Technological Architecture of Digitized Fighting Games: From Historical Arcade Hardware to Generative AI Asset Pipelines

The technological trajectory of digitized fighting games represents a critical intersection in computer graphics history, where physical performance capture first collided with hardware-constrained real-time rasterization. Emerging in the late 1980s and reaching peak commercial relevance during the mid-1990s, digitized fighting games abandoned traditional hand-drawn cel illustration in favor of frame-captured live-action footage of martial artists and actors. This technique presented distinct computational engineering challenges regarding video memory bandwidth, palette quantization, frame decimation, and sprite compression.
In contemporary software engineering, the resurgence of digitized visual styles is supported by modern real-time graphics engines, including Godot Engine, Unreal Engine 5, and Three.js. Furthermore, the integration of generative artificial intelligence pipelines—utilizing foundational image architectures such as FLUX, vision-language editing models like Qwen-Image-Edit, and 3D neural reconstruction frameworks such as TRELLIS and Hunyuan3D-2.1—has transformed digitized asset production. The labor-intensive video capture and manual blue-screen isolation workflows of the 1990s are now replaced by automated, promptable, and pose-guided neural generation systems capable of outputting production-ready 2D sprite sheets, 2.5D normal maps, and fully textured 3D meshes.

Historical Genesis and Evolution of Digitized Fighting Games

Pre-Mortal Kombat Foundations

Long before digitized graphics became synonymous with martial arts fighting games, early arcade experimentation established the viability of embedding photographic imagery into real-time interactive software. Midway's *Journey* (1983) served as an early prototype, deploying digitized black-and-white photographs of the rock band members mounted atop animated pixel bodies[1]. Williams Electronics and Midway subsequently expanded this technique with *NARC* (1988), engineered around a custom 32-bit graphics processor architecture capable of rendering dense, high-color digitized sprites depicting law enforcement officers, weapons, and particle dismemberment[1].
In 1990, Atari Games released *Pit-Fighter*, establishing the arcade framework for digitized hand-to-hand brawlers[1]. Designed to evoke an underground street-fighting environment, *Pit-Fighter* featured three playable fighters—Buzz, a heavy-set professional wrestler; Ty, a kickboxing champion; and Kato, a third-degree black belt[1]. The arcade hardware utilized

custom dynamic sprite-scaling hardware that altered sprite scales in real time as characters navigated the depth axis of an eight-directional arena[1].
Despite its initial commercial success, *Pit-Fighter* exhibited structural design flaws, including imprecise collision detection, minimal move sets, and visual artifacting when ported to 16-bit home consoles like the Sega Genesis by Tengen[3]. On these home platforms, severe cartridge ROM constraints forced dramatic reductions in sprite size, color depth, and frame counts, transforming the crowd and fighters into heavily pixelated visuals[3].

Game Title	Release Year	Developer / Publisher	Target Arcade / Console Hardware	Graphical & Mechanical Innovations
NARC	1988	Williams / Midway	Arcade (Custom TMS34010 Board)	First high-color sprite digitizing pipeline; real-time limb dismemberment[1].
Pit-Fighter	1990	Atari Games / Tengen	Arcade / Sega Genesis / SNES	Hardware-based z-axis dynamic sprite scaling; interactive crowds[1].
Mortal Kombat	1992	Midway Games	Arcade (Midway Y-Unit / TMS34010)	Five-button layout; block button; airborne juggling; Fatalities[5].
Kasumi Ninja	1994	Hand Made Software / Atari	Atari Jaguar (64-Bit Console)	High-resolution digitized graphics; deep zoom camera scaling; high gore[7].
Way of the	1994	Naughty Dog /	3DO	High

Warrior		Universal	Interactive Multiplayer	frame-rate scaling digitized sprites; Redbook CD audio soundtrack[7].
Bikini Karate Babes	2002	Creative Edge Studios	Microsoft Windows PC	Full-Motion Video (FMV) sprite extraction; 19 live-action female fighters[8].
Dong Dong Never Die	2009	Peking Duck Club (Doujin)	PC / Custom Fighter Engine	Doujin community project; full frame capture of real-world actors[9].

