# 140509\_31.md - 3D Model Generation and Customization Tool

## README

**Summary:** Develop an AI system that generates and customizes 3D models based on text descriptions, sketches, or reference images for various applications.

**Problem Statement:** 3D model creation requires specialized skills and significant time investment. Your task is to create an AI tool that generates 3D models from natural language descriptions, 2D sketches, or reference images. The system should enable model customization, provide different levels of detail, and export models in various formats for different use cases (gaming, architecture, manufacturing).

**Steps:** - Design text-to-3D and image-to-3D generation pipelines - Implement model customization tools for geometry, textures, and materials - Create level-of-detail generation for different application requirements - Build export capabilities for various 3D formats (OBJ, FBX, GLTF) - Develop quality assessment and optimization tools for generated models - Include integration with popular 3D software and game engines

**Suggested Data Requirements:** - 3D model datasets with text descriptions and metadata - Reference image collections with corresponding 3D models - Material and texture libraries - Application-specific model requirements and constraints

**Themes:** AI for creative, 3D modelling

## PRD (Product Requirements Document)

### Product Vision

Create an AI-powered 3D model generation and customization platform that democratizes 3D content creation by enabling users to generate professional-quality 3D models from natural language descriptions, sketches, or reference images without requiring specialized 3D modeling expertise.

### Target Users

* **Primary:** Game developers, indie studios, AR/VR content creators
* **Secondary:** Architects, product designers, educators, hobbyists
* **Tertiary:** Manufacturing companies, e-commerce platforms, marketing agencies

### Core Value Propositions

1. **Accessibility:** Enable non-3D artists to create professional models
2. **Speed:** Generate 3D models in minutes instead of hours/days
3. **Flexibility:** Support multiple input modalities (text, sketch, image)
4. **Customization:** Extensive model editing and refinement capabilities
5. **Industry Integration:** Seamless export to popular 3D software and engines

### Key Features

1. **Multi-Modal Input Processing:** Text descriptions, 2D sketches, reference images
2. **3D Model Generation:** Neural radiance fields, mesh generation, point clouds
3. **Advanced Customization:** Geometry editing, texture application, material assignment
4. **Level-of-Detail (LOD) Generation:** Optimized models for different use cases
5. **Format Export:** Support for OBJ, FBX, GLTF, STL, PLY formats
6. **Quality Assessment:** Automated model validation and optimization
7. **Software Integration:** Plugins for Blender, Maya, Unity, Unreal Engine

### Success Metrics

* Model generation accuracy: >85% user satisfaction
* Generation time: <5 minutes for simple models, <20 minutes for complex
* Export success rate: >98% across all supported formats
* User retention: >60% monthly active users
* Integration adoption: 50+ software partnerships

## FRD (Functional Requirements Document)

### Core Functional Requirements

#### F1: Multi-Modal Input Processing

* **F1.1:** Process natural language text descriptions for 3D model generation
* **F1.2:** Accept and interpret 2D sketches and line drawings
* **F1.3:** Analyze reference images for 3D reconstruction
* **F1.4:** Support multi-view image input for enhanced accuracy
* **F1.5:** Enable hybrid input combinations (text + image, sketch + description)

#### F2: 3D Model Generation Pipeline

* **F2.1:** Generate base 3D geometry from processed inputs
* **F2.2:** Create detailed mesh structures with proper topology
* **F2.3:** Apply procedural texturing and material assignment
* **F2.4:** Generate multiple model variations for user selection
* **F2.5:** Support style transfer between different 3D models

#### F3: Model Customization Tools

* **F3.1:** Real-time geometry editing (vertex manipulation, scaling, rotation)
* **F3.2:** Texture customization and UV mapping adjustment
* **F3.3:** Material property modification (metallic, roughness, transparency)
* **F3.4:** Color palette application and customization
* **F3.5:** Anatomical and structural constraint enforcement

