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 { "_idâ€œ: ObjectId,â€model_idâ€œ: String, // References generated_models.idâ€geometry_dataâ€œ: {â€verticesâ€œ: [Number], // Flattened array of vertex coordinatesâ€facesâ€œ: [Number], // Face indicesâ€normalsâ€œ: [Number], // Vertex normalsâ€uv_coordinatesâ€œ: [Number] // UV mapping coordinates },â€material_dataâ€œ: {â€texturesâ€œ: {â€albedoâ€œ: String, // File path to albedo textureâ€normalâ€œ: String, // File path to normal mapâ€roughnessâ€œ: String, // File path to roughness mapâ€metallicâ€œ: String // File path to metallic map },â€material_propertiesâ€œ: {â€base_colorâ€œ: [Number], // RGB valuesâ€metallic_factorâ€œ: Number,â€roughness_factorâ€œ: Number,â€transparency": Number
```