포팅 메뉴얼

1. 개발 환경

- **Front-End**

Node.js	
Vue.js	v3.5.13
Vite	v6.0.5
vue-router	v4.5.0
pinia	v2.3.0
tailwindcss	v3.4.17
eslint	v9.17.0
husky	v9.1.7
axios	v1.7.9

- **Back-End (GPU-server)**

cuda-toolkit	12.1
torch	2.5.1+cu121
ComfyUI	0.3.14
llama_cpp_python	0.3.7
transformers	4.47.1
fastapi	0.115.8
unicorn	0.34.0

- **Back-End (main-server)**

Java	openjdk 17
Gradle	v8.12.0
Spring Boot	v3.4.1
redis	v3.4.1
jpa	v3.4.1
Hibernate	v6.6.4.final
Spring Security	v3.4.1
Webflux(WebClient)	v3.4.1
lombok	v1.18.36
jackson(json 처리)	v2.18.2
badwordFiltering(욕설 필터)	v1.0.0
postgresql	v42.7.4
swagger	v2.2.0
Spring boot Devtools	v3.4.1

2. 환경 변수 형태

▼ Front_End

VITE_GOOGLE_CLIENT_ID	구글 oauth
VITE_MAIN_API_SERVER_URL	테스트 서버 IP주소 ex) http://x.x.x.x/
VITE_MAIN_API_SERVER_URL_MAIN	메인 서버 IP주소 ex) https://x.x.x.x/
VITE_TURN_ID	coturn 생성 시 작성한 ID
VITE_TURN_PW	coturn 생성 시 작성한 PW
VITE_USERS	apis/api/signup
VITE_USERS_SIGNIN	apis/api/user/login
VITE_GAME	apis/game
VITE_GAME_SHUFFLE	/shuffle
VITE_SCENE	apis/scene
VITE_SCENE_FILTERING	/filtering
VITE_SCENE_VOTE	/vote

▼ Back_End

DB_URL	DB URL (테스트 서버)

DB_URL_MAIN	DB URL (메인 서버)
DB_USERNAME	DB USERNAME (테스트 서버)
DB_USERNAME_MAIN	DB USERNAME (메인 서버)
DB_PASSWORD	DB PASSWORD (테스트 서버)
DB_PASSWORD_MAIN	DB PASSWORD (메인 서버)
DB_DRIVER	DB DRIVER 입력 ex) org.postgresql.Driver
REDIS_HOST	레디스 호스트 값 입력
WEBCLIENT_BASE_URL	GPU 서버 URL

▼ Infra

DOCKERHUB_USERNAME	도커 허브 아이디(test-server)
DOCKERHUB_PASSWORD	도커 허브 패스워드(test-server)
DOCKERHUB_USERNAME_MAIN	도커 허브 아이디(main-server)
DOCKERHUB_PASSWORD_MAIN	도커 허브 패스워드(main-server)
IMAGE_FE	도커 허브 front-end 레포지토리(test-server)
IMAGE_BE	도커 허브 back-end 레포지토리(test-server)
IMAGE_FE_MAIN	도커 허브 front-end 레포지토리(main-server)
IMAGE_BE_MAIN	도커 허브 back-end 레포지토리(main-server)
MAIN_SERVER_URI	<사용자명>@<서버 주소>
MAIN_SERVER_KEY	ssh 키 위치(home/gitlab-runner/ 하위에 위치)

S3_BUCKET_NAME	S3 버킷 이름
S3_REGION	S3 지역
S3_CREDENTIALS_ACCESS_KEY	S3에 접근 가능 IAM access-key
S3_CREDENTIALS_SECRET_KEY	S3에 접근 가능 IAM secret-key

3. 인프라 구축

- ***GitLab***
  - ▼ /main/FE/.Dockerfile.yml 생성

```
#############
# 1.  빌드   #
#############

FROM node:18-alpine AS build

# 컨테이너 안 경로 설정
WORKDIR /app

# 복사 및 종속성 설치
COPY package.json package-lock.json ./
RUN npm install

# 소스 코드 복사 및 빌드
COPY . .
RUN npm run build

################
# 2. 이미지 생성  #
################
```

```
FROM nginx:alpine

# 빌드된 파일을 Nginx의 기본 경로로 복사
COPY --from=build /app/dist /usr/share/nginx/html

# 포트 작성
EXPOSE 80

# Nginx 실행
CMD ["nginx", "-g", "daemon off;"]
```

▼ /main/BE/.Dockerfile.yml 생성

```
##############
# 1.  빌드   #
##############

# 베이스 이미지 설정
FROM gradle:8.12.0-jdk17-alpine AS build

# 컨테이너 안 경로 설정
WORKDIR /home/gradle/BE

# 종속성 관련 파일만 먼저 복사
COPY build.gradle settings.gradle ./

#  소스 코드 복사
COPY src ./src/

# Gradle 캐시를 활용하여 종속성 다운로드 및 빌드
RUN gradle build -x test


#################
# 2. 이미지 생성  #
#################

FROM openjdk:17-jdk-alpine

WORKDIR /BE

# 빌드된 JAR 파일 복사
COPY --from=build /home/gradle/BE/build/libs/*.jar app.jar

# 애플리케이션 실행
ENTRYPOINT ["java", "-jar", "app.jar"]

# 포트 노출 (Spring Boot 기본 포트)
EXPOSE 8080
```

▼ /main/**.gitlab-ci.yml 생성**

```


# 파이프라인 스테이지
stages:
  - build
  - push
  - cleanup
  - deploy


############################
# 1-1. build_FE JOB 단계 #
############################

build_FE:
  stage: build
  image: docker:27.4.0
  variables:
    IMAGE_NAME: "$IMAGE_FE"
    DOCKERFILE_PATH: FE/Dockerfile
    CONTEXT: FE
    VITE_GOOGLE_CLIENT_ID: "$VITE_GOOGLE_CLIENT_ID"
    VITE_MAIN_API_SERVER_URL: "$VITE_MAIN_API_SERVER_URL"
    VITE_USERS: "$VITE_USERS"
    VITE_USERS_SIGNIN: "$VITE_USERS_SIGNIN"
    VITE_GAME: "$VITE_GAME"
    VITE_GAME_SHUFFLE: "$VITE_GAME_SHUFFLE"
    VITE_TURN_ID: "$VITE_TURN_ID"
    VITE_TURN_PW: "$VITE_TURN_PW"
    VITE_SCENE: "$VITE_SCENE"
    VITE_SCENE_FILTERING: "$VITE_SCENE_FILTERING"
    VITE_SCENE_VOTE: "$VITE_SCENE_VOTE"
  script:
    - echo "============================"
    - echo "     FRONT-END빌드 중"
    - echo "============================"

    # 환경 변수 적용 (main 브랜치인 경우 URL & DockerHub 변경)
    - |
      if [ "$CI_COMMIT_REF_NAME" == "main" ]; then
        export VITE_MAIN_API_SERVER_URL="$VITE_MAIN_API_SERVER_URL_MAIN"
        export IMAGE_NAME="$IMAGE_FE_MAIN"
      else
        # main이 아닌 브랜치일 경우, Docker 이미지가 존재하면 이미지 삭제
        if [ -n "$(docker images -q)" ]; then
          docker compose -f /home/ubuntu/docker-compose.yml down
          docker rmi $(docker images -q)
        fi
      fi

    # 환경변수 설정
    - echo "VITE_GOOGLE_CLIENT_ID=$VITE_GOOGLE_CLIENT_ID" >> FE/.env
    - echo "VITE_MAIN_API_SERVER_URL=$VITE_MAIN_API_SERVER_URL" >> FE/.env
    - echo "VITE_USERS=$VITE_USERS" >> FE/.env
    - echo "VITE_USERS_SIGNIN=$VITE_USERS_SIGNIN" >> FE/.env
    - echo "VITE_GAME=$VITE_GAME" >> FE/.env
    - echo "VITE_GAME_SHUFFLE=$VITE_GAME_SHUFFLE" >> FE/.env
    - echo "VITE_TURN_ID=$VITE_TURN_ID" >> FE/.env
    - echo "VITE_TURN_PW=$VITE_TURN_PW" >> FE/.env
    - echo "VITE_SCENE=$VITE_SCENE" >> FE/.env
    - echo "VITE_SCENE_FILTERING=$VITE_SCENE_FILTERING" >> FE/.env
    - echo "VITE_SCENE_VOTE=$VITE_SCENE_VOTE" >> FE/.env
    # 이미지 빌드
    - docker build --no-cache -t $IMAGE_NAME:$CI_COMMIT_SHA -f $DOCKERFILE_PATH $CONTEXT
    - docker tag $IMAGE_NAME:$CI_COMMIT_SHA $IMAGE_NAME:latest

  rules:
  - if: $CI_PIPELINE_SOURCE == "push" && ($CI_COMMIT_BRANCH == "FE/dev" || $CI_COMMIT_BRANCH == "ma
    when: always
  - if: $CI_COMMIT_BRANCH == "CICD/test" || $CI_COMMIT_BRANCH == "CICD/main"
    when: always
  - when: never


##########################
# 1-2. build_BE JOB 단계 #
##########################

build_BE:
  stage: build
  image: gradle:8.12.0-jdk17-alpine
  variables:
    IMAGE_NAME: "$IMAGE_BE"
    DOCKERFILE_PATH: BE/Dockerfile
    CONTEXT: BE

  script:
    - echo "============================"
    - echo "      BACK-END빌드 중"
    - echo "============================"

    # 환경 변수 적용 (main 브랜치인 경우 DockerHub 변경)
    - |
      if [ "$CI_COMMIT_REF_NAME" == "main" ]; then
        export IMAGE_NAME="$IMAGE_BE_MAIN"
      else
        # main이 아닌 브랜치일 경우, Docker 이미지가 존재하면 이미지 삭제
        if [ -n "$(docker images -q)" ]; then
          docker compose -f /home/ubuntu/docker-compose.yml down
          docker rmi $(docker images -q)
        fi
      fi

    # 이미지 빌드
    - docker build --no-cache -t $IMAGE_NAME:$CI_COMMIT_SHA -f $DOCKERFILE_PATH $CONTEXT
    - docker tag $IMAGE_NAME:$CI_COMMIT_SHA $IMAGE_NAME:latest

  rules:
  - if: $CI_PIPELINE_SOURCE == "push" && ($CI_COMMIT_BRANCH == "BE/dev" || $CI_COMMIT_BRANCH == "ma
    when: always
  - if: $CI_COMMIT_BRANCH == "CICD/test" || $CI_COMMIT_BRANCH == "CICD/main"
    when: always
  - when: never


#########################
# 2-1. push_FE JOB 단계 #
#########################

push_FE:
  stage: push
  variables:
    IMAGE_NAME: "$IMAGE_FE"
    DOCKERHUB_ID: "$DOCKERHUB_USERNAME"
    DOCKERHUB_PW: "$DOCKERHUB_PASSWORD"
  script:
    - echo "==========================="
    - echo "  FRONT-END 이미지 푸쉬 중"
    - echo "==========================="

    # 환경 변수 적용 (main 브랜치인 경우 DockerHub 변경)
    - |
      if [ "$CI_COMMIT_REF_NAME" == "main" ]; then
        export IMAGE_NAME="$IMAGE_FE_MAIN"
        export DOCKERHUB_ID="$DOCKERHUB_USERNAME_MAIN"
        export DOCKERHUB_PW="$DOCKERHUB_PASSWORD_MAIN"
      fi

    # DockerHub로 이미지 PUSH
    - echo $DOCKERHUB_PW | docker login -u $DOCKERHUB_ID --password-stdin
    - docker push $IMAGE_NAME:$CI_COMMIT_SHA
    - docker push $IMAGE_NAME:latest

  dependencies:
    - build_FE
  rules:
  - if: $CI_PIPELINE_SOURCE == "push" && ($CI_COMMIT_BRANCH == "FE/dev" || $CI_COMMIT_BRANCH == "ma
    when: always
  - if: $CI_COMMIT_BRANCH == "CICD/test" || $CI_COMMIT_BRANCH == "CICD/main"
    when: always
  - when: never


########################
# 2-2. push_BE JOB 단계 #
########################

push_BE:
  stage: push
  variables:
    IMAGE_NAME: "$IMAGE_BE"
    DOCKERHUB_ID: "$DOCKERHUB_USERNAME"
    DOCKERHUB_PW: "$DOCKERHUB_PASSWORD"
  script:
    - echo "==========================="
    - echo "  BACK-END 이미지 푸쉬 중"
    - echo "==========================="

    # 환경 변수 적용 (main 브랜치인 경우 DockerHub 변경)
    - |
      if [ "$CI_COMMIT_REF_NAME" == "main" ]; then
        export IMAGE_NAME="$IMAGE_BE_MAIN"
        export DOCKERHUB_ID="$DOCKERHUB_USERNAME_MAIN"
        export DOCKERHUB_PW="$DOCKERHUB_PASSWORD_MAIN"
      fi

    # DockerHub로 이미지 PUSH
    - echo $DOCKERHUB_PW | docker login -u $DOCKERHUB_ID --password-stdin
    - docker push $IMAGE_NAME:$CI_COMMIT_SHA
    - docker push $IMAGE_NAME:latest

  dependencies:
    - build_BE
  rules:
  - if: $CI_PIPELINE_SOURCE == "push" && ($CI_COMMIT_BRANCH == "BE/dev" || $CI_COMMIT_BRANCH == "ma
    when: always
  - if: $CI_COMMIT_BRANCH == "CICD/test" || $CI_COMMIT_BRANCH == "CICD/main"
    when: always
  - when: never


##########################
# 3. cleanup JOB 단계  #
##########################

cleanup:
  stage: cleanup
  image: docker:27.4.0
  variables:
    FE_IMAGE_NAME: "$IMAGE_FE"
    BE_IMAGE_NAME: "$IMAGE_BE"
  script:
    - echo "=============================="
    - echo "   도커 이미지 및 캐시 정리 중..."
    - echo "=============================="

    # 불필요한 이미지 제거
    - docker image prune -f

    # 환경 변수 적용 (main 브랜치인 경우 DockerHub 변경)
    - |
      if [ "$CI_COMMIT_REF_NAME" == "main" ]; then
        export FE_IMAGE_NAME="$IMAGE_FE_MAIN"
        export BE_IMAGE_NAME="$IMAGE_BE_MAIN"
      fi

    # 특정 이미지 태그 제거
    - docker rmi $FE_IMAGE_NAME:$CI_COMMIT_SHA || true
    - docker rmi $FE_IMAGE_NAME:latest || true
    - docker rmi $BE_IMAGE_NAME:$CI_COMMIT_SHA || true
    - docker rmi $BE_IMAGE_NAME:latest || true

    # GitLab Runner가 사용하는 불필요한 빌드 파일 정리
    - rm -rf /builds/* || true
  dependencies:
    - push_FE
    - push_BE
  rules:
    - if: $CI_PIPELINE_SOURCE == "push" && ($CI_COMMIT_REF_NAME == "FE/dev" || $CI_COMMIT_REF_NAME =
      when: always
    - if: $CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_REF_NAME == "main"
      when: always
    - if: $CI_COMMIT_BRANCH == "CICD/test" || $CI_COMMIT_BRANCH == "CICD/main"
      when: always
    - when: never
```

```
##################
# 4. deploy 단계   #
##################

deploy_main:
  stage: deploy
  variables:
    DB_URL_MAIN: "$DB_URL_MAIN"
    DB_DRIVER_MAIN: "$DB_DRIVER"
    DB_USERNAME_MAIN: "$DB_USERNAME"
    DB_PASSWORD_MAIN: "$DB_PASSWORD_MAIN"
    REDIS_HOST_MAIN: "$REDIS_HOST"
    WEBCLIENT_BASE_URL: "$WEBCLIENT_BASE_URL"
    S3_BUCKET_NAME: "$S3_BUCKET_NAME"
    S3_REGION: "$S3_REGION"
    S3_CREDENTIALS_ACCESS_KEY: "$S3_CREDENTIALS_ACCESS_KEY"
    S3_CREDENTIALS_SECRET_KEY: "$S3_CREDENTIALS_SECRET_KEY"

  script:
    - echo "====================================="
    - echo "        main-server로 배포중..."
    - echo "====================================="

    # main-server .env 파일 생성
    - echo "DB_URL_MAIN=$DB_URL_MAIN" >> .env
    - echo "DB_DRIVER_MAIN=$DB_DRIVER_MAIN" >> .env
    - echo "DB_USERNAME_MAIN=$DB_USERNAME_MAIN" >> .env
    - echo "DB_PASSWORD_MAIN=$DB_PASSWORD_MAIN" >> .env
    - echo "REDIS_HOST_MAIN=$REDIS_HOST_MAIN" >> .env
    - echo "WEBCLIENT_BASE_URL=$WEBCLIENT_BASE_URL" >> .env
    - echo "S3_BUCKET_NAME=$S3_BUCKET_NAME" >> .env
    - echo "S3_REGION=$S3_REGION" >> .env
    - echo "S3_CREDENTIALS_ACCESS_KEY=$S3_CREDENTIALS_ACCESS_KEY" >> .env
    - echo "S3_CREDENTIALS_SECRET_KEY=$S3_CREDENTIALS_SECRET_KEY" >> .env


    #.env 파일을 main-server로 전송
    - scp -i "$MAIN_SERVER_KEY" .env "$MAIN_SERVER_URI:/home/ubuntu/"
    # SSH를 통해 main-server에서 명령어 실행(터미널 생성)
    - |
      ssh -tt -i "$MAIN_SERVER_KEY" "$MAIN_SERVER_URI" << 'EOF'

        # 컨테이너 off
        docker compose down
        # 기존 이미지 제거
        docker rmi $(docker images -q)
        # 컨테이너 on .env 참조
        docker compose --env-file /home/ubuntu/.env up --build -d
        # 나가기
        exit
      EOF

    - echo "====================================="
    - echo "          배포 완료!!!"
    - echo "====================================="
  dependencies:
    - cleanup
  rules:
    - if: $CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_REF_NAME == "main"
      when: always
    - if: $CI_COMMIT_BRANCH == "CICD/main"
      when: always
    - when: never

deploy_test:
  stage: deploy
  variables:
    DB_URL_TEST: "$DB_URL"
    DB_USERNAME_TEST: "$DB_USERNAME"
    DB_PASSWORD_TEST: "$DB_PASSWORD"
    DB_DRIVER_TEST: "$DB_DRIVER"
    REDIS_HOST_TEST: "$REDIS_HOST"
    WEBCLIENT_BASE_URL: "$WEBCLIENT_BASE_URL"
    S3_BUCKET_NAME: "$S3_BUCKET_NAME"
    S3_REGION: "$S3_REGION"
    S3_CREDENTIALS_ACCESS_KEY: "$S3_CREDENTIALS_ACCESS_KEY"
    S3_CREDENTIALS_SECRET_KEY: "$S3_CREDENTIALS_SECRET_KEY"
  script:
    - echo "====================================="
    - echo "        test_server로 배포중..."
    - echo "====================================="

    #  디렉토리 이동
    - cd /home/ubuntu/
    # test-server .env 파일 생성
    - echo "DB_URL_TEST=$DB_URL_TEST" >> .env
    - echo "DB_DRIVER_TEST=$DB_DRIVER_TEST" >> .env
    - echo "DB_USERNAME_TEST=$DB_USERNAME_TEST" >> .env
    - echo "DB_PASSWORD_TEST=$DB_PASSWORD_TEST" >> .env
    - echo "REDIS_HOST_TEST=$REDIS_HOST_TEST" >> .env
    - echo "WEBCLIENT_BASE_URL=$WEBCLIENT_BASE_URL" >> .env
    - echo "S3_BUCKET_NAME=$S3_BUCKET_NAME" >> .env
    - echo "S3_REGION=$S3_REGION" >> .env
    - echo "S3_CREDENTIALS_ACCESS_KEY=$S3_CREDENTIALS_ACCESS_KEY" >> .env
    - echo "S3_CREDENTIALS_SECRET_KEY=$S3_CREDENTIALS_SECRET_KEY" >> .env

    # 컨테이너 on .env 참조
    - docker compose --env-file /home/ubuntu/.env up --build -d

    - echo "====================================="
    - echo "          배포 완료!!!"
    - echo "====================================="
  dependencies:
    - cleanup
  rules:
    - if: $CI_PIPELINE_SOURCE == "push" && ($CI_COMMIT_REF_NAME == "FE/dev" || $CI_COMMIT_REF_NAME =
      when: always
    - if: $CI_COMMIT_BRANCH == "CICD/test"
      when: always
    - when: never
```

- ***AWS EC2 (main-server)***

  ▼ **Docker 및 Docker-compose 설치**

```
$ sudo apt-get update && \
  sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-common && \
  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - && \
  sudo apt-key fingerprint 0EBFCD88 && \
  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stab
  sudo apt-get update && \
  sudo apt-get install -y docker-ce && \
  sudo usermod -aG docker ubuntu && \
  newgrp docker && \
  sudo curl -L "https://github.com/docker/compose/releases/download/2.27.1/docker-compose-$(uname -s)-$(u
  sudo chmod +x /usr/local/bin/docker-compose && \
  sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

▼ 설명

1. **패키지 목록 업데이트 및 필수 패키지 설치**

```
$ sudo apt-get update && \
sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-common
```

- `sudo apt-get update` → 패키지 목록을 최신 상태로 업데이트
- `sudo apt-get install -y ...` → Docker 설치에 필요한 패키지들을 설치
  - `apt-transport-https` : HTTPS를 통한 패키지 다운로드 지원
  - `ca-certificates` : SSL 인증서 관련 패키지
  - `curl` : URL 요청을 위한 도구
  - `software-properties-common` : 추가 저장소를 관리하는 패키지

2. **Docker의 GPG 키 추가 (및 확인)**

- GPG 키 추가

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

 # GPG키 확
 # $ sudo apt-key fingerprint 0EBFCD88
```

- Docker 공식 사이트에서 GPG 키를 다운로드하여 추가 (패키지 신뢰성을 보장)

3. **Docker 공식 저장소 추가**

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs
```

- `$(lsb_release -cs)` → 현재 사용 중인 Ubuntu의 코드네임을 가져와서 저장소 URL에 적용
- `add-apt-repository` → Docker 패키지를 Ubuntu 공식 저장소가 아닌 Docker 공식 저장소에서 설치하도록 설정

4. **패키지 목록 다시 업데이트**

```
$ sudo apt-get update
```

5. **Docker (Community Edition)설치**

```
$ sudo apt-get install -y docker-ce
```

6. **현재 사용자를 Docker 그룹에 추가**

```
$ sudo usermod -aG docker ubuntu
```

- `usermod -aG docker ubuntu` → `ubuntu` 사용자를 `docker` 그룹에 추가하여 `sudo` 없이 Docker 명령을 실행할 수 있도록 함

7. **그룹 변경 적용**

```
$ newgrp docker
```

8. **Docker-compose 다운로드**

```
$ sudo curl -L "https://github.com/docker/compose/releases/download/2.27.1/docker-compose-$(uname
```

- `curl -L ...` → Docker Compose 바이너리를 GitHub에서 다운로드하여 `/usr/local/bin/docker-compose` 에 저장
- `$(uname -s)-$(uname -m)` → 현재 OS와 아키텍처에 맞는 Docker Compose 버전을 자동으로 다운로드

9. **실행 권한 부여**

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

- `chmod +x` → `docker-compose` 파일에 실행 권한 추가

10. **심볼릭 링크 생성**

```
$ sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compos
```

- `/usr/local/bin/docker-compose` 를 `/usr/bin/docker-compose` 에 연결하여 어디서든 `docker-compose` 명령을 사용할 수 있도록 설정

▼ CICD 관련 설정

/home/ubuntu/docker-compose.yml 생성

```
# 서비스 정의
services:

  # web-server
  web-server:
    image: ssafyb101main/my-fe:latest
    ports:
      - 80:80
      - 443:443
    volumes:
      - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
      - ./certbot/data:/var/www/certbot
      - ./certbot/conf:/etc/nginx/ssl
      - ./nginx/image-server/data/image_cache:/data/image_cache
  depends_on:
    - app


# web-application-server
app:
  image: ssafyb101main/my-be:latest
  ports:
    - 8080:8080
  environment:
    DB_URL: ${DB_URL_MAIN}  #RDS 엔드포인트
    DB_DRIVER: ${DB_DRIVER_MAIN}
    DB_USERNAME: ${DB_USERNAME_MAIN}
    DB_PASSWORD: ${DB_PASSWORD_MAIN}
    REDIS_HOST: ${REDIS_HOST_MAIN}
    S3_BUCKET_NAME: ${S3_BUCKET_NAME}
    S3_REGION: ${S3_REGION}
    S3_CREDENTIALS_ACCESS_KEY: ${S3_CREDENTIALS_ACCESS_KEY}
    S3_CREDENTIALS_SECRET_KEY: ${S3_CREDENTIALS_SECRET_KEY}
  depends_on:
    cache-server:
      condition: service_healthy

# postgres-db
# main-server 에서는 RDS(postgres)와 연결됩니다.


# cache-server
cache-server:
  image: redis
  ports:
    - 6379:6379
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 5s
    timeout: 30s
    retries: 5


# coturn
coturn:
  image: instrumentisto/coturn
  container_name: coturn
  restart: unless-stopped
  network_mode: "host"  # 네트워크 성능 최적화를 위해 호스트 네트워크 사용
  environment:
    - TURN_PORT=3478
    - TURNS_PORT=5349
    - TURN_USER=longagocoturn:longago123!
      # - static-auth-secret=longago123!
    - REALM=i12b101.p.ssafy.io
    - TURN_EXTERNAL_IP=3.38.94.196/172.26.3.240
  volumes:
    - ./turnserver.conf:/etc/coturn/turnserver.conf
    - ./data:/var/lib/turn
  command: ["turnserver", "-c", "/etc/coturn/turnserver.conf"]
```

▼ **Nginx 관련 설정**

/home/ubuntu/nginx/default.conf 생성

```
proxy_cache_path /data/image_cache levels=1:2 keys_zone=image_cache:100m inactive=7d max_size=50g;

server {
    listen 80;
    listen [::]:80;
    server_name i12b101.p.ssafy.io;

    location /.well-known/acme-challenge {
        allow all;
        root /var/www/certbot;
    }

    # HTTP 요청을 HTTPS로 리디렉션
    return 301 https://$host$request_uri;
}

server {
    listen       443 ssl;
    listen  [::]:443 ssl;
    server_name  i12b101.p.ssafy.io;

    ssl_certificate /etc/nginx/ssl/live/i12b101.p.ssafy.io/fullchain.pem;
    ssl_certificate_key /etc/nginx/ssl/live/i12b101.p.ssafy.io/privkey.pem;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;


    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    # SSL (Certbot)
    location /.well-known/acme-challenge {
        allow all;
    # 챌린지를 저장 위치
        root /var/www/certbot;
    }

    # API
    location /apis/ {
        proxy_pass http://app:8080/;  # Docker 내부의 백엔드 컨테이너
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

    # WebSocket 헤더 설정
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";


        # CORS 설정
        add_header 'Access-Control-Allow-Origin' 'https://i12b101.p.ssafy.io' always;  # 모든 도메인 허용
        add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS, PATCH' always;
        add_header 'Access-Control-Allow-Headers' 'Content-Type, Authorization' always;

        # OPTIONS 요청 처리 (CORS 프리플라이트 요청)
        if ($request_method = 'OPTIONS') {
            add_header 'Access-Control-Allow-Origin' 'https://i12b101.p.ssafy.io' always;
            add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS, PATCH' always;
            add_header 'Access-Control-Allow-Headers' 'Content-Type, Authorization' always;
            add_header 'Content-Length' 0;
            add_header 'Status' 204;
            return 204;
        }
    }
    # image
    location /images/ {

        proxy_cache image_cache;  # 캐시 저장소 사용
        proxy_cache_use_stale error timeout updating;
        proxy_cache_valid 200 7d;
        proxy_pass http://app:8080/;  # Docker 내부의 이미지-서버 컨테이너 (캐시가 없을 시 이동)
        proxy_ignore_headers Cache-Control Expires; # 헤더에 no-cache 무시

        proxy_set_header Host $host; # Host 헤더를 유지
        proxy_set_header X-Real-IP $remote_addr; # 클라이언트의 실제 IP주소를 백엔드 서버로 전달
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for; # 요청이 프록시를 거쳐왔다는 정보를 백엔드에 
        proxy_set_header X-Forwarded-Proto $scheme; # 클라이언트가 요청을 보낸 프로토콜(HTTP 또는 HTTPS)을 백엔드
        add_header X-Cache-Status $upstream_cache_status; # 캐시를 사용했는지 혹인

        # CORS 설정
        add_header 'Access-Control-Allow-Origin' 'https://i12b101.p.ssafy.io' always;
        add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS, PATCH' always;
        add_header 'Access-Control-Allow-Headers' 'Content-Type, Authorization' always;

        # OPTIONS 요청 처리 (CORS 프리플라이트 요청)
        if ($request_method = 'OPTIONS') {
            add_header 'Access-Control-Allow-Origin' 'https://i12b101.p.ssafy.io' always;
            add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS, PATCH' always;
            add_header 'Access-Control-Allow-Headers' 'Content-Type, Authorization' always;
            add_header 'Content-Length' 0;
            add_header 'Status' 204;
            return 204;
        }
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }

    # proxy the PHP scripts to Apache listening on 127.0.0.1:80
    #
    #location ~ \.php$ {
    #    proxy_pass   http://127.0.0.1;
    #}
    # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
    #
    #location ~ \.php$ {
    #    root           html;
    #    fastcgi_pass   127.0.0.1:9000;
    #    fastcgi_index  index.php;
    #    fastcgi_param  SCRIPT_FILENAME  /scripts$fastcgi_script_name;
    #    include        fastcgi_params;
    #}

    # deny access to .htaccess files, if Apache's document root
    # concurs with nginx's one
    #
    #location ~ /\.ht {
    #    deny  all;
    #}
}
```

▼ **SSL 발급 관련 설정**

▼ /home/ubuntu/certbot/docker-compose.yml (도메인과 이메일 변경 필수)

```
services:
  certbot:
    image: certbot/certbot:latest
    command: certonly --webroot --webroot-path=/var/www/certbot --email ssafy.b101@gmail.com --agree-tc
    volumes:
      - ~/certbot/conf:/etc/letsencrypt:rw
      - ~/certbot/logs:/var/log/letsencrypt:rw
      - ~/certbot/data:/var/www/certbot:rw
```

```
$ cd /home/ubuntu/certbot && \ docker compose down
```

▼ certbot 설치 및 SSL 발급

```

$ cd /home/ubuntu/certbot && \ docker compose up -d
```

▼ **coturn 관련 설정**

▼ /home/ubuntu/coturn/turnserver.conf 생성

({ID}:{PW}, {IP주소} , {public IP}/{private IP} 수정 필요)