#### F4: Level-of-Detail Generation

* **F4.1:** Generate high-poly models for rendering and visualization
* **F4.2:** Create optimized low-poly versions for real-time applications
* **F4.3:** Automatic LOD chain generation with multiple detail levels
* **F4.4:** Performance-based optimization for target platforms
* **F4.5:** Quality-preserving mesh decimation algorithms

#### F5: Export and Integration Capabilities

* **F5.1:** Export to standard 3D formats (OBJ, FBX, GLTF, STL, PLY)
* **F5.2:** Include material and texture data in exports
* **F5.3:** Generate format-specific optimizations
* **F5.4:** Plugin integration for popular 3D software
* **F5.5:** Direct upload to 3D asset marketplaces

#### F6: Quality Assessment and Optimization

* **F6.1:** Validate mesh topology and identify issues
* **F6.2:** Check UV mapping quality and texture resolution
* **F6.3:** Assess geometric accuracy against reference inputs
* **F6.4:** Optimize polygon count while preserving detail
* **F6.5:** Generate quality reports and improvement suggestions

#### F7: Application-Specific Features

* **F7.1:** Gaming-optimized models with proper LOD chains
* **F7.2:** Architecture visualization with accurate proportions
* **F7.3:** Manufacturing-ready models with precise dimensions
* **F7.4:** 3D printing preparation and validation
* **F7.5:** AR/VR optimization with performance constraints

## NFRD (Non-Functional Requirements Document)

### Performance Requirements

* **NFR-P1:** 3D model generation time: <5 minutes for simple models, <20 minutes for complex
* **NFR-P2:** Real-time preview updates during customization: <2 seconds
* **NFR-P3:** Export processing time: <30 seconds for standard formats
* **NFR-P4:** System response time: <1 second for UI interactions
* **NFR-P5:** Concurrent model generation support: 1000+ simultaneous users

### Scalability Requirements

* **NFR-S1:** GPU cluster scaling for compute-intensive 3D generation
* **NFR-S2:** Auto-scaling based on generation queue length
* **NFR-S3:** Distributed processing across multiple GPU nodes
* **NFR-S4:** Storage scaling for large 3D asset libraries
* **NFR-S5:** CDN optimization for 3D model streaming and preview

### Quality Requirements

* **NFR-Q1:** Generated model accuracy: >85% similarity to reference
* **NFR-Q2:** Mesh topology quality: Manifold meshes with proper edge flow
* **NFR-Q3:** Texture resolution: Support up to 4K textures
* **NFR-Q4:** Geometric precision: Sub-millimeter accuracy for manufacturing
* **NFR-Q5:** LOD quality preservation: <10% visual difference between levels

### Compatibility Requirements

* **NFR-C1:** Support for major 3D software (Blender, Maya, 3ds Max, Cinema 4D)
* **NFR-C2:** Game engine compatibility (Unity, Unreal Engine, Godot)
* **NFR-C3:** Web browser 3D viewing (WebGL, Three.js)
* **NFR-C4:** Mobile device optimization for AR applications
* **NFR-C5:** Industry-standard format compliance

### Security Requirements

* **NFR-SE1:** Secure model storage with encryption at rest
* **NFR-SE2:** User authentication and authorization for model access
* **NFR-SE3:** Intellectual property protection for generated models
* **NFR-SE4:** API security for third-party integrations
* **NFR-SE5:** Regular security audits for data protection compliance

### Usability Requirements

* **NFR-U1:** Intuitive 3D viewport with standard navigation controls
* **NFR-U2:** Mobile-responsive interface for tablets
* **NFR-U3:** Accessibility features for users with disabilities
* **NFR-U4:** Multi-language support for global markets
* **NFR-U5:** Context-sensitive help and tutorials