The Mortal Kombat Technical Paradigm

In 1992, Midway released *Mortal Kombat*, designed and programmed by a small core team consisting of Ed Boon, John Tobias, Rich Divizio, and Daniel Pesina[6]. Originally conceived as an action title starring martial artist and actor Jean-Claude Van Damme, the licensing agreement fell through, prompting the team to create an original martial arts fantasy universe[6].

Midway engineered *Mortal Kombat* on the Y-Unit arcade hardware platform, centered on a Texas Instruments TMS34010 graphics processor running at 6.25 MHz[6]. The Y-Unit system rendered an arcade screen resolution of 400×254 pixels, managing hardware-accelerated sprite planes and multi-layered parallax scrolling backgrounds[6]. Visual assets were produced by filming martial artists performing routines on Hi8 video cameras, with keyframes subsequently ingested into digital art tools[6].

Mechanically, *Mortal Kombat* diverged sharply from contemporary Japanese 2D fighting games like *Street Fighter II*[2]:

- **Dedicated Block Button**: Rather than requiring players to hold the directional joystick away from an opponent, *Mortal Kombat* introduced a discrete block button[5]. This eliminated cross-up mix-ups and allowed players to hold ground while absorbing minor chip damage[5].
- **Normalized Base Mechanics**: While Japanese fighting games gave every character

unique normal attacks, *Mortal Kombat* assigned identical basic moves—sweeps, uppercuts, high/low punches, and high/low kicks—to all characters, ensuring uniform frame timing across the roster[5].

- **Airborne Juggling**: The game introduced airborne combo chaining, allowing players to strike an airborne opponent with uppercuts or special attacks before they touched the canvas[5].
- **Fatalities and Impact**: The inclusion of explicit finishing moves executed after a match, rendered with photorealistic digitized blood spray, generated substantial public attention and served as a primary catalyst for the formation of the Entertainment Software Rating Board (ESRB)[5].

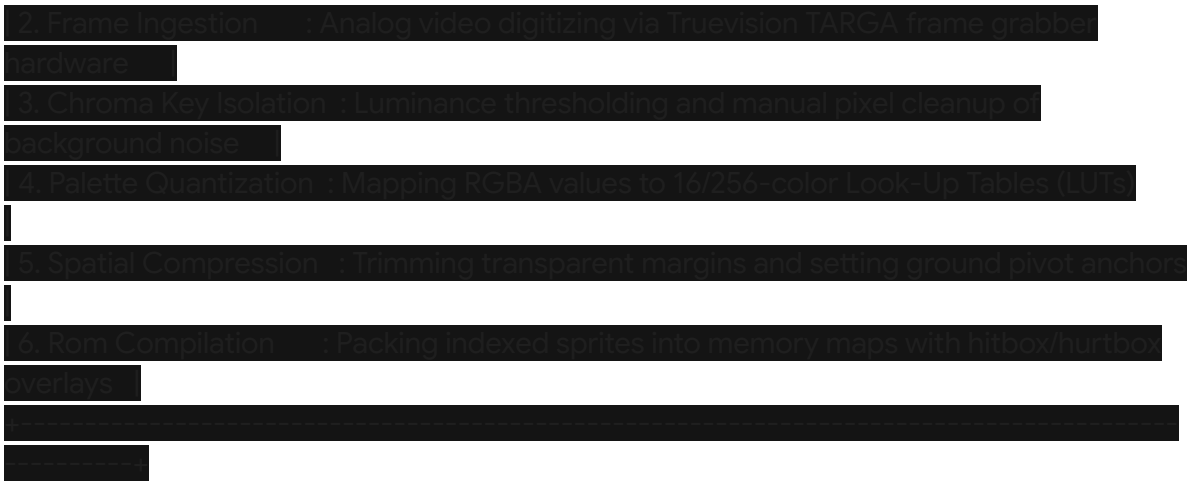
Wave of Imitators and Cult Niche Titles

Following *Mortal Kombat*'s commercial impact, rival studios attempted to capitalize on digitized aesthetics. Projects like *Kasumi Ninja* on the Atari Jaguar, *Way of the Warrior* on the 3DO, and the unreleased arcade title *Tattoo Assassins* expanded character palettes and camera zoom functions, though many suffered from input latency and poor collision alignment[7].