```
# Coturn 기본 설정
listening-port=3478
tls-listening-port=5349
fingerprint
lt-cred-mech
realm= {IP 주소}
server-name=longago
userdb=/var/lib/turn/turndb

# 사용자 인증 방식
# use-auth-secret
# static-auth-secret=longago123!
```

```
user= {ID}:{PW}
# lt-cred-mech
# userdb=/etc/turnuserdb.conf
# realm=i12b101.p.ssafy.io
# no-auth

# 네트워크 인터페이스
listening-ip=0.0.0.0
relay-ip=0.0.0.0
external-ip={public IP}/{private IP}

# TURN 릴레이 설정
total-quota=100
stale-nonce
no-multicast-peers

# 로그 설정
log-file=/var/log/turnserver.log
verbose
```

- 환경 변수 추가 필요
  - VITE_TURN_ID
  - VITE_TURN_PW

- ***AWS EC2 (runner-server & test-server)***

  ▼ **Docker, Docker-compose 설치 및 세팅 (main-server와 동일)**

```
$ sudo apt-get update && \
   sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-common && \
   curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - && \
   sudo apt-key fingerprint 0EBFCD88 && \
   sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stab
   sudo apt-get update && \
   sudo apt-get install -y docker-ce && \
   sudo usermod -aG docker ubuntu && \
   newgrp docker && \
   sudo curl -L "https://github.com/docker/compose/releases/download/2.27.1/docker-compose-$(uname -s)-$(u
   sudo chmod +x /usr/local/bin/docker-compose && \
   sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

  ▼ 설명

  1. **패키지 목록 업데이트 및 필수 패키지 설치**

```
$ sudo apt-get update && \
sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-common
```

  - `sudo apt-get update` → 패키지 목록을 최신 상태로 업데이트
  - `sudo apt-get install -y ...` → Docker 설치에 필요한 패키지들을 설치
    - `apt-transport-https` : HTTPS를 통한 패키지 다운로드 지원
    - `ca-certificates` : SSL 인증서 관련 패키지
    - `curl` : URL 요청을 위한 도구
    - `software-properties-common` : 추가 저장소를 관리하는 패키지

2. **Docker의 GPG 키 추가 (및 확인)**
   - GPG 키 추가

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

 # GPG키 확
 # $ sudo apt-key fingerprint 0EBFCD88
```

   - Docker 공식 사이트에서 GPG 키를 다운로드하여 추가 (패키지 신뢰성을 보장)

3. **Docker 공식 저장소 추가**

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs
```

   - `$(lsb_release -cs)` → 현재 사용 중인 Ubuntu의 코드네임을 가져와서 저장소 URL에 적용
   - `add-apt-repository` → Docker 패키지를 Ubuntu 공식 저장소가 아닌 Docker 공식 저장소에서 설치하도록 설정

4. **패키지 목록 다시 업데이트**

```
$ sudo apt-get update
```

5. **Docker (Community Edition)설치**

```
$ sudo apt-get install -y docker-ce
```

6. **현재 사용자를 Docker 그룹에 추가**

```
$ sudo usermod -aG docker ubuntu
```

   - `usermod -aG docker ubuntu` → `ubuntu` 사용자를 `docker` 그룹에 추가하여 `sudo` 없이 Docker 명령을 실행할 수 있도록 함

7. **그룹 변경 적용**

```
$ newgrp docker
```

8. **Docker-compose 다운로드**

```
$ sudo curl -L "https://github.com/docker/compose/releases/download/2.27.1/docker-compose-$(uname
```

   - `curl -L ...` → Docker Compose 바이너리를 GitHub에서 다운로드하여 `/usr/local/bin/docker-compose`에 저장
   - `$(uname -s)-$(uname -m)` → 현재 OS와 아키텍처에 맞는 Docker Compose 버전을 자동으로 다운로드

9. **실행 권한 부여**

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

   - `chmod +x` → `docker-compose` 파일에 실행 권한 추가

10. **심볼릭 링크 생성**

```
$ sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compos
```

- `/usr/local/bin/docker-compose` 를 `/usr/bin/docker-compose` 에 연결하여 어디서든 `docker-compose` 명령을 사용할 수 있도록 설정

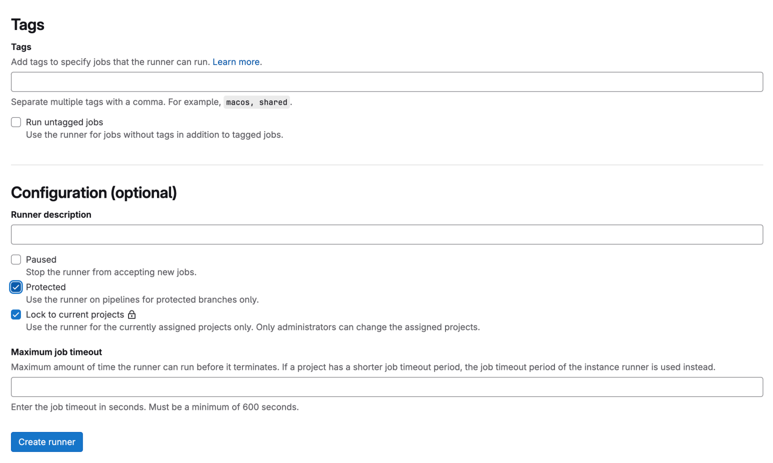
▼ gitLab-runner 설치 및 설정

▼ 설치

1. gitLab → Settings → CI/CD → Runner
2. 루트로 변경

```
$ sudo su
```

3. 다음과 같이 설정 후 **[create runner]**



Tags

Tags

Add tags to specify jobs that the runner can run. Learn more.

Separate multiple tags with a comma. For example, `macos, shared`.

[ ] Run untagged jobs
Use the runner for jobs without tags in addition to tagged jobs.

Configuration (optional)

Runner description

[ ] Paused
Stop the runner from accepting new jobs.

[✓] Protected
Use the runner on pipelines for protected branches only.

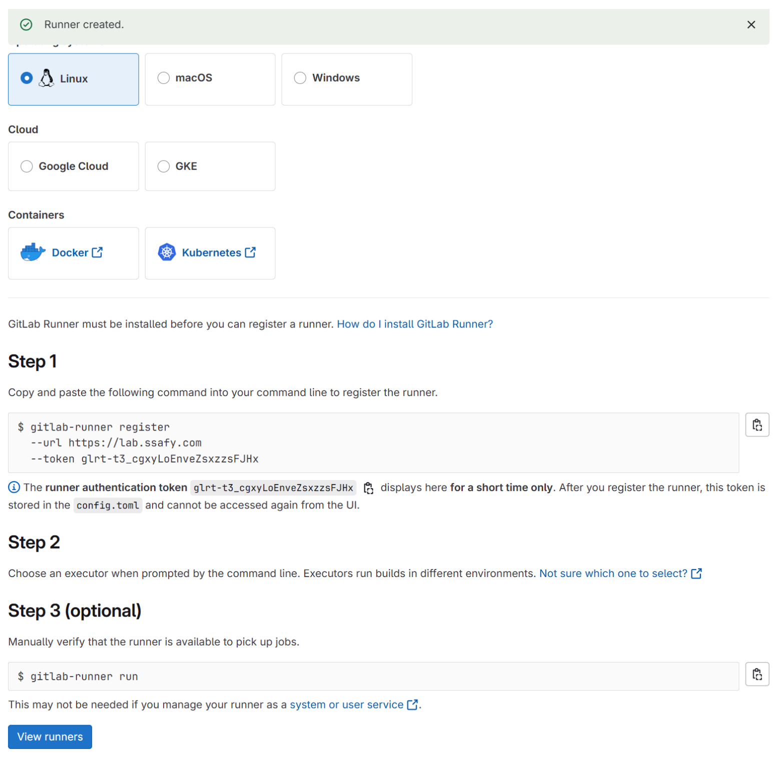
[✓] Lock to current projects
Use the runner for the currently assigned projects only. Only administrators can change the assigned projects.

Maximum job timeout

Maximum amount of time the runner can run before it terminates. If a project has a shorter job timeout period, the job timeout period of the instance runner is used instead.

Enter the job timeout in seconds. Must be a minimum of 600 seconds.

4. Linux 방식으로 설치(Executor은 shell로 설정)



Runner created.

[✓ Linux] macOS Windows

Cloud

Google Cloud GKE

Containers

Docker Kubernetes

GitLab Runner must be installed before you can register a runner. How do I install GitLab Runner?

Step 1

Copy and paste the following command into your command line to register the runner.

```
$ gitlab-runner register
  --url https://lab.ssafy.com
  --token glrt-t3_cgxyLoEnveZsxzzsFJHx
```

The **runner authentication token** `glrt-t3_cgxyLoEnveZsxzzsFJHx` displays here **for a short time only**. After you register the runner, this token is stored in the `config.toml` and cannot be accessed again from the UI.

Step 2

Choose an executor when prompted by the command line. Executors run builds in different environments. Not sure which one to select?

Step 3 (optional)

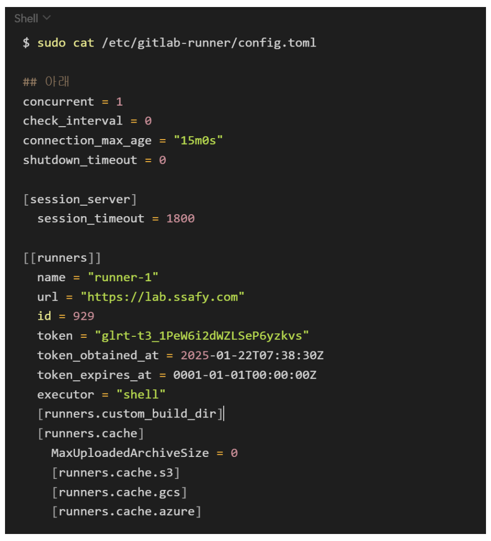
Manually verify that the runner is available to pick up jobs.

```
$ gitlab-runner run
```

This may not be needed if you manage your runner as a system or user service.

5. 해당 위치 파일에 잘 등록 되었는지 확인합니다.

```
$ sudo cat /etc/gitlab-runner/config.toml
```



Shell

```
$ sudo cat /etc/gitlab-runner/config.toml

## 아래
concurrent = 1
check_interval = 0
connection_max_age = "15m0s"
shutdown_timeout = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "runner-1"
  url = "https://lab.ssafy.com"
  id = 929
  token = "glrt-t3_1PeW6i2dWZLSeP6yzkvs"
  token_obtained_at = 2025-01-22T07:38:30Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "shell"
  [runners.custom_build_dir]
  [runners.cache]
    MaxUploadedArchiveSize = 0
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
```

정상적으로 gitlab-runner가 설정 되었다면 다음과 같이 나옵니다.

▼ 설정

1. /home/gitlab-runner/{main-server.pem}에 main-server ssh privateKey 넣기
2. 키 소유자 소유 그룹 변경

```
$ sudo chown gitlab-runner:gitlab-runner {main-server.pem}
```

3. 키 권한 변경

```
$ sudo chmod 400 {main-server.pem}
```

4. gitlab-runner가 있는 서버(test-server)에서 main-server로 접속해봅니다.

```
$ sudo su
$ cd /home/gitlab-runner/
$ ssh -i ./{main-server.pem} {main-server사용자}@{main-server 주소}
## fingerprint를 남기고, 접속이 되는지 확인 해줍니다.

## 만약, 접속이 되지 않는다면, main-server가 22번 포트가 열려있는지 확인합니다.
## main-server에 직접 들어갑니다.
$ sudo ufw status
## 22번 포트가 열려있는지 확인
## 닫겨있다면,
$ sudo ufw allow 22
$ sudo ufw enable
## 22 번 포트가 열렸는지 확인
$ sudo ufw status

## 이는 main-server로 배포하는 방법이 됩니다.
```

5. /home/ubuntu 권한 변경 (gitlab-runner가 test-server에 배포할 수 있도록 합니다)

```
$ sudo chmod 777 /home/ubuntu/
```

▼ CICD 관련 설정

/home/ubuntu/docker-compose.yml 생성 **(test-server&runner-server)**

```
# 서비스 정의
services:

  # web-server
  web-server:
    image: ssafyb101/my-fe:latest
    ports:
      - 80:80
    volumes:
      - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
      - ./nginx/data/image_cache:/data/image_cache
    depends_on:
      - app


  # web-application-server
  app:
    image: ssafyb101/my-be:latest
    ports:
      - 8080:8080
    environment:
      DB_URL: ${DB_URL_TEST}
      DB_DRIVER: ${DB_DRIVER_TEST}
      DB_USERNAME: ${DB_USERNAME_TEST}
      DB_PASSWORD: ${DB_PASSWORD_TEST}
      REDIS_HOST: ${REDIS_HOST_TEST}
      WEBCLIENT_BASE_URL: ${WEBCLIENT_BASE_URL}
      S3_BUCKET_NAME: ${S3_BUCKET_NAME}
      S3_REGION: ${S3_REGION}
      S3_CREDENTIALS_ACCESS_KEY: ${S3_CREDENTIALS_ACCESS_KEY}
      S3_CREDENTIALS_SECRET_KEY: ${S3_CREDENTIALS_SECRET_KEY}
    depends_on:
      db:
        condition: service_healthy
      cache-server:
        condition: service_healthy

  # postgres-db (테스트 단계용, 볼륨X)
  db:
    image: postgres
    ports:
      - 5432:5432
    # 가용 메모리 제한
    shm_size: 128mb
    # 환경 설정
    environment:
      POSTGRES_DB: B101
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: 1q2w3e4r
    healthcheck:
      test: ["CMD", "pg_isready", "-d", "B101", "-U", "postgres"]
      interval: 5s
      timeout: 30s
      retries: 5
    volumes:
      - ./data.sql:/docker-entrypoint-initdb.d/data.sql

  # cache-server
  cache-server:
    image: redis
    ports:
      - 6379:6379
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 30s
      retries: 5
```

▼ **Nginx 관련 설정**

/home/ubuntu/nginx/default.conf 생성

```
proxy_cache_path /data/image_cache levels=1:2 keys_zone=image_cache:100m inactive=7d max_size=50g;

server {
    listen       80;
    listen  [::]:80;
    server_name  3.39.74.250;

    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
    # API
    location /apis/ {
        proxy_pass http://app:8080/;  # Docker 내부의 백엔드 컨테이너
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

    # WebSocket 헤더 설정
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";


        # CORS 설정
        add_header 'Access-Control-Allow-Origin' 'http://localhost:5173' always;  # 모든 도메인 허용
        add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS, PATCH' always;
        add_header 'Access-Control-Allow-Headers' 'Content-Type, Authorization' always;

        # OPTIONS 요청 처리 (CORS 프리플라이트 요청)
        if ($request_method = 'OPTIONS') {
            add_header 'Access-Control-Allow-Origin' 'http://localhost:5173' always;
            add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS, PATCH' always;
            add_header 'Access-Control-Allow-Headers' 'Content-Type, Authorization' always;
            add_header 'Content-Length' 0;
            add_header 'Status' 204;
            return 204;
        }
    }

    # image
    location /images/ {

        proxy_cache image_cache;  # 캐시 저장소 사용
        proxy_cache_use_stale error timeout updating;
        proxy_cache_valid 200 7d;
        proxy_pass http://app:8080/;  # Docker 내부의 이미지-서버 컨테이너 (캐시가 없을 시 이동)
        proxy_ignore_headers Cache-Control Expires; # expires nocache

        proxy_set_header Host $host; # Host 헤더를 유지
        proxy_set_header X-Real-IP $remote_addr; # 클라이언트의 실제 IP주소를 백엔드 서버로 전달
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for; # 요청이 프록시를 거쳐왔다는 정보를 백엔드에
        proxy_set_header X-Forwarded-Proto $scheme; # 클라이언트가 요청을 보낸 프로토콜(HTTP 또는 HTTPS)을 백엔드
        add_header X-Cache-Status $upstream_cache_status; # 캐시를 사용했는지 확인

    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }

    # proxy the PHP scripts to Apache listening on 127.0.0.1:80
    #
    #location ~ \.php$ {
    #    proxy_pass   http://127.0.0.1;
    #}

    # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
    #
    #location ~ \.php$ {
    #    root           html;
    #    fastcgi_pass   127.0.0.1:9000;
    #    fastcgi_index  index.php;
    #    fastcgi_param  SCRIPT_FILENAME  /scripts$fastcgi_script_name;
    #    include        fastcgi_params;
    #}

    # deny access to .htaccess files, if Apache's document rooti
    # concurs with nginx's one
    #
    #location ~ /\.ht {
    #    deny  all;
    #}
}
```

▼ **테스트용 DB 설정**

/home/ubuntu/data.sql 생성

```
CREATE TABLE IF NOT EXISTS ending_card (
        id integer generated by default as identity,
        content varchar(255) not null,
        primary key (id)
);

CREATE TABLE IF NOT EXISTS story_card (
        id integer generated by default as identity,
        attribute varchar(255),
        keyword varchar(255) not null,
        primary key (id)
);

CREATE TABLE IF NOT EXISTS story_card_variants (
        id integer generated by default as identity,
        story_card_id integer not null,
        variant varchar(255) not null,
        primary key (id)
);


TRUNCATE TABLE story_card RESTART IDENTITY CASCADE;
TRUNCATE TABLE ending_card RESTART IDENTITY CASCADE;
TRUNCATE TABLE story_card_variants RESTART IDENTITY CASCADE;

INSERT INTO story_card (id, keyword, attribute) VALUES
-- 인물
(1, '호랑이', '인물'),
(2, '유령', '인물'),
(3, '농부', '인물'),
(4, '상인', '인물'),
(5, '신', '인물'),
(6, '외계인', '인물'),
(7, '박사', '인물'),
(8, '아이돌', '인물'),
(9, '마법사', '인물'),
(10, '마왕', '인물'),
(11, '소년/소녀', '인물'),
(12, '부자', '인물'),
(13, '탐정', '인물'),
(14, '노인', '인물'),
(15, '가난뱅이', '인물'),
(16, '공주', '인물'),
(17, '닌자', '인물'),

-- 사물
(18, '핸드폰', '사물'),
(19, '마차', '사물'),
(20, '인형', '사물'),
(21, '부적', '사물'),
(22, '지도', '사물'),
(23, '가면', '사물'),
(24, '칼', '사물'),
(25, '피리', '사물'),
(26, '지팡이', '사물'),
(27, '태양', '사물'),
(28, '날개', '사물'),
(29, '의자', '사물'),
(30, '시계', '사물'),
(31, '도장', '사물'),
(32, '보석', '사물'),
(33, 'UFO', '사물'),
(34, '덫', '사물'),
(35, '총', '사물'),
(36, '타임머신', '사물'),
(37, '감자', '사물'),

-- 장소
(38, '바다', '장소'),
(39, '다리', '장소'),
(40, '묘지', '장소'),
(41, '식당', '장소'),
(42, '박물관', '장소'),
(43, '비밀통로', '장소'),
(44, '사막', '장소'),
(45, '저택', '장소'),
(46, '천국', '장소'),

-- 사건
(47, '사망', '사건'),
(48, '배신', '사건'),
(49, '계약', '사건'),
(50, '폭발', '사건'),
(51, '승리', '사건'),
(52, '패배', '사건'),
(53, '음모', '사건'),
(54, '공연', '사건'),
(55, '식사', '사건'),
(56, '시간이 지남', '사건'),
(57, '떨어짐', '사건'),
(58, '모험', '사건'),
(59, '희생', '사건'),
(60, '실패', '사건'),
(61, '유혹', '사건'),
(62, '중단/멈춤', '사건'),
(63, '의식', '사건'),
(64, '고백', '사건'),
(65, '짝사랑', '사건'),
(66, '진화', '사건'),
(67, '텔레파시', '사건'),
(68, '노화', '사건'),
(69, '멸망', '사건'),
(70, '결투', '사건'),
(71, '부활', '사건'),

-- 상태
(72, '빛남', '상태'),
(73, '굶주림', '상태'),
(74, '착각', '상태'),
(75, '순진함', '상태'),
(76, '그리움', '상태'),
(77, '집착', '상태'),
(78, '절망', '상태'),
(79, '흥분', '상태'),
(80, '실망', '상태'),
(81, '귀찮음', '상태'),
(82, '자신만만함', '상태'),
(83, '메스꺼움', '상태'),
(84, '들뜸', '상태'),
(85, '격분', '상태'),
(86, '희열', '상태'),
(87, '호기심', '상태'),
(88, '지루함', '상태'),
(89, '간절함', '상태'),
(90, '잠재력', '상태'),
(91, '자존심', '상태'),
(92, '잘생김/이쁨', '상태'),
(93, '거대함', '상태');




INSERT INTO ending_card (id, content) VALUES
                         (1, '적이 죽음에 따라 드디어 그들은 결혼할 수 있었습니다.'),
                         (2, '그리하여 그것은 다시 사람의 모습으로 돌아올 수 있었습니다.'),
                         (3, '그리하여 그녀는 자신을 찾아온 자가 처음부터 괴물이었음을 알았습니다.'),
                         (4, '그에게 걸린 주문이 풀리면서 다음날 그들은 결혼했습니다.'),
                         (5, '그러나 그들이 아무리 열심히 찾아보려 해도 끝끝내 그것을 찾아낼 수가 없었습니다.'),
                         (6, '그래서 마녀는 자신의 가마솥 속으로 사라져버렸습니다.'),
                         (7, '그리고 그녀가 사는 동안은 그것은 절대로 사라지지 않았습니다.'),
                         (8, '매일마다 그는 자신이 반항한 댓가를 보면서 눈물을 흘렸습니다.'),
                         (9, '그리하여 악인은 우물 속으로 던져졌습니다.'),
                         (10, '악랄한 괴물은 패배하여 왕국 밖으로 쫓겨났습니다.'),
                         (11, '이것이 바로 그 왕국이 이토록 특이한 이름을 갖게된 사연입니다.'),
                         (12, '진정한 사랑이 마법을 깨트렸습니다.'),
                         (13, '그들은 같은 무덤에 묻히게 되었으며, 왕국에서는 그들을 애도했습니다.'),
                         (14, '불꽃이 높이 솟구쳐 올라 사악한 장소가 파괴되었습니다.'),
                         (15, '괴물은 파괴되고 다시 한 번 도시는 안전해졌습니다.'),
                         (16, '저주는 예언대로 사라졌습니다.'),
                         (17, '그리하여 주문은 깨어지고 그들은 자유를 얻었습니다.'),
                         (18, '그리하여 그녀는 그녀의 진짜 정체를 공개하고 그들은 결혼했습니다.'),
                         (19, '그리하여 요리사는 축제를 위해 그것을 준비했고 그것은 맛있었습니다.'),
                         (20, '그리고 왕국은 폭군의 지배가 끝나자 기뻐했습니다.'),
                         (21, '왕자의 광기는 치유되었습니다.'),
                         (22, '왕은 자신의 계약을 이행했고 모두가 행복해졌습니다.'),
                         (23, '그리하여 그들은 도난당한 것을 그 주인에게 돌려주었습니다.'),
                         (24, '그리하여 그들은 자신들을 붙잡은 이로부터 탈출해 집으로 달아났습니다.'),
                         (25, '그리하여 정당한 통치자가 다시 한 번 왕위에 올랐습니다.'),
                         (26, '그리하여 왕비는 약속했던 상을 그들에게 주었습니다.'),
                         (27, '그리하여 왕은 마음을 누그러뜨렸고 그들은 결혼했습니다.'),
                         (28, '그리하여 그는 그녀에게 자신이 왕자였음을 말해주고 그들은 오래오래 행복하게 살았습니다.'),
                         (29, '그녀는 자신의 용기로 인해 부유해졌습니다.'),
                         (30, '그녀는 저 끔찍한 곳에서 탈출해 집으로 달아났습니다.'),
                         (31, '그가 끔찍한 범죄를 저질렀음에도 불구하고 왕은 그의 목숨을 살려주기로 했습니다.'),
                         (32, '그들은 때가 되면 왕과 왕비가 될 것입니다.'),
                         (33, '그리고 아마도 그들은 아직도 왕국에서 춤을 추고 있을 것입니다.'),
                         (34, '그리하여 마을이 재건되어 번영하게 되었습니다.'),
                         (35, '그리하여 그는 그녀를 용서하고 그들은 결혼했습니다.'),
                         (36, '그리하여 그는 그의 여동생이 얼마나 충성스러운지를 알았습니다.'),
                         (37, '그건 단지 계모를 보이는대로만 판단하면 안 된다는 것을 보여줄 따름입니다.'),
                         (38, '그는 자신의 방법에서 잘못을 찾았고 이후로는 좋은 삶을 살았습니다.'),
                         (39, '그는 남은 생을 거지로 살았습니다... 그렇게 되어야 하는 거죠.'),
                         (40, '그의 상처는 치유되었으나 그의 마음은 영원히 부서진 채로 남았습니다.'),
                         (41, '그리하여 그들은 위치를 바꿨고 모든 것이 정상으로 돌아갔습니다.'),
                         (42, '그리하여 그들은 다시는 싸우지 않기로 약속했습니다.'),
                         (43, '이것이야말로 용기있고 진실된 마음은 언제나 결국엔 승리한다는 것을 보여주는 이야기입니다.')
                         (44, '이것이야말로 당신은 동료를 선택할 때 항상 신중히 해야 한다는 것을 보여주는 이야기입니다.'),
                         (45, '그리고 그들이 죽었을 때 그들이 가진 것은 그들의 자식들에게 이어졌습니다.'),
                         (46, '그리고 부모는 그들이 오래 전 잃어버린 아이와 다시 만났습니다.'),
                         (47, '그리고 그들은 그들의 사악함과 악행으로 인해 여생을 눈이 먼 채로 살았습니다.'),
                         (48, '그리고 그녀는 그날 이후로 아버지의 충고를 머릿속에 항상 담아두었습니다.'),
                         (49, '그리고 그의 어머니는 그처럼 흔치 않은 선물을 받고서 기뻐했습니다.'),
                         (50, '그래서 그 늙은 여인의 예언은 이루어졌습니다.'),
                         (51, '그녀는 자신의 보물에서 두 번 다시 눈을 떼지 않았습니다.');
```

```
INSERT INTO story_card_variants (story_card_id, variant) VALUES
-- 인물 (변형어 없음)
(1, '호랑이'),
(2, '유령'),
(3, '농부'),
(4, '상인'),
(5, '신'),
(6, '외계인'),
(7, '박사'),
(8, '아이돌'),
(9, '마법사'),
(10, '마왕'),
(11, '소녀'), (11, '소년'),
(12, '부자'),
(13, '탐정'),
(14, '노인'),
(15, '가난뱅이'),
(16, '공주'),
(17, '닌자'),

-- 사물 (변형어 없음)
(18, '핸드폰'),
(19, '마차'),
(20, '인형'),
(21, '부적'),
(22, '지도'),
(23, '가면'),
(24, '칼'),
(25, '피리'),
(26, '지팡이'),
(27, '태양'),
(28, '날개'),
(29, '의자'),
(30, '시계'),
(31, '도장'),
(32, '보석'),
(33, 'UFO'), (33, 'ufo'),
(34, '덫'),
(35, '총'),
(36, '타임머신'),
(37, '감자'),

-- 장소 (변형어 없음)
(38, '바다'),
(39, '다리'),
(40, '묘지'),
(41, '식당'),
(42, '박물관'),
(43, '비밀통로'),
(44, '사막'),
(45, '저택'),
(46, '천국'),
-- 사건 (변형어 없음)
(47, '사망'),
(48, '배신'),
(49, '계약'),
(50, '폭발'),
(51, '승리'),
(52, '패배'),
(53, '음모'),
(54, '공연'),
(55, '식사'),

-- 시간 관련 변형어 추가
(56, '시간이 지남'),
(56, '시간이 흐름'),
(56, '시간이 지나감'),
(56, '시간이 경과함'),
(56, '시간이 흘러감'),
(56, '시간이 지나가다'),
(56, '시간이 흐르다'),
(56, '시간이 경과하였다'),
(56, '시간이 소요되었다'),
(56, '시간이 지났습니다'),
(56, '시간이 좀 갔다'),
(56, '시간이 많이 갔네'),
(56, '시간이 후딱 지나갔다'),

-- 떨어짐 관련 변형어 추가
(57, '떨어짐'),
(57, '추락'),
(57, '낙하'),
(57, '하락'),
(57, '무너짐'),
(57, '넘어짐'),
(57, '미끄러짐'),
(57, '떨어졌다'),
(57, '추락했다'),
(57, '낙하했다'),
(57, '무너졌다'),
(57, '쓰러짐'),
(57, '넘어졌다'),
(57, '떨어트림'),
(57, '떨어져 나감'),
(57, '빠져나감'),
(57, '탈락'),
(57, '낙오'),
(57, '아래로 떨어짐'),

-- 추가 사건
(58, '모험'),
(59, '희생'),
(60, '실패'),
(61, '유혹'),
(62, '중단'), (62, '멈춤'),
(63, '의식'),
(64, '고백'),
(65, '짝사랑'),
(66, '진화'),
(67, '텔레파시'),
(68, '노화'),
(69, '멸망'),
(70, '결투'),
(71, '부활'),

-- 상태 (변형어 포함)
(72, '빛남'), (72, '빛난'), (72, '빛나'), (72, '빛났'), (72, '빛날'), (72, '빛내'),

(73, '굶주림'), (73, '굶주린'), (73, '굶주려'), (73, '굶주리'), (73, '굶주렸'), (73, '굶주릴'),
(73, '굶은'), (73, '굶음'), (73, '굶었'), (73, '굶을'), (73, '굶긴'), (73, '굶어'), (73, '굶겨'),
(73, '배고픔'), (73, '배고픈'), (73, '배고프다'),

(74, '착각'),

(75, '순진함'), (75, '순진한'), (75, '순진하'), (75, '순진했'), (75, '순진할'),

(76, '그리움'), (76, '그리운'), (76, '그리워'), (76, '그리웠'), (76, '그리울'),

(77, '집착'),

(78, '절망'),

(79, '흥분'),

(80, '실망'),

(81, '귀찮음'), (81, '귀찮은'), (81, '귀찮아'), (81, '귀찮았'), (81, '귀찮을'), (81, '귀찮게'),

(82, '자신만만함'),

(83, '메스꺼움'), (83, '메스꺼운'), (83, '메스꺼워'), (83, '메스꺼웠'), (83, '메스꺼울'),

(84, '들뜸'), (84, '들뜬'), (84, '들뜨'), (84, '들떴'), (84, '들뜰'),

(85, '격분'),

(86, '희열'),

(87, '호기심'),

(88, '지루함'), (88, '지루한'), (88, '지루하'), (88, '지루했'), (88, '지루할'),
(88, '따분함'), (88, '무료함'), (88, '심심한'), (88, '심심하다'), (88, '무료하다'),

(89, '간절함'), (89, '간절한'), (89, '간절하'), (89, '간절했'), (89, '간절할'),
(89, '간절하다'), (89, '간절하게'), (89, '간절히 바라다'),

(90, '잠재력'),

(91, '자존심'),

(92, '이쁨'), (92, '이쁜'), (92, '이쁘'), (92, '이쁠'), (92, '이뻤'),
(92, '예쁜'), (92, '예쁘'), (92, '예뻤'),
(92, '잘생긴'), (92, '잘생겨'), (92, '잘생겼'), (92, '잘생길'), (92, '잘생김'),
(92, '이쁘다'), (92, '예쁘다'), (92, '잘생기다'), (92, '예뻐요'), (92, '이뻐요'), (92, '예쁨이 있다'),

(93, '거대함'), (93, '거대한'), (93, '거대하'), (93, '거대했'), (93, '거대할');
```