## AD (Architecture Diagram)

graph TB  
 subgraph "Client Layer"  
 WEB[Web Application]  
 MOBILE[Mobile App]  
 PLUGINS[3D Software Plugins]  
 API\_CLIENTS[API Clients]  
 end  
   
 subgraph "Load Balancer & CDN"  
 LB[Load Balancer]  
 CDN[Content Delivery Network]  
 end  
   
 subgraph "API Gateway"  
 GATEWAY[API Gateway]  
 AUTH[Authentication Service]  
 RATE\_LIMIT[Rate Limiter]  
 end  
   
 subgraph "Core Services"  
 INPUT\_PROC[Input Processing Service]  
 MODEL\_GEN[Model Generation Service]  
 CUSTOM[Customization Service]  
 EXPORT[Export Service]  
 QUALITY[Quality Assessment Service]  
 end  
   
 subgraph "AI/ML Pipeline"  
 TEXT\_PROC[Text Processing (CLIP/BERT)]  
 IMG\_PROC[Image Processing (CNN)]  
 SKETCH\_PROC[Sketch Processing]  
 NERF[Neural Radiance Fields]  
 MESH\_GEN[Mesh Generation (DMTet)]  
 TEXTURE\_GEN[Texture Generation (GAN)]  
 LOD\_GEN[LOD Generator]  
 end  
   
 subgraph "GPU Compute Cluster"  
 GPU\_SCHED[GPU Scheduler]  
 GPU\_NODE1[GPU Node 1]  
 GPU\_NODE2[GPU Node 2]  
 GPU\_NODEN[GPU Node N]  
 end  
   
 subgraph "Data Layer"  
 POSTGRES[PostgreSQL - Metadata]  
 MONGO[MongoDB - Model Data]  
 REDIS[Redis - Cache]  
 S3[Object Storage - 3D Assets]  
 ELASTIC[Elasticsearch - Search]  
 end  
   
 subgraph "External Services"  
 TEXTURE\_DB[Texture Libraries]  
 MATERIAL\_DB[Material Databases]  
 REFERENCE\_DB[Reference Model DB]  
 MARKETPLACE[3D Marketplaces]  
 end  
   
 WEB --> LB  
 MOBILE --> LB  
 PLUGINS --> LB  
 API\_CLIENTS --> LB  
   
 LB --> GATEWAY  
 GATEWAY --> AUTH  
 GATEWAY --> RATE\_LIMIT  
   
 GATEWAY --> INPUT\_PROC  
 GATEWAY --> MODEL\_GEN  
 GATEWAY --> CUSTOM  
 GATEWAY --> EXPORT  
 GATEWAY --> QUALITY  
   
 INPUT\_PROC --> TEXT\_PROC  
 INPUT\_PROC --> IMG\_PROC  
 INPUT\_PROC --> SKETCH\_PROC  
   
 MODEL\_GEN --> NERF  
 MODEL\_GEN --> MESH\_GEN  
 MODEL\_GEN --> TEXTURE\_GEN  
 MODEL\_GEN --> LOD\_GEN  
   
 MODEL\_GEN --> GPU\_SCHED  
 GPU\_SCHED --> GPU\_NODE1  
 GPU\_SCHED --> GPU\_NODE2  
 GPU\_SCHED --> GPU\_NODEN  
   
 INPUT\_PROC --> POSTGRES  
 MODEL\_GEN --> MONGO  
 CUSTOM --> REDIS  
 EXPORT --> S3  
 QUALITY --> ELASTIC  
   
 MODEL\_GEN --> TEXTURE\_DB  
 MODEL\_GEN --> MATERIAL\_DB  
 QUALITY --> REFERENCE\_DB  
 EXPORT --> MARKETPLACE  
   
 CDN --> S3

## HLD (High Level Design)

### System Architecture Overview

The 3D Model Generation and Customization Tool employs a distributed architecture optimized for GPU-intensive AI workloads with real-time user interactions.

