity()
        self._proj_created.add(key)
        return self._proj_convs[key]

    def _project_feature(self, key: str, feat: torch.Tensor) -> torch.Tensor:
        """
        Project or verify feature map to have self.feature_dim channels.
        If projector conv wasn't present and channels == feature_dim, returns feat unchanged.
        """
        in_ch = feat.shape[1]
        proj = self._ensure_projector(key, in_ch)
        if proj is None:
            # projection not desired; assert channels match
            if in_ch != self.feature_dim:
                raise RuntimeError(
                    f"Feature '{key}' channels ({in_ch}) != feature_dim ({self.feature_dim}) "
                    "and projection disabled."
                )
            return feat
        # if proj is Identity, apply it still (fast path)
        return proj(feat)

    def forward(
        self,
        features: Dict[str, torch.Tensor],
        segmentation: Optional[torch.Tensor] = None,
        return_all_steps: bool = False,
```

```
    ) -> Dict[str, Any]:
        """
        features: dict with keys like "p2", "p4" containing tensors (B, C, H, W)
        segmentation: optional (B, 1, H_img, W_img) or similar (not strictly required)
        return_all_steps: if True returns per-step intermediate polygons & displacements
        """
        # pick features for init and refinement
        p4 = features.get("p4", None)
        p2 = features.get("p2", None)

        if p4 is None and p2 is None:
            raise ValueError("At least one of 'p4' or 'p2' must be present in features.")

        # prefer p4 for init; fallback to p2 if not present
        init_feat = p4 if p4 is not None else p2
        refine_feat = p2 if p2 is not None else p4

        B = init_feat.shape[0]

        # Project features to feature_dim (if needed)
        init_feat = self._project_feature("p4_init", init_feat)
        refine_feat = self._project_feature("p2_refine", refine_feat)

        # -- Initialize polygons --
        # Pool then MLP; ensure init_mlp expects feature_dim channels
        pooled = self.init_pool(init_feat)  # [B, C', 8, 8]
        if pooled.shape[1] != self.feature_dim:
            # If the projector returned Identity but pooled channels mismatch, try to apply a
+                    runtime projector
            pooled = self._project_feature("p4_init_postpool", pooled)

        init_logits = self.init_mlp(pooled)  # [B, max_polygons * max_points * 2]
        init_polygons = init_logits.view(B, self.max_polygons, self.max_points, 2)  # normalized
+                [0,1]

        # Iterative refinement
        polygons = init_polygons.clone()
        per_step_displacements = []
        for step in range(self.num_refinement_steps):
            # sample features at the polygon control-point locations
            displ = self._single_refine_step(polygons, refine_feat)
            per_step_displacements.append(displ)
            polygons = torch.clamp(polygons + displ, 0.0, 1.0)

        # final validity
        validity = self._predict_validity(polygons)

        out: Dict[str, Any] = {
            "polygons": polygons,  # [B, P, N, 2]
            "validity": validity,  # [B, P]
            "init_polygons": init_polygons,
            "refinement_displacements": per_step_displacements,  # list of [B, P, N, 2]
        }

        if return_all_steps:
            out["all_step_polygons"] = [
                torch.clamp(init_polygons + sum(per_step_displacements[:i + 1]), 0.0, 1.0)
                for i in range(len(per_step_displacements))
            ]

        return out

    def _single_refine_step(self, polygons: torch.Tensor, feature_map: torch.Tensor) ->
+            torch.Tensor:
        """
        One refinement step: sample features at polygon points, predict displacement (scaled),
+                return displacement.
        polygons: [B, P, N, 2] in [0,1]
        feature_map: [B, C, H, W] with C == feature_dim (or projected)
        returns displacement: [B, P, N, 2] in [-displacement_scale, displacement_scale]
        """
        B, P, N, _ = polygons.shape
        # flatten pts to sample
```

```
        coords = polygons.view(B, -1, 2)  # [B, P*N, 2], coords in [0,1]
        grid = coords * 2.0 - 1.0  # to [-1,1]
        # grid_sample expects (B, H_out, W_out, 2); use W_out=1
        grid_sample = grid.view(B, -1, 1, 2)
        sampled = F.grid_sample(
            feature_map,
            grid_sample,
            mode="bilinear",
            padding_mode=self.padding_mode,
            align_corners=self.align_corners,
        )  # [B, C, P*N, 1]
        sampled = sampled.squeeze(-1).permute(0, 2, 1).contiguous()  # [B, P*N, C]

        # combine sampled features and coords (coords in [0,1])
        input_feats = torch.cat([sampled, coords], dim=-1)  # [B, P*N, C+2]
        # predict displacements in [-1,1] via tanh on last layer
        disp = self.refine_net(input_feats)  # [B, P*N, 2], values ~[-1,1]
        disp = disp.view(B, P, N, 2)
        disp = disp * self.displacement_scale  # scale
        return disp

    def _predict_validity(self, polygons: torch.Tensor) -> torch.Tensor:
        B, P, N, _ = polygons.shape
        if N != self.max_points:
            # If someone truncated or padded points, adapt: flatten to last dim whatever it is
            poly_flat = polygons.view(B * P, -1)
        else:
            poly_flat = polygons.view(B * P, -1)
        validity = self.validity_net(poly_flat)  # [B*P, 1]
        validity = validity.view(B, P)
        return validity


# ------------------ quick unit test / smoke test ------------------
def _smoke_test():
    torch.manual_seed(0)
    B = 2
    C1 = 384  # different from feature_dim to test projector conv
    C2 = 128
    H2, W2 = 64, 64
    H4, W4 = 16, 16

    # create dummy backbone features with different channels
    p2 = torch.randn(B, C1, H2, W2)
    p4 = torch.randn(B, C2, H4, W4)
    seg = torch.rand(B, 1, H2 * 4, W2 * 4)  # just a placeholder

    dvx = DifferentiableVectorization(
        max_polygons=4,
        max_points=16,
        feature_dim=256,
        displacement_scale=0.08,
        num_refinement_steps=3,
        align_corners=False,
        padding_mode="border",
        use_proj_conv=True,
    )

    # ensure module moves projector convs to device when dvx.to(device) called
    dvx = dvx.eval()  # inference mode ok
    # Forward pass
    out = dvx({"p2": p2, "p4": p4}, seg, return_all_steps=True)
    print("polygons shape:", out["polygons"].shape)  # expected [B, P, N, 2]
    print("validity shape:", out["validity"].shape)  # expected [B, P]
    print("init shape:", out["init_polygons"].shape)
    print("refinement steps:", len(out["refinement_displacements"]))
    # check ranges
    assert out["polygons"].min().item() >= 0.0 - 1e-6
    assert out["polygons"].max().item() <= 1.0 + 1e-6
    print("smoke test passed")


if __name__ == "__main__":
```

```python
    _smoke_test()
```

■ File: models\encoder.py

```python
"""
Encoder architecture for multi-scale feature extraction
"""

import torch
import torch.nn as nn
import torch.nn.functional as F


class ResidualBlock(nn.Module):
    """Basic residual block for the encoder"""

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, stride, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1, stride, bias=False),
                nn.BatchNorm2d(out_channels),
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        return F.relu(out)


class MultiScaleEncoder(nn.Module):
    """
    Advanced encoder with skip connections and multi-scale feature extraction
    Based on ResNet architecture with Feature Pyramid Network (FPN)
    """

    def __init__(self, input_channels=3, feature_dim=512):
        super().__init__()

        # Stem
        self.stem = nn.Sequential(
            nn.Conv2d(input_channels, 64, 7, 2, 3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 2, 1),
        )

        # ResNet blocks
        self.layer1 = self._make_layer(64, 64, 2, stride=1)  # 64x64
        self.layer2 = self._make_layer(64, 128, 2, stride=2)  # 32x32
        self.layer3 = self._make_layer(128, 256, 2, stride=2)  # 16x16
        self.layer4 = self._make_layer(256, 512, 2, stride=2)  # 8x8

        # FPN lateral connections
        self.lateral4 = nn.Conv2d(512, feature_dim, 1)
        self.lateral3 = nn.Conv2d(256, feature_dim, 1)
        self.lateral2 = nn.Conv2d(128, feature_dim, 1)
        self.lateral1 = nn.Conv2d(64, feature_dim, 1)

        # FPN output layers
        self.smooth4 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
        self.smooth3 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
        self.smooth2 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
```

```
        self.smooth1 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)

        # Global context
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.global_fc = nn.Sequential(
            nn.Linear(512, feature_dim),
            nn.ReLU(),
            nn.Linear(feature_dim, feature_dim)
        )

    def _make_layer(self, in_channels, out_channels, blocks, stride=1):
        layers = []
        layers.append(ResidualBlock(in_channels, out_channels, stride))
        for _ in range(1, blocks):
            layers.append(ResidualBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        # Bottom-up pathway
        x = self.stem(x)  # 64x64

        c1 = self.layer1(x)  # 64x64
        c2 = self.layer2(c1)  # 32x32
        c3 = self.layer3(c2)  # 16x16
        c4 = self.layer4(c3)  # 8x8

        # Global context
        global_feat = self.global_pool(c4).flatten(1)
        global_feat = self.global_fc(global_feat)

        # Top-down pathway (FPN)
        p4 = self.lateral4(c4)
        p3 = self.lateral3(c3) + F.interpolate(p4, scale_factor=2)
        p2 = self.lateral2(c2) + F.interpolate(p3, scale_factor=2)
        p1 = self.lateral1(c1) + F.interpolate(p2, scale_factor=2)

        # Smooth
        p4 = self.smooth4(p4)
        p3 = self.smooth3(p3)
        p2 = self.smooth2(p2)
        p1 = self.smooth1(p1)

        return {
            "p1": p1,  # 64x64
            "p2": p2,  # 32x32
            "p3": p3,  # 16x16
            "p4": p4,  # 8x8
            "global": global_feat,
        }
```

■ File: models\extrusion.py

==

```
"""
Vectorized Differentiable 3D extrusion module for converting polygons to 3D occupancy
Optimized version with GPU-accelerated vectorized operations
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import logging


# -----------------------------------------------------------------------------
# Logging and sanitization helper
# -----------------------------------------------------------------------------
logger = logging.getLogger(__name__)
if not logger.handlers:
    handler = logging.StreamHandler()
    handler.setFormatter(logging.Formatter("%(asctime)s | %(levelname)s | %(message)s"))
```

```
    logger.addHandler(handler)
logger.setLevel(logging.INFO)


def _sanitize_normalized_height(value, sample_id=None, default=0.6):
    """
    Ensure normalized height value is finite and in [0,1].
    Returns a float in [0,1].

    Args:
        value: torch scalar tensor or float
        sample_id: optional identifier for logging (string or int)
        default: fallback normalized height
    """
    try:
        if isinstance(value, torch.Tensor):
            raw = float(value.item())
        else:
            raw = float(value)
    except Exception:
        raw = float("nan")

    # Build label for logging
    sid = f"[sample={sample_id}]" if sample_id is not None else ""

    # Check finite
    if not math.isfinite(raw):
        logger.warning(f"{sid} Invalid wall height value (not finite): {raw}; using default
+                 {default}")
        raw = default

    # Clamp to [0,1]
    if raw < 0.0 or raw > 1.0:
        logger.warning(f"{sid} Wall height normalized {raw} out of [0,1]; clamping.")
        raw = max(0.0, min(1.0, raw))

    return raw


def _sanitize_tensor(tensor, default_value=0.0, name="tensor"):
    """
    Sanitize an entire tensor by replacing NaN/Inf values with default.

    Args:
        tensor: Input tensor
        default_value: Value to replace invalid entries with
        name: Name for logging

    Returns:
        Sanitized tensor
    """
    if tensor.numel() == 0:
        return tensor

    # Check for any invalid values
    invalid_mask = ~torch.isfinite(tensor)
    num_invalid = invalid_mask.sum().item()

    if num_invalid > 0:
        logger.warning(f"Found {num_invalid} invalid values in {name}, replacing with
+                 {default_value}")
        tensor = tensor.clone()
        tensor[invalid_mask] = default_value

    return tensor


# ---------------------------------------------------------------------------
# Main extrusion module
# ---------------------------------------------------------------------------
class DifferentiableExtrusion(nn.Module):
    """
    Vectorized Differentiable 3D extrusion module
```

```
    Converts polygons + attributes to soft 3D occupancy grids
    """

    def __init__(self, voxel_size: int = 64):
        super().__init__()
        self.voxel_size = int(voxel_size)
        self.register_buffer("_coords", None)

    def _ensure_coords(self, device):
        """Initialize or update coordinate grid if needed"""
        if (self._coords is None or
            self._coords.device != device or
            self._coords.shape[0] != (self.voxel_size * self.voxel_size)):

            H = W = self.voxel_size
            y, x = torch.meshgrid(
                torch.arange(H, device=device),
                torch.arange(W, device=device),
                indexing="ij"
            )
            coords = torch.stack([x.flatten().float(), y.flatten().float()], dim=1)  # [H*W, 2]
            coords = coords / float(self.voxel_size - 1)
            self.register_buffer("_coords", coords)

    def polygon_sdf(self, polygon_xy):
        """
        Compute signed distance field for a polygon using vectorized operations.
        """
        device = polygon_xy.device
        self._ensure_coords(device)
        pts = self._coords  # [M, 2]
        P = polygon_xy.shape[0]

        if P < 2:
            return torch.full((pts.shape[0],), 1.0, device=device)

        # Sanitize polygon coordinates
        polygon_xy = _sanitize_tensor(polygon_xy, default_value=0.0, name="polygon_xy")

        v0 = polygon_xy.unsqueeze(1)
        v1 = torch.roll(polygon_xy, shifts=-1, dims=0).unsqueeze(1)
        pts_exp = pts.unsqueeze(0)

        e = v1 - v0
        v = pts_exp - v0
        e_norm_sq = (e**2).sum(dim=2, keepdim=True) + 1e-8
        t = (v * e).sum(dim=2, keepdim=True) / e_norm_sq
        t_clamped = t.clamp(0.0, 1.0)

        proj = v0 + t_clamped * e
        diff = pts_exp - proj
        dists = torch.norm(diff, dim=2)

        # Sanitize distances before min operation
        dists = _sanitize_tensor(dists, default_value=1.0, name="distances")
        min_dist_per_point, _ = dists.min(dim=0)

        x_pts = pts[:, 0].unsqueeze(0)
        y_pts = pts[:, 1].unsqueeze(0)
        x0, y0 = v0[..., 0], v0[..., 1]
        x1, y1 = v1[..., 0], v1[..., 1]

        y_crosses = ((y0 <= y_pts) & (y1 > y_pts)) | ((y1 <= y_pts) & (y0 > y_pts))
        inter_x = x0 + (x1 - x0) * ((y_pts - y0) / (y1 - y0 + 1e-8))
        crossings = (inter_x > x_pts) & y_crosses
        crossing_count = crossings.sum(dim=0)
        inside = (crossing_count % 2 == 1)

        sdf = min_dist_per_point.clone()
        sdf[inside] = -sdf[inside]

        # Final sanitization of SDF output
        sdf = _sanitize_tensor(sdf, default_value=1.0, name="sdf")
```

```
        return sdf

    def forward(self, polygons, attributes, validity_scores, sample_ids=None):
        """
        Convert polygons to 3D voxel occupancy.
        sample_ids: optional list/array of identifiers (e.g., filenames or dataset indices)
        """
        device = polygons.device
        B, N, P, _ = polygons.shape
        D = H = W = self.voxel_size

        # Sanitize input tensors
        polygons = _sanitize_tensor(polygons, default_value=0.0, name="input_polygons")
        attributes = _sanitize_tensor(attributes, default_value=0.6, name="input_attributes")
        validity_scores = _sanitize_tensor(validity_scores, default_value=0.0,
                 name="input_validity_scores")

        voxels = torch.zeros((B, D, H, W), device=device)

        for b in range(B):
            # pick identifier if available
            sid = sample_ids[b] if sample_ids is not None else b

            # Sanitize height with logging
            wall_height_normalized = attributes[b, 0]
            sanitized_norm = _sanitize_normalized_height(
                wall_height_normalized, sample_id=sid, default=0.6
            )

            wall_height_m = sanitized_norm * 5.0
            height_frac = wall_height_m / 5.0
            height_voxels = int(round(height_frac * D))
            height_voxels = max(1, min(D, height_voxels))

            # Process each polygon for this batch
            validity_mask = validity_scores[b] >= 0.5
            if not validity_mask.any():
                continue

            combined_mask = torch.zeros((H, W), device=device)
            sharpness = 100.0

            for n in range(N):
                if not validity_mask[n]:
                    continue

                polygon = polygons[b, n]  # [P, 2]

                # Filter out zero-padded vertices
                vertex_mask = (polygon.sum(dim=1) != 0.0)
                if vertex_mask.sum().item() < 3:
                    continue

                valid_polygon = polygon[vertex_mask]

                # Compute SDF for this polygon
                sdf = self.polygon_sdf(valid_polygon)
                mask = torch.sigmoid(-sdf * sharpness)
                mask_2d = mask.view(H, W)

                # Sanitize mask before combining
                mask_2d = _sanitize_tensor(mask_2d, default_value=0.0, name=f"mask_2d_b{b}_n{n}")
                combined_mask = torch.maximum(combined_mask, mask_2d)

            # Create 3D mask by extruding to the computed height
            mask_3d = combined_mask.unsqueeze(0).expand(height_voxels, -1, -1)

            # Sanitize final mask before assignment
            mask_3d = _sanitize_tensor(mask_3d, default_value=0.0, name=f"final_mask_3d_b{b}")
            voxels[b, :height_voxels] = mask_3d

        # Final sanitization of output
        voxels = _sanitize_tensor(voxels, default_value=0.0, name="output_voxels")
```

```
        return voxels


# ------------------------------------------------------------------------------
# Fast extrusion module
# ------------------------------------------------------------------------------
class DifferentiableExtrusionFast(nn.Module):
    """
    Optimized version that batches polygon processing.
    """

    def __init__(self, voxel_size: int = 64):
        super().__init__()
        self.voxel_size = int(voxel_size)
        self.register_buffer("_coords", None)

    def _ensure_coords(self, device):
        if (self._coords is None or
            self._coords.device != device or
            self._coords.shape[0] != (self.voxel_size * self.voxel_size)):

            H = W = self.voxel_size
            y, x = torch.meshgrid(
                torch.arange(H, device=device),
                torch.arange(W, device=device),
                indexing="ij"
            )
            coords = torch.stack([x.flatten().float(), y.flatten().float()], dim=1)
            coords = coords / float(self.voxel_size - 1)
            self.register_buffer("_coords", coords)

    def batch_polygon_sdf(self, polygons_batch, validity_mask):
        device = polygons_batch.device
        self._ensure_coords(device)