‼️ test-server(runner-server) 의 메모리가 너무 작다면 가상 메모리 설정 필요 (안정성 향상)

- ***DockerHub 설정***
  - ▼ 메인 서버 DockerHub
    - 환경 변수
      - DOCKERHUB_USERNAME ⇒ 도커 허브 아이디(test-server)
      - DOCKERHUB_PASSWORD ⇒ 도커 허브 패스워드(test-server)
      - IMAGE_FE ⇒ 도커 허브 front-end 레포지토리(test-server)
      - IMAGE_BE ⇒도커 허브 back-end 레포지토리(test-server)
  - ▼ 테스트 서버 DockerHub]
    - 환경 변수
      - DOCKERHUB_USERNAME_MAIN ⇒ 메인 도커 허브 아이디(main-server)
      - DOCKERHUB_PASSWORD_MAIN ⇒ 메인 도커 허브 패스워드(main-server)
      - IMAGE_FE_MAIN ⇒ 도커 허브 front-end 레포지토리(main-server)
      - IMAGE_BE_MAIN ⇒도커 허브 back-end 레포지토리(main-server)

- ***AWS IAM 설정***
  1. IAM 사용자 생성
  2. 권한 추가 → 권한 추가 → 직접 정책 연결 → AmazonS3FullAccess 정책 추가
  2. 생성 시에 받는 IAM KEY를 S3관련 환경 변수 처리
  - 환경 변수 설정 필요
    - S3_CREDENTIALS_ACCESS_KEY
    - S3_CREDENTIALS_SECRET_KEY

- ***AWS S3 설정***
  - 버킷 생성
    - public 차단
  - 환경 변수 설정 필요
    - S3_BUCKET_NAME
    - S3_REGION

- ***AWS RDS 설정***
  - DB 인스턴스 생성 (posrgres)
    - 보안 그룹 인바운드규칙에서 5432 포트를 http 열어둠
  - 환경 변수 설정 필요
    - DB_URL_MAIN ⇒ 엔드포인트
    - DB_USERNAME_MAIN ⇒ DB USERNAME (메인 서버)
    - DB_PASSWORD_MAIN ⇒ DB PASSWORD (메인 서버)

- ***AI 서버 설정***
  - ▼ **Docker 설정**

```
# Docker Pull Command
docker pull comfyui/base

# Docker Build
docker build -t comfyui/base:dev . && docker run --env-file .env -p 8188:8188 comfyui/base:dev

# DOCKER_OPTIONS
-p 1111:1111 -p 8080:8080 -p 8384:8384 -p 72299:72299 -p 8188:8188 -p 8000:8000 -p 8189:8189 -e OPEN_BUTTON_PORT=1111 -e OPEN_BUTTON_TOKEN=1 -e JUPYTER_DIR=/ -e DATA_DIRECTORY=/workspace/ -e PORTAL_CONFIG="localhost:1111:11111:/:Instance Portal|localhost:8188:18188:/:ComfyUI|localhost:8189:8189:/:FastAPI|localhost:8080:18080:/:Jupyter|localhost:8080:18080:/terminals/1:Jupyter Terminal|localhost:8384:18384:/:Syncthing" -e PROVISIONING_SCRIPT=https://raw.githubusercontent.com/vast-ai/base-image/refs/heads/main/derivatives/pytorch/derivatives/comfyui/provisioning_scripts/default.sh -e COMFYUI_ARGS="--disable-auto-launch --listen 0.0.0.0 --port 18188 --enable-cors-header"
```

(Optional)

```
"--disable-auto-launch --listen 0.0.0.0 --port 18188 --enable-cors-header"
```

▼ **poetry (의존성 관리 설정)**

```

# 1. poetry install
curl -sSL https://install.python-poetry.org | python3 -

# 2. poetry가 설치된 .local/bin 경로를 path에 추가
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc

# 3. path 설정 적용
source ~/.bashrc

# 4. poetry 환경 폴더 생성
mkdir gpu-server
cd gpu-server

# 5. poetry 환경 초기화 (초기 환경 설치)
poetry init

# 6. BE와 통신을 위한 fastAPI 설치
poetry add fastapi uvicorn requests python-multipart

# 7.poetry shell 명령어 plugin 설치
poetry self add poetry-plugin-shell

# 8. poetry 가상환경 실행
poetry shell

# 9.llama.cpp 설치
CMAKE_ARGS="-DGGML_CUDA=on" pip install llama-cpp-python
```

▼ **API 실행**

1. **API 실행 파일 다운**

   main.py

   main_low_vram.py

2. **API 실행**

```
# 1. FastAPI를 8189에서 실행
uvicorn main:app --host 0.0.0.0 --port 8189 --reload

# 2. Fastapi 포트 열려있는지 확인
netstat -tulnp | grep 8189
```

▼ AYL_LLM_Node 파일

https://github.com/Yeonri/ComfyUI_LLM_Are_You_Listening

AYL_origin.py

AYL.py

▼ **API 테스트**

API 호출 (Postman)

- **URL:** `http://{ip주소}:{fastapi포트번호}/generate`
- **Headers:** `Content-Type: application/json`
- **Method:** `POST`
- **Body (raw JSON):**

```
{
  "session_id": "abc123",
  "game_mode": 1,
  "user_sentence": "한 호랑이가 살았어요",
  "status": 0
}
```

API 호출 (curl)

```
curl -X POST "http://{ip주소}:{fastapi포트번호}/generate" \
  -H "Content-Type: application/json" \
  -d '{
     "session_id": "abc123",
     "game_mode": 1,
     "user_sentence": "한 호랑이가 살았어요",
     "status": 0
    }'
```

▼ 기타 정보

pip list

▼ pip list

```
(main) root@C.17654865:/workspace$ pip list
Package                  Version
------------------------- ------------
accelerate               1.3.0
aiohappyeyeballs         2.4.4
aiohttp                  3.11.11
aiosignal                1.3.2
albucore                 0.0.16
albumentations           1.4.15
annotated-types          0.7.0
anyio                    4.8.0
asttokens                3.0.0
attrs                    25.1.0
beautifulsoup4           4.13.1
certifi                  2025.1.31
cffi                     1.17.1
charset-normalizer       3.4.1
click                    8.1.8
clip-interrogator        0.6.0
coloredlogs              15.0.1
colour-science           0.4.6
comm                     0.2.2
contourpy                1.3.1
cryptography             44.0.0
cycler                   0.12.1
debugpy                  1.8.12
decorator                5.1.1
Deprecated               1.2.18
diffusers                0.27.2
diskcache                5.6.3
easydict                 1.13
einops                   0.8.0
eval_type_backport       0.2.2
executing                2.2.0
fastapi                  0.115.8
filelock                 3.17.0
flatbuffers              25.1.24
flet                     0.26.0
fonttools                4.56.0
frozenlist               1.5.0
fsspec                   2025.2.0
ftfy                     6.3.1
gdown                    5.2.0
gitdb                    4.0.12
GitPython                3.1.44
h11                      0.14.0
httpcore                 1.0.7
httpx                    0.28.1
huggingface-hub          0.28.1
humanfriendly            10.0
idna                     3.10
imageio                  2.37.0
```

```
importlib_metadata        8.6.1
inquirerpy                0.3.4
ipykernel                 6.29.5
ipython                   8.32.0
ipywidgets                8.1.5
jedi                      0.19.2
Jinja2                    3.1.4
jsonschema                4.23.0
jsonschema-specifications 2024.10.1
jupyter_client            8.6.3
jupyter_core              5.7.2
jupyterlab_widgets        3.0.13
kiwisolver                1.4.8
kornia                    0.8.0
kornia_rs                 0.1.8
lark                      1.2.2
lazy_loader               0.4
llama_cpp_python          0.3.7
llvmlite                  0.44.0
markdown-it-py            3.0.0
MarkupSafe                2.1.5
matplotlib                3.10.0
matplotlib-inline         0.1.7
matrix-client             0.4.0
mdurl                     0.1.2
mpmath                    1.3.0
multidict                 6.1.0
nest-asyncio              1.6.0
networkx                  3.3
numba                     0.61.0
numpy                     1.26.4
nvidia-cublas-cu12        12.1.3.1
nvidia-cuda-cupti-cu12    12.1.105
nvidia-cuda-nvrtc-cu12    12.1.105
nvidia-cuda-runtime-cu12  12.1.105
nvidia-cudnn-cu12         9.1.0.70
nvidia-cufft-cu12         11.0.2.54
nvidia-curand-cu12        10.3.2.106
nvidia-cusolver-cu12      11.4.5.107
nvidia-cusparse-cu12      12.1.0.106
nvidia-nccl-cu12          2.21.5
nvidia-nvjitlink-cu12     12.1.105
nvidia-nvtx-cu12          12.1.105
oauthlib                  3.2.2
onnxruntime               1.20.1
open_clip_torch           2.30.0
opencv-python             4.11.0.86
opencv-python-headless    4.11.0.86
packaging                 24.2
parso                     0.8.4
peft                      0.14.0
pexpect                   4.9.0
pfzy                      0.3.4
pillow                    11.0.0
pip                       24.0
pixeloe                   0.1.1
platformdirs              4.3.6
pooch                     1.8.2
```

```
prompt_toolkit           3.0.50
propcache                0.2.1
protobuf                 5.29.3
psutil                   6.1.1
ptyprocess               0.7.0
pure_eval                0.2.3
pycparser                2.22
pydantic                 2.10.6
pydantic_core            2.27.2
PyGithub                 2.5.0
Pygments                 2.19.1
PyJWT                    2.10.1
PyMatting                1.1.13
PyNaCl                   1.5.0
pyparsing                3.2.1
PySocks                  1.7.1
python-dateutil          2.9.0.post0
python-multipart         0.0.20
PyYAML                   6.0.2
pyzmq                    26.2.1
referencing              0.36.2
regex                    2024.11.6
rembg                    2.0.62
repath                   0.9.0
requests                 2.32.3
rich                     13.9.4
rpds-py                  0.22.3
safetensors              0.5.2
scikit-image             0.25.1
scipy                    1.15.1
sentencepiece            0.2.0
setuptools               65.5.0
shellingham              1.5.4
six                      1.17.0
smmap                    5.0.2
sniffio                  1.3.1
soundfile                0.13.1
soupsieve                2.6
spandrel                 0.4.1
stack-data               0.6.3
starlette                0.45.3
sympy                    1.13.1
tifffile                 2025.1.10
timm                     1.0.14
tokenizers               0.21.0
toml                     0.10.2
torch                    2.5.1+cu121
torchaudio               2.5.1+cu121
torchsde                 0.2.6
torchvision              0.20.1+cu121
tornado                  6.4.2
tqdm                     4.67.1
traitlets                5.14.3
trampoline               0.1.2
transformers             4.48.2
transparent-background   1.3.3
triton                   3.1.0
typer                    0.15.1
```

```
typing_extensions        4.12.2
urllib3                  2.3.0
uv                       0.5.26
uvicorn                  0.34.0
wcwidth                  0.2.13
wget                     3.2
wheel                    0.45.1
widgetsnbextension       4.0.13
wrapt                    1.17.2
xformers                 0.0.29.post1
yarl                     1.18.3
zipp                     3.21.0
```

comfyUI 실행법

```
python main.py --disable-auto-launch --listen 0.0.0.0 --port 18188 --enable-cors-header --use-pytorch-cross-attention --cuda-malloc --highvram --fast
```

```
python main.py --disable-auto-launch --listen 0.0.0.0 --port 18188 --enable-cors-header --use-pytorch-cross-attention --cuda-malloc --fast
```

```
python main.py --disable-auto-launch --listen 0.0.0.0 --port 18188 --enable-cors-header --use-pytorch-cross-attention --cuda-malloc --fast
```