#### 1. Client Layer Architecture

* **Web Application:** React-based 3D viewport using Three.js/Babylon.js
* **Mobile Application:** React Native with WebGL integration for 3D preview
* **Plugin System:** Native plugins for Blender, Maya, Unity, Unreal Engine
* **API Clients:** RESTful and WebSocket APIs for third-party integrations

#### 2. Input Processing Pipeline

##### Multi-Modal Input Handler

* **Text Processing:** CLIP/BERT models for semantic understanding
* **Image Analysis:** CNN-based feature extraction and depth estimation
* **Sketch Interpretation:** Specialized neural networks for line drawing analysis
* **Multi-View Reconstruction:** Photogrammetry and stereo vision algorithms

#### 3. Core AI/ML Pipeline

##### Neural 3D Generation

class Neural3DGenerator:  
 def \_\_init\_\_(self):  
 self.nerf\_model = NeRFModel() # Volume rendering  
 self.dmtet\_model = DMTetModel() # Mesh extraction  
 self.texture\_generator = TextureGAN() # Texture synthesis  
 self.style\_transfer = Style3DNet() # Style application

##### Model Generation Workflow

1. **Input Encoding:** Convert text/image inputs to latent representations
2. **Volume Generation:** Use NeRF for initial 3D volume representation
3. **Mesh Extraction:** Convert volume to mesh using differentiable marching cubes
4. **Texture Synthesis:** Generate and apply textures using GANs
5. **Quality Optimization:** Refine topology and UV mapping

#### 4. GPU Compute Architecture

##### Distributed GPU Processing

* **GPU Scheduler:** Intelligent workload distribution across GPU clusters
* **Model Parallelism:** Large models split across multiple GPUs
* **Batch Processing:** Efficient batching of similar generation tasks
* **Resource Pooling:** Dynamic GPU allocation based on demand

#### 5. Data Management Strategy

##### Storage Architecture

* **PostgreSQL:** User data, project metadata, generation parameters
* **MongoDB:** 3D model geometry data, mesh structures, materials
* **Object Storage:** Large 3D assets, textures, exported models
* **Redis:** Real-time collaboration state, preview cache
* **Elasticsearch:** Model search, similarity matching

### Key Technical Components

#### 1. NeRF-Based Volume Rendering

class NeRFVolumeRenderer:  
 def \_\_init\_\_(self, input\_encoding, density\_network, color\_network):  
 self.encoding = input\_encoding  
 self.density\_net = density\_network  
 self.color\_net = color\_network  
   
 def render\_volume(self, rays, viewpoints):  
 # Sample points along rays  
 points = self.sample\_points\_on\_rays(rays)  
   
 # Encode 3D points  
 encoded\_points = self.encoding.encode(points)  
   
 # Predict density and color  
 density = self.density\_net(encoded\_points)  
 colors = self.color\_net(encoded\_points, viewpoints)  
   
 # Volume rendering equation  
 rendered\_image = self.volume\_integrate(density, colors, rays)  
 return rendered\_image

#### 2. Mesh Generation and Optimization

class MeshGenerator:  
 def \_\_init\_\_(self):  
 self.marching\_cubes = DifferentiableMarchingCubes()  
 self.mesh\_optimizer = MeshOptimizer()  
   
 def extract\_mesh(self, volume\_representation):  
 # Extract initial mesh  
 vertices, faces = self.marching\_cubes.extract(volume\_representation)  
   
 # Optimize topology  
 optimized\_mesh = self.mesh\_optimizer.optimize(vertices, faces)  
   
 # Generate UV coordinates  
 uv\_coords = self.generate\_uv\_mapping(optimized\_mesh)  
   
 return Mesh(  
 vertices=optimized\_mesh.vertices,  
 faces=optimized\_mesh.faces,  
 uv\_coordinates=uv\_coords  
 )