        N, P, _ = polygons_batch.shape
        M = self._coords.shape[0]
        sdfs = torch.full((N, M), 1.0, device=device)

        valid_indices = torch.where(validity_mask)[0]
        if len(valid_indices) == 0:
            return sdfs

        valid_polygons = polygons_batch[valid_indices]
        for i, poly_idx in enumerate(valid_indices):
            poly = valid_polygons[i]
            vertex_mask = (poly.sum(dim=1) != 0.0)
            if vertex_mask.sum().item() >= 3:
                valid_poly = poly[vertex_mask]
                sdf = self.polygon_sdf(valid_poly)
                sdfs[poly_idx] = sdf

        return sdfs

    def polygon_sdf(self, polygon_xy):
        device = polygon_xy.device
        self._ensure_coords(device)
        pts = self._coords
        P = polygon_xy.shape[0]

        if P < 2:
            return torch.full((pts.shape[0],), 1.0, device=device)

        # Sanitize polygon coordinates
        polygon_xy = _sanitize_tensor(polygon_xy, default_value=0.0, name="polygon_xy_fast")

        v0 = polygon_xy.unsqueeze(1)
        v1 = torch.roll(polygon_xy, shifts=-1, dims=0).unsqueeze(1)
        pts_exp = pts.unsqueeze(0)

        e = v1 - v0
        v = pts_exp - v0
        e_norm_sq = (e**2).sum(dim=2, keepdim=True) + 1e-8
```

```
        t = (v * e).sum(dim=2, keepdim=True) / e_norm_sq
        t_clamped = t.clamp(0.0, 1.0)

        proj = v0 + t_clamped * e
        diff = pts_exp - proj
        dists = torch.norm(diff, dim=2)

        # Sanitize distances before min operation
        dists = _sanitize_tensor(dists, default_value=1.0, name="distances_fast")
        min_dist_per_point, _ = dists.min(dim=0)

        x_pts = pts[:, 0].unsqueeze(0)
        y_pts = pts[:, 1].unsqueeze(0)
        x0, y0 = v0[..., 0], v0[..., 1]
        x1, y1 = v1[..., 0], v1[..., 1]

        y_crosses = ((y0 <= y_pts) & (y1 > y_pts)) | ((y1 <= y_pts) & (y0 > y_pts))
        inter_x = x0 + (x1 - x0) * ((y_pts - y0) / (y1 - y0 + 1e-8))
        crossings = (inter_x > x_pts) & y_crosses
        crossing_count = crossings.sum(dim=0)
        inside = (crossing_count % 2 == 1)

        sdf = min_dist_per_point.clone()
        sdf[inside] = -sdf[inside]

        # Final sanitization of SDF output
        sdf = _sanitize_tensor(sdf, default_value=1.0, name="sdf_fast")
        return sdf

    def forward(self, polygons: torch.Tensor, attributes: torch.Tensor, validity_scores:
+             torch.Tensor) -> torch.Tensor:
        device = polygons.device
        B, N, P, _ = polygons.shape
        D = H = W = self.voxel_size

        # Sanitize input tensors
        polygons = _sanitize_tensor(polygons, default_value=0.0, name="input_polygons_fast")
        attributes = _sanitize_tensor(attributes, default_value=0.6, name="input_attributes_fast")
        validity_scores = _sanitize_tensor(validity_scores, default_value=0.0,
+                 name="input_validity_scores_fast")

        voxels = torch.zeros((B, D, H, W), device=device)

        for b in range(B):
            validity_mask = validity_scores[b] >= 0.5
            if not validity_mask.any():
                continue

            sdfs = self.batch_polygon_sdf(polygons[b], validity_mask)

            # Sanitize SDFs before sigmoid
            sdfs = _sanitize_tensor(sdfs, default_value=1.0, name=f"batch_sdfs_b{b}")

            sharpness = 100.0
            masks = torch.sigmoid(-sdfs * sharpness)
            masks_2d = masks.view(N, H, W)

            # Sanitize masks
            masks_2d = _sanitize_tensor(masks_2d, default_value=0.0, name=f"masks_2d_b{b}")

            # Sanitize height
            wall_height_normalized = attributes[b, 0]
            sanitized_norm = _sanitize_normalized_height(wall_height_normalized, sample_id=b,
+                     default=0.6)
            wall_height_m = sanitized_norm * 5.0
            height_frac = wall_height_m / 5.0
            height_voxels = int(round(height_frac * D))
            height_voxels = max(1, min(D, height_voxels))

            combined_mask = torch.zeros((H, W), device=device)
            for n in range(N):
                if validity_mask[n]:
                    combined_mask = torch.maximum(combined_mask, masks_2d[n])
```

```python

            mask_3d = combined_mask.unsqueeze(0).expand(height_voxels, -1, -1)

            # Sanitize final mask before assignment
            mask_3d = _sanitize_tensor(mask_3d, default_value=0.0, name=f"final_mask_3d_fast_b{b}")
            voxels[b, :height_voxels] = mask_3d

        # Final sanitization of output
        voxels = _sanitize_tensor(voxels, default_value=0.0, name="output_voxels_fast")
        return voxels
```

■ File: models\heads.py

==

```python
"""
Multi-task prediction heads for the Neural-Geometric 3D Model Generator
"""

import torch
import torch.nn as nn
import torch.nn.functional as F


class SegmentationHead(nn.Module):
    """Semantic segmentation head with multi-scale fusion"""

    def __init__(self, feature_dim=512, num_classes=5, dropout=0.1):
        super().__init__()

        # Multi-scale fusion
        self.fusion = nn.Sequential(
            nn.Conv2d(feature_dim * 4, feature_dim, 3, 1, 1),
            nn.BatchNorm2d(feature_dim),
            nn.ReLU(),
            nn.Dropout2d(dropout),
        )

        # Segmentation decoder
        self.decoder = nn.Sequential(
            nn.Conv2d(feature_dim, feature_dim // 2, 3, 1, 1),
            nn.BatchNorm2d(feature_dim // 2),
            nn.ReLU(),
            nn.Conv2d(feature_dim // 2, feature_dim // 4, 3, 1, 1),
            nn.BatchNorm2d(feature_dim // 4),
            nn.ReLU(),
            nn.Conv2d(feature_dim // 4, num_classes, 1),
        )

    def forward(self, features):
        # Fuse multi-scale features
        p1, p2, p3, p4 = features["p1"], features["p2"], features["p3"], features["p4"]

        # Upsample all to p1 resolution
        p2_up = F.interpolate(
            p2, size=p1.shape[-2:], mode="bilinear", align_corners=False
        )
        p3_up = F.interpolate(
            p3, size=p1.shape[-2:], mode="bilinear", align_corners=False
        )
        p4_up = F.interpolate(
            p4, size=p1.shape[-2:], mode="bilinear", align_corners=False
        )

        fused = torch.cat([p1, p2_up, p3_up, p4_up], dim=1)
        fused = self.fusion(fused)

        # Final segmentation
        seg = self.decoder(fused)
        return F.interpolate(seg, scale_factor=4, mode="bilinear", align_corners=False)


class AttributeHead(nn.Module):
```

```
    """Attribute regression head for geometric parameters"""

    def __init__(self, feature_dim=512, num_attributes=6, dropout=0.2):
        super().__init__()

        self.regressor = nn.Sequential(
            nn.Linear(feature_dim, feature_dim),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(feature_dim, feature_dim // 2),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(feature_dim // 2, num_attributes),
            nn.Sigmoid(),  # Output in [0,1] range
        )

    def forward(self, global_features):
        return self.regressor(global_features)


class SDFHead(nn.Module):
    """Signed Distance Field prediction for sharp boundaries"""

    def __init__(self, feature_dim=512, dropout=0.1):
        super().__init__()

        self.sdf_decoder = nn.Sequential(
            nn.Conv2d(feature_dim, feature_dim // 2, 3, 1, 1),
            nn.BatchNorm2d(feature_dim // 2),
            nn.ReLU(),
            nn.Dropout2d(dropout),
            nn.Conv2d(feature_dim // 2, feature_dim // 4, 3, 1, 1),
            nn.BatchNorm2d(feature_dim // 4),
            nn.ReLU(),
            nn.Conv2d(feature_dim // 4, 1, 1),
            nn.Tanh(),  # SDF in [-1, 1]
        )

    def forward(self, features):
        # Use highest resolution features
        p1 = features["p1"]
        sdf = self.sdf_decoder(p1)
        return F.interpolate(sdf, scale_factor=4, mode="bilinear", align_corners=False)
```

■ File: models\model.py

==

```
"""
Advanced loss functions for multi-task training with dynamic weighting
Enhanced with cross-modal consistency, graph constraints, and GradNorm
Modified to support conditional geometric losses via run_full_geometric flag
"""
import torch
import torch.nn as nn
import torch.nn.functional as F
from .encoder import MultiScaleEncoder
from .heads import SegmentationHead, AttributeHead, SDFHead
from .dvx import DifferentiableVectorization
from .extrusion import DifferentiableExtrusion


class L2Normalize(nn.Module):
    """L2 normalization layer"""

    def __init__(self, dim=1):
        super().__init__()
        self.dim = dim

    def forward(self, x):
        return F.normalize(x, p=2, dim=self.dim)


```

```python
class LatentEmbeddingHead(nn.Module):
    """Auxiliary head for cross-modal latent consistency"""

    def __init__(self, feature_dim: int, embedding_dim: int = 256):
        super().__init__()
        self.embedding_dim = embedding_dim

        # 2D embedding path
        self.embedding_2d = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(feature_dim, embedding_dim * 2),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(embedding_dim * 2, embedding_dim),
            L2Normalize(dim=1),  # L2 normalize for cosine similarity
        )

        # 3D embedding path (from voxel features)
        self.embedding_3d = nn.Sequential(
            nn.AdaptiveAvgPool3d((1, 1, 1)),
            nn.Flatten(),
            nn.Linear(feature_dim, embedding_dim * 2),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(embedding_dim * 2, embedding_dim),
            L2Normalize(dim=1),
        )

    def forward(
        self, features_2d: torch.Tensor, features_3d: torch.Tensor = None
    ) -> tuple:
        """
        Generate 2D and 3D embeddings for consistency loss

        Args:
            features_2d: [B, C, H, W] - 2D feature maps
            features_3d: [B, C, D, H, W] - 3D feature maps (optional)

        Returns:
            tuple: (embedding_2d, embedding_3d)
        """
        # 2D embedding
        emb_2d = self.embedding_2d(features_2d)

        # 3D embedding (if available, otherwise use 2D features reshaped)
        if features_3d is not None:
            emb_3d = self.embedding_3d(features_3d)
        else:
            # Create pseudo-3D from 2D features
            B, C, H, W = features_2d.shape
            pseudo_3d = features_2d.unsqueeze(2).expand(
                B, C, 4, H, W
            )  # Duplicate across depth
            emb_3d = self.embedding_3d(pseudo_3d)

        return emb_2d, emb_3d


class GraphStructureHead(nn.Module):
    """Head for predicting graph structure (room connectivity)"""

    def __init__(self, feature_dim: int, max_rooms: int = 10):
        super().__init__()
        self.max_rooms = max_rooms

        # Room detection branch
        self.room_detector = nn.Sequential(
            nn.Conv2d(feature_dim, feature_dim // 2, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(feature_dim // 2, max_rooms, 3, padding=1),
            nn.Sigmoid(),  # Room probability maps
        )
```

```
        # Room feature extractor
        self.room_features = nn.Sequential(
            nn.AdaptiveAvgPool2d((8, 8)),  # Pool to fixed size
            nn.Flatten(),
            nn.Linear(feature_dim * 64, 256),
            nn.ReLU(),
            nn.Linear(256, 128),  # Room feature vectors
        )

        # Adjacency predictor
        self.adjacency_net = nn.Sequential(
            nn.Linear(128 * 2, 64),  # Pairwise room features
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
            nn.Sigmoid(),  # Adjacency probability
        )

    def forward(self, features: torch.Tensor) -> dict:
        """
        Predict room graph structure

        Args:
            features: [B, C, H, W] - Feature maps

        Returns:
            dict with 'room_maps', 'room_features', 'adjacency_matrix'
        """
        B = features.shape[0]

        # Detect room probability maps
        room_maps = self.room_detector(features)  # [B, max_rooms, H, W]

        # Extract room features
        room_feats = self.room_features(features)  # [B, 128]

        # Create adjacency matrix for all room pairs
        adjacency_matrices = []

        for b in range(B):
            # Get room features for this batch item
            feat_b = room_feats[b : b + 1]  # [1, 128]

            # Create pairwise combinations
            adj_matrix = torch.zeros(
                (self.max_rooms, self.max_rooms), device=features.device
            )

            for i in range(self.max_rooms):
                for j in range(i + 1, self.max_rooms):
                    # Concatenate features for room pair
                    pair_feat = torch.cat([feat_b, feat_b], dim=1)  # [1, 256]

                    # Predict adjacency
                    adj_prob = self.adjacency_net(pair_feat)  # [1, 1]

                    # Fill symmetric matrix
                    adj_matrix[i, j] = adj_prob.squeeze()
                    adj_matrix[j, i] = adj_prob.squeeze()

            adjacency_matrices.append(adj_matrix)

        return {
            "room_maps": room_maps,
            "room_features": room_feats,
            "adjacency_matrices": torch.stack(adjacency_matrices),
        }


class NeuralGeometric3DGenerator(nn.Module):
    """
```

```
    Enhanced neural-geometric system with auxiliary heads for novel training strategies:
    - Cross-modal latent consistency
    - Graph structure prediction
    - Multi-view embeddings for dynamic curriculum
    - Conditional geometric computation via run_full_geometric flag
    """

    def __init__(
        self,
        input_channels=3,
        num_classes=5,
        feature_dim=512,
        num_attributes=6,
        voxel_size=64,
        max_polygons=20,
        max_points=50,
        use_latent_consistency=True,
        use_graph_constraints=True,
        latent_embedding_dim=256,
    ):
        super().__init__()

        # Store configuration
        self.use_latent_consistency = use_latent_consistency
        self.use_graph_constraints = use_graph_constraints
        self.feature_dim = feature_dim

        # Core components
        self.encoder = MultiScaleEncoder(input_channels, feature_dim)
        self.seg_head = SegmentationHead(feature_dim, num_classes)
        self.attr_head = AttributeHead(feature_dim, num_attributes)
        self.sdf_head = SDFHead(feature_dim)
        self.dvx = DifferentiableVectorization(max_polygons, max_points, feature_dim)
        self.extrusion = DifferentiableExtrusion(voxel_size)

        # NEW: Auxiliary heads for novel training strategies
        if use_latent_consistency:
            self.latent_head = LatentEmbeddingHead(feature_dim, latent_embedding_dim)

        if use_graph_constraints:
            self.graph_head = GraphStructureHead(feature_dim)

        # Enhanced feature processing for multi-stage training
        self.feature_enhancer = nn.Sequential(
            nn.Conv2d(feature_dim, feature_dim, 3, padding=1),
            nn.GroupNorm(32, feature_dim),
            nn.ReLU(),
            nn.Conv2d(feature_dim, feature_dim, 3, padding=1),
            nn.GroupNorm(32, feature_dim),
        )

        # lazy-created 3d voxel processor will be attached on first use
        self._voxel_processor = None

    def _select_spatial_feature(self, features):
        """
        Given encoder output (dict or tensor), select a spatial 4-D feature map
        Prefer high-resolution feature maps (p1) and avoid selecting 'global' vector.
        """
        # If encoder returned a tensor already, make sure it's 4D
        if not isinstance(features, dict):
            if features.dim() == 4:
                return features
            else:
                raise ValueError(
                    f"Encoder returned a tensor with shape {tuple(features.shape)}; "
                    "expected a 4D feature map [B, C, H, W]."
                )

        # Encoder returned dict: prefer p1,p2,p3,p4,high_res,out,main but NEVER 'global'
        preferred_keys = ["p1", "p2", "p3", "p4", "high_res", "out", "main"]
        for k in preferred_keys:
            if k in features:
```

```
                candidate = features[k]
                if isinstance(candidate, torch.Tensor) and candidate.dim() == 4:
                    return candidate

        # As a last resort, scan dict values for the first 4D tensor that isn't 'global'
        for k, v in features.items():
            if k == "global":
                continue
            if isinstance(v, torch.Tensor) and v.dim() == 4:
                return v

        # If nothing found, raise informative error rather than silently picking wrong shape
        raise RuntimeError(
            "No spatial 4D feature map found in encoder output. Encoder returned keys: "
            f"{list(features.keys())}. Ensure encoder provides at least one [B,C,H,W] tensor "
            "under keys like 'p1','p2','p3','p4','out', or 'high_res'."
        )

    def forward(self, image, run_full_geometric=True, return_aux=True):
        """
        Enhanced forward pass with auxiliary outputs and conditional geometric computation

        Args:
            image: [B, C, H, W] input images
            run_full_geometric: Whether to run heavy DVX and extrusion computations
            return_aux: Whether to compute auxiliary outputs

        Returns:
            dict with predictions, conditionally including geometric outputs
        """
        # Multi-scale feature extraction
        features = self.encoder(image)

        # Enhance features
        spatial_feat = self._select_spatial_feature(features)
        enhanced_features = self.feature_enhancer(spatial_feat)

        # keep structured features dict for heads that expect multi-scale inputs
        if isinstance(features, dict):
            features["enhanced"] = enhanced_features
            main_features = enhanced_features
        else:
            features = {"main": enhanced_features, "enhanced": enhanced_features}
            main_features = enhanced_features

        # Core predictions (always computed - these are fast)
        segmentation = self.seg_head(features)
        attributes = self.attr_head(
            features.get("global")
            if isinstance(features, dict) and "global" in features
            else main_features.mean(dim=[2, 3])
        )
        sdf = self.sdf_head(features)

        # Base outputs
        outputs = {
            "segmentation": segmentation,
            "attributes": attributes,
            "sdf": sdf,
            "features": features,
        }

        # Conditional geometric computation (heavy operations)
        if run_full_geometric:
            # DVX polygon fitting
            dvx_output = self.dvx(features, segmentation)
            polygons = dvx_output.get("polygons", None)
            validity = dvx_output.get("validity", None)

            # 3D extrusion (defensive: ensure inputs exist)
            try:
                voxels_pred = self.extrusion(polygons, attributes, validity)
            except Exception as e:
```

```
                # Log or print a helpful message for debugging; avoid crashing training
                # (Replace print with logger if you have one)
                print(f"[Warning] extrusion failed: {e}")
                voxels_pred = None

            # Add geometric outputs
            outputs.update({
                "polygons": polygons,
                "polygon_validity": validity,
                "voxels_pred": voxels_pred,
            })