text_concatenate 코드 수정

> ComfyUI/custom_nodes/pr-was-node-suite-comfyui-47064894/WAS_Node_Suite.py

▼ 코드

```
# By WASasquatch (Discord: WAS#0263)
#
# Copyright 2023 Jordan Thompson (WASasquatch)
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to
# deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
```

```
from PIL import Image, ImageFilter, ImageEnhance, ImageOps, ImageDraw, ImageChops, ImageFont
from PIL.PngImagePlugin import PngInfo
from io import BytesIO
from typing import Optional, Union, List
from urllib.request import urlopen
import comfy.diffusers_convert
import comfy.samplers
import comfy.sd
import comfy.utils
import comfy.clip_vision
import comfy.model_management
import folder_paths as comfy_paths
from comfy_extras.chainner_models import model_loading
import ast
import glob
import hashlib
import json
import nodes
import math
import numpy as np
from numba import jit
import os
import random
import re
import requests
import socket
import subprocess
import sys
import datetime
import time
import torch
from tqdm import tqdm

p310_plus = (sys.version_info >= (3, 10))

MANIFEST = {
    "name": "WAS Node Suite",
    "version": (2,2,2),
    "author": "WASasquatch",
    "project": "https://github.com/WASasquatch/was-node-suite-comfyui",
    "description": "An extensive node suite for ComfyUI with over 180 new nodes",
}

sys.path.insert(0, os.path.join(os.path.dirname(os.path.realpath(__file__)), "was_node_suite_comfyui"))
sys.path.append(comfy_paths.base_path)

#! SYSTEM HOOKS

class cstr(str):
    class color:
        END = '\33[0m'
        BOLD = '\33[1m'
        ITALIC = '\33[3m'
        UNDERLINE = '\33[4m'
        BLINK = '\33[5m'
        BLINK2 = '\33[6m'
        SELECTED = '\33[7m'

        BLACK = '\33[30m'
        RED = '\33[31m'
        GREEN = '\33[32m'
        YELLOW = '\33[33m'
        BLUE = '\33[34m'
        VIOLET = '\33[35m'
        BEIGE = '\33[36m'
        WHITE = '\33[37m'

        BLACKBG = '\33[40m'
        REDBG = '\33[41m'
        GREENBG = '\33[42m'
        YELLOWBG = '\33[43m'
        BLUEBG = '\33[44m'
        VIOLETBG = '\33[45m'
        BEIGEBG = '\33[46m'
        WHITEBG = '\33[47m'

        GREY = '\33[90m'
        LIGHTRED = '\33[91m'
        LIGHTGREEN = '\33[92m'
        LIGHTYELLOW = '\33[93m'
        LIGHTBLUE = '\33[94m'
        LIGHTVIOLET = '\33[95m'
        LIGHTBEIGE = '\33[96m'
        LIGHTWHITE = '\33[97m'

        GREYBG = '\33[100m'
        LIGHTREDBG = '\33[101m'
        LIGHTGREENBG = '\33[102m'
        LIGHTYELLOWBG = '\33[103m'
        LIGHTBLUEBG = '\33[104m'
        LIGHTVIOLETBG = '\33[105m'
        LIGHTBEIGEBG = '\33[106m'
        LIGHTWHITEBG = '\33[107m'

        @staticmethod
        def add_code(name, code):
            if not hasattr(cstr.color, name.upper()):
                setattr(cstr.color, name.upper(), code)
            else:
                raise ValueError(f"'cstr' object already contains a code with the name '{name}'.")

    def __new__(cls, text):
        return super().__new__(cls, text)

    def __getattr__(self, attr):
        if attr.lower().startswith("_cstr"):
            code = getattr(self.color, attr.upper().lstrip("_cstr"))
            modified_text = self.replace(f"__{attr[1:]}__", f"{code}")
            return cstr(modified_text)
        elif attr.upper() in dir(self.color):
            code = getattr(self.color, attr.upper())
            modified_text = f"{code}{self}{self.color.END}"
            return cstr(modified_text)
        elif attr.lower() in dir(cstr):
            return getattr(cstr, attr.lower())
        else:
            raise AttributeError(f"'cstr' object has no attribute '{attr}'")

    def print(self, **kwargs):
        print(self, **kwargs)

#! MESSAGE TEMPLATES
cstr.color.add_code("msg", f"{cstr.color.BLUE}WAS Node Suite: {cstr.color.END}")
cstr.color.add_code("warning", f"{cstr.color.BLUE}WAS Node Suite {cstr.color.LIGHTYELLOW}Warning:
{cstr.color.END}")
cstr.color.add_code("error", f"{cstr.color.RED}WAS Node Suite {cstr.color.END}Error: {cstr.color.END}")

#! GLOBALS
NODE_FILE = os.path.abspath(__file__)
MIDAS_INSTALLED = False
CUSTOM_NODES_DIR = comfy_paths.folder_names_and_paths["custom_nodes"][0][0]
MODELS_DIR =  comfy_paths.models_dir
WAS_SUITE_ROOT = os.path.dirname(NODE_FILE)
WAS_CONFIG_DIR = os.environ.get('WAS_CONFIG_DIR', WAS_SUITE_ROOT)
WAS_DATABASE = os.path.join(WAS_CONFIG_DIR, 'was_suite_settings.json')
WAS_HISTORY_DATABASE = os.path.join(WAS_CONFIG_DIR, 'was_history.json')
WAS_CONFIG_FILE = os.path.join(WAS_CONFIG_DIR, 'was_suite_config.json')
STYLES_PATH = os.path.join(WAS_CONFIG_DIR, 'styles.json')
DEFAULT_NSP_PANTRY_PATH = os.path.join(WAS_CONFIG_DIR, 'nsp_pantry.json')
ALLOWED_EXT = ('.jpeg', '.jpg', '.png',
                        '.tiff', '.gif', '.bmp', '.webp')


#! INSTALLATION CLEANUP

# Delete legacy nodes
legacy_was_nodes = ['fDOF_WAS.py', 'Image_Blank_WAS.py', 'Image_Blend_WAS.py', 'Image_Canny_Fil
ter_WAS.py', 'Canny_Filter_WAS.py', 'Image_Combine_WAS.py', 'Image_Edge_Detection_WAS.py', 'Imag
e_Film_Grain_WAS.py', 'Image_Filters_WAS.py',
                    'Image_Flip_WAS.py', 'Image_Nova_Filter_WAS.py', 'Image_Rotate_WAS.py', 'Image_Style_F
ilter_WAS.py', 'Latent_Noise_Injection_WAS.py', 'Latent_Upscale_WAS.py', 'MiDaS_Depth_Approx_WAS.
py', 'NSP_CLIPTextEncoder.py', 'Samplers_WAS.py']
legacy_was_nodes_found = []

if os.path.basename(CUSTOM_NODES_DIR) == 'was-node-suite-comfyui':
    legacy_was_nodes.append('WAS_Node_Suite.py')

f_disp = False
node_path_dir = os.getcwd()+os.sep+'ComfyUI'+os.sep+'custom_nodes'+os.sep
for f in legacy_was_nodes:
    file = f'{node_path_dir}{f}'
    if os.path.exists(file):
        if not f_disp:
            cstr("Found legacy nodes. Archiving legacy nodes...").msg.print()
            f_disp = True
        legacy_was_nodes_found.append(file)
if legacy_was_nodes_found:
    import zipfile
    from os.path import basename
    archive = zipfile.ZipFile(
        f'{node_path_dir}WAS_Legacy_Nodes_Backup_{round(time.time())}.zip', "w")
    for f in legacy_was_nodes_found:
        archive.write(f, basename(f))
        try:
            os.remove(f)
        except OSError:
            pass
    archive.close()
if f_disp:
    cstr("Legacy cleanup complete.").msg.print()

#! WAS SUITE CONFIG

was_conf_template = {
                    "run_requirements": True,
                    "suppress_uncomfy_warnings": True,
                    "show_startup_junk": True,
                    "show_inspiration_quote": True,
                    "text_nodes_type": "STRING",
                    "webui_styles": None,
                    "webui_styles_persistent_update": True,
                    "sam_model_vith_url": "https://dl.fbaipublicfiles.com/segment_anything/sam_vit_h_4b8939.
pth",
                    "sam_model_vitl_url": "https://dl.fbaipublicfiles.com/segment_anything/sam_vit_l_0b3195.pt
h",
                    "sam_model_vitb_url": "https://dl.fbaipublicfiles.com/segment_anything/sam_vit_b_01ec64.
pth",
                    "history_display_limit": 36,
                    "use_legacy_ascii_text": False,
                    "ffmpeg_bin_path": "/path/to/ffmpeg",
                    "ffmpeg_extra_codecs": {
                        "avc1": ".mp4",
                        "h264": ".mkv",
                    },
                    "wildcards_path": os.path.join(WAS_SUITE_ROOT, "wildcards"),
                    "wildcard_api": True,
                }

# Create, Load, or Update Config

def getSuiteConfig():
    global was_conf_template
    try:
        with open(WAS_CONFIG_FILE, "r") as f:
            was_config = json.load(f)
    except OSError as e:
        cstr(f"Unable to load conf file at `{WAS_CONFIG_FILE}`. Using internal config template.").error.prin
t()
        return was_conf_template
    except Exception as e:
        cstr(f"Unable to load conf file at `{WAS_CONFIG_FILE}`. Using internal config template.").error.prin
t()
        return was_conf_template
    return was_config
    return was_config

def updateSuiteConfig(conf):
    try:
        with open(WAS_CONFIG_FILE, "w", encoding='utf-8') as f:
            json.dump(conf, f, indent=4)
    except OSError as e:
        print(e)
        return False
    except Exception as e:
        print(e)
        return False
    return True

if not os.path.exists(WAS_CONFIG_FILE):
    if updateSuiteConfig(was_conf_template):
        cstr(f'Created default conf file at `{WAS_CONFIG_FILE}`.').msg.print()
        was_config = getSuiteConfig()
    else:
        cstr(f"Unable to create default conf file at `{WAS_CONFIG_FILE}`. Using internal config templat
e.").error.print()
        was_config = was_conf_template

else:
    was_config = getSuiteConfig()

    update_config = False
    for sett_ in was_conf_template.keys():
        if not was_config.__contains__(sett_):
            was_config.update({sett_: was_conf_template[sett_]})
            update_config = True

    if update_config:
        updateSuiteConfig(was_config)

# WAS Suite Locations Debug
if was_config.__contains__('show_startup_junk'):
    if was_config['show_startup_junk']:
        cstr(f"Running At: {NODE_FILE}")
        cstr(f"Running From: {WAS_SUITE_ROOT}")

# Check Write Access
if not os.access(WAS_SUITE_ROOT, os.W_OK) or not os.access(MODELS_DIR, os.W_OK):
    cstr(f"There is no write access to `{WAS_SUITE_ROOT}` or `{MODELS_DIR}`. Write access is require
d!").error.print()
    exit

# SET TEXT TYPE
TEXT_TYPE = "STRING"
if was_config and was_config.__contains__('text_nodes_type'):
    if was_config['text_nodes_type'].strip() != '':
        TEXT_TYPE = was_config['text_nodes_type'].strip()
if was_config and was_config.__contains__('use_legacy_ascii_text'):
    if was_config['use_legacy_ascii_text']:
        TEXT_TYPE = "ASCII"
        cstr("use_legacy_ascii_text is `True` in `was_suite_config.json`. `ASCII` type is deprecated and the
default will be `STRING` in the future.").warning.print()

# Convert WebUI Styles - TODO: Convert to PromptStyles class
if was_config.__contains__('webui_styles'):

    if was_config['webui_styles'] not in [None, 'None', 'none', '']:

        webui_styles_file = was_config['webui_styles']
        if was_config.__contains__('webui_styles_persistent_update'):
            styles_persist = was_config['webui_styles_persistent_update']
        else:
            styles_persist = True

        if webui_styles_file not in [None, 'none', 'None', ''] and os.path.exists(webui_styles_file):

            cstr(f"Importing styles from `{webui_styles_file}`.").msg.print()

            import csv

            styles = {}
            with open(webui_styles_file, "r", encoding="utf-8-sig", newline='') as file:
                reader = csv.DictReader(file)
                for row in reader:
                    prompt = row.get("prompt") or row.get("text", "") # Old files
                    negative_prompt = row.get("negative_prompt", "")
                    styles[row["name"]] = {
                        "prompt": prompt,
                        "negative_prompt": negative_prompt
                    }

            if styles:
                if not os.path.exists(STYLES_PATH) or styles_persist:
                    with open(STYLES_PATH, "w", encoding='utf-8') as f:
                        json.dump(styles, f, indent=4)

            del styles

            cstr(f"Styles import complete.").msg.print()

        else:
            cstr(f"Styles file `{webui_styles_file}` does not exist.").error.print()


#! SUITE SPECIFIC CLASSES & FUNCTIONS

# Freeze PIP modules
def packages(versions=False):
    import sys
    import subprocess
    try:
        result = subprocess.check_output([sys.executable, '-m', 'pip', 'freeze'], stderr=subprocess.STDO
UT)
        lines = result.decode().splitlines()
        return [line if versions else line.split('==')[0] for line in lines]
    except subprocess.CalledProcessError as e:
        print("An error occurred while fetching packages:", e.output.decode())
        return []

def install_package(package, uninstall_first: Union[List[str], str] = None):
    if os.getenv("WAS_BLOCK_AUTO_INSTALL", 'False').lower() in ('true', '1', 't'):
        cstr(f"Preventing package install of '{package}' due to WAS_BLOCK_INSTALL env").msg.print()
    else:
        if uninstall_first is None:
            return
        if isinstance(uninstall_first, str):
            uninstall_first = [uninstall_first]

        cstr(f"Uninstalling {', '.join(uninstall_first)}..")
        subprocess.check_call([sys.executable, '-s', '-m', 'pip', 'uninstall', *uninstall_first])
        cstr("Installing package...").msg.print()
        subprocess.check_call([sys.executable, '-s', '-m', 'pip', '-q', 'install', package])

# Tensor to PIL
def tensor2pil(image):
    return Image.fromarray(np.clip(255. * image.cpu().numpy().squeeze(), 0, 255).astype(np.uint8))

# PIL to Tensor
def pil2tensor(image):
    return torch.from_numpy(np.array(image).astype(np.float32) / 255.0).unsqueeze(0)

# PIL Hex
def pil2hex(image):
    return hashlib.sha256(np.array(tensor2pil(image)).astype(np.uint16).tobytes()).hexdigest()

# PIL to Mask
def pil2mask(image):
    image_np = np.array(image.convert("L")).astype(np.float32) / 255.0
    mask = torch.from_numpy(image_np)
    return 1.0 - mask

# Mask to PIL
def mask2pil(mask):
    if mask.ndim > 2:
        mask = mask.squeeze(0)
    mask_np = mask.cpu().numpy().astype('uint8')
    mask_pil = Image.fromarray(mask_np, mode="L")
    return mask_pil

# Tensor to SAM-compatible NumPy
def tensor2sam(image):
    # Convert tensor to numpy array in HWC uint8 format with pixel values in [0, 255]
    sam_image = np.clip(255. * image.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
    # Transpose the image to HWC format if it's in CHW format
    if sam_image.shape[0] == 3:
        sam_image = np.transpose(sam_image, (1, 2, 0))
    return sam_image

# SAM-compatible NumPy to tensor
def sam2tensor(image):
    # Convert the image to float32 and normalize the pixel values to [0, 1]
    float_image = image.astype(np.float32) / 255.0
    # Transpose the image from HWC format to CHW format
    chw_image = np.transpose(float_image, (2, 0, 1))
    # Convert the numpy array to a tensor
    tensor_image = torch.from_numpy(chw_image)
    return tensor_image

# Median Filter
def medianFilter(img, diameter, sigmaColor, sigmaSpace):
    import cv2 as cv
    diameter = int(diameter)
    sigmaColor = int(sigmaColor)
    sigmaSpace = int(sigmaSpace)
    img = img.convert('RGB')
    img = cv.cvtColor(np.array(img), cv.COLOR_RGB2BGR)
    img = cv.bilateralFilter(img, diameter, sigmaColor, sigmaSpace)
    img = cv.cvtColor(np.array(img), cv.COLOR_BGR2RGB)
    return Image.fromarray(img).convert('RGB')

# Resize Image
def resizeImage(image, max_size):
    width, height = image.size
    if width > height:
        if width > max_size:
            new_width = max_size
            new_height = int(height * (max_size / width))
    else:
        if height > max_size:
            new_height = max_size
            new_width = int(width * (max_size / height))
    resized_image = image.resize((new_width, new_height))
    return resized_image

# Image Seed
def image2seed(image):
    image_data = image.tobytes()
    hash_object = hashlib.sha256(image_data)
    hash_digest = hash_object.digest()
    seed = int.from_bytes(hash_digest[:4], byteorder='big')
    return seed


# SHA-256 Hash
def get_sha256(file_path):
    sha256_hash = hashlib.sha256()
    with open(file_path, 'rb') as file:
        for chunk in iter(lambda: file.read(4096), b''):
            sha256_hash.update(chunk)
    return sha256_hash.hexdigest()

# Batch Seed Generator
def seed_batch(seed, batches, seeds):
    rng = np.random.default_rng(seed)
    btch = [rng.choice(2**32 - 1, seeds, replace=False).tolist() for _ in range(batches)]
    return btch

# Download File
def download_file(url, filename=None, path=None):
    if not filename:
        filename = url.split('/')[-1]
    if not path:
        path = '.'
    save_path = os.path.join(path, filename)
    response = requests.get(url, stream=True)
    if response.status_code == requests.codes.ok:
        file_size = int(response.headers.get('Content-Length', 0))
        with open(save_path, 'wb') as file:
            with tqdm(total=file_size, unit='B', unit_scale=True, unit_divisor=1024) as progress:
                for chunk in response.iter_content(chunk_size=1024):
                    file.write(chunk)
                    progress.update(len(chunk))
        print(f"Downloaded file saved at: {save_path}")
        return True
    elif response.status_code == requests.codes.not_found:
        cstr("Error: File not found.").error.print()
    else:
        cstr(f"Error: Failed to download file. Status code: {response.status_code}").error.print()
    return False

# NSP Function

def nsp_parse(text, seed=0, noodle_key='__', nspterminology=None, pantry_path=None):
    if nspterminology is None:
        # Fetch the NSP Pantry
        if pantry_path is None:
            pantry_path = DEFAULT_NSP_PANTRY_PATH
        if not os.path.exists(pantry_path):
            response = urlopen('https://raw.githubusercontent.com/WASasquatch/noodle-soup-prompts/main/nsp_pantry.json')
            tmp_pantry = json.loads(response.read())
            # Dump JSON locally
            pantry_serialized = json.dumps(tmp_pantry, indent=4)
            with open(pantry_path, "w") as f:
                f.write(pantry_serialized)
            del response, tmp_pantry

        # Load local pantry
        with open(pantry_path, 'r') as f:
            nspterminology = json.load(f)

    if seed > 0 or seed < 0:
        random.seed(seed)

    # Parse Text
    new_text = text
    for term in nspterminology:
        # Target Noodle
        tkey = f'{noodle_key}{term}{noodle_key}'
        # How many occurrences?
        tcount = new_text.count(tkey)
        # Apply random results for each noodle counted
        for _ in range(tcount):
            new_text = new_text.replace(
                tkey, random.choice(nspterminology[term]), 1)
            seed += 1
            random.seed(seed)

    return new_text

# Simple wildcard parser:

def replace_wildcards(text, seed=None, noodle_key='__'):

    def replace_nested(text, key_path_dict):
        if re.findall(f"{noodle_key}(.+?){noodle_key}", text):
            for key, file_path in key_path_dict.items():
                with open(file_path, "r", encoding="utf-8") as file:
                    lines = file.readlines()
                if lines:
                    random_line = None
                    while not random_line:
                        line = random.choice(lines).strip()
                        if not line.startswith('#') and not line.startswith('//'):
                            random_line = line
                    text = text.replace(key, random_line)
        return text

    conf = getSuiteConfig()
    wildcard_dir = os.path.join(WAS_SUITE_ROOT, 'wildcards')
    if not os.path.exists(wildcard_dir):
        os.makedirs(wildcard_dir, exist_ok=True)
    if conf.__contains__('wildcards_path'):
        if conf['wildcards_path'] not in [None, ""]:
            wildcard_dir = conf['wildcards_path']

    cstr(f"Wildcard Path: {wildcard_dir}").msg.print()

    # Set the random seed for reproducibility
    if seed:
        random.seed(seed)

    # Create a dictionary of key to file path pairs
    key_path_dict = {}
    for root, dirs, files in os.walk(wildcard_dir):
        for file in files:
            file_path = os.path.join(root, file)
            key = os.path.relpath(file_path, wildcard_dir).replace(os.path.sep, "/").rsplit(".", 1)[0]
            key_path_dict[f"{noodle_key}{key}{noodle_key}"] = os.path.abspath(file_path)

    # Replace keys in text with random lines from corresponding files
    for key, file_path in key_path_dict.items():
        with open(file_path, "r", encoding="utf-8") as file:
            lines = file.readlines()
            if lines:
                random_line = None
                while not random_line:
                    line = random.choice(lines).strip()
                    if not line.startswith('#') and not line.startswith('//'):
                        random_line = line
                text = text.replace(key, random_line)

    # Replace sub-wildacrds in result
    text = replace_nested(text, key_path_dict)

    return text

# Parse Prompt Variables

def parse_prompt_vars(input_string, optional_vars=None):
    variables = optional_vars or {}
    pattern = r"\$\|(.*?)\|\$"
    variable_count = len(variables) + 1

    def replace_variable(match):
        nonlocal variable_count
        variable_name = f"${variable_count}"
        variables[variable_name] = match.group(1)
        variable_count += 1
        return variable_name

    output_string = re.sub(pattern, replace_variable, input_string)

    for variable_name, phrase in variables.items():
        variable_pattern = re.escape(variable_name)
        output_string = re.sub(variable_pattern, phrase, output_string)

    return output_string, variables

# Parse Dynamic Prompts

def parse_dynamic_prompt(prompt, seed):
    random.seed(seed)

    def replace_match(match):
        options = match.group(1).split('|')
        return random.choice(options)

    parse_prompt = re.sub(r'\<(.*?)\>', replace_match, prompt)
    while re.search(r'\<(.*?)\>', parse_prompt):
        parse_prompt = re.sub(r'\<(.*?)\>', replace_match, parse_prompt)

    return parse_prompt

# Ambient Occlusion Factor

@jit(nopython=True)
def calculate_ambient_occlusion_factor(rgb_normalized, depth_normalized, height, width, radius):
    occlusion_array = np.zeros((height, width), dtype=np.uint8)

    for y in range(height):
        for x in range(width):
            if radius == 0:
                occlusion_factor = 0
            else:
                y_min = max(y - radius, 0)
                y_max = min(y + radius + 1, height)
                x_min = max(x - radius, 0)
                x_max = min(x + radius + 1, width)

                neighborhood_depth = depth_normalized[y_min:y_max, x_min:x_max]
                neighborhood_rgb = rgb_normalized[y_min:y_max, x_min:x_max, :]

                depth_diff = depth_normalized[y, x] - neighborhood_depth
                rgb_diff = np.abs(rgb_normalized[y, x] - neighborhood_rgb)
                occlusion_factor = np.maximum(0, depth_diff).mean() + np.maximum(0, np.sum(rgb_diff, axis
=2)).mean()

            occlusion_value = int(255 - occlusion_factor * 255)
            occlusion_array[y, x] = occlusion_value

    return occlusion_array

# Direct Occlusion Factor
```

```
@jit(nopython=True)
def calculate_direct_occlusion_factor(rgb_normalized, depth_normalized, height, width, radius):
    occlusion_array = np.empty((int(height), int(width)), dtype=np.uint8)
    depth_normalized = depth_normalized[:, :, 0]

    for y in range(int(height)):
        for x in range(int(width)):
            if radius == 0:
                occlusion_factor = 0
            else:
                y_min = max(int(y - radius), 0)
                y_max = min(int(y + radius + 1), int(height))
                x_min = max(int(x - radius), 0)
                x_max = min(int(x + radius + 1), int(width))

                neighborhood_depth = np.zeros((y_max - y_min, x_max - x_min), dtype=np.float64)
                neighborhood_rgb = np.empty((y_max - y_min, x_max - x_min, 3))

                for i in range(y_min, y_max):
                    for j in range(x_min, x_max):
                        neighborhood_depth[i - y_min, j - x_min] = depth_normalized[i, j]
                        neighborhood_rgb[i - y_min, j - x_min, :] = rgb_normalized[i, j, :]

                depth_diff = neighborhood_depth - depth_normalized[y, x]
                rgb_diff = np.abs(neighborhood_rgb - rgb_normalized[y, x])
                occlusion_factor = np.maximum(0, depth_diff).mean() + np.maximum(0, np.sum(np.abs(rgb_d
iff), axis=2)).mean()

            occlusion_value = int(occlusion_factor * 255)
            occlusion_array[y, x] = occlusion_value

    occlusion_min = np.min(occlusion_array)
    occlusion_max = np.max(occlusion_array)
    occlusion_scaled = ((occlusion_array - occlusion_min) / (occlusion_max - occlusion_min) * 255).asty
pe(np.uint8)

    return occlusion_scaled


class PromptStyles:
    def __init__(self, styles_file, preview_length = 32):
        self.styles_file = styles_file
        self.styles = {}
        self.preview_length = preview_length

        if os.path.exists(self.styles_file):
            with open(self.styles_file, 'r') as f:
                self.styles = json.load(f)

    def add_style(self, prompt="", negative_prompt="", auto=False, name=None):
        if auto:
            date_format = '%A, %d %B %Y %I:%M %p'
            date_str = datetime.datetime.now().strftime(date_format)
            key = None
            if prompt.strip() != "":
                if len(prompt) > self.preview_length:
                    length = self.preview_length
                else:
```

```
                length = len(prompt)
            key = f"[{date_str}] Positive: {prompt[:length]} ..."
        elif negative_prompt.strip() != "":
            if len(negative_prompt) > self.preview_length:
                length = self.preview_length
            else:
                length = len(negative_prompt)
            key = f"[{date_str}] Negative: {negative_prompt[:length]} ..."
        else:
            cstr("At least a `prompt`, or `negative_prompt` input is required!").error.print()
            return
    else:
        if name == None or str(name).strip() == "":
            cstr("A `name` input is required when not using `auto=True`").error.print()
            return
        key = str(name)


    for k, v in self.styles.items():
        if v['prompt'] == prompt and v['negative_prompt'] == negative_prompt:
            return

    self.styles[key] = {"prompt": prompt, "negative_prompt": negative_prompt}

    with open(self.styles_file, "w", encoding='utf-8') as f:
        json.dump(self.styles, f, indent=4)

  def get_prompts(self):
    return self.styles

  def get_prompt(self, prompt_key):
    if prompt_key in self.styles:
        return self.styles[prompt_key]['prompt'], self.styles[prompt_key]['negative_prompt']
    else:
        cstr(f"Prompt style `{prompt_key}` was not found!").error.print()
        return None, None



# WAS SETTINGS MANAGER

class WASDatabase:
  """
  The WAS Suite Database Class provides a simple key-value database that stores
  data in a flatfile using the JSON format. Each key-value pair is associated with
  a category.

  Attributes:
    filepath (str): The path to the JSON file where the data is stored.
    data (dict): The dictionary that holds the data read from the JSON file.

  Methods:
    insert(category, key, value): Inserts a key-value pair into the database
        under the specified category.
    get(category, key): Retrieves the value associated with the specified
        key and category from the database.
    update(category, key): Update a value associated with the specified
        key and category from the database.
    delete(category, key): Deletes the key-value pair associated with the
        specified key and category from the database.
    _save(): Saves the current state of the database to the JSON file.
"""
def __init__(self, filepath):
    self.filepath = filepath
    try:
        with open(filepath, 'r') as f:
            self.data = json.load(f)
    except FileNotFoundError:
        self.data = {}

def catExists(self, category):
    return category in self.data

def keyExists(self, category, key):
    return category in self.data and key in self.data[category]

def insert(self, category, key, value):
    if not isinstance(category, str) or not isinstance(key, str):
        cstr("Category and key must be strings").error.print()
        return

    if category not in self.data:
        self.data[category] = {}
    self.data[category][key] = value
    self._save()

def update(self, category, key, value):
    if category in self.data and key in self.data[category]:
        self.data[category][key] = value
        self._save()

def updateCat(self, category, dictionary):
    self.data[category].update(dictionary)
    self._save()

def get(self, category, key):
    return self.data.get(category, {}).get(key, None)

def getDB(self):
    return self.data

def insertCat(self, category):
    if not isinstance(category, str):
        cstr("Category must be a string").error.print()
        return

    if category in self.data:
        cstr(f"The database category '{category}' already exists!").error.print()
        return
    self.data[category] = {}
    self._save()

def getDict(self, category):
    if category not in self.data:
        cstr(f"The database category '{category}' does not exist!").error.print()
        return {}
        return self.data[category]

    def delete(self, category, key):
        if category in self.data and key in self.data[category]:
            del self.data[category][key]
            self._save()

    def _save(self):
        try:
            with open(self.filepath, 'w') as f:
                json.dump(self.data, f, indent=4)
        except FileNotFoundError:
            cstr(f"Cannot save database to file '{self.filepath}'. "
                 "Storing the data in the object instead. Does the folder and node file have write permission
s?").warning.print()
        except Exception as e:
            cstr(f"Error while saving JSON data: {e}").error.print()

# Initialize the settings database
WDB = WASDatabase(WAS_DATABASE)

# WAS Token Class

class TextTokens:
    def __init__(self):
        self.WDB = WDB
        if not self.WDB.getDB().__contains__('custom_tokens'):
            self.WDB.insertCat('custom_tokens')
        self.custom_tokens = self.WDB.getDict('custom_tokens')

        self.tokens = {
            '[time]': str(time.time()).replace('.','_'),
            '[hostname]': socket.gethostname(),
            '[cuda_device]': str(comfy.model_management.get_torch_device()),
            '[cuda_name]': str(comfy.model_management.get_torch_device_name(device=comfy.model_ma
nagement.get_torch_device())),
        }

        if '.' in self.tokens['[time]']:
            self.tokens['[time]'] = self.tokens['[time]'].split('.')[0]

        try:
            self.tokens['[user]'] = os.getlogin() if os.getlogin() else 'null'
        except Exception:
            self.tokens['[user]'] = 'null'

    def addToken(self, name, value):
        self.custom_tokens.update({name: value})
        self._update()

    def removeToken(self, name):
        self.custom_tokens.pop(name)
        self._update()

    def format_time(self, format_code):
        return time.strftime(format_code, time.localtime(time.time()))

    def parseTokens(self, text):
        tokens = self.tokens.copy()
        if self.custom_tokens:
            tokens.update(self.custom_tokens)

        # Update time
        tokens['[time]'] = str(time.time())
        if '.' in tokens['[time]']:
            tokens['[time]'] = tokens['[time]'].split('.')[0]

        for token, value in tokens.items():
            if token.startswith('[time('):
                continue
            pattern = re.compile(re.escape(token))
            text = pattern.sub(value, text)

        def replace_custom_time(match):
            format_code = match.group(1)
            return self.format_time(format_code)

        text = re.sub(r'\[time\((.*?)\)\]', replace_custom_time, text)

        return text

    def _update(self):
        self.WDB.updateCat('custom_tokens', self.custom_tokens)


# Update image history

def update_history_images(new_paths):
    HDB = WASDatabase(WAS_HISTORY_DATABASE)
    if HDB.catExists("History") and HDB.keyExists("History", "Images"):
        saved_paths = HDB.get("History", "Images")
        for path_ in saved_paths:
            if not os.path.exists(path_):
                saved_paths.remove(path_)
        if isinstance(new_paths, str):
            if new_paths in saved_paths:
                saved_paths.remove(new_paths)
            saved_paths.append(new_paths)
        elif isinstance(new_paths, list):
            for path_ in new_paths:
                if path_ in saved_paths:
                    saved_paths.remove(path_)
                saved_paths.append(path_)
        HDB.update("History", "Images", saved_paths)
    else:
        if not HDB.catExists("History"):
            HDB.insertCat("History")
        if isinstance(new_paths, str):
            HDB.insert("History", "Images", [new_paths])
        elif isinstance(new_paths, list):
            HDB.insert("History", "Images", new_paths)

# Update output image history

def update_history_output_images(new_paths):
    HDB = WASDatabase(WAS_HISTORY_DATABASE)
    category = "Output_Images"
    if HDB.catExists("History") and HDB.keyExists("History", category):
        saved_paths = HDB.get("History", category)
        for path_ in saved_paths:
            if not os.path.exists(path_):
                saved_paths.remove(path_)
        if isinstance(new_paths, str):
            if new_paths in saved_paths:
                saved_paths.remove(new_paths)
            saved_paths.append(new_paths)
        elif isinstance(new_paths, list):
            for path_ in new_paths:
                if path_ in saved_paths:
                    saved_paths.remove(path_)
                saved_paths.append(path_)
        HDB.update("History", category, saved_paths)
    else:
        if not HDB.catExists("History"):
            HDB.insertCat("History")
        if isinstance(new_paths, str):
            HDB.insert("History", category, [new_paths])
        elif isinstance(new_paths, list):
            HDB.insert("History", category, new_paths)

# Update text file history

def update_history_text_files(new_paths):
    HDB = WASDatabase(WAS_HISTORY_DATABASE)
    if HDB.catExists("History") and HDB.keyExists("History", "TextFiles"):
        saved_paths = HDB.get("History", "TextFiles")
        for path_ in saved_paths:
            if not os.path.exists(path_):
                saved_paths.remove(path_)
        if isinstance(new_paths, str):
            if new_paths in saved_paths:
                saved_paths.remove(new_paths)
            saved_paths.append(new_paths)
        elif isinstance(new_paths, list):
            for path_ in new_paths:
                if path_ in saved_paths:
                    saved_paths.remove(path_)
                saved_paths.append(path_)
        HDB.update("History", "TextFiles", saved_paths)
    else:
        if not HDB.catExists("History"):
            HDB.insertCat("History")
        if isinstance(new_paths, str):
            HDB.insert("History", "TextFiles", [new_paths])
        elif isinstance(new_paths, list):
            HDB.insert("History", "TextFiles", new_paths)
# WAS Filter Class

class WAS_Tools_Class():
    """
    Contains various tools and filters for WAS Node Suite
    """
    # TOOLS
```

```
  def fig2img(self, plot):
    import io
    buf = io.BytesIO()
    plot.savefig(buf)
    buf.seek(0)
    img = Image.open(buf)
    return img

  def stitch_image(self, image_a, image_b, mode='right', fuzzy_zone=50):

    def linear_gradient(start_color, end_color, size, start, end, mode='horizontal'):
      width, height = size
      gradient = Image.new('RGB', (width, height), end_color)
      draw = ImageDraw.Draw(gradient)

      for i in range(0, start):
        if mode == "horizontal":
          draw.line((i, 0, i, height-1), start_color)
        elif mode == "vertical":
          draw.line((0, i, width-1, i), start_color)

      for i in range(start, end):
        if mode == "horizontal":
          curr_color = (
            int(start_color[0] + (float(i - start) / (end - start)) * (end_color[0] - start_color[0])),
            int(start_color[1] + (float(i - start) / (end - start)) * (end_color[1] - start_color[1])),
            int(start_color[2] + (float(i - start) / (end - start)) * (end_color[2] - start_color[2]))
          )
          draw.line((i, 0, i, height-1), curr_color)
        elif mode == "vertical":
          curr_color = (
            int(start_color[0] + (float(i - start) / (end - start)) * (end_color[0] - start_color[0])),