#### 3. Level-of-Detail Generation

class LODGenerator:  
 def \_\_init\_\_(self):  
 self.decimator = QuadricErrorMetrics()  
 self.quality\_assessor = MeshQualityAssessor()  
   
 def generate\_lod\_chain(self, base\_mesh, target\_counts):  
 lod\_chain = [base\_mesh]  
   
 for target\_count in target\_counts:  
 # Decimate mesh while preserving important features  
 decimated = self.decimator.decimate(  
 base\_mesh, target\_poly\_count=target\_count  
 )  
   
 # Assess quality and adjust if needed  
 quality\_score = self.quality\_assessor.assess(decimated, base\_mesh)  
 if quality\_score < 0.8:  
 decimated = self.improve\_decimation(decimated, base\_mesh)  
   
 lod\_chain.append(decimated)  
   
 return lod\_chain

### Real-Time Collaboration Architecture

#### WebSocket-Based Updates

* **Concurrent Editing:** Multiple users editing the same 3D model
* **Change Broadcasting:** Real-time synchronization of model modifications
* **Conflict Resolution:** Intelligent merging of simultaneous edits
* **Version Control:** Git-like versioning for 3D models

#### Performance Optimization Strategies

##### Client-Side Optimization

* **Progressive Loading:** Stream 3D models progressively
* **Level-of-Detail Streaming:** Load appropriate detail levels based on view
* **Frustum Culling:** Only render visible model parts
* **Texture Compression:** Efficient texture formats for web delivery

##### Server-Side Optimization

* **Model Caching:** Cache frequently accessed models
* **Predictive Pre-generation:** Anticipate user needs and pre-generate variants
* **Compression:** Efficient 3D model compression algorithms
* **Edge Computing:** Distribute processing closer to users

## LLD (Low Level Design)

### Detailed Component Implementation

#### 1. Input Processing Service

##### Text-to-3D Processing

class TextTo3DProcessor:  
 def \_\_init\_\_(self):  
 self.text\_encoder = CLIPTextEncoder()  
 self.shape\_decoder = Shape3DDecoder()  
 self.category\_classifier = ObjectCategoryClassifier()  
   
 async def process\_text\_input(self, text\_description: str) -> ProcessedInput:  
 # Clean and normalize text  
 cleaned\_text = self.preprocess\_text(text\_description)  
   
 # Extract semantic features  
 text\_features = self.text\_encoder.encode(cleaned\_text)  
   
 # Classify object category  
 category = self.category\_classifier.predict(text\_features)  
   
 # Generate shape parameters  
 shape\_params = self.shape\_decoder.decode(text\_features, category)  
   
 return ProcessedInput(  
 semantic\_features=text\_features,  
 category=category,  
 shape\_parameters=shape\_params,  
 input\_type="text"  
 )  
   
 def preprocess\_text(self, text: str) -> str:  
 # Remove special characters, normalize case  
 cleaned = re.sub(r'[^\w\s]', '', text.lower())  
   
 # Handle common 3D modeling terminology  
 terminology\_map = {  
 'make': 'create',  
 'build': 'create',  
 'design': 'create',  
 # ... more mappings  
 }  
   
 for old\_term, new\_term in terminology\_map.items():  
 cleaned = cleaned.replace(old\_term, new\_term)  
   
 return cleaned

##### Image-to-3D Processing

class ImageTo3DProcessor:  
 def \_\_init\_\_(self):  
 self.depth\_estimator = MiDaSDepthEstimator()  
 self.normal\_estimator = SurfaceNormalEstimator()  
 self.silhouette\_extractor = SilhouetteExtractor()  
 self.shape\_from\_shading = ShapeFromShading()  
   
 async def process\_image\_input(self, image\_data: np.ndarray) -> ProcessedInput:  
 # Estimate depth map  
 depth\_map = self.depth\_estimator.estimate\_depth(image\_data)  
   