            # NEW: Auxiliary outputs for novel training strategies (only when geometric is enabled)
            if return_aux:
                # Cross-modal consistency embeddings
                if self.use_latent_consistency:
                    if voxels_pred is not None:
                        voxel_features = self._create_3d_features_from_voxels(voxels_pred)
                        latent_2d, latent_3d = self.latent_head(main_features, voxel_features)
                    else:
                        # Fall back to pseudo-3D built from 2D features if voxels not available
                        latent_2d, latent_3d = self.latent_head(main_features, None)
                    outputs["latent_2d_embedding"] = latent_2d
                    outputs["latent_3d_embedding"] = latent_3d
        else:
            # Geometric path explicitly skipped for this stage
            outputs.update({
                "polygons": None,
                "polygon_validity": None,
                "voxels_pred": None,
            })

            # Still compute some auxiliary outputs that don't depend on geometry
            if return_aux and self.use_latent_consistency:
                # Use pseudo-3D features for 2D-only consistency inside latent head
                latent_2d, latent_3d = self.latent_head(main_features, None)
                outputs["latent_2d_embedding"] = latent_2d
                outputs["latent_3d_embedding"] = latent_3d

        # Graph structure predictions (independent of geometric computation)
        if return_aux and self.use_graph_constraints:
            graph_output = self.graph_head(main_features)
            outputs.update(graph_output)

        return outputs

    def get_latent_embeddings(self, image):
        """
        Convenience method to get just the latent embeddings
        Used by trainer for consistency loss
        """
        if not self.use_latent_consistency:
            return None, None

        with torch.no_grad():
            features = self.encoder(image)
            spatial_feat = self._select_spatial_feature(features)
            main_features = self.feature_enhancer(spatial_feat)

            # Quick forward to get segmentation/attributes
            segmentation = self.seg_head(features)
            attributes = self.attr_head(
                features.get("global")
                if isinstance(features, dict) and "global" in features
                else main_features.mean(dim=[2, 3])
            )

            # Attempt DVX + extrusion, but be defensive (may be expensive)
            dvx_output = self.dvx(features, segmentation)
            polygons = dvx_output.get("polygons", None)
            validity = dvx_output.get("validity", None)

            try:
```

```
                voxels_pred = self.extrusion(polygons, attributes, validity)
            except Exception as e:
                print(f"[Warning] get_latent_embeddings: extrusion failed: {e}")
                voxels_pred = None

            # If voxels not available, latent_head will fall back to pseudo-3D
            if voxels_pred is not None:
                voxel_features = self._create_3d_features_from_voxels(voxels_pred)
            else:
                voxel_features = None

        return self.latent_head(main_features, voxel_features)

    def _create_3d_features_from_voxels(self, voxels):
        """
        Create 3D feature representation from voxel predictions

        Args:
            voxels: [B, D, H, W] voxel predictions

        Returns:
            [B, C, D, H, W] 3D features
        """
        # Defensive check
        if voxels is None:
            raise ValueError(
                "Received voxels=None in _create_3d_features_from_voxels(). "
                "This indicates that the geometric pipeline was skipped or extrusion failed. "
                "Call this method only when voxels are available, or use latent_head(..., None) to "
                "compute pseudo-3D features from 2D."
            )

        # Ensure expected shape
        if voxels.dim() != 4:
            raise ValueError(f"voxels must have shape [B,D,H,W], got {tuple(voxels.shape)}")

        B, D, H, W = voxels.shape

        # Expand voxels to have feature channels
        # Simple approach: repeat voxel values across feature dimension
        rep_ch = max(1, self.feature_dim // 4)
        voxel_features = voxels.unsqueeze(1).expand(B, rep_ch, D, H, W).contiguous()

        # Add some learned 3D processing
        if self._voxel_processor is None:
            # Build with correct device
            device = voxels.device
            self._voxel_processor = nn.Sequential(
                nn.Conv3d(rep_ch, max(rep_ch, self.feature_dim // 2), 3, padding=1),
                nn.ReLU(),
                nn.Conv3d(max(rep_ch, self.feature_dim // 2), self.feature_dim, 3, padding=1),
            ).to(device)

        return self._voxel_processor(voxel_features)

    def get_stage_parameters(self, stage: int):
        """
        Get parameters for specific training stage
        Useful for stage-specific optimization
        """
        if stage == 1:
            # Stage 1: 2D components only
            params = []
            params.extend(list(self.encoder.parameters()))
            params.extend(list(self.seg_head.parameters()))
            params.extend(list(self.attr_head.parameters()))
            params.extend(list(self.sdf_head.parameters()))
            params.extend(list(self.feature_enhancer.parameters()))

            if self.use_latent_consistency:
                params.extend(list(self.latent_head.parameters()))

        elif stage == 2:
```

```
            # Stage 2: DVX components
            params = list(self.dvx.parameters())

        else:  # stage == 3
            # Stage 3: All parameters
            params = list(self.parameters())

        return params

    def freeze_stage_parameters(self, stages_to_freeze: list):
        """
        Freeze parameters for specific stages

        Args:
            stages_to_freeze: List of stage numbers to freeze
        """
        for stage in stages_to_freeze:
            stage_params = self.get_stage_parameters(stage)
            for param in stage_params:
                param.requires_grad = False

    def unfreeze_stage_parameters(self, stages_to_unfreeze: list):
        """
        Unfreeze parameters for specific stages

        Args:
            stages_to_unfreeze: List of stage numbers to unfreeze
        """
        for stage in stages_to_unfreeze:
            stage_params = self.get_stage_parameters(stage)
            for param in stage_params:
                param.requires_grad = True

    def get_curriculum_metrics(self):
        """
        Get metrics useful for curriculum learning decisions
        """
        metrics = {}

        # Parameter counts per stage
        for stage in [1, 2, 3]:
            stage_params = self.get_stage_parameters(stage)
            metrics[f"stage_{stage}_params"] = sum(p.numel() for p in stage_params)

        # Feature dimensions
        metrics["feature_dim"] = self.feature_dim
        metrics["has_latent_consistency"] = self.use_latent_consistency
        metrics["has_graph_constraints"] = self.use_graph_constraints

        return metrics
```

■ File: train.py

==

```
"""
Enhanced training script for the Neural-Geometric 3D Model Generator
Implements novel training strategies: dynamic curriculum, adaptive weighting, cross-modal
+         consistency
"""

import argparse
import torch
from torch.utils.data import DataLoader
from pathlib import Path
import json
import matplotlib.pyplot as plt
import numpy as np

from dataset import AdvancedFloorPlanDataset
from models.model import NeuralGeometric3DGenerator
from training.trainer import AdaptiveMultiStageTrainer, MultiStageTrainer
from utils.visualization import plot_training_history, plot_curriculum_analysis
```

```python
from config import (
    DEFAULT_DATA_CONFIG,
    DEFAULT_MODEL_CONFIG,
    DEFAULT_TRAINING_CONFIG,
    DEFAULT_LOSS_CONFIG,
    TrainingConfig,
    CurriculumConfig
)


def create_enhanced_config(args):
    """Create enhanced training configuration with novel strategies"""
    config = TrainingConfig()

    # Basic settings
    config.device = args.device or ("cuda" if torch.cuda.is_available() else "cpu")

    # Dynamic curriculum settings
    if args.dynamic_curriculum:
        config.curriculum = CurriculumConfig()
        config.curriculum.use_dynamic_curriculum = True
        config.curriculum.stage_switch_patience = args.patience
        config.curriculum.min_improvement_threshold = args.min_improvement

        # Adjust epoch limits for dynamic training
        config.max_stage1_epochs = args.max_stage1_epochs
        config.max_stage2_epochs = args.max_stage2_epochs
        config.max_stage3_epochs = args.max_stage3_epochs

        print("Dynamic curriculum learning enabled")
        print(f"  Stage switch patience: {config.curriculum.stage_switch_patience}")
        print(f"  Min improvement threshold: {config.curriculum.min_improvement_threshold}")
    else:
        # Disable dynamic curriculum for traditional training
        config.curriculum.use_dynamic_curriculum = False
        print("Using traditional fixed-epoch training")

    # GradNorm dynamic weighting
    if args.gradnorm:
        config.curriculum.use_gradnorm = True
        config.curriculum.gradnorm_alpha = args.gradnorm_alpha
        config.curriculum.gradnorm_update_freq = args.gradnorm_freq
        print(f"GradNorm dynamic weighting enabled (alpha={args.gradnorm_alpha})")

    # Topology-aware scheduling
    if args.topology_schedule != "static":
        config.curriculum.topology_schedule = args.topology_schedule
        config.curriculum.topology_start_weight = args.topology_start
        config.curriculum.topology_end_weight = args.topology_end
        print(f"Topology-aware scheduling: {args.topology_schedule}")
        print(f"  Weights: {args.topology_start} -> {args.topology_end}")

    return config


def create_enhanced_model(args):
    """Create enhanced model with auxiliary heads"""
    model = NeuralGeometric3DGenerator(
        input_channels=args.input_channels,
        num_classes=args.num_classes,
        feature_dim=args.feature_dim,
        num_attributes=args.num_attributes,
        voxel_size=args.voxel_size,
        max_polygons=args.max_polygons,
        max_points=args.max_points,
        use_latent_consistency=args.latent_consistency,
        use_graph_constraints=args.graph_constraints,
        latent_embedding_dim=args.embedding_dim
    )

    print(f"Enhanced model created:")
    print(f"  Feature dim: {args.feature_dim}")
    print(f"  Latent consistency: {args.latent_consistency}")
```

```python
    print(f"  Graph constraints: {args.graph_constraints}")

    # Print parameter counts
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f"  Total parameters: {total_params:,}")
    print(f"  Trainable parameters: {trainable_params:,}")

    return model


def visualize_training_results(history, output_dir):
    """Create comprehensive training visualizations"""
    output_dir = Path(output_dir)
    output_dir.mkdir(exist_ok=True)

    # Traditional loss curves
    plot_training_history(history, save_path=str(output_dir / "training_history.png"))

    # Novel curriculum analysis plots
    if "stage_transitions" in history and history["stage_transitions"]:
        plot_curriculum_analysis(history, save_path=str(output_dir / "curriculum_analysis.png"))

    # Dynamic weight evolution
    if "dynamic_weights" in history and history["dynamic_weights"]:
        plt.figure(figsize=(12, 8))

        # Extract weight evolution data
        epochs = [entry["epoch"] for entry in history["dynamic_weights"]]
        weight_names = list(history["dynamic_weights"][0]["weights"].keys())

        for weight_name in weight_names:
            weights = [entry["weights"].get(weight_name, 0) for entry in history["dynamic_weights"]]
            if any(w > 0.001 for w in weights):  # Only plot significant weights
                plt.plot(epochs, weights, label=weight_name, linewidth=2)

        plt.xlabel("Global Epoch")
        plt.ylabel("Loss Weight")
        plt.title("Dynamic Loss Weight Evolution")
        plt.legend()
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.savefig(output_dir / "weight_evolution.png", dpi=300)
        plt.close()

    # Component loss breakdown
    fig, axes = plt.subplots(1, 3, figsize=(18, 5))
    stage_names = ["stage1", "stage2", "stage3"]

    for idx, stage_name in enumerate(stage_names):
        if stage_name in history and "component_losses" in history[stage_name]:
            component_data = history[stage_name]["component_losses"]
            if component_data:
                # Get component names from first entry
                component_names = list(component_data[0].keys())

                for comp_name in component_names:
                    if comp_name in ['seg', 'dice', 'polygon', 'voxel', 'topology',
                                   'latent_consistency', 'graph']:
                        values = [entry.get(comp_name, 0) for entry in component_data]
                        if any(v > 0.001 for v in values):  # Only plot significant losses
                            axes[idx].plot(values, label=comp_name, linewidth=2)

                axes[idx].set_title(f"{stage_name.upper()} Component Losses")
                axes[idx].set_xlabel("Epoch")
                axes[idx].set_ylabel("Loss Value")
                axes[idx].legend()
                axes[idx].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig(output_dir / "component_losses.png", dpi=300)
    plt.close()

```

```python
    print(f"Training visualizations saved to {output_dir}")


def save_training_summary(history, config, output_dir):
    """Save comprehensive training summary"""
    output_dir = Path(output_dir)

    summary = {
        "training_config": {
            "dynamic_curriculum": config.curriculum.use_dynamic_curriculum,
            "gradnorm_enabled": config.curriculum.use_gradnorm,
            "topology_schedule": config.curriculum.topology_schedule,
            "max_epochs": [config.max_stage1_epochs, config.max_stage2_epochs, config.max_stage3_epochs]
        },
        "training_results": {},
        "novel_strategies_summary": {}
    }

    # Training results
    for stage_name, data in history.items():
        if isinstance(data, dict) and "val_loss" in data and data["val_loss"]:
            summary["training_results"][stage_name] = {
                "final_val_loss": data["val_loss"][-1],
                "best_val_loss": min(data["val_loss"]),
                "epochs_trained": len(data["val_loss"])
            }

    # Novel strategies summary
    if "stage_transitions" in history:
        summary["novel_strategies_summary"]["adaptive_transitions"] = len(history["stage_transitions"])

    if "dynamic_weights" in history:
        summary["novel_strategies_summary"]["weight_updates"] = len(history["dynamic_weights"])

    if "curriculum_events" in history:
        summary["novel_strategies_summary"]["curriculum_events"] = len(history["curriculum_events"])

    # Save as JSON
    with open(output_dir / "training_summary.json", 'w') as f:
        json.dump(summary, f, indent=2)

    print(f"Training summary saved to {output_dir / 'training_summary.json'}")


def main():
    parser = argparse.ArgumentParser(description="Enhanced Neural-Geometric 3D Model Generator Training")

    # Basic arguments
    parser.add_argument("--data_dir", type=str, default="./data/floorplans",
                        help="Path to dataset directory")
    parser.add_argument("--batch_size", type=int, default=2, help="Batch size")
    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")
    parser.add_argument("--resume", type=str, default=None, help="Resume from checkpoint")
    parser.add_argument("--output_dir", type=str, default="./checkpoints",
                        help="Output directory for checkpoints")

    # Training mode selection
    parser.add_argument("--training_mode", type=str, choices=["traditional", "adaptive"],
                        default="adaptive", help="Training mode (traditional fixed epochs vs adaptive)")
    parser.add_argument("--stage", type=str, choices=["1", "2", "3", "all"], default="all",
                        help="Training stage to run (only for traditional mode)")

    # Novel training strategies
    parser.add_argument("--dynamic-curriculum", action="store_true", default=True,
                        help="Enable adaptive stage transitioning")
    parser.add_argument("--patience", type=int, default=5,
                        help="Epochs without improvement before stage transition")
    parser.add_argument("--min-improvement", type=float, default=0.001,
```

```
                        help="Minimum relative improvement threshold")

    parser.add_argument("--gradnorm", action="store_true", default=True,
                        help="Enable GradNorm dynamic loss weighting")
    parser.add_argument("--gradnorm-alpha", type=float, default=0.12,
                        help="GradNorm restoring force parameter")
    parser.add_argument("--gradnorm-freq", type=int, default=5,
                        help="GradNorm update frequency (batches)")

    parser.add_argument("--topology-schedule", type=str,
                        choices=["static", "progressive", "linear_ramp"],
                        default="progressive", help="Topology loss scheduling strategy")
    parser.add_argument("--topology-start", type=float, default=0.1,
                        help="Starting weight for topology loss")
    parser.add_argument("--topology-end", type=float, default=1.0,
                        help="Ending weight for topology loss")

    # Model enhancements
    parser.add_argument("--latent-consistency", action="store_true", default=True,
                        help="Enable cross-modal latent consistency loss")
    parser.add_argument("--graph-constraints", action="store_true", default=True,
                        help="Enable graph-based topology constraints")
    parser.add_argument("--embedding-dim", type=int, default=256,
                        help="Latent embedding dimension")

    # Model architecture
    parser.add_argument("--input_channels", type=int, default=3, help="Input image channels")
    parser.add_argument("--num_classes", type=int, default=5, help="Number of segmentation classes")
    parser.add_argument("--feature_dim", type=int, default=768, help="Feature dimension")
    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")
    parser.add_argument("--max_polygons", type=int, default=30, help="Maximum number of polygons")
    parser.add_argument("--max_points", type=int, default=100, help="Maximum points per polygon")

    # Dynamic epoch limits
    parser.add_argument("--max-stage1-epochs", type=int, default=50, help="Max epochs for Stage 1")
    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")

    parser.add_argument("--persistent_workers",action="store_true",default=False,help="Keep DataLoader workers alive between epochs (requires num_workers > 0).")

    parser.add_argument("--prefetch_factor",type=int,default=2,help="Number of batches preloaded by each worker.")


    args = parser.parse_args()

    # Setup device
    device = args.device or ("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    import torch.backends.cudnn as cudnn
    if device == "cuda":
        cudnn.benchmark = True

    # Create output directory
    output_dir = Path(args.output_dir)
    output_dir.mkdir(exist_ok=True)

    # Create enhanced configuration
    config = create_enhanced_config(args)

    print("\n" + "="*80)
    print("NEURAL-GEOMETRIC 3D MODEL GENERATOR - ENHANCED TRAINING")
    print("="*80)
    print("Novel Training Strategies Enabled:")
    if config.curriculum.use_dynamic_curriculum:
        print("? Adaptive Stage Transitioning (Dynamic Curriculum)")
    if config.curriculum.use_gradnorm:
        print("? Multi-objective Optimization with GradNorm")
    if config.curriculum.topology_schedule != "static":
```

```
        print("? Topology-aware Loss Scheduling")
    if args.latent_consistency:
        print("? Cross-modal Latent Consistency Learning")
    if args.graph_constraints:
        print("? Graph-based Topology Constraints")
    print("="*80)

    # Create datasets
    print("\nLoading datasets...")
    train_dataset = AdvancedFloorPlanDataset(
        args.data_dir, split="train", augment=True
    )
    val_dataset = AdvancedFloorPlanDataset(
        args.data_dir, split="val", augment=False
    )

    print(f"Train samples: {len(train_dataset)}")
    print(f"Validation samples: {len(val_dataset)}")

    if len(train_dataset) == 0:
        print("Error: No training samples found!")
        return

    # Create data loaders
    train_loader = DataLoader(
        train_dataset,
        batch_size=args.batch_size,
        shuffle=True,
        num_workers=args.num_workers,
        pin_memory=True,
        drop_last=True,
        persistent_workers=args.persistent_workers if args.num_workers > 0 else False,
        prefetch_factor=args.prefetch_factor if args.num_workers > 0 else None
    )

    val_loader = DataLoader(
        val_dataset,
        batch_size=max(1, args.batch_size),
        shuffle=False,
        num_workers=max(1, args.num_workers // 2),
        pin_memory=True,
        drop_last=False,
        persistent_workers=args.persistent_workers if args.num_workers > 0 else False,
        prefetch_factor=args.prefetch_factor if args.num_workers > 0 else None
    )