            int(start_color[1] + (float(i - start) / (end - start)) * (end_color[1] - start_color[1])),
            int(start_color[2] + (float(i - start) / (end - start)) * (end_color[2] - start_color[2]))
          )
          draw.line((0, i, width-1, i), curr_color)

      for i in range(end, width if mode == 'horizontal' else height):
        if mode == "horizontal":
          draw.line((i, 0, i, height-1), end_color)
        elif mode == "vertical":
          draw.line((0, i, width-1, i), end_color)

      return gradient

    image_a = image_a.convert('RGB')
    image_b = image_b.convert('RGB')

    offset = int(fuzzy_zone / 2)
    canvas_width = int(image_a.size[0] + image_b.size[0] - fuzzy_zone) if mode == 'right' or mode ==
'left' else image_a.size[0]
    canvas_height = int(image_a.size[1] + image_b.size[1] - fuzzy_zone) if mode == 'top' or mode ==
'bottom' else image_a.size[1]
    canvas = Image.new('RGB', (canvas_width, canvas_height), (0,0,0))

    im_ax = 0
    im_ay = 0
    im_bx = 0
```

```
        im_by = 0

        image_a_mask = None
        image_b_mask = None

        if mode == 'top':

            image_a_mask = linear_gradient((0,0,0), (255,255,255), image_a.size, 0, fuzzy_zone, 'vertical')
            image_b_mask = linear_gradient((255,255,255), (0,0,0), image_b.size, int(image_b.size[1] - fuzz
y_zone), image_b.size[1], 'vertical')
            im_ay = image_b.size[1] - fuzzy_zone

        elif mode == 'bottom':

            image_a_mask = linear_gradient((255,255,255), (0,0,0), image_a.size, int(image_a.size[1] - fuzz
y_zone), image_a.size[1], 'vertical')
            image_b_mask = linear_gradient((0,0,0), (255,255,255), image_b.size, 0, fuzzy_zone, 'vertical').
convert('L')
            im_by = image_a.size[1] - fuzzy_zone

        elif mode == 'left':

            image_a_mask = linear_gradient((0,0,0), (255,255,255), image_a.size, 0, fuzzy_zone, 'horizonta
l')
            image_b_mask = linear_gradient((255,255,255), (0,0,0), image_b.size, int(image_b.size[0] - fuzz
y_zone), image_b.size[0], 'horizontal')
            im_ax = image_b.size[0] - fuzzy_zone

        elif mode == 'right':

            image_a_mask = linear_gradient((255,255,255), (0,0,0), image_a.size, int(image_a.size[0] - fuzz
y_zone), image_a.size[0], 'horizontal')
            image_b_mask = linear_gradient((0,0,0), (255,255,255), image_b.size, 0, fuzzy_zone, 'horizonta
l')
            im_bx = image_a.size[0] - fuzzy_zone

        Image.Image.paste(canvas, image_a, (im_ax, im_ay), image_a_mask.convert('L'))
        Image.Image.paste(canvas, image_b, (im_bx, im_by), image_b_mask.convert('L'))


        return canvas


    def morph_images(self, images, steps=10, max_size=512, loop=None, still_duration=30, duration=0.1,
output_path='output', filename="morph", filetype="GIF"):

        import cv2
        import imageio

        output_file = os.path.abspath(os.path.join(os.path.join(*output_path.split('/')), filename))
        output_file += ( '.png' if filetype == 'APNG' else '.gif' )

        max_width = max(im.size[0] for im in images)
        max_height = max(im.size[1] for im in images)
        max_aspect_ratio = max_width / max_height

        def padded_images():
            for im in images:
```

```
            aspect_ratio = im.size[0] / im.size[1]
            if aspect_ratio > max_aspect_ratio:
                new_height = int(max_width / aspect_ratio)
                padding = (max_height - new_height) // 2
                padded_im = Image.new('RGB', (max_width, max_height), color=(0, 0, 0))
                padded_im.paste(im.resize((max_width, new_height)), (0, padding))
            else:
                new_width = int(max_height * aspect_ratio)
                padding = (max_width - new_width) // 2
                padded_im = Image.new('RGB', (max_width, max_height), color=(0, 0, 0))
                padded_im.paste(im.resize((new_width, max_height)), (padding, 0))
            yield np.array(padded_im)

        padded_images = list(padded_images())
        padded_images.append(padded_images[0].copy())
        images = padded_images
        frames = []
        durations = []

        for i in range(len(images)-1):
            frames.append(Image.fromarray(images[i]).convert('RGB'))
            durations.append(still_duration)

            for j in range(steps):
                alpha = j / float(steps)
                morph = cv2.addWeighted(images[i], 1 - alpha, images[i+1], alpha, 0)
                frames.append(Image.fromarray(morph).convert('RGB'))
                durations.append(duration)

        frames.append(Image.fromarray(images[-1]).convert('RGB'))
        durations.insert(0, still_duration)

        if loop is not None:
            for i in range(loop):
                durations.insert(0, still_duration)
                durations.append(still_duration)

        try:
            imageio.mimsave(output_file, frames, filetype, duration=durations, loop=loop)
        except OSError as e:
            cstr(f"Unable to save output to {output_file} due to the following error:").error.print()
            print(e)
            return
        except Exception as e:
            cstr(f"\033[34mWAS NS\033[0m Error: Unable to generate GIF due to the following error:").erro
r.print()
            print(e)

        cstr(f"Morphing completed. Output saved as {output_file}").msg.print()

        return output_file

    class GifMorphWriter:
        def __init__(self, transition_frames=30, duration_ms=100, still_image_delay_ms=2500, loop=0):
            self.transition_frames = transition_frames
            self.duration_ms = duration_ms
            self.still_image_delay_ms = still_image_delay_ms
            self.loop = loop
    def write(self, image, gif_path):

        import cv2

        if not os.path.isfile(gif_path):
            with Image.new("RGBA", image.size) as new_gif:
                new_gif.paste(image.convert("RGBA"))
                new_gif.info["duration"] = self.still_image_delay_ms
                new_gif.save(gif_path, format="GIF", save_all=True, append_images=[], duration=self.still_i
mage_delay_ms, loop=0)
            cstr(f"Created new GIF animation at: {gif_path}").msg.print()
        else:
            with Image.open(gif_path) as gif:
                n_frames = gif.n_frames
                if n_frames > 0:
                    gif.seek(n_frames - 1)
                    last_frame = gif.copy()
                else:
                    last_frame = None

                end_image = image
                steps = self.transition_frames - 1 if last_frame is not None else self.transition_frames

                if last_frame is not None:
                    image = self.pad_to_size(image, last_frame.size)

                frames = self.generate_transition_frames(last_frame, image, steps)

                still_frame = end_image.copy()

                gif_frames = []
                for i in range(n_frames):
                    gif.seek(i)
                    gif_frame = gif.copy()
                    gif_frames.append(gif_frame)

                for frame in frames:
                    frame.info["duration"] = self.duration_ms
                    gif_frames.append(frame)

                still_frame.info['duration'] = self.still_image_delay_ms
                gif_frames.append(still_frame)

                gif_frames[0].save(
                    gif_path,
                    format="GIF",
                    save_all=True,
                    append_images=gif_frames[1:],
                    optimize=True,
                    loop=self.loop,
                )

                cstr(f"Edited existing GIF animation at: {gif_path}").msg.print()


    def pad_to_size(self, image, size):
        new_image = Image.new("RGBA", size, color=(0, 0, 0, 0))
        x_offset = (size[0] - image.width) // 2
        y_offset = (size[1] - image.height) // 2
        new_image.paste(image, (x_offset, y_offset))
        return new_image

    def generate_transition_frames(self, start_frame, end_image, num_frames):

        if start_frame is None:
            return []

        start_frame = start_frame.convert("RGBA")
        end_image = end_image.convert("RGBA")

        frames = []
        for i in range(1, num_frames + 1):
            weight = i / (num_frames + 1)
            frame = Image.blend(start_frame, end_image, weight)
            frames.append(frame)
        return frames

class VideoWriter:
    def __init__(self, transition_frames=30, fps=25, still_image_delay_sec=2,
                max_size=512, codec="mp4v"):
        conf = getSuiteConfig()
        self.transition_frames = transition_frames
        self.fps = fps
        self.still_image_delay_frames = round(still_image_delay_sec * fps)
        self.max_size = int(max_size)
        self.valid_codecs = ["ffv1","mp4v"]
        self.extensions = {"ffv1":".mkv","mp4v":".mp4"}
        if conf.__contains__('ffmpeg_extra_codecs'):
            self.add_codecs(conf['ffmpeg_extra_codecs'])
        self.codec = codec.lower() if codec.lower() in self.valid_codecs else "mp4v"

    def write(self, image, video_path):
        video_path += self.extensions[self.codec]
        end_image = self.rescale(self.pil2cv(image), self.max_size)

        if os.path.isfile(video_path):
            cap = cv2.VideoCapture(video_path)

            width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
            height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
            fps = int(cap.get(cv2.CAP_PROP_FPS))
            total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

            if width <= 0 or height <= 0:
                raise ValueError("Invalid video dimensions")

            temp_file_path = video_path.replace(self.extensions[self.codec], '_temp' + self.extensions[sel
f.codec])
            fourcc = cv2.VideoWriter_fourcc(*self.codec)
            out = cv2.VideoWriter(temp_file_path, fourcc, fps, (width, height), isColor=True)

            for i in tqdm(range(total_frames), desc="Copying original frames"):
                ret, frame = cap.read()
                if not ret:
                    break
                out.write(frame)

            if self.transition_frames > 0:
                cap.set(cv2.CAP_PROP_POS_FRAMES, total_frames - 1)
                ret, last_frame = cap.read()
                if ret:
                    transition_frames = self.generate_transition_frames(last_frame, end_image, self.transition
_frames)
                    for i, transition_frame in tqdm(enumerate(transition_frames), desc="Generating transition
frames", total=self.transition_frames):
                        try:
                            transition_frame_resized = cv2.resize(transition_frame, (width, height))
                            out.write(transition_frame_resized)
                        except cv2.error as e:
                            print(f"Error resizing frame {i}: {e}")
                            continue

            for i in tqdm(range(self.still_image_delay_frames), desc="Adding new frames"):
                out.write(end_image)

            cap.release()
            out.release()

            os.remove(video_path)
            os.rename(temp_file_path, video_path)

            cstr(f"Edited video at: {video_path}").msg.print()

            return video_path

        else:
            fourcc = cv2.VideoWriter_fourcc(*self.codec)
            height, width, _ = end_image.shape
            if width <= 0 or height <= 0:
                raise ValueError("Invalid image dimensions")

            out = cv2.VideoWriter(video_path, fourcc, self.fps, (width, height), isColor=True)

            for i in tqdm(range(self.still_image_delay_frames), desc="Adding new frames"):
                out.write(end_image)

            out.release()

            cstr(f"Created new video at: {video_path}").msg.print()

            return video_path

        return ""

    def create_video(self, image_folder, video_path):
        import cv2
        from tqdm import tqdm

        image_paths = sorted([os.path.join(image_folder, f) for f in os.listdir(image_folder)
                              if os.path.isfile(os.path.join(image_folder, f))
                              and os.path.join(image_folder, f).lower().endswith(ALLOWED_EXT)])

        if len(image_paths) == 0:
            cstr(f"No valid image files found in `{image_folder}` directory.").error.print()
            cstr(f"The valid formats are: {', '.join(sorted(ALLOWED_EXT))}").error.print()
            return

        output_file = video_path + self.extensions[self.codec]
        image = self.rescale(cv2.imread(image_paths[0]), self.max_size)
        height, width = image.shape[:2]
        fourcc = cv2.VideoWriter_fourcc(*self.codec)
        out = cv2.VideoWriter(output_file, fourcc, self.fps, (width, height), isColor=True)
        out.write(image)
        for _ in range(self.still_image_delay_frames - 1):
            out.write(image)

        for i in tqdm(range(len(image_paths)), desc="Writing video frames"):
            start_frame = cv2.imread(image_paths[i])
            end_frame = None
            if i+1 <= len(image_paths)-1:
                end_frame = self.rescale(cv2.imread(image_paths[i+1]), self.max_size)

            if isinstance(end_frame, np.ndarray):
                transition_frames = self.generate_transition_frames(start_frame, end_frame, self.transition_f
rames)
                transition_frames = [cv2.resize(frame, (width, height)) for frame in transition_frames]
                for _, frame in enumerate(transition_frames):
                    out.write(frame)

                for _ in range(self.still_image_delay_frames - self.transition_frames):
                    out.write(end_frame)

            else:
                out.write(start_frame)
                for _ in range(self.still_image_delay_frames - 1):
                    out.write(start_frame)

        out.release()

        if os.path.exists(output_file):
            cstr(f"Created video at: {output_file}").msg.print()
            return output_file
        else:
            cstr(f"Unable to create video at: {output_file}").error.print()
            return ""

    def extract(self, video_file, output_folder, prefix='frame_', extension="png", zero_padding_digits=-
1):
        os.makedirs(output_folder, exist_ok=True)

        video = cv2.VideoCapture(video_file)

        fps = video.get(cv2.CAP_PROP_FPS)
        frame_number = 0

        while True:
            success, frame = video.read()

            if success:
                if zero_padding_digits > 0:
                    frame_path = os.path.join(output_folder, f"{prefix}{frame_number:0{zero_padding_digit
s}}.{extension}")
                else:
                    frame_path = os.path.join(output_folder, f"{prefix}{frame_number}.{extension}")

                cv2.imwrite(frame_path, frame)
                print(f"Saved frame {frame_number} to {frame_path}")
                frame_number += 1
            else:
                break

        video.release()

    def rescale(self, image, max_size):
        f1 = max_size / image.shape[1]
        f2 = max_size / image.shape[0]
        f = min(f1, f2)
        dim = (int(image.shape[1] * f), int(image.shape[0] * f))
        resized = cv2.resize(image, dim)
        return resized

    def generate_transition_frames(self, img1, img2, num_frames):
        import cv2
        if img1 is None and img2 is None:
            return []

        if img1 is not None and img2 is not None:
            if img1.shape != img2.shape:
                img2 = cv2.resize(img2, img1.shape[:2][::-1])
        elif img1 is not None:
            img2 = np.zeros_like(img1)
        else:
            img1 = np.zeros_like(img2)

        height, width, _ = img2.shape

        frame_sequence = []
        for i in range(num_frames):
            alpha = i / float(num_frames)
            blended = cv2.addWeighted(img1, 1 - alpha, img2, alpha,
                                      gamma=0.0, dtype=cv2.CV_8U)
            frame_sequence.append(blended)

        return frame_sequence

    def pil2cv(self, img):
        import cv2
        img = np.array(img)
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        return img

    def add_codecs(self, codecs):
        if isinstance(codecs, dict):
            codec_forcc_codes = codecs.keys()
            self.valid_codecs.extend(codec_forcc_codes)
            self.extensions.update(codecs)

    def get_codecs(self):
        return self.valid_codecs
```

```
# FILTERS

class Masking:

    @staticmethod
    def crop_dominant_region(image, padding=0):
        from scipy.ndimage import label
        grayscale_image = image.convert("L")
        binary_image = grayscale_image.point(lambda x: 255 if x > 128 else 0, mode="1")
        labeled_image, num_labels = label(np.array(binary_image))
        largest_label = max(range(1, num_labels + 1), key=lambda i: np.sum(labeled_image == i))
        largest_region_mask = (labeled_image == largest_label).astype(np.uint8) * 255
        bbox = Image.fromarray(largest_region_mask, mode="L").getbbox()
        cropped_image = image.crop(bbox)
        size = max(cropped_image.size)
        padded_size = size + 2 * padding
        centered_crop = Image.new("L", (padded_size, padded_size), color="black")
        left = (padded_size - cropped_image.width) // 2
        top = (padded_size - cropped_image.height) // 2
        centered_crop.paste(cropped_image, (left, top), mask=cropped_image)

        return ImageOps.invert(centered_crop)

    @staticmethod
    def crop_minority_region(image, padding=0):
        from scipy.ndimage import label
        grayscale_image = image.convert("L")
        binary_image = grayscale_image.point(lambda x: 255 if x > 128 else 0, mode="1")
        labeled_image, num_labels = label(np.array(binary_image))
        smallest_label = min(range(1, num_labels + 1), key=lambda i: np.sum(labeled_image == i))
        smallest_region_mask = (labeled_image == smallest_label).astype(np.uint8) * 255
        bbox = Image.fromarray(smallest_region_mask, mode="L").getbbox()
        cropped_image = image.crop(bbox)
        size = max(cropped_image.size)
        padded_size = size + 2 * padding
        centered_crop = Image.new("L", (padded_size, padded_size), color="black")
        left = (padded_size - cropped_image.width) // 2
        top = (padded_size - cropped_image.height) // 2
        centered_crop.paste(cropped_image, (left, top), mask=cropped_image)

        return ImageOps.invert(centered_crop)

    @staticmethod
    def crop_region(mask, region_type, padding=0):
        from scipy.ndimage import label, find_objects
        binary_mask = np.array(mask.convert("L")) > 0
        bbox = mask.getbbox()
        if bbox is None:
            return mask, (mask.size, (0, 0, 0, 0))

        bbox_width = bbox[2] - bbox[0]
        bbox_height = bbox[3] - bbox[1]

        side_length = max(bbox_width, bbox_height) + 2 * padding

        center_x = (bbox[2] + bbox[0]) // 2
        center_y = (bbox[3] + bbox[1]) // 2

        crop_x = center_x - side_length // 2
        crop_y = center_y - side_length // 2

        crop_x = max(crop_x, 0)
        crop_y = max(crop_y, 0)
        crop_x2 = min(crop_x + side_length, mask.width)
        crop_y2 = min(crop_y + side_length, mask.height)

        cropped_mask = mask.crop((crop_x, crop_y, crop_x2, crop_y2))
        crop_data = (cropped_mask.size, (crop_x, crop_y, crop_x2, crop_y2))

        return cropped_mask, crop_data

    @staticmethod
    def dominant_region(image, threshold=128):
        from scipy.ndimage import label
        image = ImageOps.invert(image.convert("L"))
        binary_image = image.point(lambda x: 255 if x > threshold else 0, mode="1")
        l, n = label(np.array(binary_image))
        sizes = np.bincount(l.flatten())
        dominant = 0
        try:
            dominant = np.argmax(sizes[1:]) + 1
        except ValueError:
            pass
        dominant_region_mask = (l == dominant).astype(np.uint8) * 255
        result = Image.fromarray(dominant_region_mask, mode="L")
        return result.convert("RGB")

    @staticmethod
    def minority_region(image, threshold=128):
        from scipy.ndimage import label
        image = image.convert("L")
        binary_image = image.point(lambda x: 255 if x > threshold else 0, mode="1")
        labeled_array, num_features = label(np.array(binary_image))
        sizes = np.bincount(labeled_array.flatten())
        smallest_region = 0
        try:
            smallest_region = np.argmin(sizes[1:]) + 1
        except ValueError:
            pass
        smallest_region_mask = (labeled_array == smallest_region).astype(np.uint8) * 255
        inverted_mask = Image.fromarray(smallest_region_mask, mode="L")
        rgb_image = Image.merge("RGB", [inverted_mask, inverted_mask, inverted_mask])

        return rgb_image

    @staticmethod
    def arbitrary_region(image, size, threshold=128):
        from skimage.measure import label, regionprops
        image = image.convert("L")
        binary_image = image.point(lambda x: 255 if x > threshold else 0, mode="1")
        labeled_image = label(np.array(binary_image))
        regions = regionprops(labeled_image)

        image_area = binary_image.size[0] * binary_image.size[1]
        scaled_size = size * image_area / 10000

        filtered_regions = [region for region in regions if region.area >= scaled_size]
        if len(filtered_regions) > 0:
            filtered_regions.sort(key=lambda region: region.area)
            smallest_region = filtered_regions[0]
            region_mask = (labeled_image == smallest_region.label).astype(np.uint8) * 255
            result = Image.fromarray(region_mask, mode="L")
            return result

        return image

    @staticmethod
    def smooth_region(image, tolerance):
        from scipy.ndimage import gaussian_filter
        image = image.convert("L")
        mask_array = np.array(image)
        smoothed_array = gaussian_filter(mask_array, sigma=tolerance)
        threshold = np.max(smoothed_array) / 2
        smoothed_mask = np.where(smoothed_array >= threshold, 255, 0).astype(np.uint8)
        smoothed_image = Image.fromarray(smoothed_mask, mode="L")
        return ImageOps.invert(smoothed_image.convert("RGB"))

    @staticmethod
    def erode_region(image, iterations=1):
        from scipy.ndimage import binary_erosion
        image = image.convert("L")
        binary_mask = np.array(image) > 0
        eroded_mask = binary_erosion(binary_mask, iterations=iterations)
        eroded_image = Image.fromarray(eroded_mask.astype(np.uint8) * 255, mode="L")
        return ImageOps.invert(eroded_image.convert("RGB"))

    @staticmethod
    def dilate_region(image, iterations=1):
        from scipy.ndimage import binary_dilation
        image = image.convert("L")
        binary_mask = np.array(image) > 0
        dilated_mask = binary_dilation(binary_mask, iterations=iterations)
        dilated_image = Image.fromarray(dilated_mask.astype(np.uint8) * 255, mode="L")
        return ImageOps.invert(dilated_image.convert("RGB"))

    @staticmethod
    def fill_region(image):
        from scipy.ndimage import binary_fill_holes
        image = image.convert("L")
        binary_mask = np.array(image) > 0
        filled_mask = binary_fill_holes(binary_mask)
        filled_image = Image.fromarray(filled_mask.astype(np.uint8) * 255, mode="L")
        return ImageOps.invert(filled_image.convert("RGB"))

    @staticmethod
    def combine_masks(*masks):
        if len(masks) < 1:
            raise ValueError("\033[34mWAS NS\033[0m Error: At least one mask must be provided.")
        dimensions = masks[0].size
        for mask in masks:
            if mask.size != dimensions:
                raise ValueError("\033[34mWAS NS\033[0m Error: All masks must have the same dimensio
ns.")

            inverted_masks = [mask.convert("L") for mask in masks]
            combined_mask = Image.new("L", dimensions, 255)
            for mask in inverted_masks:
                combined_mask = Image.fromarray(np.minimum(np.array(combined_mask), np.array(mask)),
mode="L")

            return combined_mask

        @staticmethod
        def threshold_region(image, black_threshold=0, white_threshold=255):
            gray_image = image.convert("L")
            mask_array = np.array(gray_image)
            mask_array[mask_array < black_threshold] = 0
            mask_array[mask_array > white_threshold] = 255
            thresholded_image = Image.fromarray(mask_array, mode="L")
            return ImageOps.invert(thresholded_image)

        @staticmethod
        def floor_region(image):
            gray_image = image.convert("L")
            mask_array = np.array(gray_image)
            non_black_pixels = mask_array[mask_array > 0]

            if non_black_pixels.size > 0:
                threshold_value = non_black_pixels.min()
                mask_array[mask_array > threshold_value] = 255  # Set whites to 255
                mask_array[mask_array <= threshold_value] = 0  # Set blacks to 0

            thresholded_image = Image.fromarray(mask_array, mode="L")
            return ImageOps.invert(thresholded_image)

        @staticmethod
        def ceiling_region(image, offset=30):
            if offset < 0:
                offset = 0
            elif offset > 255:
                offset = 255
            grayscale_image = image.convert("L")
            mask_array = np.array(grayscale_image)
            mask_array[mask_array < 255 - offset] = 0
            mask_array[mask_array >= 250] = 255
            filtered_image = Image.fromarray(mask_array, mode="L")
            return ImageOps.invert(filtered_image)

        @staticmethod
        def gaussian_region(image, radius=5.0):
            image = ImageOps.invert(image.convert("L"))
            image = image.filter(ImageFilter.GaussianBlur(radius=int(radius)))
            return image.convert("RGB")

    # SHADOWS AND HIGHLIGHTS ADJUSTMENTS

    def shadows_and_highlights(self, image, shadow_thresh=30, highlight_thresh=220, shadow_factor=
0.5, highlight_factor=1.5, shadow_smooth=None, highlight_smooth=None, simplify_masks=None):

        if 'pilgram' not in packages():
            install_package('pilgram')

        import pilgram

        alpha = None
        if image.mode.endswith('A'):
            alpha = image.getchannel('A')
            image = image.convert('RGB')

        grays = image.convert('L')

        if shadow_smooth is not None or highlight_smooth is not None and simplify_masks is not None:
            simplify = float(simplify_masks)
            grays = grays.filter(ImageFilter.GaussianBlur(radius=simplify))

        shadow_mask = Image.eval(grays, lambda x: 255 if x < shadow_thresh else 0)
        highlight_mask = Image.eval(grays, lambda x: 255 if x > highlight_thresh else 0)

        image_shadow = image.copy()
        image_highlight = image.copy()

        if shadow_smooth is not None:
            shadow_mask = shadow_mask.filter(ImageFilter.GaussianBlur(radius=shadow_smooth))
        if highlight_smooth is not None:
            highlight_mask = highlight_mask.filter(ImageFilter.GaussianBlur(radius=highlight_smooth))

        image_shadow = Image.eval(image_shadow, lambda x: x * shadow_factor)
        image_highlight = Image.eval(image_highlight, lambda x: x * highlight_factor)

        if shadow_smooth is not None:
            shadow_mask = shadow_mask.filter(ImageFilter.GaussianBlur(radius=shadow_smooth))
        if highlight_smooth is not None:
            highlight_mask = highlight_mask.filter(ImageFilter.GaussianBlur(radius=highlight_smooth))

        result = image.copy()
        result.paste(image_shadow, shadow_mask)
        result.paste(image_highlight, highlight_mask)
        result = pilgram.css.blending.color(result, image)

        if alpha:
            result.putalpha(alpha)

        return (result, shadow_mask, highlight_mask)

    # DRAGAN PHOTOGRAPHY FILTER


    def dragan_filter(self, image, saturation=1, contrast=1, sharpness=1, brightness=1, highpass_radius=
3, highpass_samples=1, highpass_strength=1, colorize=True):

        if 'pilgram' not in packages():
            install_package('pilgram')

        import pilgram

        alpha = None
        if image.mode == 'RGBA':
            alpha = image.getchannel('A')
        grayscale_image = image if image.mode == 'L' else image.convert('L')

        contrast_enhancer = ImageEnhance.Contrast(grayscale_image)
        contrast_image = contrast_enhancer.enhance(contrast)

        saturation_enhancer = ImageEnhance.Color(contrast_image) if image.mode != 'L' else None
        saturation_image = contrast_image if saturation_enhancer is None else saturation_enhancer.enhan
ce(saturation)

        sharpness_enhancer = ImageEnhance.Sharpness(saturation_image)
        sharpness_image = sharpness_enhancer.enhance(sharpness)

        brightness_enhancer = ImageEnhance.Brightness(sharpness_image)
        brightness_image = brightness_enhancer.enhance(brightness)

        blurred_image = brightness_image.filter(ImageFilter.GaussianBlur(radius=-highpass_radius))
        highpass_filter = ImageChops.subtract(image, blurred_image.convert('RGB'))
        blank_image = Image.new('RGB', image.size, (127, 127, 127))
        highpass_image = ImageChops.screen(blank_image, highpass_filter.resize(image.size))
        if not colorize:
            highpass_image = highpass_image.convert('L').convert('RGB')
        highpassed_image = pilgram.css.blending.overlay(brightness_image.convert('RGB'), highpass_im
age)
        for _ in range((highpass_samples if highpass_samples > 0 else 1)):
            highpassed_image = pilgram.css.blending.overlay(highpassed_image, highpass_image)

        final_image = ImageChops.blend(brightness_image.convert('RGB'), highpassed_image, highpass_
strength)

        if colorize:
            final_image = pilgram.css.blending.color(final_image, image)

        if alpha:
            final_image.putalpha(alpha)

        return final_image

    def sparkle(self, image):

        if 'pilgram' not in packages():
            install_package('pilgram')

        import pilgram

        image = image.convert('RGBA')
        contrast_enhancer = ImageEnhance.Contrast(image)
        image = contrast_enhancer.enhance(1.25)
        saturation_enhancer = ImageEnhance.Color(image)
        image = saturation_enhancer.enhance(1.5)

        bloom = image.filter(ImageFilter.GaussianBlur(radius=20))
        bloom = ImageEnhance.Brightness(bloom).enhance(1.2)
        bloom.putalpha(128)
        bloom = bloom.convert(image.mode)
        image = Image.alpha_composite(image, bloom)

        width, height = image.size
        particles = Image.new('RGBA', (width, height), (0, 0, 0, 0))
        draw = ImageDraw.Draw(particles)
        for i in range(5000):
            x = random.randint(0, width)
            y = random.randint(0, height)
            r = random.randint(0, 255)
            g = random.randint(0, 255)
            b = random.randint(0, 255)
            draw.point((x, y), fill=(r, g, b, 255))
        particles = particles.filter(ImageFilter.GaussianBlur(radius=1))
        particles.putalpha(128)

        particles2 = Image.new('RGBA', (width, height), (0, 0, 0, 0))
        draw = ImageDraw.Draw(particles2)
        for i in range(5000):
            x = random.randint(0, width)
            y = random.randint(0, height)
            r = random.randint(0, 255)
            g = random.randint(0, 255)
            b = random.randint(0, 255)
            draw.point((x, y), fill=(r, g, b, 255))
        particles2 = particles2.filter(ImageFilter.GaussianBlur(radius=1))
        particles2.putalpha(128)

        image = pilgram.css.blending.color_dodge(image, particles)
        image = pilgram.css.blending.lighten(image, particles2)

        return image

    def digital_distortion(self, image, amplitude=5, line_width=2):

        im = np.array(image)

        x, y, z = im.shape
        sine_wave = amplitude * np.sin(np.linspace(-np.pi, np.pi, y))
        sine_wave = sine_wave.astype(int)

        left_distortion = np.zeros((x, y, z), dtype=np.uint8)
        right_distortion = np.zeros((x, y, z), dtype=np.uint8)
        for i in range(y):
            left_distortion[:, i, :] = np.roll(im[:, i, :], -sine_wave[i], axis=0)
            right_distortion[:, i, :] = np.roll(im[:, i, :], sine_wave[i], axis=0)

        distorted_image = np.maximum(left_distortion, right_distortion)
        scan_lines = np.zeros((x, y), dtype=np.float32)
        scan_lines[::line_width, :] = 1
        scan_lines = np.minimum(scan_lines * amplitude*50.0, 1)  # Scale scan line values
        scan_lines = np.tile(scan_lines[:, :, np.newaxis], (1, 1, z))  # Add channel dimension
        distorted_image = np.where(scan_lines > 0, np.random.permutation(im), distorted_image)
        distorted_image = np.roll(distorted_image, np.random.randint(0, y), axis=1)

        distorted_image = Image.fromarray(distorted_image)

        return distorted_image

    def signal_distortion(self, image, amplitude):
            img_array = np.array(image)
            row_shifts = np.random.randint(-amplitude, amplitude + 1, size=img_array.shape[0])
            distorted_array = np.zeros_like(img_array)

            for y in range(img_array.shape[0]):
                x_shift = row_shifts[y]
                x_shift = x_shift + y % (amplitude * 2) - amplitude
                distorted_array[y,:] = np.roll(img_array[y,:], x_shift, axis=0)

            distorted_image = Image.fromarray(distorted_array)

            return distorted_image

        def tv_vhs_distortion(self, image, amplitude=10):
            np_image = np.array(image)
            offset_variance = int(image.height / amplitude)
            row_shifts = np.random.randint(-offset_variance, offset_variance + 1, size=image.height)
            distorted_array = np.zeros_like(np_image)

            for y in range(np_image.shape[0]):
                x_shift = row_shifts[y]
                x_shift = x_shift + y % (offset_variance * 2) - offset_variance
                distorted_array[y,:] = np.roll(np_image[y,:], x_shift, axis=0)

            h, w, c = distorted_array.shape
            x_scale = np.linspace(0, 1, w)
            y_scale = np.linspace(0, 1, h)
            x_idx = np.broadcast_to(x_scale, (h, w))
            y_idx = np.broadcast_to(y_scale.reshape(h, 1), (h, w))
            noise = np.random.rand(h, w, c) * 0.1
            distortion = np.sin(x_idx * 50) * 0.5 + np.sin(y_idx * 50) * 0.5
            distorted_array = distorted_array + distortion[:, :, np.newaxis] + noise

            distorted_image = Image.fromarray(np.uint8(distorted_array))
            distorted_image = distorted_image.resize((image.width, image.height))

            image_enhance = ImageEnhance.Color(image)
            image = image_enhance.enhance(0.5)

            effect_image = ImageChops.overlay(image, distorted_image)
            result_image = ImageChops.overlay(image, effect_image)
            result_image = ImageChops.blend(image, result_image, 0.25)

            return result_image

        def gradient(self, size, mode='horizontal', colors=None, tolerance=0):

            if isinstance(colors, str):
                colors = json.loads(colors)

            if colors is None:
                colors = {0: [255, 0, 0], 50: [0, 255, 0], 100: [0, 0, 255]}

            colors = {int(k): [int(c) for c in v] for k, v in colors.items()}

            colors[0] = colors[min(colors.keys())]
            colors[255] = colors[max(colors.keys())]
            img = Image.new('RGB', size, color=(0, 0, 0))

            color_stop_positions = sorted(colors.keys())
            color_stop_count = len(color_stop_positions)
            spectrum = []
            for i in range(256):
                start_pos = max(p for p in color_stop_positions if p <= i)
                end_pos = min(p for p in color_stop_positions if p >= i)
                start = colors[start_pos]
                end = colors[end_pos]

                if start_pos == end_pos:
                    factor = 0
                else:
                    factor = (i - start_pos) / (end_pos - start_pos)

                r = round(start[0] + (end[0] - start[0]) * factor)
