Script Documentation : refresh.swith, generate index, and deploy

**Overview**

This Swift script automatically generates PNG thumbnail images from USDZ (Universal Scene Description) 3D model files. It's designed for batch processing with intelligent caching, error logging, and automatic deployment integration.

**Features**

* **Batch Processing**: Processes all USDZ files in a directory
* **Smart Caching**: Only regenerates thumbnails when USDZ files are newer than existing PNGs
* **Error Logging**: Redirects verbose USD errors to a log file
* **Deployment Integration**: Automatically runs index generation and deployment scripts
* **Clean Output**: Minimal console output with clear status indicators

**Usage**

swift refresh.swift [directory]

* If no directory is specified, uses current directory (.)
* Creates PNG thumbnails alongside USDZ files
* Runs post-processing scripts if they exist

**Script Breakdown**

**1. Imports and Setup**

#!/usr/bin/env swift

import Foundation

import SceneKit

import SceneKit.ModelIO

import AppKit

* **Shebang**: Allows script to run directly without swift command
* **Foundation**: Core system functionality
* **SceneKit/ModelIO**: 3D rendering and model loading
* **AppKit**: macOS image handling (NSImage, NSBitmapImageRep)

**2. ARQLThumbnailGenerator Class**

This class handles the core thumbnail generation using Apple's SceneKit framework.

class ARQLThumbnailGenerator {

private let device = MTLCreateSystemDefaultDevice()!

* Creates a Metal device for hardware-accelerated rendering
* Metal provides GPU-based rendering for better performance

**thumbnail() Method**

func thumbnail(for url: URL, size: CGSize, time: TimeInterval = 0) -> NSImage?

**Parameters:**

* url: File path to the USDZ model
* size: Desired thumbnail dimensions (we use 512x512)
* time: Animation time point (default 0 for static)

**Process:**

1. **Renderer Setup**: Creates SCNRenderer with automatic lighting
2. **Model Loading**: Uses ModelIO to load USDZ file
3. **Texture Loading**: Calls loadTextures() to ensure materials display correctly
4. **Scene Creation**: Converts ModelIO asset to SceneKit scene
5. **Validation**: Checks if scene has geometry (prevents empty thumbnails)
6. **Rendering**: Takes high-quality snapshot with 4x antialiasing

**3. Command Line Argument Handling**

let args = CommandLine.arguments

let directory = args.count >= 2 ? args[1] : "."

* Defaults to current directory if no path provided
* Maintains backward compatibility while improving usability

**4. File Discovery and Logging Setup**

let files = try FileManager.default.contentsOfDirectory(atPath: directory).filter { $0.hasSuffix(".usdz") }

let logPath = directory + "/refresh.log"

let logURL = URL(fileURLWithPath: logPath)

FileManager.default.createFile(atPath: logPath, contents: nil, attributes: nil)

let logHandle = FileHandle(forWritingAtPath: logPath)!

* **File Discovery**: Finds all .usdz files in the target directory
* **Log File**: Creates/overwrites refresh.log for error and status capture
* **FileHandle**: Uses modern Swift FileHandle for better logging control

**5. Main Processing Loop**

The core logic that processes each USDZ file with intelligent caching.

**Timestamp Comparison**

if FileManager.default.fileExists(atPath: outputPath) {

// Compare modification dates

if usdzDate > pngDate {

shouldGenerate = usdzDate > pngDate

}

}

**Smart Caching Logic:**

* Checks if PNG thumbnail already exists
* Compares modification dates of USDZ and PNG files
* Only regenerates if USDZ is newer than existing PNG
* Skips unchanged files with "- filename (up to date)" message

**Error Suppression and Logging**

let originalStderr = dup(STDERR\_FILENO)

dup2(logHandle.fileDescriptor, STDERR\_FILENO)

// ... thumbnail generation ...

dup2(originalStderr, STDERR\_FILENO)

// Log success/failure to file

let logMessage = "SUCCESS: Generated thumbnail for \(file)\n"

logHandle.write(logMessage.data(using: .utf8)!)