    # Create enhanced model
    print("\nInitializing enhanced model...")
    model = create_enhanced_model(args)

    # Create appropriate trainer
    if args.training_mode == "adaptive":
        print("\nUsing Adaptive Multi-Stage Trainer with Novel Strategies")
        trainer = AdaptiveMultiStageTrainer(
            model=model,
            train_loader=train_loader,
            val_loader=val_loader,
            device=device,
            config=config
        )
    else:
        print("\nUsing Traditional Multi-Stage Trainer")
        trainer = MultiStageTrainer(
            model=model,
            train_loader=train_loader,
            val_loader=val_loader,
            device=device,
            config=config
        )

    # Resume from checkpoint if specified
    if args.resume:
        print(f"Resuming from checkpoint: {args.resume}")
```

```
        trainer.load_checkpoint(args.resume)

    # Run training
    if args.training_mode == "adaptive" or args.stage == "all":
        print("\nStarting adaptive multi-stage training with novel strategies...")
        history = trainer.train_all_stages()
    else:
        # Traditional single-stage training
        stage_num = int(args.stage)
        print(f"Training Stage {stage_num} only...")
        if stage_num == 1:
            trainer.train_stage1()
        elif stage_num == 2:
            trainer.train_stage2()
        elif stage_num == 3:
            trainer.train_stage3()
        history = trainer.history

    # Save final model
    final_model_path = output_dir / "final_enhanced_model.pth"
    if hasattr(trainer, '_save_checkpoint'):
        trainer._save_checkpoint(str(final_model_path))
    print(f"Final model saved to: {final_model_path}")

    # Create comprehensive visualizations
    print("\nGenerating training analysis...")
    visualize_training_results(history, output_dir)

    # Save training summary
    save_training_summary(history, config, output_dir)

    print(f"\n? Enhanced training completed successfully!")
    print(f"? Results saved to: {output_dir}")
    print("\nNovel contributions implemented:")
    print("- Dynamic curriculum learning with adaptive stage transitions")
    print("- Multi-objective optimization with gradient-based reweighting")
    print("- Topology-aware progressive constraint injection")
    print("- Cross-modal latent consistency learning")
    print("- Graph-based architectural constraint learning")


if __name__ == "__main__":
    main()
```

■ File: training\losses.py

==

```
"""
Advanced loss functions for multi-task training with dynamic weighting
Enhanced with cross-modal consistency, graph constraints, and GradNorm
Modified to support conditional geometric losses via run_full_geometric flag
FIXED: Dynamic loss component initialization for stage transitions
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
import cv2
import numpy as np
from typing import Dict, Optional, Tuple, List
import networkx as nx


class DynamicLossWeighter:
    def __init__(self, loss_names: List[str], alpha: float = 0.12, device: str = 'cuda'):
        self.loss_names = loss_names
        self.alpha = alpha
        self.device = device

        # Initialize weights for all known loss components
        self.weights = {name: 1.0 for name in loss_names}
        self.initial_task_losses = {}
```

```
        # Add running normalization to prevent raw magnitude issues
        self.running_mean_losses = {name: 0.0 for name in loss_names}
        self.running_std_losses = {name: 1.0 for name in loss_names}  # NEW
        self.update_count = 0

        print(f"[DynamicWeighter] Initialized with loss components: {loss_names}")

    def update_weights(self, task_losses: Dict[str, torch.Tensor],
                       shared_parameters, update_freq: int = 10):
        """Update loss weights using GradNorm with stability improvements"""
        if self.update_count % update_freq != 0:
            self.update_count += 1
            return self.weights

        # Initialize tracking for new loss components
        newly_initialized = []
        for name, loss in task_losses.items():
            if name not in self.initial_task_losses:
                loss_val = loss.item() if torch.is_tensor(loss) else float(loss)
                # Use log-scale initialization for stability
                self.initial_task_losses[name] = max(np.log(abs(loss_val) + 1e-6), -10.0)

                if name not in self.weights:
                    self.weights[name] = 1.0
                if name not in self.running_mean_losses:
                    self.running_mean_losses[name] = loss_val
                if name not in self.running_std_losses:
                    self.running_std_losses[name] = max(abs(loss_val), 1e-3)

                newly_initialized.append(name)

        # Update running statistics with EMA
        for name, loss in task_losses.items():
            loss_val = loss.item() if torch.is_tensor(loss) else float(loss)
            if name in self.running_mean_losses:
                # Exponential moving average for mean and std
                self.running_mean_losses[name] = 0.9 * self.running_mean_losses[name] + 0.1 *
+                        loss_val

                # Update running std using Welford's algorithm
                delta = loss_val - self.running_mean_losses[name]
                self.running_std_losses[name] = 0.9 * self.running_std_losses[name] + 0.1 *
+                        abs(delta)
                self.running_std_losses[name] = max(self.running_std_losses[name], 1e-3)

        # Calculate normalized relative decrease rates
        loss_ratios = {}
        for name, loss in task_losses.items():
            if name in self.initial_task_losses and self.initial_task_losses[name] > -9.0:
                # Normalize current loss by running statistics
                current_loss = self.running_mean_losses.get(name, loss.item())
                normalized_current = current_loss / (self.running_std_losses[name] + 1e-6)

                initial_loss = self.initial_task_losses[name]
                # Use log-space ratios for stability
                loss_ratios[name] = np.exp(min(max(normalized_current - initial_loss, -5.0), 5.0))

        if not loss_ratios:
            self.update_count += 1
            return self.weights

        # Calculate gradient norms with improved stability
        grad_norms = {}
        for name, loss in task_losses.items():
            if name in loss_ratios:
                if not torch.is_tensor(loss) or not loss.requires_grad:
                    continue
                if not torch.isfinite(loss):
                    continue

                try:
                    grads = torch.autograd.grad(
                        loss, shared_parameters,
```

```
                        retain_graph=True, create_graph=False, allow_unused=True
                    )

                    grad_norm_sq = 0.0
                    valid_grads = False
                    for grad in grads:
                        if grad is not None and torch.isfinite(grad).all():
                            # Apply gradient norm stabilization
                            clipped_grad = torch.clamp(grad, -10.0, 10.0)
                            grad_norm_sq += clipped_grad.norm().item() ** 2
                            valid_grads = True

                    if valid_grads and grad_norm_sq > 0:
                        # Use log-scale gradient norms
                        grad_norms[name] = np.log(grad_norm_sq ** 0.5 + 1e-8)

                except Exception as e:
                    continue

        if not grad_norms:
            self.update_count += 1
            return self.weights

        # Normalize gradient norms
        mean_grad_norm = np.mean(list(grad_norms.values()))

        # Update weights with improved stability
        for name in grad_norms.keys():
            if name in loss_ratios:
                # Calculate target gradient in log space
                target_grad_log = mean_grad_norm + self.alpha * np.log(loss_ratios[name] + 1e-8)
                current_grad_log = grad_norms[name]

                # Calculate weight update with damping
                weight_update_log = target_grad_log - current_grad_log
                weight_update = np.exp(np.clip(weight_update_log, -1.0, 1.0))  # Stronger clipping

                # Apply update with momentum and stronger constraints
                current_weight = self.weights.get(name, 1.0)
                new_weight = 0.8 * current_weight + 0.2 * weight_update  # More conservative
                self.weights[name] = float(np.clip(new_weight, 0.1, 2.0))  # Tighter bounds

        self.update_count += 1
        return self.weights

class GraphTopologyExtractor:
    """Extracts graph structure from segmentation for topology constraints"""

    @staticmethod
    def extract_room_graph(segmentation: torch.Tensor) -> Dict[str, torch.Tensor]:
        """Extract room connectivity graph from segmentation mask"""
        B, C, H, W = segmentation.shape
        device = segmentation.device

        # Get room predictions (assume classes: 0=bg, 1=wall, 2=door, 3=window, 4=room)
        room_probs = F.softmax(segmentation, dim=1)
        room_mask = room_probs[:, 4] if C > 4 else torch.zeros((B, H, W), device=device)
        wall_mask = room_probs[:, 1] if C > 1 else torch.zeros((B, H, W), device=device)

        # Simple connectivity: rooms connected if they share wall boundary
        adjacency_matrices = []
        room_features = []

        for b in range(B):
            room_b = room_mask[b].detach().cpu().numpy()
            wall_b = wall_mask[b].detach().cpu().numpy()

            # Find connected components (rooms)
            try:
                from scipy import ndimage
                labeled_rooms, num_rooms = ndimage.label(room_b > 0.5)

                # Create adjacency matrix
```

```
                adj_matrix = np.zeros((max(num_rooms, 1), max(num_rooms, 1)))
                room_centroids = []

                for i in range(1, num_rooms + 1):
                    room_i_mask = (labeled_rooms == i)
                    if np.sum(room_i_mask) > 0:
                        centroid = ndimage.center_of_mass(room_i_mask)
                        room_centroids.append(centroid)

                        # Check connectivity to other rooms through walls
                        for j in range(i + 1, num_rooms + 1):
                            room_j_mask = (labeled_rooms == j)
                            if np.sum(room_j_mask) > 0:
                                # Check if rooms are connected via wall adjacency
                                connectivity = GraphTopologyExtractor._check_room_connectivity(
                                    room_i_mask, room_j_mask, wall_b
                                )
                                adj_matrix[i-1, j-1] = connectivity
                                adj_matrix[j-1, i-1] = connectivity

                # Convert to tensor
                adj_tensor = torch.from_numpy(adj_matrix).float().to(device)
                centroids_tensor = torch.from_numpy(np.array(room_centroids) if room_centroids else
                         np.zeros((0, 2))).float().to(device)

            except ImportError:
                # Fallback if scipy not available
                adj_tensor = torch.zeros((1, 1), device=device)
                centroids_tensor = torch.zeros((0, 2), device=device)
            except Exception as e:
                # General fallback for any other issues
                print(f"Warning: Graph extraction failed: {e}")
                adj_tensor = torch.zeros((1, 1), device=device)
                centroids_tensor = torch.zeros((0, 2), device=device)

            adjacency_matrices.append(adj_tensor)
            room_features.append(centroids_tensor)

        return {
            "adjacency_matrices": adjacency_matrices,
            "room_features": room_features
        }

    @staticmethod
    def _check_room_connectivity(room1_mask, room2_mask, wall_mask):
        """Check if two rooms are connected through walls"""
        try:
            from scipy.ndimage import binary_dilation

            # Dilate room masks to check wall adjacency
            dilated1 = binary_dilation(room1_mask, iterations=2)
            dilated2 = binary_dilation(room2_mask, iterations=2)

            # Check overlap through wall areas
            wall_overlap = (dilated1 & dilated2) & (wall_mask > 0.3)
            return float(np.sum(wall_overlap) > 0)
        except ImportError:
            # Simple distance-based fallback
            return 0.0


class ResearchGradeLoss(nn.Module):
    """
    Multi-task loss with stage-aware dynamic weighting:
    - Stage 1: segmentation, dice, sdf, attributes, topology, graph
    - Stage 2: + polygon (DVX)
    - Stage 3: + voxel, latent_consistency (full geometric)

    FIXED: Dynamic initialization handles new loss components during stage transitions
    """

    def __init__(
        self,
```

```
        seg_weight: float = 1.0,
        dice_weight: float = 1.0,
        sdf_weight: float = 0.5,
        attr_weight: float = 1.0,
        polygon_weight: float = 1.0,
        voxel_weight: float = 1.0,
        topology_weight: float = 0.5,
        latent_consistency_weight: float = 0.5,
        graph_constraint_weight: float = 0.3,
        enable_dynamic_weighting: bool = True,
        gradnorm_alpha: float = 0.12,
        device: str = 'cuda',
        weight_update_freq: int = 10,
        weight_momentum: float = 0.9,
    ):
        super().__init__()

        # Store initial weights for all possible loss components
        self.initial_weights = {
            'seg': float(seg_weight),
            'dice': float(dice_weight),
            'sdf': float(sdf_weight),
            'attr': float(attr_weight),
            'polygon': float(polygon_weight),
            'voxel': float(voxel_weight),
            'topology': float(topology_weight),
            'latent_consistency': float(latent_consistency_weight),
            'graph': float(graph_constraint_weight)
        }

        # Current weights (will be dynamically updated)
        self.weights = self.initial_weights.copy()

        # Core losses
        self.ce_loss = nn.CrossEntropyLoss()
        self.mse_loss = nn.MSELoss()
        self.l1_loss = nn.L1Loss()
        self.cosine_loss = nn.CosineEmbeddingLoss()

        # Dynamic weighting with all possible loss names
        self.enable_dynamic_weighting = enable_dynamic_weighting
        if enable_dynamic_weighting:
            all_loss_names = list(self.initial_weights.keys())
            self.loss_weighter = DynamicLossWeighter(
                all_loss_names, alpha=gradnorm_alpha, device=device,
            )
            self.update_freq = weight_update_freq
            self.momentum = weight_momentum
            print(f"[ResearchGradeLoss] Dynamic weighting enabled for: {all_loss_names}")

        self.device = device

    def update_loss_weights(self, new_weights: Dict[str, float]):
        """Update loss weights (called by trainer for curriculum scheduling)"""
        for key, value in new_weights.items():
            if key in self.weights:
                self.weights[key] = float(value)

    def forward(self, predictions: dict, targets: dict, shared_parameters=None,
+             run_full_geometric=True):
        """Compute multi-task loss with proper normalization and aggregation"""
        # Input validation and sanitization
        predictions = self._sanitize_predictions(predictions)
        targets = self._sanitize_targets(targets)

        device = self._get_device_from_inputs(predictions, targets)
        losses = {}
        total_loss = torch.tensor(0.0, device=device, requires_grad=True)

        # ---- STAGE 1 LOSSES with proper scaling ----
        if "segmentation" in predictions and "mask" in targets:
            seg_pred = predictions["segmentation"]
            seg_target = targets["mask"].long()
```

```

            # Scale CE loss by number of pixels for consistency
            ce_loss = self.ce_loss(seg_pred, seg_target)
            losses["seg"] = ce_loss

            dice_loss = self._dice_loss(seg_pred, seg_target)
            losses["dice"] = dice_loss

        if "sdf" in predictions and "mask" in targets:
            sdf_pred = predictions["sdf"]
            sdf_pred = torch.clamp(sdf_pred, -1.0, 1.0)
            sdf_target = self._mask_to_sdf(targets["mask"])
            sdf_target = sdf_target.to(sdf_pred.device).type_as(sdf_pred)
            # Normalize SDF loss by spatial dimensions
            sdf_loss = self.mse_loss(sdf_pred, sdf_target)
            losses["sdf"] = sdf_loss

        if "attributes" in predictions and "attributes" in targets:
            pred_attr = predictions["attributes"].float()
            tgt_attr = targets["attributes"].float().to(pred_attr.device)
            # Normalize attribute loss by number of attributes
            attr_loss = self.l1_loss(pred_attr, tgt_attr) / pred_attr.shape[-1]
            losses["attr"] = attr_loss

        # Apply proper scaling to topology losses
        if "segmentation" in predictions:
            topology_loss = self._topology_loss(predictions["segmentation"])
            # Scale topology loss to reasonable magnitude
            losses["topology"] = topology_loss * 0.5

            graph_loss = self._graph_topology_loss(predictions["segmentation"])
            # Graph loss is already scaled in the function above
            losses["graph"] = graph_loss

        # ---- GEOMETRIC LOSSES with normalization ----
        if run_full_geometric:
            if ("polygons" in predictions and predictions["polygons"] is not None and
                "polygons_gt" in targets):
                poly_loss = self._polygon_loss(predictions, targets["polygons_gt"])
                # Normalize polygon loss by number of polygons and points
                if "polygons" in predictions and predictions["polygons"] is not None:
                    B, P, N, _ = predictions["polygons"].shape
                    poly_loss = poly_loss / (P * N)  # Normalize by polygon complexity
                losses["polygon"] = poly_loss
            else:
                losses["polygon"] = torch.tensor(0.0, device=device)

            if ("voxels_pred" in predictions and predictions["voxels_pred"] is not None and
                "voxels_gt" in targets):
                pred_vox = predictions["voxels_pred"].float()
                tgt_vox = targets["voxels_gt"].float().to(pred_vox.device)
                voxel_loss = self._voxel_iou_loss(pred_vox, tgt_vox)
                losses["voxel"] = voxel_loss
            else:
                losses["voxel"] = torch.tensor(0.0, device=device)

            if ("latent_2d_embedding" in predictions and "latent_3d_embedding" in predictions and
                predictions["latent_2d_embedding"] is not None and
+                        predictions["latent_3d_embedding"] is not None):
                consistency_loss = self._latent_consistency_loss(
                    predictions["latent_2d_embedding"],
                    predictions["latent_3d_embedding"]
                )
                losses["latent_consistency"] = consistency_loss
            else:
                losses["latent_consistency"] = torch.tensor(0.0, device=device)
        else:
            losses["polygon"] = torch.tensor(0.0, device=device)
            losses["voxel"] = torch.tensor(0.0, device=device)
            losses["latent_consistency"] = torch.tensor(0.0, device=device)

        # ---- IMPROVED WEIGHTING AND AGGREGATION ----
        active_losses = {
```

```
            name: loss for name, loss in losses.items()
            if isinstance(loss, torch.Tensor) and loss.requires_grad and loss.item() > 1e-8
        }

        if self.enable_dynamic_weighting and shared_parameters is not None and active_losses:
            try:
                dynamic_weights = self.loss_weighter.update_weights(
                    active_losses, shared_parameters, self.update_freq
                )

                # Apply weights with additional stability checks
                for name, loss in losses.items():
                    if name in self.weights and isinstance(loss, torch.Tensor) and torch.isfinite(loss):
                        weight = dynamic_weights.get(name, self.weights[name])
                        # Apply weight with gradient scaling for stability
                        weighted_loss = weight * loss
                        if torch.isfinite(weighted_loss):
                            total_loss = total_loss + weighted_loss

            except Exception as e:
                print(f"[ResearchGradeLoss] Dynamic weighting failed: {e}, falling back to static weights")
                # Fallback to static weights
                for name, loss in losses.items():
                    if name in self.weights and isinstance(loss, torch.Tensor) and torch.isfinite(loss):
                        total_loss = total_loss + self.weights[name] * loss
        else:
            # Static weights with stability
            for name, loss in losses.items():
                if name in self.weights and isinstance(loss, torch.Tensor) and torch.isfinite(loss):
                    total_loss = total_loss + self.weights[name] * loss