                g = round(start[1] + (end[1] - start[1]) * factor)
                b = round(start[2] + (end[2] - start[2]) * factor)
                spectrum.append((r, g, b))

            draw = ImageDraw.Draw(img)
            if mode == 'horizontal':
                for x in range(size[0]):
                    pos = int(x * 100 / (size[0] - 1))
                    color = spectrum[pos]
                    if tolerance > 0:
                        color = tuple([round(c / tolerance) * tolerance for c in color])
                    draw.line((x, 0, x, size[1]), fill=color)
            elif mode == 'vertical':
                for y in range(size[1]):
                    pos = int(y * 100 / (size[1] - 1))
                    color = spectrum[pos]
                    if tolerance > 0:
                        color = tuple([round(c / tolerance) * tolerance for c in color])
                    draw.line((0, y, size[0], y), fill=color)

            blur = 1.5
            if size[0] > 512 or size[1] > 512:
                multiplier = max(size[0], size[1]) / 512
                if multiplier < 1.5:
                    multiplier = 1.5
                blur =  blur * multiplier

            img = img.filter(ImageFilter.GaussianBlur(radius=blur))

            return img

        # Version 2 optimized based on Mark Setchell's ideas
        def gradient_map(self, image, gradient_map_input, reverse=False):

            # Reverse the image
            if reverse:
                gradient_map_input = gradient_map_input.transpose(Image.FLIP_LEFT_RIGHT)

            # Convert image to Numpy array and average RGB channels
            # grey = self.greyscale(np.array(image))
            grey = np.array(image.convert('L'))
```

```
        # Convert gradient map to Numpy array
        cmap = np.array(gradient_map_input.convert('RGB'))

        # smush the map into the proper size -- 256 gradient colors
        cmap = cv2.resize(cmap, (256, 256))

        # lop off a single row for the LUT mapper
        cmap = cmap[0,:,:].reshape((256, 1, 3)).astype(np.uint8)

        # map with our "custom" LUT
        result = cv2.applyColorMap(grey, cmap)

        # Convert result to PIL image
        return Image.fromarray(result)

    def greyscale(self, image):
        if image.dtype in [np.float16, np.float32, np.float64]:
            image = np.clip(image * 255, 0, 255).astype(np.uint8)
        cc = image.shape[2] if image.ndim == 3 else 1
        if cc == 1:
            return image
        typ = cv2.COLOR_BGR2HSV
        if cc == 4:
            typ = cv2.COLOR_BGRA2GRAY
        image = cv2.cvtColor(image, typ)[:,:,2]
        return np.expand_dims(image, -1)

    # Generate Perlin Noise (Finally in house version)

    def perlin_noise(self, width, height, octaves, persistence, scale, seed=None):

        @jit(nopython=True)
        def fade(t):
            return 6 * t**5 - 15 * t**4 + 10 * t**3


        @jit(nopython=True)
        def lerp(t, a, b):
            return a + t * (b - a)


        @jit(nopython=True)
        def grad(hash, x, y, z):
            h = hash & 15
            u = x if h < 8 else y
            v = y if h < 4 else (x if h == 12 or h == 14 else z)
            return (u if (h & 1) == 0 else -u) + (v if (h & 2) == 0 else -v)


        @jit(nopython=True)
        def noise(x, y, z, p):
            X = np.int32(np.floor(x)) & 255
            Y = np.int32(np.floor(y)) & 255
            Z = np.int32(np.floor(z)) & 255

            x -= np.floor(x)
            y -= np.floor(y)
            z -= np.floor(z)

            u = fade(x)
            v = fade(y)
            w = fade(z)

            A = p[X] + Y
            AA = p[A] + Z
            AB = p[A + 1] + Z
            B = p[X + 1] + Y
            BA = p[B] + Z
            BB = p[B + 1] + Z

            return lerp(w, lerp(v, lerp(u, grad(p[AA], x, y, z), grad(p[BA], x - 1, y, z)),
                                   lerp(u, grad(p[AB], x, y - 1, z), grad(p[BB], x - 1, y - 1, z))),
                           lerp(v, lerp(u, grad(p[AA + 1], x, y, z - 1), grad(p[BA + 1], x - 1, y, z - 1)),
                                   lerp(u, grad(p[AB + 1], x, y - 1, z - 1), grad(p[BB + 1], x - 1, y - 1, z - 1))))

        if seed:
            random.seed(seed)

        p = np.arange(256, dtype=np.int32)
        random.shuffle(p)
        p = np.concatenate((p, p))

        noise_map = np.zeros((height, width))
        amplitude = 1.0
        total_amplitude = 0.0

        for octave in range(octaves):
            frequency = 2 ** octave
            total_amplitude += amplitude

            for y in range(height):
                for x in range(width):
                    nx = x / scale * frequency
                    ny = y / scale * frequency
                    noise_value = noise(nx, ny, 0, p) * amplitude
                    current_value = noise_map[y, x]
                    noise_map[y, x] = current_value + noise_value

            amplitude *= persistence

        min_value = np.min(noise_map)
        max_value = np.max(noise_map)
        noise_map = np.interp(noise_map, (min_value, max_value), (0, 255)).astype(np.uint8)
        image = Image.fromarray(noise_map, mode='L').convert("RGB")

        return image


    # Generate Perlin Power Fractal (Based on in-house perlin noise)

    def perlin_power_fractal(self, width, height, octaves, persistence, lacunarity, exponent, scale, seed=
None):

        @jit(nopython=True)
        def fade(t):
```

```
    return 6 * t**5 - 15 * t**4 + 10 * t**3

@jit(nopython=True)
def lerp(t, a, b):
    return a + t * (b - a)

@jit(nopython=True)
def grad(hash, x, y, z):
    h = hash & 15
    u = x if h < 8 else y
    v = y if h < 4 else (x if h == 12 or h == 14 else z)
    return (u if (h & 1) == 0 else -u) + (v if (h & 2) == 0 else -v)

@jit(nopython=True)
def noise(x, y, z, p):
    X = np.int32(np.floor(x)) & 255
    Y = np.int32(np.floor(y)) & 255
    Z = np.int32(np.floor(z)) & 255

    x -= np.floor(x)
    y -= np.floor(y)
    z -= np.floor(z)

    u = fade(x)
    v = fade(y)
    w = fade(z)

    A = p[X] + Y
    AA = p[A] + Z
    AB = p[A + 1] + Z
    B = p[X + 1] + Y
    BA = p[B] + Z
    BB = p[B + 1] + Z

    return lerp(w, lerp(v, lerp(u, grad(p[AA], x, y, z), grad(p[BA], x - 1, y, z)),
                        lerp(u, grad(p[AB], x, y - 1, z), grad(p[BB], x - 1, y - 1, z))),
                lerp(v, lerp(u, grad(p[AA + 1], x, y, z - 1), grad(p[BA + 1], x - 1, y, z - 1)),
                        lerp(u, grad(p[AB + 1], x, y - 1, z - 1), grad(p[BB + 1], x - 1, y - 1, z - 1))))

if seed:
    random.seed(seed)

p = np.arange(256, dtype=np.int32)
random.shuffle(p)
p = np.concatenate((p, p))

noise_map = np.zeros((height, width))
amplitude = 1.0
total_amplitude = 0.0

for octave in range(octaves):
    frequency = lacunarity ** octave
    amplitude *= persistence
    total_amplitude += amplitude

    for y in range(height):
        for x in range(width):
            nx = x / scale * frequency
```

```
                ny = y / scale * frequency
                noise_value = noise(nx, ny, 0, p) * amplitude ** exponent
                current_value = noise_map[y, x]
                noise_map[y, x] = current_value + noise_value

        min_value = np.min(noise_map)
        max_value = np.max(noise_map)
        noise_map = np.interp(noise_map, (min_value, max_value), (0, 255)).astype(np.uint8)
        image = Image.fromarray(noise_map, mode='L').convert("RGB")

        return image

    # Worley Noise Generator
    class worley_noise:

        def __init__(self, height=512, width=512, density=50, option=0, use_broadcast_ops=True, flat=Fals
e, seed=None):

            self.height = height
            self.width = width
            self.density = density
            self.use_broadcast_ops = use_broadcast_ops
            self.seed = seed
            self.generate_points_and_colors()
            self.calculate_noise(option)
            self.image = self.generateImage(option, flat_mode=flat)

        def generate_points_and_colors(self):
            rng = np.random.default_rng(self.seed)
            self.points = rng.integers(0, self.width, (self.density, 2))
            self.colors = rng.integers(0, 256, (self.density, 3))

        def calculate_noise(self, option):
            self.data = np.zeros((self.height, self.width))
            for h in range(self.height):
                for w in range(self.width):
                    distances = np.sqrt(np.sum((self.points - np.array([w, h])) ** 2, axis=1))
                    self.data[h, w] = np.sort(distances)[option]

        def broadcast_calculate_noise(self, option):
            xs = np.arange(self.width)
            ys = np.arange(self.height)
            x_dist = np.power(self.points[:, 0, np.newaxis] - xs, 2)
            y_dist = np.power(self.points[:, 1, np.newaxis] - ys, 2)
            d = np.sqrt(x_dist[:, :, np.newaxis] + y_dist[:, np.newaxis, :])
            distances = np.sort(d, axis=0)
            self.data = distances[option]

        def generateImage(self, option, flat_mode=False):
            if flat_mode:
                flat_color_data = np.zeros((self.height, self.width, 3), dtype=np.uint8)
                for h in range(self.height):
                    for w in range(self.width):
                        closest_point_idx = np.argmin(np.sum((self.points - np.array([w, h])) ** 2, axis=1))
                        flat_color_data[h, w, :] = self.colors[closest_point_idx]
                return Image.fromarray(flat_color_data, 'RGB')
            else:
                min_val, max_val = np.min(self.data), np.max(self.data)
            data_scaled = (self.data - min_val) / (max_val - min_val) * 255
            data_scaled = data_scaled.astype(np.uint8)
            return Image.fromarray(data_scaled, 'L')

    # Make Image Seamless

    def make_seamless(self, image, blending=0.5, tiled=False, tiles=2):

        if 'img2texture' not in packages():
            install_package('git+https://github.com/WASasquatch/img2texture.git')

        from img2texture import img2tex
        from img2texture._tiling import tile

        texture = img2tex(src=image, dst=None, pct=blending, return_result=True)
        if tiled:
            texture = tile(source=texture, target=None, horizontal=tiles, vertical=tiles, return_result=True)

        return texture

    # Image Displacement Warp

    def displace_image(self, image, displacement_map, amplitude):

        image = image.convert('RGB')
        displacement_map = displacement_map.convert('L')
        width, height = image.size
        result = Image.new('RGB', (width, height))

        for y in range(height):
            for x in range(width):

                # Calculate the displacements n' stuff
                displacement = displacement_map.getpixel((x, y))
                displacement_amount = amplitude * (displacement / 255)
                new_x = x + int(displacement_amount)
                new_y = y + int(displacement_amount)

                # Apply mirror reflection at edges and corners
                if new_x < 0:
                    new_x = abs(new_x)
                elif new_x >= width:
                    new_x = 2 * width - new_x - 1

                if new_y < 0:
                    new_y = abs(new_y)
                elif new_y >= height:
                    new_y = 2 * height - new_y - 1

                if new_x < 0:
                    new_x = abs(new_x)
                if new_y < 0:
                    new_y = abs(new_y)

                if new_x >= width:
                    new_x = 2 * width - new_x - 1
                if new_y >= height:
                    new_y = 2 * height - new_y - 1
                # Consider original image color at new location for RGB results, oops
                pixel = image.getpixel((new_x, new_y))
                result.putpixel((x, y), pixel)

        return result

    # Analyze Filters

    def black_white_levels(self, image):

        if 'matplotlib' not in packages():
            install_package('matplotlib')

        import matplotlib.pyplot as plt

        # convert to grayscale
        image = image.convert('L')

        # Calculate the histogram of grayscale intensities
        hist = image.histogram()

        # Find the minimum and maximum grayscale intensity values
        min_val = 0
        max_val = 255
        for i in range(256):
            if hist[i] > 0:
                min_val = i
                break
        for i in range(255, -1, -1):
            if hist[i] > 0:
                max_val = i
                break

        # Create a graph of the grayscale histogram
        plt.figure(figsize=(16, 8))
        plt.hist(image.getdata(), bins=256, range=(0, 256), color='black', alpha=0.7)
        plt.xlim([0, 256])
        plt.ylim([0, max(hist)])
        plt.axvline(min_val, color='red', linestyle='dashed')
        plt.axvline(max_val, color='red', linestyle='dashed')
        plt.title('Black and White Levels')
        plt.xlabel('Intensity')
        plt.ylabel('Frequency')

        return self.fig2img(plt)

    def channel_frequency(self, image):

        if 'matplotlib' not in packages():
            install_package('matplotlib')

        import matplotlib.pyplot as plt

        # Split the image into its RGB channels
        r, g, b = image.split()

        # Calculate the frequency of each color in each channel
```

```
        r_freq = r.histogram()
        g_freq = g.histogram()
        b_freq = b.histogram()

        # Create a graph to hold the frequency maps
        fig, axs = plt.subplots(1, 3, figsize=(16, 4))
        axs[0].set_title('Red Channel')
        axs[1].set_title('Green Channel')
        axs[2].set_title('Blue Channel')

        # Plot the frequency of each color in each channel
        axs[0].plot(range(256), r_freq, color='red')
        axs[1].plot(range(256), g_freq, color='green')
        axs[2].plot(range(256), b_freq, color='blue')

        # Set the axis limits and labels
        for ax in axs:
            ax.set_xlim([0, 255])
            ax.set_xlabel('Color Intensity')
            ax.set_ylabel('Frequency')

        return self.fig2img(plt)

    def generate_palette(self, img, n_colors=16, cell_size=128, padding=0, font_path=None, font_size=1
5, mode='chart'):
        if 'scikit-learn' not in packages():
            install_package('scikit-learn')

        from sklearn.cluster import KMeans

        img = img.resize((img.width // 2, img.height // 2), resample=Image.BILINEAR)
        pixels = np.array(img)
        pixels = pixels.reshape((-1, 3))
        kmeans = KMeans(n_clusters=n_colors, random_state=0, n_init='auto').fit(pixels)
        cluster_centers = np.uint8(kmeans.cluster_centers_)

        # Get the sorted indices based on luminance
        luminance = np.sqrt(np.dot(cluster_centers, [0.299, 0.587, 0.114]))
        sorted_indices = np.argsort(luminance)

        # Rearrange the cluster centers and luminance based on sorted indices
        cluster_centers = cluster_centers[sorted_indices]
        luminance = luminance[sorted_indices]

        # Group colors by their individual types
        reds = []
        greens = []
        blues = []
        others = []

        for i in range(n_colors):
            color = cluster_centers[i]
            color_type = np.argmax(color)  # Find the dominant color component

            if color_type == 0:
                reds.append((color, luminance[i]))
            elif color_type == 1:
                greens.append((color, luminance[i]))
    elif color_type == 2:
        blues.append((color, luminance[i]))
    else:
        others.append((color, luminance[i]))

# Sort each color group by luminance
reds.sort(key=lambda x: x[1])
greens.sort(key=lambda x: x[1])
blues.sort(key=lambda x: x[1])
others.sort(key=lambda x: x[1])

# Combine the sorted color groups
sorted_colors = reds + greens + blues + others

if mode == 'back_to_back':
    # Calculate the size of the palette image based on the number of colors
    palette_width = n_colors * cell_size
    palette_height = cell_size
else:
    # Calculate the number of rows and columns based on the number of colors
    num_rows = int(np.sqrt(n_colors))
    num_cols = int(np.ceil(n_colors / num_rows))

    # Calculate the size of the palette image based on the number of rows and columns
    palette_width = num_cols * cell_size
    palette_height = num_rows * cell_size

palette_size = (palette_width, palette_height)

palette = Image.new('RGB', palette_size, color='white')
draw = ImageDraw.Draw(palette)
if font_path:
    font = ImageFont.truetype(font_path, font_size)
else:
    font = ImageFont.load_default()

hex_palette = []
for i, (color, _) in enumerate(sorted_colors):
    if mode == 'back_to_back':
        cell_x = i * cell_size
        cell_y = 0
    else:
        row = i % num_rows
        col = i // num_rows
        cell_x = col * cell_size
        cell_y = row * cell_size

    cell_width = cell_size
    cell_height = cell_size

    color = tuple(color)

    cell = Image.new('RGB', (cell_width, cell_height), color=color)
    palette.paste(cell, (cell_x, cell_y))

    if mode != 'back_to_back':
        text_x = cell_x + (cell_width / 2)
        text_y = cell_y + cell_height + padding
```

```
                draw.text((text_x + 1, text_y + 1), f"R: {color[0]} G: {color[1]} B: {color[2]}", font=font, fill='blac
k', anchor='ms')
                draw.text((text_x, text_y), f"R: {color[0]} G: {color[1]} B: {color[2]}", font=font, fill='white', anc
hor='ms')

            hex_palette.append('#%02x%02x%02x' % color)

        return palette, '\n'.join(hex_palette)


from transformers import BlipProcessor, BlipForConditionalGeneration, BlipForQuestionAnswering

class BlipWrapper:
    def __init__(self, caption_model_id="Salesforce/blip-image-captioning-base", vqa_model_id="Salesf
orce/blip-vqa-base", device="cuda", cache_dir=None):
        self.device = torch.device(device='cuda' if device == "cuda" and torch.cuda.is_available() else 'cp
u')
        self.caption_processor = BlipProcessor.from_pretrained(caption_model_id, cache_dir=cache_dir)
        self.caption_model = BlipForConditionalGeneration.from_pretrained(caption_model_id, cache_dir=
cache_dir).to(self.device)
        self.vqa_processor = BlipProcessor.from_pretrained(vqa_model_id, cache_dir=cache_dir)
        self.vqa_model = BlipForQuestionAnswering.from_pretrained(vqa_model_id, cache_dir=cache_dir).
to(self.device)

    def generate_caption(self, image: Image.Image, min_length=50, max_length=100, num_beams=5, no
_repeat_ngram_size=2, early_stopping=False):
        self.caption_model.eval()
        inputs = self.caption_processor(images=image, return_tensors="pt").to(self.device)
        outputs = self.caption_model.generate(**inputs, min_length=min_length, max_length=max_length,
num_beams=num_beams, no_repeat_ngram_size=no_repeat_ngram_size, early_stopping=early_stoppin
g)
        return self.caption_processor.decode(outputs[0], skip_special_tokens=True)

    def answer_question(self, image: Image.Image, question: str, min_length=50, max_length=100, num_
beams=5, no_repeat_ngram_size=2, early_stopping=False):
        self.vqa_model.eval()
        inputs = self.vqa_processor(images=image, text=question, return_tensors="pt").to(self.device)
        answer_ids = self.vqa_model.generate(**inputs, min_length=min_length, max_length=max_length,
num_beams=num_beams, no_repeat_ngram_size=no_repeat_ngram_size, early_stopping=early_stoppin
g)
        return self.vqa_processor.decode(answer_ids[0], skip_special_tokens=True)


#! IMAGE FILTER NODES

# IMAGE SHADOW AND HIGHLIGHT ADJUSTMENTS

class WAS_Shadow_And_Highlight_Adjustment:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "shadow_threshold": ("FLOAT", {"default": 75, "min": 0.0, "max": 255.0, "step": 0.1}),
                "shadow_factor": ("FLOAT", {"default": 1.5, "min": -12.0, "max": 12.0, "step": 0.1}),
                "shadow_smoothing": ("FLOAT", {"default": 0.25, "min": -255.0, "max": 255.0, "step": 0.1}),
                "highlight_threshold": ("FLOAT", {"default": 175, "min": 0.0, "max": 255.0, "step": 0.1}),
                "highlight_factor": ("FLOAT", {"default": 0.5, "min": -12.0, "max": 12.0, "step": 0.1}),
                "highlight_smoothing": ("FLOAT", {"default": 0.25, "min": -255.0, "max": 255.0, "step": 0.1}),
                "simplify_isolation": ("FLOAT", {"default": 0, "min": -255.0, "max": 255.0, "step": 0.1}),
            }
        }

    RETURN_TYPES = ("IMAGE","IMAGE","IMAGE")
    RETURN_NAMES = ("image","shadow_map","highlight_map")
    FUNCTION = "apply_shadow_and_highlight"

    CATEGORY = "WAS Suite/Image/Adjustment"

    def apply_shadow_and_highlight(self, image, shadow_threshold=30, highlight_threshold=220, shado
w_factor=1.5, highlight_factor=0.5, shadow_smoothing=0, highlight_smoothing=0, simplify_isolation=0):

        WTools = WAS_Tools_Class()

        result, shadows, highlights = WTools.shadows_and_highlights(tensor2pil(image), shadow_threshol
d, highlight_threshold, shadow_factor, highlight_factor, shadow_smoothing, highlight_smoothing, simplif
y_isolation)
        result, shadows, highlights = WTools.shadows_and_highlights(tensor2pil(image), shadow_threshol
d, highlight_threshold, shadow_factor, highlight_factor, shadow_smoothing, highlight_smoothing, simplif
y_isolation)

        return (pil2tensor(result), pil2tensor(shadows), pil2tensor(highlights) )


# IMAGE PIXELATE

class WAS_Image_Pixelate:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "pixelation_size": ("FLOAT", {"default": 164, "min": 16, "max": 480, "step": 1}),
                "num_colors": ("FLOAT", {"default": 16, "min": 2, "max": 256, "step": 1}),
                "init_mode": (["k-means++", "random", "none"],),
                "max_iterations": ("FLOAT", {"default": 100, "min": 1, "max": 256, "step": 1}),
                "dither": (["False", "True"],),
                "dither_mode": (["FloydSteinberg", "Ordered"],),
            },
            "optional": {
                "color_palettes": ("LIST", {"forceInput": True}),
                "color_palette_mode": (["Brightness", "BrightnessAndTonal", "Linear", "Tonal"],),
                "reverse_palette":(["False","True"],),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "image_pixelate"
    CATEGORY = "WAS Suite/Image/Process"

    def image_pixelate(self, images, pixelation_size=164, num_colors=16, init_mode='random', max_iterat
ions=100,
                       color_palettes=None, color_palette_mode="Linear", reverse_palette='False', dither='Fals
e', dither_mode='FloydSteinberg'):

        if 'scikit-learn' not in packages():
            install_package('scikit-learn')

        pixelation_size = int(pixelation_size)
        num_colors = int(num_colors)
        max_iterations = int(max_iterations)
        color_palette_mode = color_palette_mode
        dither = (dither == 'True')

        color_palettes_list = []
        if color_palettes:
            for palette in color_palettes:
                color_palettes_list.append([color.strip() for color in palette.splitlines() if not color.startswith
('//') or not color.startswith(';')])

        reverse_palette = (True if reverse_palette == 'True' else False)

        return ( self.pixel_art_batch(images, pixelation_size, num_colors, init_mode, max_iterations, 42,
                (color_palettes_list if color_palettes_list else None), color_palette_mode, reverse_palette, dith
er, dither_mode), )

    def pixel_art_batch(self, batch, min_size, num_colors=16, init_mode='random', max_iter=100, random
_state=42,
                        palette=None, palette_mode="Linear", reverse_palette=False, dither=False, dither_mo
de='FloydSteinberg'):

        from sklearn.cluster import KMeans

        hex_palette_to_rgb = lambda hex: tuple(int(hex[i:i+2], 16) for i in (0, 2, 4))

        def flatten_colors(image, num_colors, init_mode='random', max_iter=100, random_state=42):
            np_image = np.array(image)
            pixels = np_image.reshape(-1, 3)
            kmeans = KMeans(n_clusters=num_colors, init=init_mode, max_iter=max_iter, tol=1e-3, random
_state=random_state, n_init='auto')
            labels = kmeans.fit_predict(pixels)
            colors = kmeans.cluster_centers_.astype(np.uint8)
            flattened_pixels = colors[labels]
            flattened_image = flattened_pixels.reshape(np_image.shape)
            return Image.fromarray(flattened_image)

        def dither_image(image, mode, nc):

            def clamp(value, min_value=0, max_value=255):
                return max(min(value, max_value), min_value)

            def get_new_val(old_val, nc):
                return np.round(old_val * (nc - 1)) / (nc - 1)

            def fs_dither(img, nc):
```

```
        arr = np.array(img, dtype=float) / 255
        new_width, new_height = img.size

        for ir in range(new_height):
            for ic in range(new_width):
                old_val = arr[ir, ic].copy()
                new_val = get_new_val(old_val, nc)
                arr[ir, ic] = new_val
                err = old_val - new_val

                if ic < new_width - 1:
                    arr[ir, ic + 1] += err * 7/16
                if ir < new_height - 1:
                    if ic > 0:
                        arr[ir + 1, ic - 1] += err * 3/16
                    arr[ir + 1, ic] += err * 5/16
                    if ic < new_width - 1:
                        arr[ir + 1, ic + 1] += err / 16

        carr = np.array(arr * 255, dtype=np.uint8)
        return Image.fromarray(carr)

    def ordered_dither(img, nc):
        width, height = img.size
        dither_matrix = [
            [0, 8, 2, 10],
            [12, 4, 14, 6],
            [3, 11, 1, 9],
            [15, 7, 13, 5]
        ]
        dithered_image = Image.new('RGB', (width, height))
        num_colors = min(2 ** int(np.log2(nc)), 16)

        for y in range(height):
            for x in range(width):
                old_pixel = img.getpixel((x, y))
                threshold = dither_matrix[x % 4][y % 4] * num_colors
                new_pixel = tuple(int(c * num_colors / 256) * (256 // num_colors) for c in old_pixel)
                error = tuple(old - new for old, new in zip(old_pixel, new_pixel))
                dithered_image.putpixel((x, y), new_pixel)

                if x < width - 1:
                    neighboring_pixel = img.getpixel((x + 1, y))
                    neighboring_pixel = tuple(int(c * num_colors / 256) * (256 // num_colors) for c in neigh
boring_pixel)
                    neighboring_error = tuple(neighboring - new for neighboring, new in zip(neighboring_
pixel, new_pixel))
                    neighboring_pixel = tuple(int(clamp(pixel + error * 7 / 16)) for pixel, error in zip(neighb
oring_pixel, neighboring_error))
                    img.putpixel((x + 1, y), neighboring_pixel)

                if x < width - 1 and y < height - 1:
                    neighboring_pixel = img.getpixel((x + 1, y + 1))
                    neighboring_pixel = tuple(int(c * num_colors / 256) * (256 // num_colors) for c in neigh
boring_pixel)
                    neighboring_error = tuple(neighboring - new for neighboring, new in zip(neighboring_
pixel, new_pixel))
                    neighboring_pixel = tuple(int(clamp(pixel + error * 1 / 16)) for pixel, error in zip(neighbo
ring_pixel, neighboring_error))
                    img.putpixel((x + 1, y + 1), neighboring_pixel)

                if y < height - 1:
                    neighboring_pixel = img.getpixel((x, y + 1))
                    neighboring_pixel = tuple(int(c * num_colors / 256) * (256 // num_colors) for c in neigh
boring_pixel)
                    neighboring_error = tuple(neighboring - new for neighboring, new in zip(neighboring_
pixel, new_pixel))
                    neighboring_pixel = tuple(int(clamp(pixel + error * 5 / 16)) for pixel, error in zip(neighb
oring_pixel, neighboring_error))
                    img.putpixel((x, y + 1), neighboring_pixel)

                if x > 0 and y < height - 1:
                    neighboring_pixel = img.getpixel((x - 1, y + 1))
                    neighboring_pixel = tuple(int(c * num_colors / 256) * (256 // num_colors) for c in neigh
boring_pixel)
                    neighboring_error = tuple(neighboring - new for neighboring, new in zip(neighboring_
pixel, new_pixel))
                    neighboring_pixel = tuple(int(clamp(pixel + error * 3 / 16)) for pixel, error in zip(neighb
oring_pixel, neighboring_error))
                    img.putpixel((x - 1, y + 1), neighboring_pixel)

            return dithered_image

        if mode == 'FloydSteinberg':
            return fs_dither(image, nc)
        elif mode == 'Ordered':
            return ordered_dither(image, nc)
        else:
            cstr(f"Inavlid dithering mode `{mode}` selected.").error.print()
            return image

        return image

    def color_palette_from_hex_lines(image, colors, palette_mode='Linear', reverse_palette=False):

        def color_distance(color1, color2):
            r1, g1, b1 = color1
            r2, g2, b2 = color2
            return np.sqrt((r1 - r2)**2 + (g1 - g2)**2 + (b1 - b2)**2)

        def find_nearest_color_index(color, palette):
            distances = [color_distance(color, palette_color) for palette_color in palette]
            return distances.index(min(distances))

        def find_nearest_color_index_tonal(color, palette):
            distances = [color_distance_tonal(color, palette_color) for palette_color in palette]
            return distances.index(min(distances))

        def find_nearest_color_index_both(color, palette):
            distances = [color_distance_both(color, palette_color) for palette_color in palette]
            return distances.index(min(distances))

        def color_distance_tonal(color1, color2):
            r1, g1, b1 = color1
            r2, g2, b2 = color2
            l1 = 0.299 * r1 + 0.587 * g1 + 0.114 * b1
            l2 = 0.299 * r2 + 0.587 * g2 + 0.114 * b2
            return abs(l1 - l2)

        def color_distance_both(color1, color2):
            r1, g1, b1 = color1
            r2, g2, b2 = color2
            l1 = 0.299 * r1 + 0.587 * g1 + 0.114 * b1
            l2 = 0.299 * r2 + 0.587 * g2 + 0.114 * b2
            return abs(l1 - l2) + sum(abs(c1 - c2) for c1, c2 in zip(color1, color2))

        def color_distance(color1, color2):
            return sum(abs(c1 - c2) for c1, c2 in zip(color1, color2))

        color_palette = [hex_palette_to_rgb(color.lstrip('#')) for color in colors]

        if reverse_palette:
            color_palette = color_palette[::-1]

        np_image = np.array(image)
        labels = np_image.reshape(image.size[1], image.size[0], -1)
        width, height = image.size
        new_image = Image.new("RGB", image.size)

        if palette_mode == 'Linear':
            color_palette_indices = list(range(len(color_palette)))
        elif palette_mode == 'Brightness':
            color_palette_indices = sorted(range(len(color_palette)), key=lambda i: sum(color_palette[i]) /
3)
        elif palette_mode == 'Tonal':
            color_palette_indices = sorted(range(len(color_palette)), key=lambda i: color_distance(color_
palette[i], (128, 128, 128)))
        elif palette_mode == 'BrightnessAndTonal':
            color_palette_indices = sorted(range(len(color_palette)), key=lambda i: (sum(color_palette[i])
/ 3, color_distance(color_palette[i], (128, 128, 128))))
        else:
            raise ValueError(f"Unsupported mapping mode: {palette_mode}")

        for x in range(width):
            for y in range(height):
                pixel_color = labels[y, x, :]

                if palette_mode == 'Linear':
                color_index = pixel_color[0] % len(color_palette)
                elif palette_mode == 'Brightness':
                color_index = find_nearest_color_index(pixel_color, [color_palette[i] for i in color_palette_
indices])
                elif palette_mode == 'Tonal':
                color_index = find_nearest_color_index_tonal(pixel_color, [color_palette[i] for i in color_p
alette_indices])
                elif palette_mode == 'BrightnessAndTonal':
                color_index = find_nearest_color_index_both(pixel_color, [color_palette[i] for i in color_pa
lette_indices])
                else:
                raise ValueError(f"Unsupported mapping mode: {palette_mode}")

                color = color_palette[color_palette_indices[color_index]]
                new_image.putpixel((x, y), color)
        return new_image

    pil_images = [tensor2pil(image) for image in batch]
    pixel_art_images = []
    original_sizes = []
    total_images = len(pil_images)
    for image in pil_images:
        width, height = image.size
        original_sizes.append((width, height))
        if max(width, height) > min_size:
            if width > height:
                new_width = min_size
                new_height = int(height * (min_size / width))
            else:
                new_height = min_size
                new_width = int(width * (min_size / height))
            pixel_art_images.append(image.resize((new_width, int(new_height)), Image.NEAREST))
        else:
            pixel_art_images.append(image)
    if init_mode != 'none':
        pixel_art_images = [flatten_colors(image, num_colors, init_mode) for image in pixel_art_images]
    if dither:
        pixel_art_images = [dither_image(image, dither_mode, num_colors) for image in pixel_art_image
s]
    if palette:
        pixel_art_images = [color_palette_from_hex_lines(pixel_art_image, palette[i], palette_mode, rever
se_palette) for i, pixel_art_image in enumerate(pixel_art_images)]
    else:
        pixel_art_images = pixel_art_images
    pixel_art_images = [image.resize(size, Image.NEAREST) for image, size in zip(pixel_art_images, ori
ginal_sizes)]

    tensor_images = [pil2tensor(image) for image in pixel_art_images]

    batch_tensor = torch.cat(tensor_images, dim=0)
    return batch_tensor

# SIMPLE IMAGE ADJUST

class WAS_Image_Filters:
    def __init__(self):
    pass

    @classmethod
    def INPUT_TYPES(cls):
    return {
        "required": {
            "image": ("IMAGE",),
            "brightness": ("FLOAT", {"default": 0.0, "min": -1.0, "max": 1.0, "step": 0.01}),
            "contrast": ("FLOAT", {"default": 1.0, "min": -1.0, "max": 2.0, "step": 0.01}),
            "saturation": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 5.0, "step": 0.01}),
            "sharpness": ("FLOAT", {"default": 1.0, "min": -5.0, "max": 5.0, "step": 0.01}),
            "blur": ("INT", {"default": 0, "min": 0, "max": 16, "step": 1}),
            "gaussian_blur": ("FLOAT", {"default": 0.0, "min": 0.0, "max": 1024.0, "step": 0.1}),
            "edge_enhance": ("FLOAT", {"default": 0.0, "min": 0.0, "max": 1.0, "step": 0.01}),
            "detail_enhance": (["false", "true"],),
        },
    }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_filters"

    CATEGORY = "WAS Suite/Image/Filter"

    def image_filters(self, image, brightness, contrast, saturation, sharpness, blur, gaussian_blur, edge_e
nhance, detail_enhance):


    tensors = []
    if len(image) > 1:
        for img in image:

            pil_image = None

            # Apply NP Adjustments
            if brightness > 0.0 or brightness < 0.0:
                # Apply brightness
                img = np.clip(img + brightness, 0.0, 1.0)

            if contrast > 1.0 or contrast < 1.0:
                # Apply contrast
                img = np.clip(img * contrast, 0.0, 1.0)

            # Apply PIL Adjustments
            if saturation > 1.0 or saturation < 1.0:
                # PIL Image
                pil_image = tensor2pil(img)
                # Apply saturation
                pil_image = ImageEnhance.Color(pil_image).enhance(saturation)

            if sharpness > 1.0 or sharpness < 1.0:
                # Assign or create PIL Image
                pil_image = pil_image if pil_image else tensor2pil(img)
                # Apply sharpness
                pil_image = ImageEnhance.Sharpness(pil_image).enhance(sharpness)

            if blur > 0:
                # Assign or create PIL Image
                pil_image = pil_image if pil_image else tensor2pil(img)
                # Apply blur
                for _ in range(blur):