In 2002, Creative Edge Studios released *Bikini Karate Babes*, followed by its sequel *Babes with Blades*, utilizing full-motion video (FMV) capture of 19 female actors[8]. The engine extracted sprite frames directly from digital video sequences, applying digital post-processing to ensure uniform lighting across characters[8]. Similarly, the 2009 Chinese doujin title *Dong Dong Never Die* demonstrated that digitized sprite work could maintain precise competitive hitboxes when integrated into flexible 2D engines like MUGEN[9].

Technical Mechanics of Historical Digitization Pipelines

The production methodology for 1990s digitized fighting games required rigid constraints to convert analog video into digital sprite sheets compatible with restrictive arcade ROM cartridge limits.



Video Ingestion and Chroma Keying

Actors executed martial arts techniques on stage floors painted with neutral blue or green backdrops. Performances were recorded onto analog videotape formats (Hi8, Sony Betacam) at standard NTSC frame rates (29.97 fps). Individual frames were captured into digital graphics workstations via video frame grabber boards, such as the Truevision TARGA series. Chroma keying was conducted using luminance and color-difference thresholding algorithms. Because analog videotape introduced motion blur, shadow bleed, and tape noise, automated green-screen keying left edge artifacts around character silhouettes. Digital artists manually erased residual boundary pixels frame-by-frame to produce clean sprite edges.

Palette Indexing and Memory Optimization

Arcade graphics boards relied on indexed color palettes to conserve video memory. Instead of storing 16-bit or 24-bit RGB values per pixel, each pixel stored a 4-bit or 8-bit memory address pointing to a Color Look-Up Table (LUT).

This architecture enabled color swapping[10]. In *Mortal Kombat*, the ninja fighters (Scorpion, Sub-Zero, and Reptile) were sampled from a single base set of digitized animation frames filmed with Daniel Pesina in a yellow costume[6]. By altering the active values in the graphics card's hardware LUT at runtime, the system displayed the same underlying sprite data as yellow, blue, or green without requiring extra ROM space[10].

```
0x00 0x02 0x01  ---->  Index 0x03: RGB( 50, 50, 50)  ---->  Shadow Black
```

Frame Decimation and Collision Geometry

Due to cartridge ROM limitations (often restricted to 2MB–8MB total capacity for graphics, audio, and code), developers could not store full 30fps motion captures. The animation processing pipeline relied on three key optimization strategies:

- **Frame Decimation**: Continuous video captured at 30fps was downsampled to 10–12 keyframes per second. Eliminating intermediate frames preserved memory while creating faster move startup times essential for competitive input response.
- **Bounding Box Trimming**: Outer transparent pixels surrounding the character silhouette were cropped. Each frame was anchored to a ground pivot point coordinate (X, Y) to prevent sprite jitter during playback.
- **Hitbox and Hurtbox Mapping**: Rectangular collision arrays were overlaid onto each cropped frame. Hurtboxes defined vulnerable character anatomical regions, while Hitboxes marked active strike zones generated during specific attack frames.

Modern Engine Implementations: Full-Stack Re-Architectures

Modern game engines allow developers to construct 2.5D digitized fighting games featuring sub-pixel precision, dynamic lighting, real-time GLSL/HLSL shaders, and deterministic physics execution.