        # Final loss scaling and validation
        total_loss = torch.clamp(total_loss, 0.0, 100.0)  # Prevent explosion

        if not torch.isfinite(total_loss):
            print("[ResearchGradeLoss] Warning: Non-finite total loss detected, using fallback")
            total_loss = torch.tensor(1.0, device=device, requires_grad=True)

        losses["total"] = total_loss
        return total_loss, losses

    def __call__(self, predictions: dict, targets: dict, shared_parameters=None, run_full_geometric=True):
        """Trainer compatibility method"""
        return self.forward(predictions, targets, shared_parameters, run_full_geometric)

    def _sanitize_predictions(self, predictions: dict) -> dict:
        """Sanitize prediction tensors"""
        sanitized = {}
        for name, tensor in predictions.items():
            if torch.is_tensor(tensor):
                if torch.isnan(tensor).any() or torch.isinf(tensor).any():
                    print(f"WARNING: NaN/Inf in predictions[{name}] - zeroing out")
                    sanitized[name] = torch.zeros_like(tensor)
                else:
                    sanitized[name] = tensor
            else:
                sanitized[name] = tensor
        return sanitized

    def _sanitize_targets(self, targets: dict) -> dict:
        """Sanitize target tensors"""
        sanitized = {}
        for name, tensor in targets.items():
            if torch.is_tensor(tensor):
                if torch.isnan(tensor).any() or torch.isinf(tensor).any():
                    print(f"WARNING: NaN/Inf in targets[{name}] - zeroing out")
                    sanitized[name] = torch.zeros_like(tensor)
                else:
                    sanitized[name] = tensor
```

```
            else:
                sanitized[name] = tensor
        return sanitized

    def _get_device_from_inputs(self, predictions, targets):
        """Helper to determine device from inputs"""
        for pred_dict in [predictions, targets]:
            for value in pred_dict.values():
                if torch.is_tensor(value):
                    return value.device
        return self.device

    # ---- LOSS COMPONENT IMPLEMENTATIONS ----

    def _latent_consistency_loss(self, embedding_2d: torch.Tensor, embedding_3d: torch.Tensor) -> torch.Tensor:
        """Cross-modal latent consistency loss"""
        if embedding_2d.shape != embedding_3d.shape:
            min_dim = min(embedding_2d.shape[-1], embedding_3d.shape[-1])
            embedding_2d = embedding_2d[..., :min_dim]
            embedding_3d = embedding_3d[..., :min_dim]

        target = torch.ones(embedding_2d.shape[0], device=embedding_2d.device)
        cosine_loss = self.cosine_loss(embedding_2d, embedding_3d, target)
        l2_loss = F.mse_loss(embedding_2d, embedding_3d)

        return 0.7 * cosine_loss + 0.3 * l2_loss

    def _graph_topology_loss(self, segmentation_logits: torch.Tensor) -> torch.Tensor:
        """Graph-based topology constraints with proper normalization"""
        try:
            graph_data = GraphTopologyExtractor.extract_room_graph(segmentation_logits)
            device = segmentation_logits.device

            total_graph_loss = torch.tensor(0.0, device=device)
            batch_size = segmentation_logits.shape[0]
            valid_batches = 0

            for b in range(batch_size):
                if b < len(graph_data["adjacency_matrices"]):
                    adj_matrix = graph_data["adjacency_matrices"][b]
                    if adj_matrix.numel() == 0:
                        continue

                    # Normalize by matrix size to prevent scale explosion
                    matrix_size = max(adj_matrix.shape[0], 1)
                    norm_factor = 1.0 / (matrix_size + 1e-6)

                    degrees = adj_matrix.sum(dim=1)
                    isolation_penalty = torch.exp(-degrees).mean() * norm_factor

                    max_reasonable_connections = min(4, adj_matrix.shape[0] - 1)
                    over_connection_penalty = F.relu(degrees - max_reasonable_connections).mean() * norm_factor

                    if b < len(graph_data["room_features"]) and graph_data["room_features"][b].numel() > 0:
                        room_features = graph_data["room_features"][b]
                        if room_features.shape[0] > 1:
                            feature_distances = torch.cdist(room_features, room_features)
                            # Normalize distance computation
                            mean_distance = feature_distances.mean()
                            normalized_distances = feature_distances / (mean_distance + 1e-6)
                            smoothness_loss = (adj_matrix * normalized_distances).sum() / (adj_matrix.sum() + 1e-6)
                            smoothness_loss = smoothness_loss * norm_factor
                        else:
                            smoothness_loss = torch.tensor(0.0, device=device)
                    else:
                        smoothness_loss = torch.tensor(0.0, device=device)

                    # Apply strong penalty scaling to keep graph loss in reasonable range
                    batch_graph_loss = (0.4 * isolation_penalty +
```

```
                                     0.3 * over_connection_penalty +
                                     0.3 * smoothness_loss) * 0.1  # Scale down by 10x

                    total_graph_loss = total_graph_loss + batch_graph_loss
                    valid_batches += 1

            # Average over valid batches and apply final normalization
            if valid_batches > 0:
                return total_graph_loss / valid_batches
            else:
                return torch.tensor(0.0, device=segmentation_logits.device)

        except Exception as e:
            return torch.tensor(0.0, device=segmentation_logits.device)

    def _dice_loss(self, pred: torch.Tensor, target: torch.Tensor, smooth: float = 1e-6) -> torch.Tensor:
        """Dice loss implementation"""
        pred_soft = F.softmax(pred, dim=1)
        B, C = pred_soft.shape[:2]

        dice_losses = []
        for c in range(C):
            pred_c = pred_soft[:, c, :, :]
            target_c = (target == c).float().to(pred_c.device)
            intersection = (pred_c * target_c).view(B, -1).sum(dim=1)
            union = pred_c.view(B, -1).sum(dim=1) + target_c.view(B, -1).sum(dim=1)
            dice = (2.0 * intersection + smooth) / (union + smooth)
            dice_losses.append((1.0 - dice).mean())

        return torch.stack(dice_losses).mean()

    def _mask_to_sdf(self, mask: torch.Tensor) -> torch.Tensor:
        """Convert mask to SDF"""
        device = mask.device if torch.is_tensor(mask) else self.device
        if not torch.is_tensor(mask):
            mask = torch.tensor(mask, device=device)

        B, H, W = mask.shape
        sdf = torch.zeros((B, 1, H, W), dtype=torch.float32, device=device)

        for b in range(B):
            mask_np = mask[b].detach().cpu().numpy().astype(np.uint8)
            try:
                dist_inside = cv2.distanceTransform((mask_np > 0).astype(np.uint8), cv2.DIST_L2, 5)
                dist_outside = cv2.distanceTransform((mask_np == 0).astype(np.uint8), cv2.DIST_L2, 5)
                sdf_np = dist_inside.astype(np.float32) - dist_outside.astype(np.float32)
                sdf_np = np.tanh(sdf_np / 10.0).astype(np.float32)
                sdf[b, 0] = torch.from_numpy(sdf_np)
            except Exception:
                sdf[b, 0] = torch.zeros_like(mask[b].float())

        return sdf

    def _polygon_loss(self, predictions: dict, targets: dict) -> torch.Tensor:
        """Polygon/DVX loss"""
        pred_polys = predictions.get("polygons")
        tgt_polys = targets.get("polygons")
        valid_mask = targets.get("valid_mask")

        if pred_polys is None or tgt_polys is None:
            return torch.tensor(0.0, device=pred_polys.device if pred_polys is not None else self.device)

        pred_polys = pred_polys.float()
        tgt_polys = tgt_polys.float().to(pred_polys.device)

        point_loss = self.mse_loss(pred_polys, tgt_polys)

        pred_valid = predictions.get("polygon_validity")
        if pred_valid is None or valid_mask is None:
            validity_loss = torch.tensor(0.0, device=pred_polys.device)
```

```
        else:
            pred_valid = pred_valid.float().to(pred_polys.device)
            valid_mask_f = valid_mask.float().to(pred_polys.device)
            validity_loss = self.mse_loss(pred_valid, valid_mask_f)

        smoothness_loss = self._polygon_smoothness(pred_polys)
        rect_loss = self._rectilinearity_loss(pred_polys)

        return point_loss + 0.1 * validity_loss + 0.05 * smoothness_loss + 0.1 * rect_loss

    def _polygon_smoothness(self, polygons: torch.Tensor) -> torch.Tensor:
        """Polygon smoothness loss"""
        if polygons is None or polygons.numel() == 0:
            return torch.tensor(0.0, device=polygons.device if polygons is not None else self.device)

        p1 = polygons
        p2 = torch.roll(polygons, -1, dims=2)
        p3 = torch.roll(polygons, -2, dims=2)
        curvature = torch.norm(p1 - 2.0 * p2 + p3, dim=-1)
        return curvature.mean()

    def _rectilinearity_loss(self, polygons: torch.Tensor) -> torch.Tensor:
        """Encourage axis-aligned structure"""
        if polygons is None or polygons.numel() == 0:
            return torch.tensor(0.0, device=polygons.device if polygons is not None else self.device)

        edges = torch.roll(polygons, -1, dims=2) - polygons
        edge_norms = torch.norm(edges, dim=-1, keepdim=True)
        edges_normalized = edges / (edge_norms + 1e-6)

        edge1 = edges_normalized
        edge2 = torch.roll(edges_normalized, -1, dims=2)

        cos_angles = (edge1 * edge2).sum(dim=-1)
        cos2 = cos_angles ** 2
        perp_penalty = cos2
        parallel_penalty = (cos2 - 1.0) ** 2
        angle_penalty = torch.minimum(perp_penalty, parallel_penalty)
        return angle_penalty.mean()

    def _voxel_iou_loss(self, pred_voxels: torch.Tensor, target_voxels: torch.Tensor) -> torch.Tensor:
        """3D voxel IoU loss"""
        pred_prob = torch.sigmoid(torch.clamp(pred_voxels, -10.0, 10.0))
        target = target_voxels.float().to(pred_prob.device)

        intersection = (pred_prob * target).view(pred_prob.shape[0], -1).sum(dim=1)
        union = (pred_prob.view(pred_prob.shape[0], -1).sum(dim=1) +
                target.view(target.shape[0], -1).sum(dim=1) - intersection)

        iou = (intersection + 1e-6) / (union + 1e-6)
        return (1.0 - iou).mean()

    def _topology_loss(self, segmentation_logits: torch.Tensor) -> torch.Tensor:
        """Traditional topology loss"""
        seg_soft = F.softmax(segmentation_logits, dim=1)
        C = seg_soft.shape[1]
        device = seg_soft.device

        walls = seg_soft[:, 1] if C > 1 else torch.zeros_like(seg_soft[:, 0])
        doors = seg_soft[:, 2] if C > 2 else torch.zeros_like(walls)
        windows = seg_soft[:, 3] if C > 3 else torch.zeros_like(walls)

        door_wall_overlap = doors * walls
        window_wall_overlap = windows * walls

        door_penalty = torch.maximum(doors - door_wall_overlap, torch.zeros_like(doors))
        window_penalty = torch.maximum(windows - window_wall_overlap, torch.zeros_like(windows))

        connectivity_loss = self._connectivity_loss(walls)

```

```
        return door_penalty.mean() + window_penalty.mean() + 0.1 * connectivity_loss

    def _connectivity_loss(self, wall_prob: torch.Tensor) -> torch.Tensor:
        """Connectivity loss for walls"""
        if wall_prob is None or wall_prob.numel() == 0:
            return torch.tensor(0.0, device=wall_prob.device if wall_prob is not None else self.device)

        kernel = torch.ones((1, 1, 3, 3), device=wall_prob.device, dtype=wall_prob.dtype) / 9.0
        neighbors = F.conv2d(wall_prob.unsqueeze(1), kernel, padding=1).squeeze(1)

        isolation_penalty = wall_prob * torch.exp(-neighbors)
        return isolation_penalty.mean()


class LossScheduler:
    """Manages curriculum-based loss weight scheduling"""

    def __init__(self, config):
        self.config = config
        self.loss_schedules = config.loss_schedule

    def get_scheduled_weights(self, current_stage: int, current_epoch: int,
                              stage_epoch: int, total_stage_epochs: int,
                              base_weights: Dict[str, float]) -> Dict[str, float]:
        """Calculate loss weights based on curriculum schedule"""
        scheduled_weights = base_weights.copy()

        for loss_name, schedule_type in self.loss_schedules.items():
            if loss_name not in scheduled_weights:
                continue

            base_weight = scheduled_weights[loss_name]

            if schedule_type == "static":
                continue

            elif schedule_type == "progressive":
                if loss_name == "topology":
                    start_weight = self.config.topology_start_weight
                    end_weight = self.config.topology_end_weight
                    ramp_epochs = self.config.topology_ramp_epochs
                    progress = min(current_epoch / ramp_epochs, 1.0)
                    scheduled_weights[loss_name] = start_weight + progress * (end_weight - start_weight)

            elif schedule_type == "linear_ramp":
                progress = stage_epoch / max(total_stage_epochs, 1)
                scheduled_weights[loss_name] = base_weight * progress

            elif schedule_type == "exponential":
                progress = stage_epoch / max(total_stage_epochs, 1)
                scheduled_weights[loss_name] = base_weight * (progress ** 2)

            elif schedule_type == "early_decay":
                if current_stage > 1:
                    scheduled_weights[loss_name] = base_weight * 0.3

            elif schedule_type == "staged_ramp":
                if current_stage == 2:
                    progress = stage_epoch / max(total_stage_epochs, 1)
                    scheduled_weights[loss_name] = base_weight * progress
                elif current_stage < 2:
                    scheduled_weights[loss_name] = 0.0

            elif schedule_type == "late_ramp":
                if current_stage == 3:
                    progress = stage_epoch / max(total_stage_epochs, 1)
                    scheduled_weights[loss_name] = base_weight * progress
                elif current_stage < 3:
                    scheduled_weights[loss_name] = 0.0

            elif schedule_type == "mid_ramp":
```

```
                if current_stage >= 2:
                    if current_stage == 2:
                        progress = min(stage_epoch / (total_stage_epochs * 0.5), 1.0)
                        scheduled_weights[loss_name] = base_weight * progress
                    else:
                        scheduled_weights[loss_name] = base_weight
                else:
                    scheduled_weights[loss_name] = 0.0

            elif schedule_type == "delayed_ramp":
                if current_epoch >= self.config.graph_start_epoch:
                    epochs_since_start = current_epoch - self.config.graph_start_epoch
                    ramp_duration = 50
                    progress = min(epochs_since_start / ramp_duration, 1.0)
                    scheduled_weights[loss_name] = self.config.graph_end_weight * progress
                else:
                    scheduled_weights[loss_name] = 0.0

        return scheduled_weights
```

■ File: training\trainer.py

==

```
"""
Dynamic Multi-stage training system with adaptive curriculum learning
Implements novel training strategies: dynamic stage transitions, topology-aware scheduling,
multi-objective optimization, and cross-modal consistency learning
"""

import torch
import torch.nn.utils

# training/trainer.py - Fixed AMP imports
from torch.amp import autocast, GradScaler
import time
import numpy as np
import random
from pathlib import Path
from tqdm import tqdm
from typing import Dict, List, Optional, Tuple
from collections import deque

from .losses import ResearchGradeLoss, LossScheduler
from config import DEFAULT_TRAINING_CONFIG, DEFAULT_LOSS_CONFIG, StageTransitionCriteria


class CurriculumState:
    """Tracks curriculum learning state and metrics"""

    def __init__(self, config):
        self.config = config

        # Loss history for plateau detection
        self.loss_history = {
            "stage1": deque(maxlen=config.plateau_detection_window * 2),
            "stage2": deque(maxlen=config.plateau_detection_window * 2),
            "stage3": deque(maxlen=config.plateau_detection_window * 2),
        }

        # Component loss tracking
        self.component_losses = {
            "segmentation": deque(maxlen=20),
            "dice": deque(maxlen=20),
            "polygon": deque(maxlen=20),
            "voxel": deque(maxlen=20),
            "topology": deque(maxlen=20),
            "latent_consistency": deque(maxlen=20),
            "graph": deque(maxlen=20),
        }

        # Gradient magnitude tracking for dynamic weighting
        self.gradient_norms = {
```

```
                name: deque(maxlen=config.gradient_norm_window)
                for name in self.component_losses.keys()
            }

            # Stage transition tracking
            self.epochs_without_improvement = 0
            self.best_val_loss = float("inf")
            self.stage_transition_epochs = []

            # Dynamic weights history
            self.weight_history = []

    def update_loss_history(self, stage: str, val_loss: float):
        """Update validation loss history for plateau detection"""
        if stage in self.loss_history:
            self.loss_history[stage].append(val_loss)

        # Update improvement tracking
        if val_loss < self.best_val_loss:
            self.best_val_loss = val_loss
            self.epochs_without_improvement = 0
        else:
            self.epochs_without_improvement += 1

    def update_component_losses(self, loss_components: Dict[str, float]):
        """Update individual loss component history"""
        for name, loss_val in loss_components.items():
            if name in self.component_losses:
                self.component_losses[name].append(loss_val)

    def should_transition(self, current_stage: int) -> bool:
        """Check if should transition to next stage"""
        if current_stage == 1:
            val_losses = list(self.loss_history["stage1"])
            return StageTransitionCriteria.should_transition_from_stage1(
                [], val_losses, self.config
            )
        elif current_stage == 2:
            polygon_losses = list(self.component_losses["polygon"])
            return StageTransitionCriteria.should_transition_from_stage2(
                polygon_losses, self.config
            )

        return False


class AdaptiveMultiStageTrainer:
    """
    Advanced multi-stage trainer with dynamic curriculum learning:
    - Adaptive stage transitioning based on performance plateaus
    - Topology-aware loss scheduling
    - Multi-objective optimization with dynamic weighting
    - Cross-modal latent consistency learning
    - Graph-based topology constraints
    """

    # Class constant for rolling checkpoint path
    ROLLING_CHECKPOINT = "latest_checkpoint.pth"

    def __init__(self, model, train_loader, val_loader, device=None, config=None):
        if config is None:
            config = DEFAULT_TRAINING_CONFIG

        self.model = model.to(device or config.device)
        self.train_loader = train_loader
        self.val_loader = val_loader
        self.device = device or config.device
        self.config = config

        # Initialize curriculum state
        self.curriculum_state = CurriculumState(config.curriculum)
        self.loss_scheduler = LossScheduler(config.curriculum)

```

```python
        # Training state tracking for resume functionality
        self.current_stage = 1
        self.current_epoch = 0
        self.global_epoch = 0
        self.stage_epoch = 0
        self.stage_start_time = None
        self.epoch_times = []