**Comprehensive Logging:**

* USD library outputs verbose error messages to stderr
* These errors are redirected to refresh.log during processing
* SUCCESS/FAILED messages are explicitly written to log file
* Console shows only clean ✓/✗ status indicators
* All processing details available in log for debugging

**Image Processing and Saving**

let data = thumbnail.tiffRepresentation!

let bitmap = NSBitmapImageRep(data: data)!

let pngData = bitmap.representation(using: .png, properties: [:])!

try pngData.write(to: outputURL)

**Image Conversion Process:**

1. SceneKit returns NSImage (native macOS format)
2. Convert to TIFF representation (intermediate format)
3. Create bitmap representation for format control
4. Convert to PNG with no compression properties
5. Write directly to filesystem

**6. Status Reporting**

print("✓ \(file)") // Success

print("✗ \(file)") // Failure

print("- \(file) (up to date)") // Skipped

**Output Indicators:**

* **✓**: Successfully generated new thumbnail
* **✗**: Failed to generate (usually corrupted USDZ)
* **-**: Skipped because existing thumbnail is current

**7. Post-Processing Scripts**

**Index Generation**

let generateIndexScript = directory + "/generate\_index\_with\_USDZ.sh"

if FileManager.default.fileExists(atPath: generateIndexScript) {

// Run script

}

* Looks for generate\_index\_with\_USDZ.sh in the same directory
* Typically generates HTML index pages or catalogs
* Runs before deployment to ensure fresh content

**Deployment**

let deployScript = directory + "/deploy.sh"

if FileManager.default.fileExists(atPath: deployScript) {

// Run script with proper working directory

}

* Executes deploy.sh for automated deployment
* Sets working directory correctly for Git operations
* Reports exit codes for debugging

**Error Handling**

**Common Issues and Solutions**

**"Runtime Error: Failed to open layer"**

* Indicates corrupted or invalid USDZ file
* Script continues processing other files
* Error details saved to log file

**"Empty scene" warnings**

* USDZ loaded but contains no visible geometry
* Often happens with corrupted files
* Results in failed thumbnail generation

**Git deployment failures**

* Usually requires git pull before push
* Script reports exit codes for troubleshooting
* Check authentication and repository access

**Performance Considerations**

**Caching Benefits:**

* First run: Processes all files (slower)
* Subsequent runs: Only processes changed files (much faster)
* Typical speedup: 10-100x for unchanged files

**Hardware Requirements:**

* Requires Metal-compatible GPU
* Benefits from more GPU memory for complex models
* CPU usage minimal due to GPU acceleration

**File Size Impact:**

* Larger USDZ files take longer to process
* Complex materials/textures increase render time
* 512x512 PNG output is good balance of quality/size

**Integration Examples**

**Basic Usage**

# Process current directory

./refresh.swift

# Process specific directory

./refresh.swift /path/to/models

**With Deployment Pipeline**

# Your workflow might be:

1. Edit USDZ files

2. Run refresh.swift (generates PNGs + runs scripts)

3. generate\_index\_with\_USDZ.sh creates HTML catalog

4. deploy.sh pushes to GitHub Pages

**Troubleshooting**

# Check log file for detailed errors

cat refresh.log

# Force regeneration (delete existing PNGs)

rm \*.png && ./refresh.swift

# Test single file

swift -c "print(ARQLThumbnailGenerator().thumbnail(for: URL(fileURLWithPath: \"test.usdz\"), size: CGSize(width: 512, height: 512)))"

**Output Files**

**Generated Files:**

* filename.png: Thumbnail for filename.usdz
* refresh.log: Error and status log

**File Locations:**

* All files created in same directory as USDZ files
* No subdirectories created
* Maintains flat file structure for web deployment

**Deployment Scripts Documentation**

This document explains the two shell scripts that handle website generation and deployment for USDZ file galleries.

**generate\_index\_with\_USDZ.sh**

**Overview**

Creates a responsive HTML gallery page that displays all USDZ files with thumbnails as downloadable links. Designed for GitHub Pages deployment with modern styling.

**Purpose**

* **Gallery Generation**: Creates visual catalog of 3D models
* **Thumbnail Integration**: Shows PNG previews alongside download links
* **GitHub Pages Ready**: Includes .nojekyll file for proper deployment
* **Responsive Design**: Works on desktop and mobile devices

**Script Breakdown**

**1. Header and Setup**

#!/bin/bash

echo "--- Generating index.html and .nojekyll file ---"

* Standard bash script with status messaging
* Indicates start of generation process

**2. GitHub Pages Configuration**

touch .nojekyll

echo ".nojekyll file ensured for GitHub Pages."

**Why .nojekyll is needed:**

* GitHub Pages uses Jekyll by default
* Jekyll ignores files starting with underscore
* .nojekyll tells GitHub to serve files directly
* Essential for proper asset loading (CSS, images, etc.)