Engine Platform	Rendering Pipeline	Lighting Architecture	Frame Timing & Input Handling	Deployment Capabilities
Godot 4.x	CanvasItem / CharacterBody 2D	2D Canvas Shaders with dynamic Normal Maps	Deterministic _physics_process fixed ticks	Desktop Native, WebAssembly (WASM), WebGL2
Unreal Engine 5	Paper2D / PaperZD Orthographic Planes	Deferred Renderer with Lumen 3D Lighting	Fixed C++ 60Hz tick decoupled from render rate	High-End PC, Consoles, Cloud Pixel Streaming
Three.js	WebGL / WebGPU	Custom GLSL Fragment	requestAnimationFrame with	Native Cross-Platform

	THREE.Sprite & Mesh Planes	Shaders (PBR/Normals)	WebWorker WASM Physics	Web Browsers

Godot 4.x Implementation

Godot 4 provides a dedicated 2D rendering pipeline that supports 3D-style dynamic lighting on 2D planes. Characters are instantiated as a CharacterBody2D node containing an AnimatedSprite2D, an AnimationPlayer, and various Area2D collision nodes.
Dynamic palette swapping and 2D normal-mapped dynamic lighting are processed directly on the GPU using a custom GLSL Canvas Item Shader:

```
OpenGL Shading Language
shader_type canvas_item;

uniform sampler2D palette_texture : filter_nearest;
uniform float palette_index : hint_range(0.0, 10.0, 1.0);
uniform float num_palettes = 10.0;
uniform sampler2D normal_map : hint_normal;

void fragment() {
    vec4 base_color = texture(TEXTURE, UV);

    // Read indexed channel to derive color palette coordinate
    float index = base_color.r;
    vec2 palette_uv = vec2(index, (palette_index + 0.5) / num_palettes);
    vec4 swapped_color = texture(palette_texture, palette_uv);

    swapped_color.a *= base_color.a;

    COLOR = swapped_color;
    NORMAL = texture(normal_map, UV).rgb;
}
```

State management is evaluated inside the _physics_process(delta) loop using a Finite State Machine (FSM) that reads an input buffer to evaluate motion commands (such as quarter-circle inputs) deterministically.

Unreal Engine 5 Architecture

Unreal Engine 5 handles 2.5D digitized fighting games by rendering 2D textured sprite cards within fully realized 3D environments illuminated by deferred lighting.

1. **Sprite & Flipbook Integration**: Character frames are imported into Paper2D or PaperZD texture atlases using uncompressed 32-bit RGBA PNG files.
2. **Normal Map and PBR Materials**: Sprites use custom lit materials containing Normal, Roughness, and Specular texture maps. Unreal's Lumen dynamic lighting system illuminates 2D character silhouettes as environmental lights move across the 3D stage.
3. **Deterministic Combat Logic**: Combat mechanics decouple visual interpolation from state calculations. A fixed-tick C++ combat engine executes at $60.0\ \text{Hz}$, maintaining frame structures that calculate move startup, active hit windows, recovery periods, and frame advantage on hit or block.

Three.js WebGL / WebGPU Implementation

For web-based digitized fighting games, Three.js provides direct access to GPU rendering pipelines. To recreate real-time video-based digitized characters (similar to *Bikini Karate Babes*), Three.js maps an HTML5 <video> element directly onto a WebGL plane via THREE.VideoTexture. Real-time chroma keying is executed inside a fragment shader:

```JavaScript
const video = document.getElementById('fighter_video');
const videoTexture = new THREE.VideoTexture(video);
videoTexture.minFilter = THREE.NearestFilter;
videoTexture.magFilter = THREE.NearestFilter;

const chromaShaderMaterial = new THREE.ShaderMaterial({
  uniforms: {
    u_texture: { value: videoTexture },
    u_keyColor: { value: new THREE.Color(0x00ff00) },
    u_threshold: { value: 0.4 }
  },
  vertexShader: `
    varying vec2 vUv;
    void main() {
      vUv = uv;
      gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    }
```

```
  `
  fragmentShader: `
    uniform sampler2D u_texture;
    uniform vec3 u_keyColor;
    uniform float u_threshold;
    varying vec2 vUv;

    void main() {
      vec4 texColor = texture2D(u_texture, vUv);
      float dist = distance(texColor.rgb, u_keyColor);
      if (dist < u_threshold) {
        discard;
      }
      gl_FragColor = texColor;
    }
  `,
  transparent: true
});

const spriteMesh = new THREE.Mesh(new THREE.PlaneGeometry(2.0, 3.5), chromaShaderMaterial);
scene.add(spriteMesh);
```

Physics computations run in a dedicated Web Worker using a WebAssembly (WASM) build of the Rapier2D physics engine, shielding frame calculations from main-thread JavaScript execution pauses.