 # Extract surface normals  
 normal\_map = self.normal\_estimator.estimate\_normals(image\_data)  
   
 # Extract object silhouette  
 silhouette = self.silhouette\_extractor.extract(image\_data)  
   
 # Combine cues for 3D shape inference  
 shape\_cues = self.combine\_shape\_cues(depth\_map, normal\_map, silhouette)  
   
 # Generate initial 3D representation  
 voxel\_grid = self.shape\_from\_shading.reconstruct(shape\_cues)  
   
 return ProcessedInput(  
 depth\_map=depth\_map,  
 normal\_map=normal\_map,  
 silhouette=silhouette,  
 voxel\_representation=voxel\_grid,  
 input\_type="image"  
 )

#### 2. Neural 3D Generation Engine

##### NeRF-Based Generation

class NeRF3DGenerator:  
 def \_\_init\_\_(self):  
 self.positional\_encoding = PositionalEncoding(num\_freqs=10)  
 self.density\_network = DensityMLP(hidden\_dims=[256, 256, 256])  
 self.color\_network = ColorMLP(hidden\_dims=[128, 128, 128])  
 self.volume\_renderer = VolumeRenderer()  
   
 async def generate\_from\_text(self, text\_features: torch.Tensor) -> NeRFModel:  
 # Initialize NeRF parameters from text features  
 initial\_params = self.initialize\_nerf\_from\_text(text\_features)  
   
 # Optimize NeRF through iterative refinement  
 optimized\_nerf = await self.optimize\_nerf(initial\_params)  
   
 return optimized\_nerf  
   
 def initialize\_nerf\_from\_text(self, text\_features: torch.Tensor) -> dict:  
 # Use text features to initialize NeRF network weights  
 init\_weights = self.text\_to\_nerf\_mapper(text\_features)  
   
 return {  
 'density\_weights': init\_weights['density'],  
 'color\_weights': init\_weights['color'],  
 'scene\_bounds': self.estimate\_scene\_bounds(text\_features)  
 }  
   
 async def optimize\_nerf(self, initial\_params: dict) -> NeRFModel:  
 nerf\_model = NeRFModel(initial\_params)  
 optimizer = torch.optim.Adam(nerf\_model.parameters(), lr=1e-4)  
   
 for iteration in range(1000): # Optimization iterations  
 # Sample random rays  
 rays = self.sample\_random\_rays(batch\_size=1024)  
   
 # Render using current NeRF  
 rendered\_colors = nerf\_model.render(rays)  
   
 # Compute loss (using priors and consistency constraints)  
 loss = self.compute\_loss(rendered\_colors, rays)  
   
 # Optimize  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
   
 if iteration % 100 == 0:  
 print(f"Iteration {iteration}, Loss: {loss.item()}")  
   
 return nerf\_model

##### Mesh Extraction and Refinement

class MeshExtractor:  
 def \_\_init\_\_(self):  
 self.marching\_cubes = MarchingCubes(resolution=128)  
 self.mesh\_simplifier = QuadricErrorSimplifier()  
 self.uv\_unwrapper = AngleBasedUVUnwrapper()  
   
 def extract\_mesh\_from\_nerf(self, nerf\_model: NeRFModel) -> Mesh:  
 # Sample density values on a 3D grid  
 grid\_points = self.generate\_grid\_points(resolution=128)  
 density\_values = nerf\_model.query\_density(grid\_points)  
   
 # Extract mesh using marching cubes  
 vertices, faces = self.marching\_cubes.extract(  
 density\_values.reshape(128, 128, 128)  
 )  
   
 # Refine mesh topology  
 refined\_vertices, refined\_faces = self.refine\_mesh\_topology(  
 vertices, faces  
 )  
   
 # Generate UV coordinates  
 uv\_coordinates = self.uv\_unwrapper.unwrap(refined\_vertices, refined\_faces)  
   
 # Create mesh object  
 mesh = Mesh(  
 vertices=refined\_vertices,  
 faces=refined\_faces,  
 uv\_coordinates=uv\_coordinates  
 )  
   
 return mesh  
   
 def refine\_mesh\_topology(self, vertices: np.ndarray, faces: np.ndarray):  
 # Remove duplicate vertices  
 unique\_vertices, vertex\_map = np.unique(vertices, axis=0, return\_inverse=True)  
 updated\_faces = vertex\_map[faces]  
   