        # Add AMP and optimization settings - Updated for new PyTorch API
        self.use_amp = getattr(config, "use_mixed_precision", True)
        self.scaler = GradScaler("cuda", enabled=self.use_amp)
        self.accumulation_steps = getattr(config, "accumulation_steps", 1)
        self.dvx_step_freq = getattr(config, "dvx_step_freq", 1)
        self.voxel_size_stage = getattr(config, "voxel_size_stage", None)
        self.image_size_stage = getattr(config, "image_size_stage", None)
        self._step = 0

        # Enhanced optimizers with better hyperparameters
        self.optimizer_2d = torch.optim.AdamW(
            list(self.model.encoder.parameters())
            + list(self.model.seg_head.parameters())
            + list(self.model.attr_head.parameters())
            + list(self.model.sdf_head.parameters()),
            lr=config.stage1_lr,
            weight_decay=config.stage1_weight_decay,
            betas=(0.9, 0.999),
        )

        self.optimizer_dvx = torch.optim.AdamW(
            self.model.dvx.parameters(),
            lr=config.stage2_lr,
            weight_decay=config.stage2_weight_decay,
            betas=(0.9, 0.999),
        )

        self.optimizer_full = torch.optim.AdamW(
            self.model.parameters(),
            lr=config.stage3_lr,
            weight_decay=config.stage3_weight_decay,
            betas=(0.9, 0.999),
        )

        # Enhanced learning rate schedulers with proper minimum LR
        if config.use_cosine_restarts:
            self.scheduler_2d = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
                self.optimizer_2d, T_0=20, T_mult=1,
                eta_min=config.stage1_lr * 0.1  # Min LR is 10% of initial
            )
            self.scheduler_dvx = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
                self.optimizer_dvx, T_0=15, T_mult=1,
                eta_min=config.stage2_lr * 0.1
            )
            self.scheduler_full = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
                self.optimizer_full, T_0=30, T_mult=1,
                eta_min=config.stage3_lr * 0.1
            )
        else:
            self.scheduler_2d = torch.optim.lr_scheduler.CosineAnnealingLR(
                self.optimizer_2d, T_max=config.max_stage1_epochs,
                eta_min=config.stage1_lr * 0.1  # Min LR is 10% of initial
            )
            self.scheduler_dvx = torch.optim.lr_scheduler.CosineAnnealingLR(
                self.optimizer_dvx, T_max=config.max_stage2_epochs,
                eta_min=config.stage2_lr * 0.1
            )
            self.scheduler_full = torch.optim.lr_scheduler.CosineAnnealingLR(
                self.optimizer_full, T_max=config.max_stage3_epochs,
                eta_min=config.stage3_lr * 0.1
            )

        # Enhanced loss function with dynamic weighting
        base_loss_kwargs = {
            k: v
```

```
            for k, v in DEFAULT_LOSS_CONFIG.__dict__.items()
            if k != "enable_dynamic_weighting"
        }
        self.loss_fn = ResearchGradeLoss(
            **base_loss_kwargs,
            enable_dynamic_weighting=bool(config.curriculum.use_gradnorm),
            gradnorm_alpha=float(config.curriculum.gradnorm_alpha),
            device=self.device,
        )

        self.history = {
            "stage1": {"train_loss": [], "val_loss": [], "component_losses": []},
            "stage2": {"train_loss": [], "val_loss": [], "component_losses": []},
            "stage3": {"train_loss": [], "val_loss": [], "component_losses": []},
            "stage_transitions": [],
            "dynamic_weights": [],
            "curriculum_events": [],
        }

    def _get_eta_string(self, epoch, total_epochs):
        """Calculate and format ETA string"""
        if len(self.epoch_times) == 0:
            return "ETA: calculating..."

        avg_epoch_time = sum(self.epoch_times) / len(self.epoch_times)
        remaining_epochs = total_epochs - epoch - 1
        eta_seconds = avg_epoch_time * remaining_epochs

        if eta_seconds < 60:
            return f"ETA: {int(eta_seconds)}s"
        elif eta_seconds < 3600:
            return f"ETA: {int(eta_seconds // 60)}m {int(eta_seconds % 60)}s"
        else:
            hours = int(eta_seconds // 3600)
            minutes = int((eta_seconds % 3600) // 60)
            return f"ETA: {hours}h {minutes}m"

    def _get_shared_parameters(self):
        """Get shared parameters for GradNorm weighting"""
        # Return encoder parameters as shared across tasks
        return list(self.model.encoder.parameters())

    def _update_loss_weights_for_curriculum(
        self, current_stage: int, stage_epoch: int, total_stage_epochs: int
    ):
        """Update loss weights based on curriculum schedule"""
        base_weights = {
            "seg": self.loss_fn.initial_weights["seg"],
            "dice": self.loss_fn.initial_weights["dice"],
            "sdf": self.loss_fn.initial_weights["sdf"],
            "attr": self.loss_fn.initial_weights["attr"],
            "polygon": self.loss_fn.initial_weights["polygon"],
            "voxel": self.loss_fn.initial_weights["voxel"],
            "topology": self.loss_fn.initial_weights["topology"],
            "latent_consistency": self.loss_fn.initial_weights["latent_consistency"],
            "graph": self.loss_fn.initial_weights["graph"],
        }

        scheduled_weights = self.loss_scheduler.get_scheduled_weights(
            current_stage,
            self.global_epoch,
            stage_epoch,
            total_stage_epochs,
            base_weights,
        )

        self.loss_fn.update_loss_weights(scheduled_weights)

        # Log weight changes
        self.history["dynamic_weights"].append(
            {
                "epoch": self.global_epoch,
                "stage": current_stage,
```

```
                "weights": scheduled_weights.copy(),
            }
        )

    def _train_epoch(self, mode="stage1"):
        """Enhanced training epoch with improved stability and speed"""
        self.model.train()
        total_loss = 0
        component_loss_sums = {}

        # Select optimizer and apply gradient scaling
        if mode == "stage1":
            optimizer = self.optimizer_2d
        elif mode == "stage2":
            optimizer = self.optimizer_dvx
        else:
            optimizer = self.optimizer_full

        # Improved progress tracking
        train_pbar = tqdm(
            self.train_loader, desc=f"Training {mode.upper()}", leave=False, ncols=120
        )

        batch_count = 0
        epoch_start_time = time.time()

        # Add gradient accumulation tracking
        accumulated_loss = 0.0

        for batch_idx, batch in enumerate(train_pbar):
            self._step += 1
            batch = {
                k: v.to(self.device, non_blocking=True) if torch.is_tensor(v) else v
                for k, v in batch.items()
            }

            # Smart geometric computation gating
            run_full_geometric = (
                mode == "stage3" or  # Always run in final stage
                (mode == "stage2" and self._step % 1 == 0) or  # Every other step in stage 2
                (mode == "stage1" and self._step % 2 == 0)  # Every 4th step in stage 1
            )

            with autocast("cuda", enabled=self.use_amp):
                predictions = self.model(
                    batch["image"], run_full_geometric=run_full_geometric
                )

                targets = self._prepare_targets(batch, mode)

                shared_params = (
                    self._get_shared_parameters()
                    if self.config.curriculum.use_gradnorm
                    else None
                )

                loss, loss_components = self.loss_fn(
                    predictions,
                    targets,
                    shared_params,
                    run_full_geometric=run_full_geometric,
                )

                # Scale for accumulation
                loss = loss / self.accumulation_steps
                accumulated_loss += loss.item()

            # Backward pass with stability
            self.scaler.scale(loss).backward()

            # Gradient accumulation and update
            if ((batch_idx + 1) % self.accumulation_steps) == 0:
                # Enhanced gradient clipping
```

```
                self.scaler.unscale_(optimizer)

                # Adaptive gradient clipping based on loss magnitude
                max_grad_norm = min(self.config.grad_clip_norm * (1.0 + accumulated_loss), 2.0)
                torch.nn.utils.clip_grad_norm_(
                    self.model.parameters(), max_grad_norm
                )

                self.scaler.step(optimizer)
                self.scaler.update()
                optimizer.zero_grad()

                # Reset accumulation
                accumulated_loss = 0.0

            current_loss = loss.item() * self.accumulation_steps
            total_loss += current_loss

            # Track components with better averaging
            for name, component_loss in loss_components.items():
                if name != "total":
                    loss_val = (
                        component_loss.item()
                        if torch.is_tensor(component_loss)
                        else component_loss
                    )
                    if name not in component_loss_sums:
                        component_loss_sums[name] = []
                    component_loss_sums[name].append(loss_val)

            batch_count += 1

            # Less frequent but more informative logging
            if (batch_idx + 1) % 100 == 0:
                elapsed = time.time() - epoch_start_time
                avg_time_per_batch = elapsed / (batch_idx + 1)

                # Show meaningful component averages
                recent_components = {}
                for name, vals in component_loss_sums.items():
                    if len(vals) >= 10:  # Only show if we have enough samples
                        recent_avg = np.mean(vals[-10:])  # Last 10 batches
                        if recent_avg > 0.01:  # Only show significant components
                            recent_components[name] = recent_avg

                comp_str = ", ".join([f"{k}:{v:.3f}" for k, v in recent_components.items()])
                print(f"[Epoch {self.global_epoch}] Batch {batch_idx+1} | "
                      f"{avg_time_per_batch:.2f}s/batch | loss:{total_loss/batch_count:.4f} | {comp_str}")

            # Update progress with meaningful info
            train_pbar.set_postfix({
                "loss": f"{current_loss:.4f}",
                "lr": f"{optimizer.param_groups[0]['lr']:.6f}"
            })

        # Calculate proper component averages
        avg_component_losses = {}
        for name, loss_list in component_loss_sums.items():
            if loss_list:
                avg_component_losses[name] = np.mean(loss_list)
            else:
                avg_component_losses[name] = 0.0

        return total_loss / batch_count, avg_component_losses

    def _prepare_targets(self, batch, mode):
        """Prepare targets based on training mode"""
        if mode == "stage1":
            return {"mask": batch["mask"], "attributes": batch["attributes"]}
        elif mode == "stage2":
            return {
                "polygons_gt": {
```

```
                    "polygons": batch["polygons_gt"]["polygons"].to(self.device),
                    "valid_mask": batch["polygons_gt"]["valid_mask"].to(self.device),
                }
            }
        else:  # stage3
            return {
                "mask": batch["mask"],
                "attributes": batch["attributes"],
                "voxels_gt": batch["voxels_gt"],
                "polygons_gt": {
                    "polygons": batch["polygons_gt"]["polygons"].to(self.device),
                    "valid_mask": batch["polygons_gt"]["valid_mask"].to(self.device),
                },
            }

    def _validate(self, mode="stage1"):
        """Enhanced validation with consistent loss computation"""
        self.model.eval()
        total_loss = 0
        component_loss_sums = {}

        val_pbar = tqdm(
            self.val_loader, desc=f"Validating {mode.upper()}", leave=False, ncols=120
        )

        batch_count = 0
        with torch.no_grad():
            for batch in val_pbar:
                batch = {
                    k: v.to(self.device, non_blocking=True) if torch.is_tensor(v) else v
                    for k, v in batch.items()
                }

                with autocast("cuda", enabled=self.use_amp):
                    # ALWAYS run full geometric in validation for consistency
                    predictions = self.model(batch["image"], run_full_geometric=True)

                    targets = self._prepare_targets(batch, "stage3")  # Use full targets

                    # Use same loss computation as training but without dynamic weighting
                    loss, loss_components = self.loss_fn(
                        predictions, targets, shared_parameters=None, run_full_geometric=True
                    )

                current_loss = loss.item()
                total_loss += current_loss

                # Track component losses properly
                for name, component_loss in loss_components.items():
                    if name != "total":
                        loss_val = (
                            component_loss.item()
                            if torch.is_tensor(component_loss)
                            else component_loss
                        )
                        if name not in component_loss_sums:
                            component_loss_sums[name] = []
                        component_loss_sums[name].append(loss_val)

                batch_count += 1
                val_pbar.set_postfix({"loss": f"{current_loss:.4f}"})

        # Calculate proper averages
        avg_component_losses = {}
        for name, loss_list in component_loss_sums.items():
            if loss_list:
                avg_component_losses[name] = np.mean(loss_list)
            else:
                avg_component_losses[name] = 0.0

        return total_loss / batch_count, avg_component_losses

    def train_stage_adaptive(self, stage: int, max_epochs: int, min_epochs: int):
```

```
        """
        Train a stage with adaptive termination based on curriculum learning

        Args:
            stage: Stage number (1, 2, 3)
            max_epochs: Maximum epochs for this stage
            min_epochs: Minimum epochs before considering transition
        """
        print("=" * 60)
        print(f"STAGE {stage}: Adaptive Training with Dynamic Curriculum")
        print("=" * 60)

        self.current_stage = stage
        self.stage_start_time = time.time()

        # Only reset if not resuming
        if not hasattr(self, "epoch_times") or self.epoch_times is None:
            self.epoch_times = []

        start_epoch = int(self.stage_epoch or 0)

        # Set parameter gradients for current stage
        self._configure_stage_parameters(stage)

        mode_name = f"stage{stage}"

        for epoch in range(start_epoch, max_epochs):
            epoch_start_time = time.time()
            self.stage_epoch = epoch
            self.global_epoch += 1

            # Update loss weights based on curriculum
            self._update_loss_weights_for_curriculum(stage, epoch, max_epochs)

            print(
                f"\nStage {stage} - Epoch {epoch+1}/{max_epochs} (Global: {self.global_epoch})"
            )

            # Training and validation
            train_loss, train_components = self._train_epoch(mode_name)
            val_loss, val_components = self._validate(mode_name)

            # Record epoch time
            epoch_time = time.time() - epoch_start_time
            self.epoch_times.append(epoch_time)

            if len(self.epoch_times) > 10:
                self.epoch_times.pop(0)

            # Update curriculum state
            self.curriculum_state.update_loss_history(mode_name, val_loss)
            self.curriculum_state.update_component_losses(val_components)

            # Store training history
            self.history[mode_name]["train_loss"].append(train_loss)
            self.history[mode_name]["val_loss"].append(val_loss)
            self.history[mode_name]["component_losses"].append(val_components)

            # Update learning rate
            if stage == 1:
                self.scheduler_2d.step()
                current_lr = self.optimizer_2d.param_groups[0]['lr']
            elif stage == 2:
                self.scheduler_dvx.step()
                current_lr = self.optimizer_dvx.param_groups[0]['lr']
            else:
                self.scheduler_full.step()
                current_lr = self.optimizer_full.param_groups[0]['lr']

            # Log learning rate every 10 epochs
            if (epoch + 1) % 10 == 0:
                print(f"Learning rate at epoch {epoch + 1}: {current_lr:.6f}")

```

```
            # Display comprehensive results
            self._display_epoch_results(
                epoch,
                max_epochs,
                train_loss,
                val_loss,
                train_components,
                val_components,
                epoch_time,
            )

            # Check for adaptive stage transition
            if epoch >= min_epochs:
                should_transition = self.curriculum_state.should_transition(stage)
                if should_transition:
                    print(
                        f"\n? ADAPTIVE TRANSITION: Stage {stage} converged after {epoch+1} epochs"
                    )
                    print(
                        "   Detected performance plateau - transitioning to next stage"
                    )

                    self.history["stage_transitions"].append(
                        {
                            "from_stage": stage,
                            "epoch": epoch + 1,
                            "global_epoch": self.global_epoch,
                            "reason": "performance_plateau",
                        }
                    )

                    self.history["curriculum_events"].append(
                        {
                            "type": "stage_transition",
                            "stage": stage,
                            "epoch": self.global_epoch,
                            "details": f"Converged after {epoch+1} epochs",
                        }
                    )
                    break

            # Save rolling checkpoint
            if (epoch + 1) % self.config.checkpoint_freq == 0:
                self._save_rolling_checkpoint()

        print(f"\nStage {stage} completed after {epoch+1} epochs")

    def _configure_stage_parameters(self, stage: int):
        """Configure which parameters require gradients for each stage"""
        # First freeze everything
        for param in self.model.parameters():
            param.requires_grad = False

        if stage == 1:
            # Stage 1: Segmentation + Attributes (2D only)
            for param in self.model.encoder.parameters():
                param.requires_grad = True
            for param in self.model.seg_head.parameters():
                param.requires_grad = True
            for param in self.model.attr_head.parameters():
                param.requires_grad = True
            for param in self.model.sdf_head.parameters():
                param.requires_grad = True

        elif stage == 2:
            # Stage 2: DVX training (polygon fitting) - keep encoder frozen initially
            for param in self.model.dvx.parameters():
                param.requires_grad = True
            # Optionally unfreeze encoder in later epochs
            if self.stage_epoch > 10:
                for param in self.model.encoder.parameters():
                    param.requires_grad = True

```

```
        else:  # stage == 3
            # Stage 3: End-to-end fine-tuning (all parameters)
            for param in self.model.parameters():
                param.requires_grad = True

    def _display_epoch_results(
        self,
        epoch: int,
        total_epochs: int,
        train_loss: float,
        val_loss: float,
        train_components: Dict,
        val_components: Dict,
        epoch_time: float,
    ):
        """Display comprehensive epoch results with curriculum information"""
        eta_str = self._get_eta_string(epoch, total_epochs)

        print(f"Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}")
        print(f"Epoch time: {epoch_time:.1f}s, {eta_str}")

        # Show significant component losses
        significant_components = {
            k: v
            for k, v in val_components.items()
            if v > 0.01
            and k
            in [
                "seg",
                "dice",
                "polygon",
                "voxel",
                "topology",
                "latent_consistency",
                "graph",
            ]
        }
        if significant_components:
            comp_str = ", ".join(
                [f"{k}: {v:.3f}" for k, v in significant_components.items()]
            )
            print(f"Components: {comp_str}")

        # Show current loss weights for active components
        active_weights = {k: v for k, v in self.loss_fn.weights.items() if v > 0.001}
        if active_weights:
            weight_str = ", ".join([f"{k}: {v:.3f}" for k, v in active_weights.items()])
            print(f"Weights: {weight_str}")

        # Show curriculum status
        plateau_epochs = self.curriculum_state.epochs_without_improvement
        if plateau_epochs > 0:
            print(f"Plateau: {plateau_epochs} epochs without improvement")

    def _save_rolling_checkpoint(self):
        """Enhanced checkpoint saving with curriculum state, RNG state, and scaler state"""
        checkpoint = {
            "model_state_dict": self.model.state_dict(),
            "optimizer_2d_state_dict": self.optimizer_2d.state_dict(),
            "optimizer_dvx_state_dict": self.optimizer_dvx.state_dict(),
            "optimizer_full_state_dict": self.optimizer_full.state_dict(),
            "scheduler_2d_state_dict": self.scheduler_2d.state_dict(),
            "scheduler_dvx_state_dict": self.scheduler_dvx.state_dict(),
            "scheduler_full_state_dict": self.scheduler_full.state_dict(),
            "scaler_state_dict": self.scaler.state_dict(),  # Add AMP scaler state
            "loss_fn_state": {
                "weights": self.loss_fn.weights,
                "initial_weights": self.loss_fn.initial_weights,
            },
            "history": self.history,
            "config": self.config,
            "current_stage": self.current_stage,
            "current_epoch": self.current_epoch,
```