Generative AI Asset Pipelines for Digitized Fighting Games

Modern generative AI models remove the need for physical film sets, studio lighting, and manual green-screen isolation by using models such as FLUX, Qwen-Image-Edit, TRELLIS, and Hunyuan3D-2.1.

Pipeline Phase	Primary Model Architecture	Input Conditioning	Output Asset Specification
Character Identity Generation	FLUX.1 [Dev] + Custom LoRA	Text Prompts + Fine-Tuned Style LoRA Weights	Master character reference portrait ($2048 \times$)
Pose-Guided	Qwen-Image-Edit	Master Portrait +	Target pose

Frame Generation	(2509 / 2511)	DWPose Skeleton + Masked Ref	keyframe image with preserved identity[11]
Automated Matting & Normal Baking	RMBG-2.0 / Segment Anything 2	Generated Pose Images	Transparent RGBA sprite cutout + 2D Normal Map
3D Mesh Generation & PBR Materials	TRELLIS / Tencent Hunyuan3D-2.1	Single Keyframe Image Condition	3D Mesh Geometry + Baked PBR Material Maps[13]

Character Synthesis via FLUX and Style LoRAs

Character generation begins by producing a high-resolution master image using the FLUX.1 diffusion architecture. Developers fine-tune character consistency using Low-Rank Adaptations (LoRAs). A LoRA injects trainable rank decomposition matrices into FLUX's attention layers, fixing facial structure, body proportions, and costume details across generated frames.

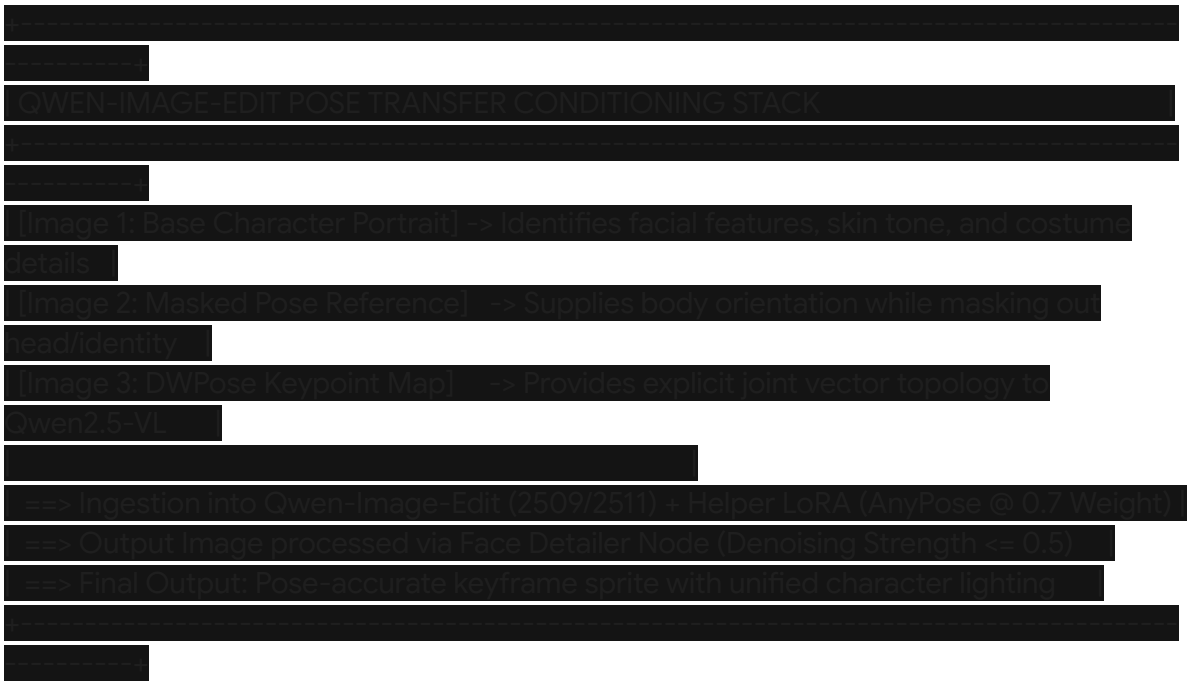
Pose-Guided Animation Sequences via Qwen-Image-Edit and DWPose