 # Remove degenerate faces  
 valid\_faces = self.filter\_degenerate\_faces(updated\_faces)  
   
 # Smooth mesh if needed  
 if self.needs\_smoothing(unique\_vertices, valid\_faces):  
 smoothed\_vertices = self.laplacian\_smooth(unique\_vertices, valid\_faces)  
 return smoothed\_vertices, valid\_faces  
   
 return unique\_vertices, valid\_faces

#### 3. Texture Generation System

##### Procedural Texture Generator

class TextureGenerator:  
 def \_\_init\_\_(self):  
 self.texture\_gan = TextureGAN(latent\_dim=512)  
 self.style\_transfer = NeuralStyleTransfer()  
 self.material\_mapper = MaterialPropertyMapper()  
   
 async def generate\_texture(self, mesh: Mesh, style\_description: str) -> TextureSet:  
 # Encode style description  
 style\_features = self.encode\_style\_description(style\_description)  
   
 # Generate base texture using GAN  
 base\_texture = self.texture\_gan.generate(style\_features)  
   
 # Create material maps (normal, roughness, metallic)  
 material\_maps = self.generate\_material\_maps(base\_texture, style\_features)  
   
 # Project texture onto mesh UV coordinates  
 projected\_texture = self.project\_texture\_to\_mesh(  
 base\_texture, mesh.uv\_coordinates  
 )  
   
 return TextureSet(  
 albedo\_map=projected\_texture,  
 normal\_map=material\_maps['normal'],  
 roughness\_map=material\_maps['roughness'],  
 metallic\_map=material\_maps['metallic']  
 )  
   
 def generate\_material\_maps(self, base\_texture: np.ndarray, style\_features: torch.Tensor):  
 # Generate normal map from height information  
 normal\_map = self.generate\_normal\_map(base\_texture)  
   
 # Predict material properties from style  
 material\_properties = self.material\_mapper.predict(style\_features)  
   
 # Generate roughness and metallic maps  
 roughness\_map = self.generate\_roughness\_map(  
 base\_texture, material\_properties['roughness']  
 )  
 metallic\_map = self.generate\_metallic\_map(  
 base\_texture, material\_properties['metallic']  
 )  
   
 return {  
 'normal': normal\_map,  
 'roughness': roughness\_map,  
 'metallic': metallic\_map  
 }

#### 4. Level-of-Detail System

##### Adaptive LOD Generator

class AdaptiveLODGenerator:  
 def \_\_init\_\_(self):  
 self.mesh\_decimator = QEMDecimator()  
 self.quality\_metrics = MeshQualityMetrics()  
 self.performance\_predictor = PerformancePredictor()  
   
 def generate\_adaptive\_lod(self, base\_mesh: Mesh, target\_platforms: List[str]) -> LODChain:  
 lod\_chain = LODChain()  
   
 for platform in target\_platforms:  
 # Determine optimal poly count for platform  
 target\_polycount = self.performance\_predictor.optimal\_polycount(platform)  
   
 # Generate LOD level  
 lod\_mesh = self.generate\_lod\_level(base\_mesh, target\_polycount)  
   
 # Validate quality  
 quality\_score = self.quality\_metrics.assess(lod\_mesh, base\_mesh)  
   
 if quality\_score < 0.7: # Quality threshold  
 lod\_mesh = self.improve\_lod\_quality(lod\_mesh, base\_mesh)  
   
 lod\_chain.add\_level(platform, lod\_mesh)  
   
 return lod\_chain  
   
 def generate\_lod\_level(self, base\_mesh: Mesh, target\_polycount: int) -> Mesh:  
 # Calculate decimation ratio  
 current\_polycount = len(base\_mesh.faces)  
 decimation\_ratio = target\_polycount / current\_polycount  
   
 if decimation\_ratio >= 1.0:  
 return base\_mesh # No decimation needed  
   
 # Perform quadric error metric decimation  
 decimated\_mesh = self.mesh\_decimator.decimate(  
 base\_mesh, target\_ratio=decimation\_ratio  
 )  
   