**3. HTML Template Structure**

The script builds a complete HTML document with modern styling:

**DOCTYPE and Meta Tags:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

* HTML5 standard structure
* UTF-8 encoding for international filenames
* Responsive viewport for mobile compatibility

**Styling Framework:**

<script src="https://cdn.tailwindcss.com"></script>

* Uses Tailwind CSS for rapid, modern styling
* CDN delivery for reliability and speed
* No local dependencies required

**Custom CSS:**

body {

font-family: "Inter", sans-serif;

background-color: #f0f4f8;

color: #334155;

}

* **Inter font**: Clean, modern typeface
* **Light blue-gray background**: Professional appearance
* **Dark gray text**: Good contrast and readability

**4. Page Layout Components**

**Header Section:**

<header class="bg-blue-600 text-white shadow-lg py-4">

* **Blue gradient header** with navigation
* **Shadow effect** for depth
* **Responsive navigation** with hover effects

**Main Content Area:**

<main class="flex-grow py-8">

<div class="container bg-white p-8 rounded-xl shadow-lg">

* **Flex-grow**: Ensures footer stays at bottom
* **White content area** with rounded corners
* **Subtle shadow** for card-like appearance

**Footer:**

<footer class="bg-gray-800 text-white py-6 mt-auto">

* **Dark footer** with copyright information
* **Auto-margin top** pushes to bottom of viewport

**5. File Processing Logic**

**IFS Configuration:**

IFS=\n'

for filename in $(ls -1 \*.usdz 2>/dev/null | sort); do

**Why IFS matters:**

* Default IFS splits on spaces, breaking filenames with spaces
* Setting IFS=\n' makes loop split only on newlines
* Essential for files like "w67b 106 bed and me.usdz"
* 2>/dev/null suppresses error if no USDZ files exist

**Thumbnail Detection:**

thumbnail\_file="${filename%.usdz}.png"

if [ -f "$thumbnail\_file" ]; then

thumbnail\_html="<img src=\"./${thumbnail\_file// /%20}\"..."

else

thumbnail\_html="<div class=\"thumbnail-placeholder\">📦</div>"

fi

**Process:**

1. **Generate thumbnail filename** by replacing .usdz with .png
2. **Check if thumbnail exists** using file test
3. **Create appropriate HTML**:
   * If PNG exists: <img> tag with proper encoding
   * If no PNG: Placeholder div with package emoji
4. **URL encoding**: Spaces become %20 for web compatibility

**URL Encoding:**

ENCODED\_FILENAME="${filename// /%20}"

* **Bash parameter expansion**: ${filename// /%20} replaces all spaces with %20
* **Web standard**: Ensures links work with spaces in filenames
* **Download attribute**: Preserves original filename when downloading

**6. HTML Generation**

**Dynamic List Building:**

INDEX\_HTML\_CONTENT+="

<li class=\"file-item\">

${thumbnail\_html}

<a href=\"./${ENCODED\_FILENAME}\" download=\"${filename}\">

${filename}

</a>

</li>"

**Structure:**

* **Flex layout**: Thumbnail, filename, download arrow
* **Download attribute**: Forces download instead of browser preview
* **Hover effects**: Scale and color transitions
* **Accessibility**: Proper semantic markup

**File Output:**

echo "$INDEX\_HTML\_CONTENT" > index.html

* Writes complete HTML to index.html
* Overwrites existing file
* Creates web-ready gallery page

**deploy.sh**

**Overview**

Automates Git workflow for deploying changes to remote repositories. Handles the standard add-commit-push cycle with error checking and user feedback.

**Purpose**

* **Automated Deployment**: Reduces manual Git commands
* **Error Handling**: Provides clear feedback on failures
* **GitHub Pages Integration**: Optimized for static site deployment
* **Zero-downtime**: Checks for changes before committing

**Script Breakdown**

**1. Configuration Variables**

GIT\_BRANCH="main"

COMMIT\_MESSAGE="Updated content"

**Customizable Settings:**

* **Branch target**: Usually "main" or "master"
* **Default message**: Generic but descriptive
* **Easy modification**: Change once, applies to all commits

**2. Git Add Process**

echo "Adding all changes to Git staging area..."

git add .