To generate complete animation sequences (such as sweeps, uppercuts, or jump kicks), pipelines utilize **Qwen-Image-Edit** (versions 2509 and 2511)[11]. The model employs a dual-stream architecture[16]:

- **Qwen2.5-VL**: Manages visual semantic reasoning and structural prompt adherence[16].
- **VAE Encoder**: Maintains visual texture, costume detail, and facial features across edits[16].

To execute pose transfers with high fidelity, ComfyUI workflows use a three-image conditioning setup[11]:

1. **Image 1 (Main Character)**: The base FLUX character portrait establishing facial identity and costume details[12].
2. **Image 2 (Pose Reference)**: A photograph or mannequin render demonstrating the target pose[12]. This image is masked to strip away background elements, hair, and facial traits to prevent identity bleed[12].
3. **Image 3 (Pose Skeleton)**: A structural skeleton generated from Image 2 using DWPose keypoint estimation[12].



When generating complex martial arts poses (such as ground sweeps or spinning kicks), Qwen-Image-Edit is paired with helper LoRAs like **AnyPose** set to a weight strength of 0.7[12]. The output is processed through a Face Detailer node using a specialized facial LoRA[12]. The denoising strength inside the Face Detailer must be capped at ≤ 0.5 to avoid lighting mismatches between the face and body[12].

3D Mesh Generation & PBR Asset Pipelines: TRELLIS and Hunyuan3D-2.1

While 2D sprites provide high rendering performance, modern hybrid engines often generate full 3D meshes for dynamic camera movements during special moves or precise 2.5D normal map baking.

Microsoft TRELLIS

TRELLIS converts single 2D images into 3D assets using structured 3D latents[13]. The architecture combines sparse voxel grids with Flexicubes shape decoders to extract volumetric geometry from concept renders, generating low-polygon topology suitable for real-time applications[13].

Tencent Hunyuan3D-2.1

Hunyuan3D-2.1 provides an open-source framework that decouples 3D mesh creation from texture synthesis into a two-stage generation pipeline[14]:

1. **Geometry Generation (Hunyuan3D-DiT)**: A 3.3-billion parameter Flow-Matching Diffusion Transformer generates bare, untextured 3D meshes from a single image condition[14].

2. **PBR Texture Synthesis (Hunyuan3D-Paint)**: A 2-billion parameter model maps physically-based rendering textures onto generated geometry[14]. Utilizing normal and position maps derived from the mesh, Hunyuan3D-Paint bakes multi-view textures across four core PBR channels: Albedo, Normal, Roughness, and Metallic maps[14].