 # Preserve UV coordinates through decimation  
 preserved\_uvs = self.preserve\_uv\_coordinates(  
 base\_mesh, decimated\_mesh  
 )  
 decimated\_mesh.uv\_coordinates = preserved\_uvs  
   
 return decimated\_mesh

### Database Schema Implementation

#### PostgreSQL Schema

-- Users and projects  
CREATE TABLE users (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 username VARCHAR(100) UNIQUE NOT NULL,  
 email VARCHAR(255) UNIQUE NOT NULL,  
 subscription\_tier VARCHAR(50) DEFAULT 'free',  
 gpu\_credits\_remaining INTEGER DEFAULT 100,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
CREATE TABLE projects (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 user\_id UUID REFERENCES users(id) ON DELETE CASCADE,  
 name VARCHAR(255) NOT NULL,  
 description TEXT,  
 project\_type VARCHAR(50) NOT NULL, -- 'gaming', 'architecture', 'manufacturing'  
 status VARCHAR(50) DEFAULT 'active',  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- 3D model generations  
CREATE TABLE model\_generations (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 project\_id UUID REFERENCES projects(id) ON DELETE CASCADE,  
 input\_type VARCHAR(50) NOT NULL, -- 'text', 'image', 'sketch'  
 input\_data JSONB NOT NULL,  
 generation\_parameters JSONB,  
 status VARCHAR(50) DEFAULT 'pending', -- 'pending', 'processing', 'completed', 'failed'  
 gpu\_time\_used INTEGER DEFAULT 0, -- in seconds  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 completed\_at TIMESTAMP  
);  
  
-- Generated 3D models  
CREATE TABLE generated\_models (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 generation\_id UUID REFERENCES model\_generations(id) ON DELETE CASCADE,  
 model\_name VARCHAR(255) NOT NULL,  
 file\_path TEXT NOT NULL,  
 file\_size\_bytes BIGINT,  
 polygon\_count INTEGER,  
 vertex\_count INTEGER,  
 texture\_resolution VARCHAR(20),  
 quality\_score FLOAT,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- LOD chains  
CREATE TABLE lod\_chains (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 base\_model\_id UUID REFERENCES generated\_models(id) ON DELETE CASCADE,  
 platform\_target VARCHAR(100) NOT NULL,  
 lod\_level INTEGER NOT NULL,  
 model\_file\_path TEXT NOT NULL,  
 polygon\_count INTEGER,  
 file\_size\_bytes BIGINT,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Export history  
CREATE TABLE model\_exports (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 model\_id UUID REFERENCES generated\_models(id) ON DELETE CASCADE,  
 export\_format VARCHAR(10) NOT NULL,  
 export\_settings JSONB,  
 file\_path TEXT NOT NULL,  
 exported\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

#### MongoDB Collections

```javascript // Model geometry data { "\_id“: ObjectId,”model\_id“: String, // References generated\_models.id”geometry\_data“: {”vertices“: [Number], // Flattened array of vertex coordinates”faces“: [Number], // Face indices”normals“: [Number], // Vertex normals”uv\_coordinates“: [Number] // UV mapping coordinates },”material\_data“: {”textures“: {”albedo“: String, // File path to albedo texture”normal“: String, // File path to normal map”roughness“: String, // File path to roughness map”metallic“: String // File path to metallic map },”material\_properties“: {”base\_color“: [Number], // RGB values”metallic\_factor“: Number,”roughness\_factor“: Number,”transparency": Number