                    pil_image = pil_image.filter(ImageFilter.BLUR)

            if gaussian_blur > 0.0:
                # Assign or create PIL Image
                pil_image = pil_image if pil_image else tensor2pil(img)
                # Apply Gaussian blur
                pil_image = pil_image.filter(
                    ImageFilter.GaussianBlur(radius=gaussian_blur))

            if edge_enhance > 0.0:
                # Assign or create PIL Image
                pil_image = pil_image if pil_image else tensor2pil(img)
                # Edge Enhancement
                edge_enhanced_img = pil_image.filter(ImageFilter.EDGE_ENHANCE_MORE)
                # Blend Mask
                blend_mask = Image.new(
                    mode="L", size=pil_image.size, color=(round(edge_enhance * 255)))
                # Composite Original and Enhanced Version
                pil_image = Image.composite(
                    edge_enhanced_img, pil_image, blend_mask)
                # Clean-up
                del blend_mask, edge_enhanced_img

            if detail_enhance == "true":
                pil_image = pil_image if pil_image else tensor2pil(img)
                pil_image = pil_image.filter(ImageFilter.DETAIL)

            # Output image
            out_image = (pil2tensor(pil_image) if pil_image else img.unsqueeze(0))

            tensors.append(out_image)

        tensors = torch.cat(tensors, dim=0)

    else:

        pil_image = None
        img = image

        # Apply NP Adjustments
        if brightness > 0.0 or brightness < 0.0:
            # Apply brightness
            img = np.clip(img + brightness, 0.0, 1.0)

        if contrast > 1.0 or contrast < 1.0:
            # Apply contrast
            img = np.clip(img * contrast, 0.0, 1.0)

        # Apply PIL Adjustments
        if saturation > 1.0 or saturation < 1.0:
            # PIL Image
            pil_image = tensor2pil(img)
            # Apply saturation
            pil_image = ImageEnhance.Color(pil_image).enhance(saturation)

        if sharpness > 1.0 or sharpness < 1.0:
            # Assign or create PIL Image
            pil_image = pil_image if pil_image else tensor2pil(img)
            # Apply sharpness
            pil_image = ImageEnhance.Sharpness(pil_image).enhance(sharpness)

        if blur > 0:
            # Assign or create PIL Image
            pil_image = pil_image if pil_image else tensor2pil(img)
            # Apply blur
            for _ in range(blur):
                pil_image = pil_image.filter(ImageFilter.BLUR)

        if gaussian_blur > 0.0:
            # Assign or create PIL Image
            pil_image = pil_image if pil_image else tensor2pil(img)
            # Apply Gaussian blur
            pil_image = pil_image.filter(
                ImageFilter.GaussianBlur(radius=gaussian_blur))

            if edge_enhance > 0.0:
                # Assign or create PIL Image
                pil_image = pil_image if pil_image else tensor2pil(img)
                # Edge Enhancement
                edge_enhanced_img = pil_image.filter(ImageFilter.EDGE_ENHANCE_MORE)
                # Blend Mask
                blend_mask = Image.new(
                    mode="L", size=pil_image.size, color=(round(edge_enhance * 255)))
                # Composite Original and Enhanced Version
                pil_image = Image.composite(
                    edge_enhanced_img, pil_image, blend_mask)
                # Clean-up
                del blend_mask, edge_enhanced_img

            if detail_enhance == "true":
                pil_image = pil_image if pil_image else tensor2pil(img)
                pil_image = pil_image.filter(ImageFilter.DETAIL)

            # Output image
            out_image = (pil2tensor(pil_image) if pil_image else img)

            tensors = out_image

        return (tensors, )

# RICHARDSON LUCY SHARPEN

class WAS_Lucy_Sharpen:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "iterations": ("INT", {"default": 2, "min": 1, "max": 12, "step": 1}),
                "kernel_size": ("INT", {"default": 3, "min": 1, "max": 16, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "sharpen"

    CATEGORY = "WAS Suite/Image/Filter"

    def sharpen(self, images, iterations, kernel_size):

        tensors = []
        if len(images) > 1:
            for img in images:
                tensors.append(pil2tensor(self.lucy_sharpen(tensor2pil(img), iterations, kernel_size)))
            tensors = torch.cat(tensors, dim=0)
        else:
            return (pil2tensor(self.lucy_sharpen(tensor2pil(images), iterations, kernel_size)),)
        return (tensors,)


    def lucy_sharpen(self, image, iterations=10, kernel_size=3):

        from scipy.signal import convolve2d

        image_array = np.array(image, dtype=np.float32) / 255.0
        kernel = np.ones((kernel_size, kernel_size), dtype=np.float32) / (kernel_size ** 2)
        sharpened_channels = []

        padded_image_array = np.pad(image_array, ((kernel_size, kernel_size), (kernel_size, kernel_size),
(0, 0)), mode='edge')

        for channel in range(3):
            channel_array = padded_image_array[:, :, channel]

            for _ in range(iterations):
                blurred_channel = convolve2d(channel_array, kernel, mode='same')
                ratio = channel_array / (blurred_channel + 1e-6)
                channel_array *= convolve2d(ratio, kernel, mode='same')

            sharpened_channels.append(channel_array)

        cropped_sharpened_image_array = np.stack(sharpened_channels, axis=-1)[kernel_size:-kernel_siz
e, kernel_size:-kernel_size, :]
        sharpened_image_array = np.clip(cropped_sharpened_image_array * 255.0, 0, 255).astype(np.uint
8)
        sharpened_image = Image.fromarray(sharpened_image_array)
        return sharpened_image

# IMAGE STYLE FILTER

class WAS_Image_Style_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "style": ([
                    "1977",
                    "aden",
                    "brannan",
                    "brooklyn",
                    "clarendon",
                    "earlybird",
                    "fairy tale",
                    "gingham",
                    "hudson",
                    "inkwell",
                    "kelvin",
                    "lark",
                    "lofi",
                    "maven",
                    "mayfair",
                    "moon",
                    "nashville",
                    "perpetua",
                    "reyes",
                    "rise",
                    "slumber",
                    "stinson",
                    "toaster",
                    "valencia",
                    "walden",
                    "willow",
                    "xpro2"
                ],),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_style_filter"

    CATEGORY = "WAS Suite/Image/Filter"

    def image_style_filter(self, image, style):

        # Install Pilgram
        if 'pilgram' not in packages():
            install_package('pilgram')

        # Import Pilgram module
        import pilgram

        # WAS Filters
        WTools = WAS_Tools_Class()

        # Apply blending
        tensors = []
        for img in image:
            if style == "1977":
                tensors.append(pil2tensor(pilgram._1977(tensor2pil(img))))
            elif style == "aden":
                tensors.append(pil2tensor(pilgram.aden(tensor2pil(img))))
            elif style == "brannan":
                tensors.append(pil2tensor(pilgram.brannan(tensor2pil(img))))
            elif style == "brooklyn":
                tensors.append(pil2tensor(pilgram.brooklyn(tensor2pil(img))))
            elif style == "clarendon":
                tensors.append(pil2tensor(pilgram.clarendon(tensor2pil(img))))
            elif style == "earlybird":
                tensors.append(pil2tensor(pilgram.earlybird(tensor2pil(img))))
            elif style == "fairy tale":
                tensors.append(pil2tensor(WTools.sparkle(tensor2pil(img))))
            elif style == "gingham":
                tensors.append(pil2tensor(pilgram.gingham(tensor2pil(img))))
            elif style == "hudson":
                tensors.append(pil2tensor(pilgram.hudson(tensor2pil(img))))
            elif style == "inkwell":
                tensors.append(pil2tensor(pilgram.inkwell(tensor2pil(img))))
            elif style == "kelvin":
                tensors.append(pil2tensor(pilgram.kelvin(tensor2pil(img))))
            elif style == "lark":
                tensors.append(pil2tensor(pilgram.lark(tensor2pil(img))))
            elif style == "lofi":
                tensors.append(pil2tensor(pilgram.lofi(tensor2pil(img))))
            elif style == "maven":
                tensors.append(pil2tensor(pilgram.maven(tensor2pil(img))))
            elif style == "mayfair":
                tensors.append(pil2tensor(pilgram.mayfair(tensor2pil(img))))
            elif style == "moon":
                tensors.append(pil2tensor(pilgram.moon(tensor2pil(img))))
            elif style == "nashville":
                tensors.append(pil2tensor(pilgram.nashville(tensor2pil(img))))
            elif style == "perpetua":
                tensors.append(pil2tensor(pilgram.perpetua(tensor2pil(img))))
            elif style == "reyes":
                tensors.append(pil2tensor(pilgram.reyes(tensor2pil(img))))
            elif style == "rise":
                tensors.append(pil2tensor(pilgram.rise(tensor2pil(img))))
            elif style == "slumber":
                tensors.append(pil2tensor(pilgram.slumber(tensor2pil(img))))
            elif style == "stinson":
                tensors.append(pil2tensor(pilgram.stinson(tensor2pil(img))))
            elif style == "toaster":
                tensors.append(pil2tensor(pilgram.toaster(tensor2pil(img))))
            elif style == "valencia":
                tensors.append(pil2tensor(pilgram.valencia(tensor2pil(img))))
            elif style == "walden":
                tensors.append(pil2tensor(pilgram.walden(tensor2pil(img))))
            elif style == "willow":
                tensors.append(pil2tensor(pilgram.willow(tensor2pil(img))))
            elif style == "xpro2":
                tensors.append(pil2tensor(pilgram.xpro2(tensor2pil(img))))
            else:
                tensors.append(img)

        tensors = torch.cat(tensors, dim=0)

        return (tensors, )


# IMAGE CROP FACE

class WAS_Image_Crop_Face:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "crop_padding_factor": ("FLOAT", {"default": 0.25, "min": 0.0, "max": 2.0, "step": 0.01}),
                "cascade_xml": ([
                                "lbpcascade_animeface.xml",
                                "haarcascade_frontalface_default.xml",
                                "haarcascade_frontalface_alt.xml",
                                "haarcascade_frontalface_alt2.xml",
                                "haarcascade_frontalface_alt_tree.xml",
                        "haarcascade_profileface.xml",
                        "haarcascade_upperbody.xml",
                        "haarcascade_eye.xml"
                        ],),
                }
            }

    RETURN_TYPES = ("IMAGE", "CROP_DATA")
    FUNCTION = "image_crop_face"

    CATEGORY = "WAS Suite/Image/Process"

    def image_crop_face(self, image, cascade_xml=None, crop_padding_factor=0.25):
        return self.crop_face(tensor2pil(image), cascade_xml, crop_padding_factor)

    def crop_face(self, image, cascade_name=None, padding=0.25):

        import cv2

        img = np.array(image.convert('RGB'))

        face_location = None

        cascades = [ os.path.join(os.path.join(WAS_SUITE_ROOT, 'res'), 'lbpcascade_animeface.xml'),
                    os.path.join(os.path.join(WAS_SUITE_ROOT, 'res'), 'haarcascade_frontalface_default.xml'),
                    os.path.join(os.path.join(WAS_SUITE_ROOT, 'res'), 'haarcascade_frontalface_alt.xml'),
                    os.path.join(os.path.join(WAS_SUITE_ROOT, 'res'), 'haarcascade_frontalface_alt2.xml'),
                    os.path.join(os.path.join(WAS_SUITE_ROOT, 'res'), 'haarcascade_frontalface_alt_tree.xml'),
                    os.path.join(os.path.join(WAS_SUITE_ROOT, 'res'), 'haarcascade_profileface.xml'),
                    os.path.join(os.path.join(WAS_SUITE_ROOT, 'res'), 'haarcascade_upperbody.xml') ]

        if cascade_name:
            for cascade in cascades:
                if os.path.basename(cascade) == cascade_name:
                    cascades.remove(cascade)
                    cascades.insert(0, cascade)
                    break

        faces = None
        if not face_location:
            for cascade in cascades:
                if not os.path.exists(cascade):
                    cstr(f"Unable to find cascade XML file at `{cascade}`. Did you pull the latest files from http
s://github.com/WASasquatch/was-node-suite-comfyui repo?").error.print()
                    return (pil2tensor(Image.new("RGB", (512,512), (0,0,0))), False)
                face_cascade = cv2.CascadeClassifier(cascade)
                gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5)
                if len(faces) != 0:
                    cstr(f"Face found with: {os.path.basename(cascade)}").msg.print()
                    break
            if len(faces) == 0:
                cstr("No faces found in the image!").warning.print()
                return (pil2tensor(Image.new("RGB", (512,512), (0,0,0))), False)
        else:
            cstr("Face found with: face_recognition model").warning.print()
            faces = face_location
        # Assume there is only one face in the image
        x, y, w, h = faces[0]

        # Check if the face region aligns with the edges of the original image
        left_adjust = max(0, -x)
        right_adjust = max(0, x + w - img.shape[1])
        top_adjust = max(0, -y)
        bottom_adjust = max(0, y + h - img.shape[0])

        # Check if the face region is near any edges, and if so, pad in the opposite direction
        if left_adjust < w:
            x += right_adjust
        elif right_adjust < w:
            x -= left_adjust
        if top_adjust < h:
            y += bottom_adjust
        elif bottom_adjust < h:
            y -= top_adjust

        w -= left_adjust + right_adjust
        h -= top_adjust + bottom_adjust

        # Calculate padding around face
        face_size = min(h, w)
        y_pad = int(face_size * padding)
        x_pad = int(face_size * padding)

        # Calculate square coordinates around face
        center_x = x + w // 2
        center_y = y + h // 2
        half_size = (face_size + max(x_pad, y_pad)) // 2
        top = max(0, center_y - half_size)
        bottom = min(img.shape[0], center_y + half_size)
        left = max(0, center_x - half_size)
        right = min(img.shape[1], center_x + half_size)

        # Ensure square crop of the original image
        crop_size = min(right - left, bottom - top)
        left = center_x - crop_size // 2
        right = center_x + crop_size // 2
        top = center_y - crop_size // 2
        bottom = center_y + crop_size // 2

        # Crop face from original image
        face_img = img[top:bottom, left:right, :]

        # Resize image
        size = max(face_img.copy().shape[:2])
        pad_h = (size - face_img.shape[0]) // 2
        pad_w = (size - face_img.shape[1]) // 2
        face_img = cv2.copyMakeBorder(face_img, pad_h, pad_h, pad_w, pad_w, cv2.BORDER_CONSTAN
T, value=[0,0,0])
        min_size = 64 # Set minimum size for padded image
        if size < min_size:
            size = min_size
        face_img = cv2.resize(face_img, (size, size))

        # Convert numpy array back to PIL image
        face_img = Image.fromarray(face_img)

        # Resize image to a multiple of 64
        original_size = face_img.size
        face_img.resize((((face_img.size[0] // 64) * 64 + 64), ((face_img.size[1] // 64) * 64 + 64)))

        # Return face image and coordinates
        return (pil2tensor(face_img.convert('RGB')), (original_size, (left, top, right, bottom)))


# IMAGE PASTE FACE CROP

class WAS_Image_Paste_Face_Crop:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "crop_image": ("IMAGE",),
                "crop_data": ("CROP_DATA",),
                "crop_blending": ("FLOAT", {"default": 0.25, "min": 0.0, "max": 1.0, "step": 0.01}),
                "crop_sharpening": ("INT", {"default": 0, "min": 0, "max": 3, "step": 1}),
            }
        }

    RETURN_TYPES = ("IMAGE", "IMAGE")
    RETURN_NAMES = ("IMAGE", "MASK_IMAGE")
    FUNCTION = "image_paste_face"

    CATEGORY = "WAS Suite/Image/Process"

    def image_paste_face(self, image, crop_image, crop_data=None, crop_blending=0.25, crop_sharpeni
ng=0):

        if crop_data == False:
            cstr("No valid crop data found!").error.print()
            return (image, pil2tensor(Image.new("RGB", tensor2pil(image).size, (0,0,0))))

        result_image, result_mask = self.paste_image(tensor2pil(image), tensor2pil(crop_image), crop_dat
a, crop_blending, crop_sharpening)
        return(result_image, result_mask)

    def paste_image(self, image, crop_image, crop_data, blend_amount=0.25, sharpen_amount=1):

        def lingrad(size, direction, white_ratio):
            image = Image.new('RGB', size)
            draw = ImageDraw.Draw(image)
            if direction == 'vertical':
                black_end = int(size[1] * (1 - white_ratio))
                range_start = 0
                range_end = size[1]
                range_step = 1
                for y in range(range_start, range_end, range_step):
                    color_ratio = y / size[1]
                    if y <= black_end:
```

```
                color = (0, 0, 0)
            else:
                color_value = int(((y - black_end) / (size[1] - black_end)) * 255)
                color = (color_value, color_value, color_value)
            draw.line([(0, y), (size[0], y)], fill=color)
    elif direction == 'horizontal':
        black_end = int(size[0] * (1 - white_ratio))
        range_start = 0
        range_end = size[0]
        range_step = 1
        for x in range(range_start, range_end, range_step):
            color_ratio = x / size[0]
            if x <= black_end:
                color = (0, 0, 0)
            else:
                color_value = int(((x - black_end) / (size[0] - black_end)) * 255)
                color = (color_value, color_value, color_value)
            draw.line([(x, 0), (x, size[1])], fill=color)

    return image.convert("L")

crop_size, (top, left, right, bottom) = crop_data
crop_image = crop_image.resize(crop_size)

if sharpen_amount > 0:
    for _ in range(int(sharpen_amount)):
        crop_image = crop_image.filter(ImageFilter.SHARPEN)

blended_image = Image.new('RGBA', image.size, (0, 0, 0, 255))
blended_mask = Image.new('L', image.size, 0)
crop_padded = Image.new('RGBA', image.size, (0, 0, 0, 0))
blended_image.paste(image, (0, 0))
crop_padded.paste(crop_image, (top, left))
crop_mask = Image.new('L', crop_image.size, 0)

if top > 0:
    gradient_image = ImageOps.flip(lingrad(crop_image.size, 'vertical', blend_amount))
    crop_mask = ImageChops.screen(crop_mask, gradient_image)

if left > 0:
    gradient_image = ImageOps.mirror(lingrad(crop_image.size, 'horizontal', blend_amount))
    crop_mask = ImageChops.screen(crop_mask, gradient_image)

if right < image.width:
    gradient_image = lingrad(crop_image.size, 'horizontal', blend_amount)
    crop_mask = ImageChops.screen(crop_mask, gradient_image)

if bottom < image.height:
    gradient_image = lingrad(crop_image.size, 'vertical', blend_amount)
    crop_mask = ImageChops.screen(crop_mask, gradient_image)

crop_mask = ImageOps.invert(crop_mask)
blended_mask.paste(crop_mask, (top, left))
blended_mask = blended_mask.convert("L")
blended_image.paste(crop_padded, (0, 0), blended_mask)

return (pil2tensor(blended_image.convert("RGB")), pil2tensor(blended_mask.convert("RGB")))
```

```
# IMAGE CROP LOCATION

class WAS_Image_Crop_Location:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "top": ("INT", {"default":0, "max": 10000000, "min":0, "step":1}),
                "left": ("INT", {"default":0, "max": 10000000, "min":0, "step":1}),
                "right": ("INT", {"default":256, "max": 10000000, "min":0, "step":1}),
                "bottom": ("INT", {"default":256, "max": 10000000, "min":0, "step":1}),
            }
        }

    RETURN_TYPES = ("IMAGE", "CROP_DATA")
    FUNCTION = "image_crop_location"

    CATEGORY = "WAS Suite/Image/Process"

    def image_crop_location(self, image, top=0, left=0, right=256, bottom=256):
        image = tensor2pil(image)
        img_width, img_height = image.size

        # Calculate the final coordinates for cropping
        crop_top = max(top, 0)
        crop_left = max(left, 0)
        crop_bottom = min(bottom, img_height)
        crop_right = min(right, img_width)

        # Ensure that the cropping region has non-zero width and height
        crop_width = crop_right - crop_left
        crop_height = crop_bottom - crop_top
        if crop_width <= 0 or crop_height <= 0:
            raise ValueError("Invalid crop dimensions. Please check the values for top, left, right, and botto
m.")

        # Crop the image and resize
        crop = image.crop((crop_left, crop_top, crop_right, crop_bottom))
        crop_data = (crop.size, (crop_left, crop_top, crop_right, crop_bottom))
        crop = crop.resize((((crop.size[0] // 8) * 8), ((crop.size[1] // 8) * 8)))

        return (pil2tensor(crop), crop_data)


# IMAGE SQUARE CROP LOCATION

class WAS_Image_Crop_Square_Location:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "x": ("INT", {"default":0, "max": 24576, "min":0, "step":1}),
                "y": ("INT", {"default":0, "max": 24576, "min":0, "step":1}),
                "size": ("INT", {"default":256, "max": 4096, "min":5, "step":1}),
            }
        }

    RETURN_TYPES = ("IMAGE", "CROP_DATA")
    FUNCTION = "image_crop_location"

    CATEGORY = "WAS Suite/Image/Process"

    def image_crop_location(self, image, x=256, y=256, size=512):

        image = tensor2pil(image)
        img_width, img_height = image.size
        exp_size = size // 2
        left = max(x - exp_size, 0)
        top = max(y - exp_size, 0)
        right = min(x + exp_size, img_width)
        bottom = min(y + exp_size, img_height)

        if right - left < size:
            if right < img_width:
                right = min(right + size - (right - left), img_width)
            elif left > 0:
                left = max(left - (size - (right - left)), 0)
        if bottom - top < size:
            if bottom < img_height:
                bottom = min(bottom + size - (bottom - top), img_height)
            elif top > 0:
                top = max(top - (size - (bottom - top)), 0)

        crop = image.crop((left, top, right, bottom))

        # Original Crop Data
        crop_data = (crop.size, (left, top, right, bottom))

        # Output resize
        crop = crop.resize((((crop.size[0] // 8) * 8), ((crop.size[1] // 8) * 8)))

        return (pil2tensor(crop), crop_data)


# IMAGE SQUARE CROP LOCATION

class WAS_Image_Tile_Batch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "num_tiles": ("INT", {"default":4, "max": 64, "min":2, "step":1}),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("IMAGES",)
    FUNCTION = "tile_image"

    CATEGORY = "WAS Suite/Image/Process"

    def tile_image(self, image, num_tiles=6):
        image = tensor2pil(image.squeeze(0))
        img_width, img_height = image.size

        num_rows = int(num_tiles ** 0.5)
        num_cols = (num_tiles + num_rows - 1) // num_rows
        tile_width = img_width // num_cols
        tile_height = img_height // num_rows

        tiles = []
        for y in range(0, img_height, tile_height):
            for x in range(0, img_width, tile_width):
                tile = image.crop((x, y, x + tile_width, y + tile_height))
                tiles.append(pil2tensor(tile))

        tiles = torch.stack(tiles, dim=0).squeeze(1)

        return (tiles, )


# IMAGE PASTE CROP

class WAS_Image_Paste_Crop:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
                "required": {
                    "image": ("IMAGE",),
                    "crop_image": ("IMAGE",),
                    "crop_data": ("CROP_DATA",),
                    "crop_blending": ("FLOAT", {"default": 0.25, "min": 0.0, "max": 1.0, "step": 0.01}),
                    "crop_sharpening": ("INT", {"default": 0, "min": 0, "max": 3, "step": 1}),
                }
            }

    RETURN_TYPES = ("IMAGE", "IMAGE")
    FUNCTION = "image_paste_crop"

    CATEGORY = "WAS Suite/Image/Process"

    def image_paste_crop(self, image, crop_image, crop_data=None, crop_blending=0.25, crop_sharpen
ing=0):

        if crop_data == False:
            cstr("No valid crop data found!").error.print()
            return (image, pil2tensor(Image.new("RGB", tensor2pil(image).size, (0,0,0))))
        result_image, result_mask = self.paste_image(tensor2pil(image), tensor2pil(crop_image), crop_dat
a, crop_blending, crop_sharpening)

        return (result_image, result_mask)

    def paste_image(self, image, crop_image, crop_data, blend_amount=0.25, sharpen_amount=1):

        def lingrad(size, direction, white_ratio):
            image = Image.new('RGB', size)
            draw = ImageDraw.Draw(image)
            if direction == 'vertical':
                black_end = int(size[1] * (1 - white_ratio))
                range_start = 0
                range_end = size[1]
                range_step = 1
                for y in range(range_start, range_end, range_step):
                    color_ratio = y / size[1]
                    if y <= black_end:
                        color = (0, 0, 0)
                    else:
                        color_value = int(((y - black_end) / (size[1] - black_end)) * 255)
                        color = (color_value, color_value, color_value)
                    draw.line([(0, y), (size[0], y)], fill=color)
            elif direction == 'horizontal':
                black_end = int(size[0] * (1 - white_ratio))
                range_start = 0
                range_end = size[0]
                range_step = 1
                for x in range(range_start, range_end, range_step):
                    color_ratio = x / size[0]
                    if x <= black_end:
                        color = (0, 0, 0)
                    else:
                        color_value = int(((x - black_end) / (size[0] - black_end)) * 255)
                        color = (color_value, color_value, color_value)
                    draw.line([(x, 0), (x, size[1])], fill=color)

            return image.convert("L")

        crop_size, (left, top, right, bottom) = crop_data
        crop_image = crop_image.resize(crop_size)

        if sharpen_amount > 0:
            for _ in range(int(sharpen_amount)):
                crop_image = crop_image.filter(ImageFilter.SHARPEN)

        blended_image = Image.new('RGBA', image.size, (0, 0, 0, 255))
        blended_mask = Image.new('L', image.size, 0)
        crop_padded = Image.new('RGBA', image.size, (0, 0, 0, 0))
        blended_image.paste(image, (0, 0))
        crop_padded.paste(crop_image, (left, top))
        crop_mask = Image.new('L', crop_image.size, 0)

        if top > 0:
            gradient_image = ImageOps.flip(lingrad(crop_image.size, 'vertical', blend_amount))
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        if left > 0:
            gradient_image = ImageOps.mirror(lingrad(crop_image.size, 'horizontal', blend_amount))
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        if right < image.width:
            gradient_image = lingrad(crop_image.size, 'horizontal', blend_amount)
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        if bottom < image.height:
            gradient_image = lingrad(crop_image.size, 'vertical', blend_amount)
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        crop_mask = ImageOps.invert(crop_mask)
        blended_mask.paste(crop_mask, (left, top))
        blended_mask = blended_mask.convert("L")
        blended_image.paste(crop_padded, (0, 0), blended_mask)

        return (pil2tensor(blended_image.convert("RGB")), pil2tensor(blended_mask.convert("RGB")))


# IMAGE PASTE CROP BY LOCATION

class WAS_Image_Paste_Crop_Location:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
                "required": {
                    "image": ("IMAGE",),
                    "crop_image": ("IMAGE",),
                    "top": ("INT", {"default":0, "max": 10000000, "min":0, "step":1}),
                    "left": ("INT", {"default":0, "max": 10000000, "min":0, "step":1}),
                    "right": ("INT", {"default":256, "max": 10000000, "min":0, "step":1}),
                    "bottom": ("INT", {"default":256, "max": 10000000, "min":0, "step":1}),
                    "crop_blending": ("FLOAT", {"default": 0.25, "min": 0.0, "max": 1.0, "step": 0.01}),
                    "crop_sharpening": ("INT", {"default": 0, "min": 0, "max": 3, "step": 1}),
                }
            }

    RETURN_TYPES = ("IMAGE", "IMAGE")
    FUNCTION = "image_paste_crop_location"

    CATEGORY = "WAS Suite/Image/Process"

    def image_paste_crop_location(self, image, crop_image, top=0, left=0, right=256, bottom=256, crop_
blending=0.25, crop_sharpening=0):
        result_image, result_mask = self.paste_image(tensor2pil(image), tensor2pil(crop_image), top, left,
right, bottom, crop_blending, crop_sharpening)
        return (result_image, result_mask)

    def paste_image(self, image, crop_image, top=0, left=0, right=256, bottom=256, blend_amount=0.2
5, sharpen_amount=1):

        image = image.convert("RGBA")
        crop_image = crop_image.convert("RGBA")

        def inset_border(image, border_width=20, border_color=(0)):
            width, height = image.size
            bordered_image = Image.new(image.mode, (width, height), border_color)
            bordered_image.paste(image, (0, 0))
            draw = ImageDraw.Draw(bordered_image)
            draw.rectangle((0, 0, width-1, height-1), outline=border_color, width=border_width)
            return bordered_image

        img_width, img_height = image.size

        # Ensure that the coordinates are within the image bounds
        top = min(max(top, 0), img_height)
        left = min(max(left, 0), img_width)
        bottom = min(max(bottom, 0), img_height)
        right = min(max(right, 0), img_width)

        crop_size = (right - left, bottom - top)
        crop_img = crop_image.resize(crop_size)
        crop_img = crop_img.convert("RGBA")

        if sharpen_amount > 0:
            for _ in range(sharpen_amount):
                crop_img = crop_img.filter(ImageFilter.SHARPEN)

        if blend_amount > 1.0:
            blend_amount = 1.0
        elif blend_amount < 0.0:
            blend_amount = 0.0
        blend_ratio = (max(crop_size) / 2) * float(blend_amount)

        blend = image.copy()
        mask = Image.new("L", image.size, 0)

        mask_block = Image.new("L", crop_size, 255)
        mask_block = inset_border(mask_block, int(blend_ratio/2), (0))

        Image.Image.paste(mask, mask_block, (left, top))
        blend.paste(crop_img, (left, top), crop_img)

        mask = mask.filter(ImageFilter.BoxBlur(radius=blend_ratio/4))
        mask = mask.filter(ImageFilter.GaussianBlur(radius=blend_ratio/4))

        blend.putalpha(mask)
        image = Image.alpha_composite(image, blend)

        return (pil2tensor(image), pil2tensor(mask.convert('RGB')))


# IMAGE GRID IMAGE

class WAS_Image_Grid_Image_Batch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "border_width": ("INT", {"default":3, "min": 0, "max": 100, "step":1}),
                "number_of_columns": ("INT", {"default":6, "min": 1, "max": 24, "step":1}),
                "max_cell_size": ("INT", {"default":256, "min":32, "max":2048, "step":1}),
                "border_red": ("INT", {"default":0, "min": 0, "max": 255, "step":1}),
                "border_green": ("INT", {"default":0, "min": 0, "max": 255, "step":1}),
                "border_blue": ("INT", {"default":0, "min": 0, "max": 255, "step":1}),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "smart_grid_image"

    CATEGORY = "WAS Suite/Image/Process"

    def smart_grid_image(self, images, number_of_columns=6, max_cell_size=256, add_border=False, b
order_red=255, border_green=255, border_blue=255, border_width=3):

        cols = number_of_columns
        border_color = (border_red, border_green, border_blue)

        images_resized = []
        max_row_height = 0

        for tensor_img in images:
            img = tensor2pil(tensor_img)
            img_w, img_h = img.size
            aspect_ratio = img_w / img_h

            if img_w > img_h:
                cell_w = min(img_w, max_cell_size)
                cell_h = int(cell_w / aspect_ratio)
            else:
                cell_h = min(img_h, max_cell_size)
                cell_w = int(cell_h * aspect_ratio)

            img_resized = img.resize((cell_w, cell_h))

            if add_border:
                img_resized = ImageOps.expand(img_resized, border=border_width // 2, fill=border_color)

            images_resized.append(img_resized)
            max_row_height = max(max_row_height, cell_h)

        max_row_height = int(max_row_height)
        total_images = len(images_resized)
        rows = math.ceil(total_images / cols)

        grid_width = cols * max_cell_size + (cols - 1) * border_width
        grid_height = rows * max_row_height + (rows - 1) * border_width

        new_image = Image.new('RGB', (grid_width, grid_height), border_color)

        for i, img in enumerate(images_resized):
            x = (i % cols) * (max_cell_size + border_width)
            y = (i // cols) * (max_row_height + border_width)

            img_w, img_h = img.size
            paste_x = x + (max_cell_size - img_w) // 2
            paste_y = y + (max_row_height - img_h) // 2

            new_image.paste(img, (paste_x, paste_y, paste_x + img_w, paste_y + img_h))

        if add_border:
            new_image = ImageOps.expand(new_image, border=border_width, fill=border_color)

        return (pil2tensor(new_image), )


# IMAGE GRID IMAGE FROM PATH

class WAS_Image_Grid_Image:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images_path": ("STRING", {"default":"./ComfyUI/input/", "multiline": False}),
                "pattern_glob": ("STRING", {"default":"*", "multiline": False}),
                "include_subfolders": (["false", "true"],),
                "border_width": ("INT", {"default":3, "min": 0, "max": 100, "step":1}),
                "number_of_columns": ("INT", {"default":6, "min": 1, "max": 24, "step":1}),
                "max_cell_size": ("INT", {"default":256, "min":32, "max":1280, "step":1}),
                "border_red": ("INT", {"default":0, "min": 0, "max": 255, "step":1}),
                "border_green": ("INT", {"default":0, "min": 0, "max": 255, "step":1}),
                "border_blue": ("INT", {"default":0, "min": 0, "max": 255, "step":1}),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "create_grid_image"

    CATEGORY = "WAS Suite/Image/Process"

    def create_grid_image(self, images_path, pattern_glob="*", include_subfolders="false", number_of_c
olumns=6,
                            max_cell_size=256, border_width=3, border_red=0, border_green=0, border_blue=0):

        if not os.path.exists(images_path):
            cstr(f"The grid image path `{images_path}` does not exist!").error.print()
            return (pil2tensor(Image.new("RGB", (512,512), (0,0,0))),)

        paths = glob.glob(os.path.join(images_path, pattern_glob), recursive=(False if include_subfolders
== "false" else True))
        image_paths = []
        for path in paths:
            if path.lower().endswith(ALLOWED_EXT) and os.path.exists(path):
                image_paths.append(path)

        grid_image = self.smart_grid_image(image_paths, int(number_of_columns), (int(max_cell_size), int
(max_cell_size)),
                                        (False if border_width <= 0 else True), (int(border_red),
                                        int(border_green), int(border_blue)), int(border_width))

        return (pil2tensor(grid_image),)