**What git add . does:**

* **Stages all changes** in current directory and subdirectories
* **Includes new files**: Newly generated PNGs and updated HTML
* **Includes modifications**: Changed USDZ files or updated content
* **Includes deletions**: Removed files are staged for deletion

**3. Change Detection**

if git diff --cached --quiet; then

echo "No changes to commit. Exiting."

exit 0

fi

**Smart Skip Logic:**

* **git diff --cached**: Shows staged changes
* **--quiet flag**: No output, just exit code
* **Exit code 0**: No differences found
* **Early exit**: Prevents empty commits and unnecessary pushes

**Why this matters:**

* Saves time on repeated runs
* Prevents cluttering Git history with empty commits
* Provides clear feedback when nothing needs deployment

**4. Commit Process**

echo "Committing changes with message: \"$COMMIT\_MESSAGE\""

git commit -m "$COMMIT\_MESSAGE"

**Commit Creation:**

* **Fixed message**: Simple but identifiable
* **All staged changes**: Includes thumbnails, HTML, and any other updates
* **Local operation**: Fast, doesn't require network

**5. Remote Push**

echo "Pushing changes to $GIT\_BRANCH branch on GitHub..."

git push origin "$GIT\_BRANCH"

**Push Process:**

* **Remote sync**: Sends local commits to GitHub
* **Branch specific**: Pushes to configured branch only
* **Triggers deployment**: GitHub Pages rebuilds site automatically

**6. Exit Status Handling**

if [ $? -eq 0 ]; then

echo "--- Git Deployment Successful! ---"

echo "Your changes should now be deploying to your live site."

else

echo "--- Git Deployment Failed! ---"

echo "Please check the error messages above..."

fi

**Error Reporting:**

* **$?**: Exit code of last command (git push)
* **Exit code 0**: Success
* **Non-zero**: Failure (authentication, conflicts, network)
* **User guidance**: Suggests common solutions

**Common Issues and Solutions**

**generate\_index\_with\_USDZ.sh Issues**

**No USDZ files found:**

* Script handles gracefully with 2>/dev/null
* Creates empty gallery page
* No error messages

**Filenames with special characters:**

* IFS handling manages spaces correctly
* URL encoding prevents link breakage
* HTML escaping prevents injection

**Missing thumbnails:**

* Graceful fallback to 📦 emoji placeholder
* Consistent layout maintained
* No broken image links

**deploy.sh Issues**

**Authentication failures:**

* Usually requires Personal Access Token
* Check GitHub authentication settings
* Verify repository permissions

**Merge conflicts:**

* Requires git pull before push
* Script doesn't handle this automatically
* Manual intervention needed

**Network issues:**

* Retry the deployment
* Check internet connection
* Verify GitHub status

**Integration Workflow**

**Complete Process Flow**

1. **Edit USDZ files** (add/modify/remove models)
2. **Run refresh.swift** (generates PNGs)
3. **generate\_index\_with\_USDZ.sh** (creates gallery HTML)
4. **deploy.sh** (pushes to GitHub Pages)
5. **GitHub Pages rebuilds** (site updates automatically)

**File Dependencies**

project-directory/

├── \*.usdz # 3D model files

├── \*.png # Generated thumbnails

├── index.html # Generated gallery page

├── .nojekyll # GitHub Pages config

├── refresh.swift # Thumbnail generator

├── generate\_index\_with\_USDZ.sh # Gallery generator

└── deploy.sh # Git deployment

**Typical Output**

# From refresh.swift

✓ model1.usdz

✓ model2.usdz

- model3.usdz (up to date)

Done: 2 success, 0 failed

# From index generator

--- Generating index.html and .nojekyll file ---

.nojekyll file ensured for GitHub Pages.

index.html has been generated with links to all .usdz files.

--- Generation Complete ---

# From deployment

--- Starting Git Deployment ---

Adding all changes to Git staging area...

Committing changes with message: "Updated content"

Pushing changes to main branch on GitHub...

--- Git Deployment Successful! ---

Your changes should now be deploying to your live site.

**Customization Options**

**generate\_index\_with\_USDZ.sh**

* **Styling**: Modify CSS in the <style> section
* **Colors**: Change Tailwind classes (bg-blue-600, etc.)
* **Layout**: Adjust file-item structure
* **Branding**: Update header text and footer

**deploy.sh**

* **Branch**: Change GIT\_BRANCH variable
* **Commit message**: Modify COMMIT\_MESSAGE
* **Pre-push checks**: Add additional validation
* **Post-deploy actions**: Add notification webhooks