```
            "global_epoch": self.global_epoch,
            "stage_epoch": self.stage_epoch,
            "epoch_times": self.epoch_times,
            "step_counter": self._step,  # Save step counter for DVX gating
            "curriculum_state": {
                "loss_history": dict(self.curriculum_state.loss_history),
                "component_losses": dict(self.curriculum_state.component_losses),
                "epochs_without_improvement": self.curriculum_state.epochs_without_improvement,
                "best_val_loss": self.curriculum_state.best_val_loss,
                "stage_transition_epochs": self.curriculum_state.stage_transition_epochs,
            },
            "rng_state": {
                "torch": torch.get_rng_state(),
                "cuda": torch.cuda.get_rng_state_all()
                if torch.cuda.is_available()
                else None,
                "numpy": np.random.get_state(),
                "python": random.getstate(),
            },
        }

        checkpoint_path = self.ROLLING_CHECKPOINT
        torch.save(checkpoint, checkpoint_path)
        print(f"Rolling checkpoint saved: {checkpoint_path}")

    def load_checkpoint(self, filename):
        """
        Enhanced checkpoint loading with architecture compatibility handling
        Safely handles model architecture changes by filtering incompatible parameters
        """
        print(f"Loading checkpoint: {filename}")
        checkpoint = torch.load(filename, map_location=self.device, weights_only=False)

        # === SAFE MODEL STATE LOADING ===
        model_state = checkpoint["model_state_dict"]
        current_model_keys = set(self.model.state_dict().keys())

        # Filter parameters into compatible and incompatible
        compatible_state = {}
        incompatible_keys = []
        missing_keys = []

        # Check each parameter in the checkpoint
        for key, value in model_state.items():
            if key in current_model_keys:
                # Check if tensor shapes match
                current_param = self.model.state_dict()[key]
                if current_param.shape == value.shape:
                    compatible_state[key] = value
                else:
                    incompatible_keys.append(f"{key} (shape mismatch: {value.shape} ->
+                             {current_param.shape})")
            else:
                incompatible_keys.append(f"{key} (parameter not found in current model)")

        # Check for missing parameters in checkpoint
        for key in current_model_keys:
            if key not in model_state:
                missing_keys.append(key)

        # Load compatible parameters only
        loaded_keys, unexpected_keys = self.model.load_state_dict(compatible_state, strict=False)

        # Report parameter loading status
        print(f"? Successfully loaded {len(compatible_state)} compatible parameters")

        if incompatible_keys:
            print(f"? Skipped {len(incompatible_keys)} incompatible parameters:")
            for key in incompatible_keys[:10]:  # Show first 10
                print(f"    - {key}")
            if len(incompatible_keys) > 10:
                print(f"    ... and {len(incompatible_keys) - 10} more")

```

```
        if missing_keys:
            print(f"? {len(missing_keys)} parameters will use random initialization:")
            for key in missing_keys[:10]:  # Show first 10
                print(f"    - {key}")
            if len(missing_keys) > 10:
                print(f"    ... and {len(missing_keys) - 10} more")

        # === OPTIMIZER STATES LOADING ===
        try:
            self.optimizer_2d.load_state_dict(checkpoint["optimizer_2d_state_dict"])
            print("? Loaded optimizer_2d state")
        except Exception as e:
            print(f"? Could not load optimizer_2d state: {e}")

        try:
            self.optimizer_dvx.load_state_dict(checkpoint["optimizer_dvx_state_dict"])
            print("? Loaded optimizer_dvx state")
        except Exception as e:
            print(f"? Could not load optimizer_dvx state: {e}")

        try:
            self.optimizer_full.load_state_dict(checkpoint["optimizer_full_state_dict"])
            print("? Loaded optimizer_full state")
        except Exception as e:
            print(f"? Could not load optimizer_full state: {e}")

        # === AMP SCALER STATE ===
        if "scaler_state_dict" in checkpoint:
            try:
                self.scaler.load_state_dict(checkpoint["scaler_state_dict"])
                print("? Loaded AMP scaler state")
            except Exception as e:
                print(f"? Could not load scaler state: {e}")

        # === SCHEDULER STATES ===
        scheduler_mappings = [
            ("scheduler_2d_state_dict", self.scheduler_2d),
            ("scheduler_dvx_state_dict", self.scheduler_dvx),
            ("scheduler_full_state_dict", self.scheduler_full),
        ]

        for state_key, scheduler_obj in scheduler_mappings:
            if state_key in checkpoint:
                try:
                    scheduler_obj.load_state_dict(checkpoint[state_key])
                    print(f"? Loaded {state_key.replace('_state_dict', '')} scheduler")
                except Exception as e:
                    print(f"? Could not load {state_key}: {e}")

        # === LOSS FUNCTION STATE ===
        if "loss_fn_state" in checkpoint:
            try:
                loaded_weights = checkpoint["loss_fn_state"]["weights"]
                if isinstance(loaded_weights, dict):
                    # Handle device transfer for tensor weights
                    self.loss_fn.weights = {
                        k: (v.to(self.device) if torch.is_tensor(v) else v)
                        for k, v in loaded_weights.items()
                    }
                else:
                    self.loss_fn.weights = loaded_weights

                self.loss_fn.initial_weights = checkpoint["loss_fn_state"]["initial_weights"]
                print("? Loaded loss function weights")
            except Exception as e:
                print(f"? Could not load loss function state: {e}")

        # === TRAINING HISTORY ===
        if "history" in checkpoint:
            self.history = checkpoint["history"]
            print("? Loaded training history")

        # === TRAINING STATE VARIABLES ===
```

```
        state_variables = [
            ("current_stage", "current_stage"),
            ("current_epoch", "current_epoch"),
            ("global_epoch", "global_epoch"),
            ("stage_epoch", "stage_epoch"),
            ("epoch_times", "epoch_times"),
            ("step_counter", "_step"),
        ]

        for checkpoint_key, attr_name in state_variables:
            if checkpoint_key in checkpoint:
                setattr(self, attr_name, checkpoint[checkpoint_key])
                print(f"? Restored {checkpoint_key}: {getattr(self, attr_name)}")

        # === CURRICULUM STATE RESTORATION ===
        if "curriculum_state" in checkpoint:
            try:
                cs = checkpoint["curriculum_state"]

                # Restore loss history deques
                for key, history in cs.get("loss_history", {}).items():
                    self.curriculum_state.loss_history[key] = deque(
                        history, maxlen=self.config.curriculum.plateau_detection_window * 2
                    )

                # Restore component loss deques
                for key, history in cs.get("component_losses", {}).items():
                    self.curriculum_state.component_losses[key] = deque(history, maxlen=20)

                # Restore curriculum metrics
                self.curriculum_state.epochs_without_improvement =
                         cs.get("epochs_without_improvement", 0)
                self.curriculum_state.best_val_loss = cs.get("best_val_loss", float("inf"))
                self.curriculum_state.stage_transition_epochs = cs.get("stage_transition_epochs",
                         [])

                print("? Restored curriculum learning state")
            except Exception as e:
                print(f"? Could not restore curriculum state: {e}")

        # === RNG STATE RESTORATION ===
        if "rng_state" in checkpoint:
            try:
                rs = checkpoint["rng_state"]

                # Torch RNG (CPU)
                if "torch" in rs and rs["torch"] is not None:
                    torch_state = rs["torch"]
                    if torch.is_tensor(torch_state) and torch_state.dtype == torch.uint8:
                        torch.set_rng_state(torch_state)
                    else:
                        torch.set_rng_state(torch.tensor(torch_state, dtype=torch.uint8))

                # CUDA RNG (all devices)
                if "cuda" in rs and rs["cuda"] is not None and torch.cuda.is_available():
                    cuda_state = rs["cuda"]
                    cuda_tensors = []
                    for s in cuda_state:
                        if torch.is_tensor(s) and s.dtype == torch.uint8:
                            cuda_tensors.append(s)
                        else:
                            cuda_tensors.append(torch.tensor(s, dtype=torch.uint8))
                    torch.cuda.set_rng_state_all(cuda_tensors)

                # NumPy RNG
                if "numpy" in rs and rs["numpy"] is not None:
                    np.random.set_state(rs["numpy"])

                # Python random RNG
                if "python" in rs and rs["python"] is not None:
                    random.setstate(rs["python"])

                print("? Restored RNG states for reproducibility")
```

```
            except Exception as e:
                print(f"? Could not restore RNG states: {e}")

        # === DATALOADER STATE (if available) ===
        if "dataloader_state" in checkpoint:
            try:
                dl_state = checkpoint["dataloader_state"]
                if (dl_state.get("train_sampler_state") is not None and
                    hasattr(self.train_loader.sampler, "__dict__")):
                    self.train_loader.sampler.__dict__.update(dl_state["train_sampler_state"])

                if (dl_state.get("val_sampler_state") is not None and
                    hasattr(self.val_loader.sampler, "__dict__")):
                    self.val_loader.sampler.__dict__.update(dl_state["val_sampler_state"])

                print("? Restored dataloader sampler states")
            except Exception as e:
                print(f"? Could not restore dataloader states: {e}")

        # === FINAL REPORT ===
        print("\n" + "="*60)
        print("CHECKPOINT LOADING SUMMARY")
        print("="*60)
        print(f"? Checkpoint loaded: {filename}")
        print(f"? Resuming from Stage {self.current_stage}, Global Epoch {self.global_epoch}")
        print(f"? Model parameters: {len(compatible_state)}/{len(model_state)} loaded successfully")

        if hasattr(self, 'curriculum_state'):
            print(f"? Curriculum state: {self.curriculum_state.epochs_without_improvement} epochs without improvement")

        if incompatible_keys:
            print(f"? Architecture changes detected: {len(incompatible_keys)} parameters skipped")
            print("  This is normal after model architecture updates.")

        if missing_keys:
            print(f"? New parameters detected: {len(missing_keys)} will use random initialization")
            print("  These will be learned quickly during resumed training.")

        print("="*60)
        print("Ready to resume adaptive multi-stage training!")
        print("="*60)

    def train_all_stages(self):
        """
        Run complete adaptive multi-stage training pipeline

        This is the main entry point that orchestrates the dynamic curriculum learning
        """
        if Path(self.ROLLING_CHECKPOINT).exists():
            print(f"Found existing checkpoint: {self.ROLLING_CHECKPOINT}")
            print("Resuming adaptive training from checkpoint...")
            self.load_checkpoint(self.ROLLING_CHECKPOINT)
        else:
            print("Starting fresh adaptive training pipeline...")
            self.current_stage = 1
            self.current_epoch = 0
            self.global_epoch = 0

        print("\n" + "=" * 80)
        print("ADAPTIVE MULTI-STAGE TRAINING WITH DYNAMIC CURRICULUM")
        print("Novel Training Strategies:")
        print("? Adaptive Stage Transitioning (Dynamic Curriculum)")
        print("? Topology-aware Loss Scheduling")
        print("? Multi-objective Optimization with Dynamic Weighting")
        print("? Cross-modal Latent Consistency Learning")
        print("? Graph-based Topology Constraints")
        print("=" * 80)

        # Stage 1: Adaptive 2D training
        if self.current_stage <= 1:
            print("\n? STAGE 1: Adaptive 2D Segmentation + Attributes Training")
            self.train_stage_adaptive(
```

```
                stage=1,
                max_epochs=self.config.max_stage1_epochs,
                min_epochs=self.config.min_stage1_epochs,
            )
            self.current_stage = 2
            self.stage_epoch = 0
            print("\nStage 1 completed. Transitioning to Stage 2...")

        # Stage 2: Adaptive DVX training
        if self.current_stage <= 2:
            print("\n? STAGE 2: Adaptive DVX Polygon Fitting Training")
            self.train_stage_adaptive(
                stage=2,
                max_epochs=self.config.max_stage2_epochs,
                min_epochs=self.config.min_stage2_epochs,
            )
            self.current_stage = 3
            self.stage_epoch = 0
            print("\nStage 2 completed. Transitioning to Stage 3...")

        # Stage 3: Adaptive end-to-end fine-tuning
        if self.current_stage <= 3:
            print("\n? STAGE 3: Adaptive End-to-End Fine-tuning with Full Loss Suite")
            self.train_stage_adaptive(
                stage=3,
                max_epochs=self.config.max_stage3_epochs,
                min_epochs=self.config.min_stage3_epochs,
            )
            print("\nStage 3 completed!")

        print("\n" + "=" * 80)
        print("? ALL ADAPTIVE TRAINING STAGES COMPLETED!")
        print("=" * 80)

        # Generate training report
        self._generate_training_report()

        # Save final model
        self._save_checkpoint("final_adaptive_model.pth")

        # Clean up rolling checkpoint
        if Path(self.ROLLING_CHECKPOINT).exists():
            Path(self.ROLLING_CHECKPOINT).unlink()
            print(f"Cleaned up rolling checkpoint: {self.ROLLING_CHECKPOINT}")

        return self.history

    def _generate_training_report(self):
        """Generate comprehensive training report with curriculum insights"""
        print("\n" + "=" * 60)
        print("ADAPTIVE TRAINING REPORT")
        print("=" * 60)

        # Stage transition summary
        if self.history["stage_transitions"]:
            print("\n? Stage Transitions:")
            for transition in self.history["stage_transitions"]:
                print(
                    f"  ? Stage {transition['from_stage']} ? {transition['from_stage']+1}: "
                    f"Epoch {transition['epoch']} (Global: {transition['global_epoch']})"
                )
                print(f"    Reason: {transition['reason']}")

        # Dynamic weight evolution
        if self.history["dynamic_weights"]:
            print(
                f"\n??  Dynamic Weight Updates: {len(self.history['dynamic_weights'])} updates"
            )
            final_weights = self.history["dynamic_weights"][-1]["weights"]
            print("  Final loss weights:")
            for name, weight in final_weights.items():
                if weight > 0.001:
                    print(f"    {name}: {weight:.3f}")
```

```

        # Curriculum events
        if self.history["curriculum_events"]:
            print(
                f"\n? Curriculum Events: {len(self.history['curriculum_events'])} events"
            )
            for event in self.history["curriculum_events"][-5:]:  # Show last 5 events
                print(
                    f"  ? {event['type']} at global epoch {event['epoch']}: {event['details']}"
                )

        # Performance summary
        print("\n? Final Performance:")
        for stage_name, data in self.history.items():
            if isinstance(data, dict) and "val_loss" in data and data["val_loss"]:
                final_loss = data["val_loss"][-1]
                best_loss = min(data["val_loss"])
                print(
                    f"  ? {stage_name.upper()}: Final={final_loss:.4f}, Best={best_loss:.4f}"
                )

        print("\n? Training completed with novel adaptive curriculum strategies!")
        print("=" * 60)

    def _save_checkpoint(self, filename):
        """Save final training checkpoint"""
        checkpoint = {
            "model_state_dict": self.model.state_dict(),
            "optimizer_2d_state_dict": self.optimizer_2d.state_dict(),
            "optimizer_dvx_state_dict": self.optimizer_dvx.state_dict(),
            "optimizer_full_state_dict": self.optimizer_full.state_dict(),
            "scheduler_2d_state_dict": self.scheduler_2d.state_dict(),
            "scheduler_dvx_state_dict": self.scheduler_dvx.state_dict(),
            "scheduler_full_state_dict": self.scheduler_full.state_dict(),
            "scaler_state_dict": self.scaler.state_dict(),
            "loss_fn_state": {
                "weights": self.loss_fn.weights,
                "initial_weights": self.loss_fn.initial_weights,
            },
            "history": self.history,
            "config": self.config,
            "final_stage": self.current_stage,
            "total_epochs": self.global_epoch,
            "training_complete": True,
            "curriculum_summary": {
                "stage_transitions": len(self.history["stage_transitions"]),
                "weight_updates": len(self.history["dynamic_weights"]),
                "curriculum_events": len(self.history["curriculum_events"]),
            },
        }
        torch.save(checkpoint, filename)
        print(f"Final model saved: {filename}")


# Legacy compatibility class
class MultiStageTrainer(AdaptiveMultiStageTrainer):
    """
    Legacy wrapper for backward compatibility
    Redirects to the new adaptive trainer
    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        print("Note: Using enhanced AdaptiveMultiStageTrainer with dynamic curriculum")

    def train_stage1(self, epochs=None):
        """Legacy method - redirects to adaptive training"""
        max_epochs = epochs or self.config.max_stage1_epochs
        min_epochs = self.config.min_stage1_epochs
        return self.train_stage_adaptive(1, max_epochs, min_epochs)

    def train_stage2(self, epochs=None):
        """Legacy method - redirects to adaptive training"""
```

```
        max_epochs = epochs or self.config.max_stage2_epochs
        min_epochs = self.config.min_stage2_epochs
        return self.train_stage_adaptive(2, max_epochs, min_epochs)

    def train_stage3(self, epochs=None):
        """Legacy method - redirects to adaptive training"""
        max_epochs = epochs or self.config.max_stage3_epochs
        min_epochs = self.config.min_stage3_epochs
        return self.train_stage_adaptive(3, max_epochs, min_epochs)
```

■ File: utils\visualization.py

==

```
"""
Visualization and utility functions
"""

import matplotlib.pyplot as plt
import numpy as np
import cv2
import torch
from pathlib import Path
from evaluation.metrics import compute_iou


def plot_training_history(history, save_path="training_history.png"):
    """Plot training curves for all stages"""
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    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 was executed
            axes[idx].plot(data["train_loss"], label="Train", linewidth=2)
            axes[idx].plot(data["val_loss"], label="Validation", linewidth=2)
            axes[idx].set_title(f"{stage.upper()} Training")
            axes[idx].set_xlabel("Epoch")
            axes[idx].set_ylabel("Loss")
            axes[idx].legend()
            axes[idx].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig(save_path, dpi=300, bbox_inches="tight")
    plt.show()


def plot_curriculum_analysis(history, save_path="curriculum_analysis.png"):
    """Plot curriculum learning analysis including stage transitions and adaptive behavior"""
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Plot 1: Stage transition timeline
    if "stage_transitions" in history and history["stage_transitions"]:
        transitions = history["stage_transitions"]

        # Extract transition epochs and reasons
        transition_epochs = [t["epoch"] for t in transitions]
        transition_stages = [t["from_stage"] + " ? " + t["to_stage"] for t in transitions]
        transition_reasons = [t.get("reason", "threshold") for t in transitions]