Programmatic Integration Script

The following Python script orchestrates pose transfer via Qwen-Image-Edit and generates a textured 3D mesh using Hunyuan3D-2.1:

```python
import os
import torch
from PIL import Image
from diffusers import QwenImageEditPipeline
from hy3dshape.pipelines import Hunyuan3DDiTFlowMatchingPipeline
from hy3dpaint.textureGenPipeline import Hunyuan3DPaintPipeline
Hunyuan3DPaintConfig

def generate_digitized_character_asset(
    portrait_path: str,
    skeleton_path: str,
    pose_prompt: str,
    output_dir: str
):
    os.makedirs(output_dir, exist_ok=True)

    # Step 1: Execute Qwen-Image-Edit Pose Transfer
    qwen_pipe = QwenImageEditPipeline.from_pretrained(
        "Qwen/Qwen-Image-Edit"
        torch_dtype=torch.bfloat16
    ).to("cuda")

    char_img = Image.open(portrait_path).convert("RGB")
    skel_img = Image.open(skeleton_path).convert("RGB")

    inputs = {
        "image": char_img,
        "image_2": skel_img,
        "prompt": f"The person in Image 1 adopts the pose from Image 2. {pose_prompt}"
```

```
        "true_cfg_scale": 4.0,
        "num_inference_steps": 30,
    }

    with torch.inference_mode():
        sprite_output = qwen_pipe(**inputs).images[0]

    sprite_path = os.path.join(output_dir, "keyframe_sprite.png")
    sprite_output.save(sprite_path)

    del qwen_pipe
    torch.cuda.empty_cache()

    # Step 2: Generate Geometry with Hunyuan3D-DiT
    shape_pipe = Hunyuan3DDiTFlowMatchingPipeline.from_pretrained(
        'tencent/Hunyuan3D-2.1'
    )
    bare_mesh = shape_pipe(image=sprite_path)[0]
    mesh_path = os.path.join(output_dir, "bare_mesh.obj")
    bare_mesh.export(mesh_path)

    del shape_pipe
    torch.cuda.empty_cache()

    # Step 3: Bake PBR Textures with Hunyuan3D-Paint
    paint_config = Hunyuan3DPaintConfig(max_num_view=6, resolution=1024)
    paint_pipe = Hunyuan3DPaintPipeline(paint_config)

    textured_mesh = paint_pipe(mesh_path=mesh_path, image_path=sprite_path)
    final_path = os.path.join(output_dir, "character_3d.glb")
    textured_mesh.export(final_path)

if __name__ == "__main__":
    generate_digitized_character_asset(
        portrait_path="./inputs/hero.png",
        skeleton_path="./inputs/dwpose_kick.png",
        pose_prompt="Executing a high side kick, martial arts tournament style",
        output_dir="./exports/fighter_01"
    )
```

Comparative Synthesis and Strategic Outlook

The development of digitized fighting games demonstrates an ongoing balance between visual

fidelity, computational limits, and workflow complexity.

Development Metric	Historical Pipeline (1990s)	Modern Engine Implementation	Generative AI Pipeline
Asset Ingestion	Physical filming on blue screens with Hi8/Betacam tape[6].	High-resolution digital video capture or 3D character renders.	Text-to-image synthesis (FLUX) and pose-guided editing[11].
Color & Palette Processing	16/256 color quantization using hardware LUT color swaps[6].	Full 32-bit RGBA color channels with real-time GLSL canvas shaders.	Automated ML matting (RMBG-2.0 / SAM 2) and PBR texture maps[14].
Frame Rate & Memory	Decimated 10–12 fps keyframes constrained by 2MB–8MB ROMs[6].	Uncompressed texture atlases running at native 60fps or higher.	Programmatic generation of complete move matrices via pose transfer[12].
3D & Depth Support	Pseudo z-axis scaling via simple hardware stretching[1].	Hybrid 2.5D rendering planes with real-time dynamic lighting.	Automated single-image 3D reconstruction via TRELLIS / Hunyuan3D[13].

Summary of Engineering Principles

1. **Historical Precedents**: Early digitized titles like *Pit-Fighter* established hardware-scaled sprite mechanics[1], while *Mortal Kombat* refined the genre through standardized frame data, block buttons, mid-air combos, and palette-swapped Look-Up Tables[5].
2. **Modern Engine Capabilities**: Game engines like Godot 4, Unreal Engine 5, and Three.js remove historical memory limits, enabling full 32-bit RGBA color depths, dynamic GPU-driven normal maps, and deterministic fixed-tick combat logic.
3. **AI Pipeline Efficiency**: Generative frameworks streamline asset creation by combining FLUX identity preservation, Qwen-Image-Edit pose transfers, and automated 3D mesh reconstruction tools like TRELLIS and Hunyuan3D-2.1[12]. This reduces dependence on physical studio infrastructure while maintaining the distinct visual character of digitized fighting games.