        # Create timeline
        y_positions = range(len(transition_epochs))
        colors = ['red' if 'patience' in reason else 'green' for reason in transition_reasons]

        axes[0, 0].barh(y_positions, transition_epochs, color=colors, alpha=0.7)
        axes[0, 0].set_yticks(y_positions)
        axes[0, 0].set_yticklabels(transition_stages)
        axes[0, 0].set_xlabel("Epoch")
        axes[0, 0].set_title("Stage Transition Timeline")
        axes[0, 0].grid(True, alpha=0.3)

        # Add legend
        axes[0, 0].legend(['Patience-based', 'Threshold-based'], loc='lower right')
    else:
```

```
        axes[0, 0].text(0.5, 0.5, "No stage transitions recorded",
                       ha='center', va='center', transform=axes[0, 0].transAxes)
        axes[0, 0].set_title("Stage Transition Timeline")

    # Plot 2: Loss component evolution
    if "dynamic_weights" in history and history["dynamic_weights"]:
        weight_data = history["dynamic_weights"]
        epochs = [entry["epoch"] for entry in weight_data]

        # Plot each loss component weight
        weight_names = list(weight_data[0]["weights"].keys()) if weight_data else []
        for weight_name in weight_names[:5]:  # Limit to top 5 for readability
            weights = [entry["weights"].get(weight_name, 0) for entry in weight_data]
            if any(w > 0.001 for w in weights):  # Only plot significant weights
                axes[0, 1].plot(epochs, weights, label=weight_name, linewidth=2, marker='o',
                         markersize=3)

        axes[0, 1].set_xlabel("Global Epoch")
        axes[0, 1].set_ylabel("Loss Weight")
        axes[0, 1].set_title("Dynamic Loss Weight Evolution")
        axes[0, 1].legend()
        axes[0, 1].grid(True, alpha=0.3)
    else:
        axes[0, 1].text(0.5, 0.5, "No dynamic weights recorded",
                       ha='center', va='center', transform=axes[0, 1].transAxes)
        axes[0, 1].set_title("Dynamic Loss Weight Evolution")

    # Plot 3: Curriculum progress indicators
    if "curriculum_events" in history and history["curriculum_events"]:
        events = history["curriculum_events"]
        event_types = {}

        for event in events:
            event_type = event.get("type", "unknown")
            if event_type not in event_types:
                event_types[event_type] = []
            event_types[event_type].append(event["epoch"])

        # Plot event timeline
        y_offset = 0
        for event_type, epochs in event_types.items():
            axes[1, 0].scatter(epochs, [y_offset] * len(epochs),
                              label=event_type, s=50, alpha=0.7)
            y_offset += 1

        axes[1, 0].set_xlabel("Epoch")
        axes[1, 0].set_ylabel("Event Type")
        axes[1, 0].set_title("Curriculum Learning Events")
        axes[1, 0].legend()
        axes[1, 0].grid(True, alpha=0.3)
    else:
        axes[1, 0].text(0.5, 0.5, "No curriculum events recorded",
                       ha='center', va='center', transform=axes[1, 0].transAxes)
        axes[1, 0].set_title("Curriculum Learning Events")

    # Plot 4: Stage performance comparison
    stage_names = ["stage1", "stage2", "stage3"]
    stage_performance = {}

    for stage_name in stage_names:
        if stage_name in history and isinstance(history[stage_name], dict):
            stage_data = history[stage_name]
            if "val_loss" in stage_data and stage_data["val_loss"]:
                stage_performance[stage_name] = {
                    "final_loss": stage_data["val_loss"][-1],
                    "best_loss": min(stage_data["val_loss"]),
                    "epochs": len(stage_data["val_loss"])
                }

    if stage_performance:
        stages = list(stage_performance.keys())
        final_losses = [stage_performance[s]["final_loss"] for s in stages]
        best_losses = [stage_performance[s]["best_loss"] for s in stages]
```

```

        x = np.arange(len(stages))
        width = 0.35

        axes[1, 1].bar(x - width/2, final_losses, width, label='Final Loss', alpha=0.8)
        axes[1, 1].bar(x + width/2, best_losses, width, label='Best Loss', alpha=0.8)

        axes[1, 1].set_xlabel("Training Stage")
        axes[1, 1].set_ylabel("Validation Loss")
        axes[1, 1].set_title("Stage Performance Comparison")
        axes[1, 1].set_xticks(x)
        axes[1, 1].set_xticklabels([s.upper() for s in stages])
        axes[1, 1].legend()
        axes[1, 1].grid(True, alpha=0.3)

        # Add epoch count annotations
        for i, stage in enumerate(stages):
            epochs = stage_performance[stage]["epochs"]
            axes[1, 1].text(i, max(final_losses) * 0.9, f'{epochs} epochs',
                           ha='center', va='bottom', fontsize=9)
    else:
        axes[1, 1].text(0.5, 0.5, "No stage performance data",
                       ha='center', va='center', transform=axes[1, 1].transAxes)
        axes[1, 1].set_title("Stage Performance Comparison")

    plt.tight_layout()
    plt.savefig(save_path, dpi=300, bbox_inches="tight")
    plt.close()

    print(f"Curriculum analysis saved to {save_path}")


def visualize_predictions(image, predictions, targets=None, save_path=None):
    """Visualize model predictions"""
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # Original image
    if len(image.shape) == 4:
        img_np = image[0].permute(1, 2, 0).cpu().numpy()
    else:
        img_np = image.permute(1, 2, 0).cpu().numpy()

    axes[0, 0].imshow(img_np)
    axes[0, 0].set_title("Input Image")
    axes[0, 0].axis('off')

    # Predicted segmentation
    if "segmentation" in predictions:
        seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0].cpu().numpy()
        axes[0, 1].imshow(seg_pred, cmap='tab10')
        axes[0, 1].set_title("Predicted Segmentation")
        axes[0, 1].axis('off')

    # Ground truth segmentation (if available)
    if targets and "mask" in targets:
        gt_mask = targets["mask"][0].cpu().numpy()
        axes[0, 2].imshow(gt_mask, cmap='tab10')
        axes[0, 2].set_title("Ground Truth Segmentation")
        axes[0, 2].axis('off')

    # SDF prediction
    if "sdf" in predictions:
        sdf_pred = predictions["sdf"][0, 0].cpu().numpy()
        im = axes[1, 0].imshow(sdf_pred, cmap='RdBu', vmin=-1, vmax=1)
        axes[1, 0].set_title("Predicted SDF")
        axes[1, 0].axis('off')
        plt.colorbar(im, ax=axes[1, 0])

    # Polygon visualization
    if "polygons" in predictions:
        poly_vis = visualize_polygons(
            predictions["polygons"][0],
            predictions["polygon_validity"][0],
```

```
            image_size=(256, 256)
        )
        axes[1, 1].imshow(poly_vis)
        axes[1, 1].set_title("Predicted Polygons")
        axes[1, 1].axis('off')

    # 3D voxel slice
    if "voxels_pred" in predictions:
        voxels = torch.sigmoid(predictions["voxels_pred"][0]).cpu().numpy()
        # Show middle slice
        mid_slice = voxels[voxels.shape[0]//2]
        axes[1, 2].imshow(mid_slice, cmap='viridis')
        axes[1, 2].set_title("3D Voxels (Mid Slice)")
        axes[1, 2].axis('off')

    plt.tight_layout()

    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches="tight")

    plt.show()


def visualize_polygons(polygons, validity, image_size=(256, 256), threshold=0.5):
    """Visualize predicted polygons"""
    vis_img = np.zeros((*image_size, 3), dtype=np.uint8)

    for poly_idx, (polygon, valid_score) in enumerate(zip(polygons, validity)):
        if valid_score > threshold:
            # Convert to image coordinates
            points = polygon.cpu().numpy() * np.array(image_size)

            # Remove zero-padded points
            valid_points = points[points.sum(axis=1) > 0]

            if len(valid_points) >= 3:
                points_int = valid_points.astype(np.int32)

                # Different colors for different polygons
                color = plt.cm.tab10(poly_idx % 10)[:3]
                color = tuple(int(c * 255) for c in color)

                cv2.polylines(vis_img, [points_int], True, color, 2)

                # Add polygon index
                center = points_int.mean(axis=0).astype(int)
                cv2.putText(vis_img, str(poly_idx), tuple(center),
                           cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1)

    return vis_img


def save_model_outputs(predictions, output_dir, sample_id):
    """Save all model outputs for detailed analysis"""
    output_dir = Path(output_dir)
    output_dir.mkdir(exist_ok=True)

    sample_dir = output_dir / sample_id
    sample_dir.mkdir(exist_ok=True)

    # Save segmentation
    if "segmentation" in predictions:
        seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0].cpu().numpy()
        cv2.imwrite(str(sample_dir / "segmentation.png"), seg_pred * 50)

    # Save SDF
    if "sdf" in predictions:
        sdf_pred = predictions["sdf"][0, 0].cpu().numpy()
        sdf_normalized = ((sdf_pred + 1) * 127.5).astype(np.uint8)
        cv2.imwrite(str(sample_dir / "sdf.png"), sdf_normalized)

    # Save attributes
    if "attributes" in predictions:
```

```python
        attrs = predictions["attributes"][0].cpu().numpy()
        np.save(sample_dir / "attributes.npy", attrs)

    # Save polygons
    if "polygons" in predictions:
        polygons = predictions["polygons"][0].cpu().numpy()
        validity = predictions["polygon_validity"][0].cpu().numpy()

        np.save(sample_dir / "polygons.npy", polygons)
        np.save(sample_dir / "polygon_validity.npy", validity)

    # Save voxels
    if "voxels_pred" in predictions:
        voxels = torch.sigmoid(predictions["voxels_pred"][0]).cpu().numpy()
        np.save(sample_dir / "voxels.npy", voxels)


def create_comparison_grid(input_images, predictions, targets=None, num_samples=4):
    """Create a comparison grid showing inputs, predictions, and targets"""
    fig, axes = plt.subplots(num_samples, 4, figsize=(16, 4 * num_samples))

    for i in range(min(num_samples, len(input_images))):
        # Input image
        img = input_images[i].permute(1, 2, 0).cpu().numpy()
        axes[i, 0].imshow(img)
        axes[i, 0].set_title(f"Sample {i+1}: Input")
        axes[i, 0].axis('off')

        # Predicted segmentation
        seg_pred = torch.argmax(predictions["segmentation"][i], dim=0).cpu().numpy()
        axes[i, 1].imshow(seg_pred, cmap='tab10')
        axes[i, 1].set_title("Predicted Seg")
        axes[i, 1].axis('off')

        # Ground truth segmentation (if available)
        if targets and "mask" in targets:
            gt_mask = targets["mask"][i].cpu().numpy()
            axes[i, 2].imshow(gt_mask, cmap='tab10')
            axes[i, 2].set_title("GT Segmentation")
        else:
            axes[i, 2].text(0.5, 0.5, "No GT", ha='center', va='center',
                           transform=axes[i, 2].transAxes)
            axes[i, 2].set_title("GT Segmentation")
        axes[i, 2].axis('off')

        # Polygon overlay
        poly_vis = visualize_polygons(
            predictions["polygons"][i],
            predictions["polygon_validity"][i]
        )
        axes[i, 3].imshow(poly_vis)
        axes[i, 3].set_title("Predicted Polygons")
        axes[i, 3].axis('off')

    plt.tight_layout()
    return fig


def analyze_failure_cases(predictions, targets, threshold_iou=0.5):
    """Analyze failure cases for debugging"""
    failure_indices = []

    for i, (pred_seg, gt_mask) in enumerate(zip(predictions["segmentation"], targets["mask"])):
        seg_pred = torch.argmax(pred_seg, dim=0)
        iou = compute_iou(seg_pred, gt_mask)

        if iou < threshold_iou:
            failure_indices.append({
                "index": i,
                "iou": iou,
                "pred_classes": torch.unique(seg_pred).tolist(),
                "gt_classes": torch.unique(gt_mask).tolist()
            })
```

```python

    return failure_indices


class ProgressiveVisualization:
    """Track and visualize training progress"""

    def __init__(self, save_dir="./training_progress"):
        self.save_dir = Path(save_dir)
        self.save_dir.mkdir(exist_ok=True)

    def log_epoch_results(self, epoch, stage, predictions, targets, sample_image):
        """Log results for a specific epoch"""
        epoch_dir = self.save_dir / f"{stage}_epoch_{epoch}"
        epoch_dir.mkdir(exist_ok=True)

        # Save prediction visualization
        fig = plt.figure(figsize=(12, 8))
        visualize_predictions(sample_image, predictions, targets)
        plt.savefig(epoch_dir / "predictions.png", dpi=150, bbox_inches="tight")
        plt.close()

        # Save individual outputs
        save_model_outputs(predictions, epoch_dir, "sample")

    def create_training_animation(self, stage, metric_name="total_loss"):
        """Create animated GIF showing training progress"""
        # This would create an animation of training progress
        # Implementation depends on having saved epoch results
        pass


def compute_architectural_metrics(predictions, image_size=(256, 256)):
    """Compute architecture-specific metrics"""
    metrics = {}

    if "segmentation" in predictions:
        seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0]

        # Room count
        room_mask = (seg_pred == 0).cpu().numpy().astype(np.uint8)
        contours, _ = cv2.findContours(room_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        room_count = len([c for c in contours if cv2.contourArea(c) > 100])
        metrics["room_count"] = room_count

        # Wall connectivity
        wall_mask = (seg_pred == 1).cpu().numpy().astype(np.uint8)
        wall_components = cv2.connectedComponents(wall_mask)[0] - 1  # Subtract background
        metrics["wall_components"] = max(0, wall_components)

        # Door and window counts
        door_pixels = (seg_pred == 2).sum().item()
        window_pixels = (seg_pred == 3).sum().item()
        metrics["door_pixels"] = door_pixels
        metrics["window_pixels"] = window_pixels

    if "polygons" in predictions:
        validity = predictions["polygon_validity"][0]
        valid_polygons = (validity > 0.5).sum().item()
        metrics["valid_polygon_count"] = valid_polygons

        # Average polygon area
        polygons = predictions["polygons"][0]
        areas = []
        for poly_idx, (polygon, valid) in enumerate(zip(polygons, validity)):
            if valid > 0.5:
                # Compute polygon area using shoelace formula
                points = polygon.cpu().numpy() * np.array(image_size)
                valid_points = points[points.sum(axis=1) > 0]
                if len(valid_points) >= 3:
                    area = compute_polygon_area(valid_points)
                    areas.append(area)

```

```
        metrics["avg_polygon_area"] = np.mean(areas) if areas else 0.0

    return metrics


def compute_polygon_area(points):
    """Compute polygon area using shoelace formula"""
    if len(points) < 3:
        return 0.0

    x = points[:, 0]
    y = points[:, 1]

    # Shoelace formula
    area = 0.5 * abs(sum(x[i] * y[i+1] - x[i+1] * y[i] for i in range(-1, len(x)-1)))
    return area


def create_model_summary_report(model, sample_input, save_path="model_summary.txt"):
    """Create detailed model summary report"""
    with open(save_path, "w") as f:
        f.write("Neural-Geometric 3D Model Generator - Model Summary\n")
        f.write("=" * 60 + "\n\n")

        # Model architecture
        f.write("MODEL ARCHITECTURE:\n")
        f.write("-" * 20 + "\n")

        total_params = sum(p.numel() for p in model.parameters())
        trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

        f.write(f"Total parameters: {total_params:,}\n")
        f.write(f"Trainable parameters: {trainable_params:,}\n")
        f.write(f"Model size: {total_params * 4 / 1024 / 1024:.2f} MB\n\n")

        # Component breakdown
        f.write("COMPONENT PARAMETERS:\n")
        f.write("-" * 25 + "\n")

        encoder_params = sum(p.numel() for p in model.encoder.parameters())
        seg_params = sum(p.numel() for p in model.seg_head.parameters())
        attr_params = sum(p.numel() for p in model.attr_head.parameters())
        sdf_params = sum(p.numel() for p in model.sdf_head.parameters())
        dvx_params = sum(p.numel() for p in model.dvx.parameters())
        ext_params = sum(p.numel() for p in model.extrusion.parameters())

        f.write(f"Encoder: {encoder_params:,} ({encoder_params/total_params*100:.1f}%)\n")
        f.write(f"Segmentation Head: {seg_params:,} ({seg_params/total_params*100:.1f}%)\n")
        f.write(f"Attribute Head: {attr_params:,} ({attr_params/total_params*100:.1f}%)\n")
        f.write(f"SDF Head: {sdf_params:,} ({sdf_params/total_params*100:.1f}%)\n")
        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")

        # Forward pass analysis
        f.write("FORWARD PASS ANALYSIS:\n")
        f.write("-" * 25 + "\n")

        model.eval()
        with torch.no_grad():
            predictions = model(sample_input)

            for key, value in predictions.items():
                if torch.is_tensor(value):
                    f.write(f"{key}: {list(value.shape)} - {value.dtype}\n")
                else:
                    f.write(f"{key}: {type(value)}\n")

    print(f"Model summary saved to {save_path}")


def debug_gradient_flow(model, loss):
    """Debug gradient flow through the model"""
    print("Gradient Flow Analysis:")
```

```python
    print("-" * 30)

    total_norm = 0
    component_norms = {}

    for name, param in model.named_parameters():
        if param.grad is not None:
            param_norm = param.grad.norm().item()
            total_norm += param_norm ** 2

            # Group by component
            component = name.split('.')[0]
            if component not in component_norms:
                component_norms[component] = 0
            component_norms[component] += param_norm ** 2

    total_norm = total_norm ** 0.5

    print(f"Total gradient norm: {total_norm:.4f}")
    print("Component gradient norms:")

    for component, norm in component_norms.items():
        norm = norm ** 0.5
        print(f"  {component}: {norm:.4f} ({norm/total_norm*100:.1f}%)")


def create_3d_visualization(voxels, output_path="3d_preview.png"):
    """Create 3D visualization of voxel prediction"""
    try:
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D

        # Convert to binary
        if isinstance(voxels, torch.Tensor):
            voxels = voxels.cpu().numpy()

        binary_voxels = voxels > 0.5

        # Get occupied voxel coordinates
        occupied = np.where(binary_voxels)

        if len(occupied[0]) == 0:
            print("No occupied voxels to visualize")
            return

        # Create 3D plot
        fig = plt.figure(figsize=(10, 8))
        ax = fig.add_subplot(111, projection='3d')

        # Plot occupied voxels
        ax.scatter(occupied[0], occupied[1], occupied[2],
                   c=occupied[2], cmap='viridis', s=1, alpha=0.6)

        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        ax.set_title('3D Voxel Occupancy')

        plt.savefig(output_path, dpi=150, bbox_inches="tight")
        plt.close()

        print(f"3D visualization saved to {output_path}")

    except ImportError:
        print("3D visualization requires matplotlib with 3D support")
```