Works cited

1. Pit-Fighter - Bad Game Hall of Fame, https://www.badgamehalloffame.com/pit-fighter/
2. Mortal Kombat Origin Story | Birth of ESRB & Fighting Games - GOG.com, https://www.gog.com/blog/finish-him-experience-the-classic-mortal-kombat-games-once-more/
3. Pit Fighter - Sega-16, https://www.sega-16.com/2007/04/pit-fighter/
4. Mortal Kombat (Game Boy) - Bad Game Hall of Fame, https://www.badgamehalloffame.com/mortal-kombat-gb/
5. Mortal Kombat (1992 video game) - Wikipedia, https://en.wikipedia.org/wiki/Mortal_Kombat_(1992_video_game)
6. Revisiting Mortal Kombat: the legend, the tech and the console ports | Digital Foundry, https://www.digitalfoundry.net/articles/digitalfoundry-2020-revisiting-mortal-kombat-the-legend-the-tech-the-ports
7. Top 5 Strangest Street Fighter and Mortal Kombat Clones - OC Weekly, https://www.ocweekly.com/top-5-strangest-street-fighter-and-mortal-kombat-clones-6582163/
8. Crapshoot: Bikini Karate Babes, which does exactly what it says on the tin | PC Gamer, https://www.pcgamer.com/saturday-crapshoot-bikini-karate-babes/
9. Steam Curators, https://store.steampowered.com/curators/curatorsinformative/?l=english&appid=350910
10. Mortal Kombat - Revenant Publications, https://revenantpublications.com/tag/mortal-kombat/
11. How to Use Qwen Image Edit 2509 on RunDiffusion, https://www.rundiffusion.com/how-to-use-qwen-image-edit-2509-on-rundiffusion
12. Pose Transfer in ComfyUI with Qwen Edit 2511 (High-Success Workflow) - MyAIForce, https://myaiforce.com/qwen-image-edit-pose-transfer/
13. hunyuan-3d/v3.1/smart-topology vs TRELLIS-image-large — comparison, examples, use cases - AIModels.fyi, https://www.aimodels.fyi/models/compare/hunyuan-3d-v31-smart-topology-fal-ai-vs-trellis-image-large-microsoft
14. GitHub - Tencent-Hunyuan/Hunyuan3D-2.1: From Images to High-Fidelity 3D Assets with Production-Ready PBR Material, https://github.com/tencent-hunyuan/hunyuan3d-2.1
15. Hunyuan3D 2.0: Scaling Diffusion Models for High Resolution Textured 3D Assets Generation - arXiv, https://arxiv.org/html/2501.12202v5
16. Qwen/Qwen-Image-Edit - Hugging Face, https://huggingface.co/Qwen/Qwen-Image-Edit
17. Qwen-Image-Edit-2509 Pose Transfer - No LoRA Required : r/StableDiffusion - Reddit,

https://www.reddit.com/r/StableDiffusion/comments/1np8lfv/qwenimageedit2509_pose_transfer_no_lora_required/
18. Qwen Image Edit 2511 Pose Transfer And Anime Stylize LoRAs Made It Easy To Edit!, https://www.youtube.com/watch?v=RQAX_IrcgrM
19. Transfer ANY Pose (Even Hard Ones) with Qwen Edit 2511 — High Success Workflow!, https://www.youtube.com/watch?v=ZphPp-9ou34
20. Voxhammer 2: State-of-the-art 3D Model Editing - Faz, https://www.fazzaidi.com/vh2
21. GitHub - Tencent-Hunyuan/Hunyuan3D-2: High-Resolution 3D Assets Generation with Large Scale Hunyuan3D Diffusion Models., https://github.com/Tencent-Hunyuan/Hunyuan3D-2