    def smart_grid_image(self, images, cols=6, size=(256,256), add_border=False, border_color=(0,0,0),
border_width=3):

        # calculate row height
        max_width, max_height = size
        row_height = 0
        images_resized = []
        for image in images:
            img = Image.open(image).convert('RGB')

            img_w, img_h = img.size
            aspect_ratio = img_w / img_h
            if aspect_ratio > 1: # landscape
                thumb_w = min(max_width, img_w-border_width)
                thumb_h = thumb_w / aspect_ratio
            else: # portrait
                thumb_h = min(max_height, img_h-border_width)
                thumb_w = thumb_h * aspect_ratio

            # pad the image to match the maximum size and center it within the cell
            pad_w = max_width - int(thumb_w)
            pad_h = max_height - int(thumb_h)
            left = pad_w // 2
            top = pad_h // 2
            right = pad_w - left
            bottom = pad_h - top
            padding = (left, top, right, bottom)  # left, top, right, bottom
            img_resized = ImageOps.expand(img.resize((int(thumb_w), int(thumb_h))), padding)

            if add_border:
                img_resized_bordered = ImageOps.expand(img_resized, border=border_width//2, fill=border_
color)

            images_resized.append(img_resized)
            row_height = max(row_height, img_resized.size[1])
        row_height = int(row_height)

        # calculate the number of rows
        total_images = len(images_resized)
        rows = math.ceil(total_images / cols)

        # create empty image to put thumbnails
        new_image = Image.new('RGB', (cols*size[0]+(cols-1)*border_width, rows*row_height+(rows-1)*b
order_width), border_color)

        for i, img in enumerate(images_resized):
            if add_border:
                border_img = ImageOps.expand(img, border=border_width//2, fill=border_color)
                x = (i % cols) * (size[0]+border_width)
                y = (i // cols) * (row_height+border_width)
                if border_img.size == (size[0], size[1]):
                    new_image.paste(border_img, (x, y, x+size[0], y+size[1]))
                else:
                    # Resize image to match size parameter
                    border_img = border_img.resize((size[0], size[1]))
                    new_image.paste(border_img, (x, y, x+size[0], y+size[1]))
            else:
                x = (i % cols) * (size[0]+border_width)
                y = (i // cols) * (row_height+border_width)
                if img.size == (size[0], size[1]):
                    new_image.paste(img, (x, y, x+img.size[0], y+img.size[1]))
                else:
                    # Resize image to match size parameter
                    img = img.resize((size[0], size[1]))
                    new_image.paste(img, (x, y, x+size[0], y+size[1]))

        new_image = ImageOps.expand(new_image, border=border_width, fill=border_color)

        return new_image

# IMAGE MORPH GIF

class WAS_Image_Morph_GIF:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_a": ("IMAGE",),
                "image_b": ("IMAGE",),
                "transition_frames": ("INT", {"default":30, "min":2, "max":60, "step":1}),
                "still_image_delay_ms": ("FLOAT", {"default":2500.0, "min":0.1, "max":60000.0, "step":0.1}),
                "duration_ms": ("FLOAT", {"default":0.1, "min":0.1, "max":60000.0, "step":0.1}),
                "loops": ("INT", {"default":0, "min":0, "max":100, "step":1}),
                "max_size": ("INT", {"default":512, "min":128, "max":1280, "step":1}),
                "output_path": ("STRING", {"default": "./ComfyUI/output", "multiline": False}),
                "filename": ("STRING", {"default": "morph", "multiline": False}),
                "filetype": (["GIF", "APNG"],),
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

    RETURN_TYPES = ("IMAGE","IMAGE",TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("image_a_pass","image_b_pass","filepath_text","filename_text")
    FUNCTION = "create_morph_gif"

    CATEGORY = "WAS Suite/Animation"

    def create_morph_gif(self, image_a, image_b, transition_frames=10, still_image_delay_ms=10, duratio
n_ms=0.1, loops=0, max_size=512,
                              output_path="./ComfyUI/output", filename="morph", filetype="GIF"):

        tokens = TextTokens()
        WTools = WAS_Tools_Class()

        if 'imageio' not in packages():
            install_package('imageio')

        if filetype not in ["APNG", "GIF"]:
            filetype = "GIF"
        if output_path.strip() in [None, "", "."]:
            output_path = "./ComfyUI/output"
        output_path = tokens.parseTokens(os.path.join(*output_path.split('/')))
        if not os.path.exists(output_path):
            os.makedirs(output_path, exist_ok=True)

        if image_a == None:
            image_a = pil2tensor(Image.new("RGB", (512,512), (0,0,0)))
        if image_b == None:
            image_b = pil2tensor(Image.new("RGB", (512,512), (255,255,255)))

        if transition_frames < 2:
            transition_frames = 2
        elif transition_frames > 60:
            transition_frames = 60

        if duration_ms < 0.1:
            duration_ms = 0.1
        elif duration_ms > 60000.0:
            duration_ms = 60000.0

        output_file = WTools.morph_images([tensor2pil(image_a), tensor2pil(image_b)], steps=int(transitio
n_frames), max_size=int(max_size), loop=int(loops),
                              still_duration=int(still_image_delay_ms), duration=int(duration_ms), output_path=outpu
t_path,
                              filename=tokens.parseTokens(filename), filetype=filetype)

        return (image_a, image_b, output_file)


# IMAGE MORPH GIF WRITER

class WAS_Image_Morph_GIF_Writer:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "transition_frames": ("INT", {"default":30, "min":2, "max":60, "step":1}),
                "image_delay_ms": ("FLOAT", {"default":2500.0, "min":0.1, "max":60000.0, "step":0.1}),
                "duration_ms": ("FLOAT", {"default":0.1, "min":0.1, "max":60000.0, "step":0.1}),
                "loops": ("INT", {"default":0, "min":0, "max":100, "step":1}),
                "max_size": ("INT", {"default":512, "min":128, "max":1280, "step":1}),
                "output_path": ("STRING", {"default": comfy_paths.output_directory, "multiline": False}),
                "filename": ("STRING", {"default": "morph_writer", "multiline": False}),
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

    RETURN_TYPES = ("IMAGE",TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("image_pass","filepath_text","filename_text")
    FUNCTION = "write_to_morph_gif"
    CATEGORY = "WAS Suite/Animation/Writer"

    def write_to_morph_gif(self, image, transition_frames=10, image_delay_ms=10, duration_ms=0.1, loop
s=0, max_size=512,
                        output_path="./ComfyUI/output", filename="morph"):

        if 'imageio' not in packages():
            install_package("imageio")

        if output_path.strip() in [None, "", "."]:
            output_path = "./ComfyUI/output"

        if image is None:
            image = pil2tensor(Image.new("RGB", (512, 512), (0, 0, 0))).unsqueeze(0)

        if transition_frames < 2:
            transition_frames = 2
        elif transition_frames > 60:
            transition_frames = 60

        if duration_ms < 0.1:
            duration_ms = 0.1
        elif duration_ms > 60000.0:
            duration_ms = 60000.0

        tokens = TextTokens()
        output_path = os.path.abspath(os.path.join(*tokens.parseTokens(output_path).split('/')))
        output_file = os.path.join(output_path, tokens.parseTokens(filename) + '.gif')

        if not os.path.exists(output_path):
            os.makedirs(output_path, exist_ok=True)

        WTools = WAS_Tools_Class()
        GifMorph = WTools.GifMorphWriter(int(transition_frames), int(duration_ms), int(image_delay_ms))

        for img in image:
            pil_img = tensor2pil(img)
            GifMorph.write(pil_img, output_file)

        return (image, output_file, filename)

# IMAGE MORPH GIF BY PATH

class WAS_Image_Morph_GIF_By_Path:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "transition_frames": ("INT", {"default":30, "min":2, "max":60, "step":1}),
                "still_image_delay_ms": ("FLOAT", {"default":2500.0, "min":0.1, "max":60000.0, "step":0.1}),
                "duration_ms": ("FLOAT", {"default":0.1, "min":0.1, "max":60000.0, "step":0.1}),
                "loops": ("INT", {"default":0, "min":0, "max":100, "step":1}),
                "max_size": ("INT", {"default":512, "min":128, "max":1280, "step":1}),
                "input_path": ("STRING",{"default":"./ComfyUI", "multiline": False}),
                "input_pattern": ("STRING",{"default":"*", "multiline": False}),
                "output_path": ("STRING", {"default": "./ComfyUI/output", "multiline": False}),
                "filename": ("STRING", {"default": "morph", "multiline": False}),
                "filetype": (["GIF", "APNG"],),
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("filepath_text","filename_text")
    FUNCTION = "create_morph_gif"

    CATEGORY = "WAS Suite/Animation"

    def create_morph_gif(self, transition_frames=30, still_image_delay_ms=2500, duration_ms=0.1, loops
=0, max_size=512,
                        input_path="./ComfyUI/output", input_pattern="*", output_path="./ComfyUI/output", fil
ename="morph", filetype="GIF"):

        if 'imageio' not in packages():
            install_package("imageio")

        if not os.path.exists(input_path):
            cstr(f"The input_path `{input_path}` does not exist!").error.print()
            return ("",)

        images = self.load_images(input_path, input_pattern)
        if not images:
            cstr(f"The input_path `{input_path}` does not contain any valid images!").msg.print()
            return ("",)

        if filetype not in ["APNG", "GIF"]:
            filetype = "GIF"
        if output_path.strip() in [None, "", "."]:
            output_path = "./ComfyUI/output"

        if transition_frames < 2:
            transition_frames = 2
        elif transition_frames > 60:
            transition_frames = 60

        if duration_ms < 0.1:
            duration_ms = 0.1
        elif duration_ms > 60000.0:
            duration_ms = 60000.0

        tokens = TextTokens()
        WTools = WAS_Tools_Class()

        output_file = WTools.morph_images(images, steps=int(transition_frames), max_size=int(max_siz
e), loop=int(loops), still_duration=int(still_image_delay_ms),
                                    duration=int(duration_ms), output_path=tokens.parseTokens(os.path.join(*o
utput_path.split('/'))),
                                    filename=tokens.parseTokens(filename), filetype=filetype)
        return (output_file,filename)


    def load_images(self, directory_path, pattern):
        images = []
        for file_name in glob.glob(os.path.join(directory_path, pattern), recursive=False):
            if file_name.lower().endswith(ALLOWED_EXT):
                images.append(Image.open(file_name).convert("RGB"))
        return images


# COMBINE NODE

class WAS_Image_Blending_Mode:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_a": ("IMAGE",),
                "image_b": ("IMAGE",),
                "mode": ([
                    "add",
                    "color",
                    "color_burn",
                    "color_dodge",
                    "darken",
                    "difference",
                    "exclusion",
                    "hard_light",
                    "hue",
                    "lighten",
                    "multiply",
                    "overlay",
                    "screen",
                    "soft_light"
                ],),
                "blend_percentage": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 1.0, "step": 0.01}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "image_blending_mode"

    CATEGORY = "WAS Suite/Image"

    def image_blending_mode(self, image_a, image_b, mode='add', blend_percentage=1.0):

        # Install Pilgram
        if 'pilgram' not in packages():
            install_package("pilgram")

        # Import Pilgram module
        import pilgram
        # Convert images to PIL
        img_a = tensor2pil(image_a)
        img_b = tensor2pil(image_b)

        # Apply blending
        if mode:
            if mode == "color":
                out_image = pilgram.css.blending.color(img_a, img_b)
            elif mode == "color_burn":
                out_image = pilgram.css.blending.color_burn(img_a, img_b)
            elif mode == "color_dodge":
                out_image = pilgram.css.blending.color_dodge(img_a, img_b)
            elif mode == "darken":
                out_image = pilgram.css.blending.darken(img_a, img_b)
            elif mode == "difference":
                out_image = pilgram.css.blending.difference(img_a, img_b)
            elif mode == "exclusion":
                out_image = pilgram.css.blending.exclusion(img_a, img_b)
            elif mode == "hard_light":
                out_image = pilgram.css.blending.hard_light(img_a, img_b)
            elif mode == "hue":
                out_image = pilgram.css.blending.hue(img_a, img_b)
            elif mode == "lighten":
                out_image = pilgram.css.blending.lighten(img_a, img_b)
            elif mode == "multiply":
                out_image = pilgram.css.blending.multiply(img_a, img_b)
            elif mode == "add":
                out_image = pilgram.css.blending.normal(img_a, img_b)
            elif mode == "overlay":
                out_image = pilgram.css.blending.overlay(img_a, img_b)
            elif mode == "screen":
                out_image = pilgram.css.blending.screen(img_a, img_b)
            elif mode == "soft_light":
                out_image = pilgram.css.blending.soft_light(img_a, img_b)
            else:
                out_image = img_a

        out_image = out_image.convert("RGB")

        # Blend image
        blend_mask = Image.new(mode="L", size=img_a.size,
                               color=(round(blend_percentage * 255)))
        blend_mask = ImageOps.invert(blend_mask)
        out_image = Image.composite(img_a, out_image, blend_mask)

        return (pil2tensor(out_image), )


# IMAGE BLEND NODE

class WAS_Image_Blend:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_a": ("IMAGE",),
                "image_b": ("IMAGE",),
                "blend_percentage": ("FLOAT", {"default": 0.5, "min": 0.0, "max": 1.0, "step": 0.01}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "image_blend"

    CATEGORY = "WAS Suite/Image"

    def image_blend(self, image_a, image_b, blend_percentage):

        # Convert images to PIL
        img_a = tensor2pil(image_a)
        img_b = tensor2pil(image_b)

        # Blend image
        blend_mask = Image.new(mode="L", size=img_a.size,
                               color=(round(blend_percentage * 255)))
        blend_mask = ImageOps.invert(blend_mask)
        img_result = Image.composite(img_a, img_b, blend_mask)

        del img_a, img_b, blend_mask

        return (pil2tensor(img_result), )



# IMAGE MONITOR DISTORTION FILTER

class WAS_Image_Monitor_Distortion_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "mode": (["Digital Distortion", "Signal Distortion", "TV Distortion"],),
                "amplitude": ("INT", {"default": 5, "min": 1, "max": 255, "step": 1}),
                "offset": ("INT", {"default": 10, "min": 1, "max": 255, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "image_monitor_filters"

    CATEGORY = "WAS Suite/Image/Filter"

    def image_monitor_filters(self, image, mode="Digital Distortion", amplitude=5, offset=5):

        # Convert images to PIL
        image = tensor2pil(image)
        # WAS Filters
        WTools = WAS_Tools_Class()

        # Apply image effect
        if mode:
            if mode == 'Digital Distortion':
                image = WTools.digital_distortion(image, amplitude, offset)
            elif mode == 'Signal Distortion':
                image = WTools.signal_distortion(image, amplitude)
            elif mode == 'TV Distortion':
                image = WTools.tv_vhs_distortion(image, amplitude)
            else:
                image = image

        return (pil2tensor(image), )



# IMAGE PERLIN NOISE

class WAS_Image_Perlin_Noise:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "width": ("INT", {"default": 512, "max": 2048, "min": 64, "step": 1}),
                "height": ("INT", {"default": 512, "max": 2048, "min": 64, "step": 1}),
                "scale": ("INT", {"default": 100, "max": 2048, "min": 2, "step": 1}),
                "octaves": ("INT", {"default": 4, "max": 8, "min": 0, "step": 1}),
                "persistence": ("FLOAT", {"default": 0.5, "max": 100.0, "min": 0.01, "step": 0.01}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "perlin_noise"

    CATEGORY = "WAS Suite/Image/Generate/Noise"

    def perlin_noise(self, width, height, scale, octaves, persistence, seed):

        WTools = WAS_Tools_Class()

        image = WTools.perlin_noise(width, height, octaves, persistence, scale, seed)

        return (pil2tensor(image), )


# IMAGE PERLIN POWER FRACTAL

class WAS_Image_Perlin_Power_Fractal:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "width": ("INT", {"default": 512, "max": 8192, "min": 64, "step": 1}),
                "height": ("INT", {"default": 512, "max": 8192, "min": 64, "step": 1}),
                "scale": ("INT", {"default": 100, "max": 2048, "min": 2, "step": 1}),
                "octaves": ("INT", {"default": 4, "max": 8, "min": 0, "step": 1}),
                "persistence": ("FLOAT", {"default": 0.5, "max": 100.0, "min": 0.01, "step": 0.01}),
                "lacunarity": ("FLOAT", {"default": 2.0, "max": 100.0, "min": 0.01, "step": 0.01}),
                "exponent": ("FLOAT", {"default": 2.0, "max": 100.0, "min": 0.01, "step": 0.01}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "perlin_power_fractal"

    CATEGORY = "WAS Suite/Image/Generate/Noise"

    def perlin_power_fractal(self, width, height, scale, octaves, persistence, lacunarity, exponent, seed):

        WTools = WAS_Tools_Class()

        image = WTools.perlin_power_fractal(width, height, octaves, persistence, lacunarity, exponent, sc
ale, seed)

        return (pil2tensor(image), )


# IMAGE VORONOI NOISE FILTER

class WAS_Image_Voronoi_Noise_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "width": ("INT", {"default": 512, "max": 4096, "min": 64, "step": 1}),
                "height": ("INT", {"default": 512, "max": 4096, "min": 64, "step": 1}),
                "density": ("INT", {"default": 50, "max": 256, "min": 10, "step": 2}),
                "modulator": ("INT", {"default": 0, "max": 8, "min": 0, "step": 1}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            },
            "optional": {
                "flat": (["False", "True"],),
                "RGB_output": (["True", "False"],),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "voronoi_noise_filter"

    CATEGORY = "WAS Suite/Image/Generate/Noise"
    def voronoi_noise_filter(self, width, height, density, modulator, seed, flat="False", RGB_output="Tru
e"):

        WTools = WAS_Tools_Class()

        image = WTools.worley_noise(height=height, width=width, density=density, option=modulator, us
e_broadcast_ops=True, seed=seed, flat=(flat == "True")).image

        if RGB_output == "True":
            image = image.convert("RGB")
        else:
            image = image.convert("L")

        return (pil2tensor(image), )

# IMAGE POWER NOISE

class WAS_Image_Power_Noise:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "width": ("INT", {"default": 512, "max": 4096, "min": 64, "step": 1}),
                "height": ("INT", {"default": 512, "max": 4096, "min": 64, "step": 1}),
                "frequency": ("FLOAT", {"default": 0.5, "max": 10.0, "min": 0.0, "step": 0.01}),
                "attenuation": ("FLOAT", {"default": 0.5, "max": 10.0, "min": 0.0, "step": 0.01}),
                "noise_type": (["grey", "white", "pink", "blue", "green", "mix"],),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "power_noise"

    CATEGORY = "WAS Suite/Image/Generate/Noise"

    def power_noise(self, width, height, frequency, attenuation, noise_type, seed):

        noise_image = self.generate_power_noise(width, height, frequency, attenuation, noise_type, seed)

        return (pil2tensor(noise_image), )

    def generate_power_noise(self, width, height, frequency=None, attenuation=None, noise_type="whit
e", seed=None):
        def white_noise(width, height):
            noise = np.random.random((height, width))
            return noise

        def grey_noise(width, height, attenuation):
            noise = np.random.normal(0, attenuation, (height, width))
            return noise

        def blue_noise(width, height, frequency, attenuation):
```

```
        noise = grey_noise(width, height, attenuation)
        scale = 1.0 / (width * height)
        fy = np.fft.fftfreq(height)[:, np.newaxis] ** 2
        fx = np.fft.fftfreq(width) ** 2
        f = fy + fx
        power = np.sqrt(f)
        power[0, 0] = 1
        noise = np.fft.ifft2(np.fft.fft2(noise) / power)
        noise *= scale / noise.std()
        return np.real(noise)

    def green_noise(width, height, frequency, attenuation):
        noise = grey_noise(width, height, attenuation)
        scale = 1.0 / (width * height)
        fy = np.fft.fftfreq(height)[:, np.newaxis] ** 2
        fx = np.fft.fftfreq(width) ** 2
        f = fy + fx
        power = np.sqrt(f)
        power[0, 0] = 1
        noise = np.fft.ifft2(np.fft.fft2(noise) / np.sqrt(power))
        noise *= scale / noise.std()
        return np.real(noise)

    def pink_noise(width, height, frequency, attenuation):
        noise = grey_noise(width, height, attenuation)
        scale = 1.0 / (width * height)
        fy = np.fft.fftfreq(height)[:, np.newaxis] ** 2
        fx = np.fft.fftfreq(width) ** 2
        f = fy + fx
        power = np.sqrt(f)
        power[0, 0] = 1
        noise = np.fft.ifft2(np.fft.fft2(noise) * power)
        noise *= scale / noise.std()
        return np.real(noise)

    def blue_noise_mask(width, height, frequency, attenuation, seed, num_masks=3):
        masks = []
        for i in range(num_masks):
            mask_seed = seed + i
            np.random.seed(mask_seed)
            mask = blue_noise(width, height, frequency, attenuation)
            masks.append(mask)
        return masks

    def blend_noise(width, height, masks, noise_types, attenuations):
        blended_image = Image.new("L", (width, height), color=0)
        fy = np.fft.fftfreq(height)[:, np.newaxis] ** 2
        fx = np.fft.fftfreq(width) ** 2
        f = fy + fx
        i = 0
        for mask, noise_type, attenuation in zip(masks, noise_types, attenuations):
            mask = Image.fromarray((255 * (mask - np.min(mask)) / (np.max(mask) - np.min(mask))).ast
ype(np.uint8).real)
            if noise_type == "white":
                noise = white_noise(width, height)
                noise = Image.fromarray((255 * (noise - np.min(noise)) / (np.max(noise) - np.min(noise))).a
stype(np.uint8).real)
            elif noise_type == "grey":
```

```
          noise = grey_noise(width, height, attenuation)
          noise = Image.fromarray((255 * (noise - np.min(noise)) / (np.max(noise) - np.min(noise))).a
stype(np.uint8).real)
        elif noise_type == "pink":
          noise = pink_noise(width, height, frequency, attenuation)
          noise = Image.fromarray((255 * (noise - np.min(noise)) / (np.max(noise) - np.min(noise))).a
stype(np.uint8).real)
        elif noise_type == "green":
          noise = green_noise(width, height, frequency, attenuation)
          noise = Image.fromarray((255 * (noise - np.min(noise)) / (np.max(noise) - np.min(noise))).a
stype(np.uint8).real)
        elif noise_type == "blue":
          noise = blue_noise(width, height, frequency, attenuation)
          noise = Image.fromarray((255 * (noise - np.min(noise)) / (np.max(noise) - np.min(noise))).a
stype(np.uint8).real)

        blended_image = Image.composite(blended_image, noise, mask)
        i += 1

      return np.asarray(blended_image)

    def shorten_to_range(value, min_value, max_value):
      range_length = max_value - min_value + 1
      return ((value - min_value) % range_length) + min_value

    if seed is not None:
      if seed > 4294967294:
        seed = shorten_to_range(seed, 0, 4294967293)
        cstr(f"Seed too large for power noise; rescaled to: {seed}").warning.print()

      np.random.seed(seed)

    if noise_type == "white":
      noise = white_noise(width, height)
    elif noise_type == "grey":
      noise = grey_noise(width, height, attenuation)
    elif noise_type == "pink":
      if frequency is None:
        cstr("Pink noise requires a frequency value.").error.print()
        return None
      noise = pink_noise(width, height, frequency, attenuation)
    elif noise_type == "green":
      if frequency is None:
        cstr("Green noise requires a frequency value.").error.print()
        return None
      noise = green_noise(width, height, frequency, attenuation)
    elif noise_type == "blue":
      if frequency is None:
        cstr("Blue noise requires a frequency value.").error.print()
        return None
      noise = blue_noise(width, height, frequency, attenuation)
    elif noise_type == "mix":
      if frequency is None:
        cstr("Mix noise requires a frequency value.").error.print()
        return None
      if seed is None:
        cstr("Mix noise requires a seed value.").error.print()
        return None
```

```
            blue_noise_masks = blue_noise_mask(width, height, frequency, attenuation, seed=seed, num_m
asks=3)
            noise_types = ["white", "grey", "pink", "green", "blue"]
            attenuations = [attenuation] * len(noise_types)
            noise = blend_noise(width, height, blue_noise_masks, noise_types, attenuations)
        else:
            cstr(f"Unsupported noise type `{noise_type}`").error.print()
            return None
        if noise_type != 'mix':
            noise = 255 * (noise - np.min(noise)) / (np.max(noise) - np.min(noise))
        noise_image = Image.fromarray(noise.astype(np.uint8).real)

        return noise_image.convert("RGB")

# IMAGE TO NOISE

class WAS_Image_To_Noise:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "num_colors": ("INT", {"default": 16, "max": 256, "min": 2, "step": 2}),
                "black_mix": ("INT", {"default": 0, "max": 20, "min": 0, "step": 1}),
                "gaussian_mix": ("FLOAT", {"default": 0.0, "max": 1024, "min": 0, "step": 0.1}),
                "brightness": ("FLOAT", {"default": 1.0, "max": 2.0, "min": 0.0, "step": 0.01}),
                "output_mode": (["batch","list"],),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    OUTPUT_IS_LIST = (False,)
    FUNCTION = "image_to_noise"

    CATEGORY = "WAS Suite/Image/Generate/Noise"

    def image_to_noise(self, images, num_colors, black_mix, gaussian_mix, brightness, output_mode, se
ed):

        noise_images = []
        for image in images:
            noise_images.append(pil2tensor(self.image2noise(tensor2pil(image), num_colors, black_mix, br
ightness, gaussian_mix, seed)))
        if output_mode == "list":
            self.OUTPUT_IS_LIST = (True,)
        else:
            noise_images = torch.cat(noise_images, dim=0)
        return (noise_images, )

    def image2noise(self, image, num_colors=16, black_mix=0, brightness=1.0, gaussian_mix=0, seed=
0):
        random.seed(int(seed))
        image = image.quantize(colors=num_colors)
        image = image.convert("RGBA")
        pixel_data = list(image.getdata())
        random.shuffle(pixel_data)
        randomized_image = Image.new("RGBA", image.size)
        randomized_image.putdata(pixel_data)

        width, height = image.size
        black_noise = Image.new("RGBA", (width, height), (0, 0, 0, 0))

        for _ in range(black_mix):
            for x in range(width):
                for y in range(height):
                    if random.randint(0,1) == 1:
                        black_noise.putpixel((x, y), (0, 0, 0, 255))

        randomized_image = Image.alpha_composite(randomized_image, black_noise)
        enhancer = ImageEnhance.Brightness(randomized_image)
        randomized_image = enhancer.enhance(brightness)

        if gaussian_mix > 0:
            original_noise = randomized_image.copy()
            randomized_gaussian = randomized_image.filter(ImageFilter.GaussianBlur(radius=gaussian_mi
x))
            randomized_image = Image.blend(randomized_image, randomized_gaussian, 0.65)
            randomized_image = Image.blend(randomized_image, original_noise, 0.25)

        return randomized_image

# IMAGE MAKE SEAMLESS

class WAS_Image_Make_Seamless:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "blending": ("FLOAT", {"default": 0.4, "max": 1.0, "min": 0.0, "step": 0.01}),
                "tiled": (["true", "false"],),
                "tiles": ("INT", {"default": 2, "max": 6, "min": 2, "step": 2}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "make_seamless"

    CATEGORY = "WAS Suite/Image/Process"

    def make_seamless(self, images, blending, tiled, tiles):

        WTools = WAS_Tools_Class()

        seamless_images = []
        for image in images:
            seamless_images.append(pil2tensor(WTools.make_seamless(tensor2pil(image), blending, tiled,
tiles)))

        seamless_images = torch.cat(seamless_images, dim=0)

        return (seamless_images, )


# IMAGE DISPLACEMENT WARP

class WAS_Image_Displacement_Warp:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "displacement_maps": ("IMAGE",),
                "amplitude": ("FLOAT", {"default": 25.0, "min": -4096, "max": 4096, "step": 0.1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "displace_image"

    CATEGORY = "WAS Suite/Image/Transform"

    def displace_image(self, images, displacement_maps, amplitude):

        WTools = WAS_Tools_Class()

        displaced_images = []
        for i in range(len(images)):
            img = tensor2pil(images[i])
            if i < len(displacement_maps):
                disp = tensor2pil(displacement_maps[i])
            else:
                disp = tensor2pil(displacement_maps[-1])
            disp = self.resize_and_crop(disp, img.size)
            displaced_images.append(pil2tensor(WTools.displace_image(img, disp, amplitude)))

        displaced_images = torch.cat(displaced_images, dim=0)

        return (displaced_images, )


    def resize_and_crop(self, image, target_size):
        width, height = image.size
        target_width, target_height = target_size
        aspect_ratio = width / height
        target_aspect_ratio = target_width / target_height

        if aspect_ratio > target_aspect_ratio:
            new_height = target_height
            new_width = int(new_height * aspect_ratio)
        else:
            new_width = target_width
            new_height = int(new_width / aspect_ratio)

        image = image.resize((new_width, new_height))
        left = (new_width - target_width) // 2
        top = (new_height - target_height) // 2
        right = left + target_width
        bottom = top + target_height
        image = image.crop((left, top, right, bottom))

        return image

# IMAGE TO BATCH

class WAS_Image_Batch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
            },
            "optional": {
                "images_a": ("IMAGE",),
                "images_b": ("IMAGE",),
                "images_c": ("IMAGE",),
                "images_d": ("IMAGE",),
                # "images_e": ("IMAGE",),
                # "images_f": ("IMAGE",),
                # Theoretically, an infinite number of image input parameters can be added.
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("image",)
    FUNCTION = "image_batch"
    CATEGORY = "WAS Suite/Image"

    def _check_image_dimensions(self, tensors, names):
        reference_dimensions = tensors[0].shape[1:]  # Ignore batch dimension
        mismatched_images = [names[i] for i, tensor in enumerate(tensors) if tensor.shape[1:] != reference
_dimensions]

        if mismatched_images:
            raise ValueError(f"WAS Image Batch Warning: Input image dimensions do not match for images:
{mismatched_images}")

    def image_batch(self, **kwargs):
        batched_tensors = [kwargs[key] for key in kwargs if kwargs[key] is not None]
        image_names = [key for key in kwargs if kwargs[key] is not None]

        if not batched_tensors:
            raise ValueError("At least one input image must be provided.")

        self._check_image_dimensions(batched_tensors, image_names)
        batched_tensors = torch.cat(batched_tensors, dim=0)
        return (batched_tensors,)


# Latent TO BATCH

class WAS_Latent_Batch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
            },
            "optional": {
                "latent_a": ("LATENT",),
                "latent_b": ("LATENT",),
                "latent_c": ("LATENT",),
                "latent_d": ("LATENT",),
            },
        }

    RETURN_TYPES = ("LATENT",)
    RETURN_NAMES = ("latent",)
    FUNCTION = "latent_batch"
    CATEGORY = "WAS Suite/Latent"

    def _check_latent_dimensions(self, tensors, names):
        dimensions = [(tensor["samples"].shape) for tensor in tensors]
        if len(set(dimensions)) > 1:
            mismatched_indices = [i for i, dim in enumerate(dimensions) if dim[1] != dimensions[0][1]]
            mismatched_latents = [names[i] for i in mismatched_indices]
            if mismatched_latents:
                raise ValueError(f"WAS latent Batch Warning: Input latent dimensions do not match for latent
s: {mismatched_latents}")

    def latent_batch(self, **kwargs):
        batched_tensors = [kwargs[key] for key in kwargs if kwargs[key] is not None]
        latent_names = [key for key in kwargs if kwargs[key] is not None]

        if not batched_tensors:
            raise ValueError("At least one input latent must be provided.")

        self._check_latent_dimensions(batched_tensors, latent_names)
        samples_out = {}
        samples_out["samples"]  = torch.cat([tensor["samples"] for tensor in batched_tensors], dim=0)
        samples_out["batch_index"] = []
        for tensor in batched_tensors:
            cindex = tensor.get("batch_index", list(range(tensor["samples"].shape[0])))
            samples_out["batch_index"] += cindex
        return (samples_out,)


# MASK TO BATCH

class WAS_Mask_Batch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "optional": {
                "masks_a": ("MASK",),
                "masks_b": ("MASK",),
                "masks_c": ("MASK",),
                "masks_d": ("MASK",),
                # "masks_e": ("MASK",),
                # "masks_f": ("MASK",),
                # Theoretically, an infinite number of mask input parameters can be added.
            },
        }

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("masks",)
    FUNCTION = "mask_batch"
    CATEGORY = "WAS Suite/Image/Masking"

    def _check_mask_dimensions(self, tensors, names):
        dimensions = [tensor.shape[1:] for tensor in tensors]  # Exclude the batch dimension (if present)
        if len(set(dimensions)) > 1:
            mismatched_indices = [i for i, dim in enumerate(dimensions) if dim != dimensions[0]]
            mismatched_masks = [names[i] for i in mismatched_indices]
            raise ValueError(f"WAS Mask Batch Warning: Input mask dimensions do not match for masks:
{mismatched_masks}")

    def mask_batch(self, **kwargs):
        batched_tensors = [kwargs[key] for key in kwargs if kwargs[key] is not None]
        mask_names = [key for key in kwargs if kwargs[key] is not None]

        if not batched_tensors:
            raise ValueError("At least one input mask must be provided.")

        self._check_mask_dimensions(batched_tensors, mask_names)
        batched_tensors = torch.stack(batched_tensors, dim=0)
        batched_tensors = batched_tensors.unsqueeze(1)  # Add a channel dimension
        return (batched_tensors,)

# IMAGE GENERATE COLOR PALETTE

class WAS_Image_Color_Palette:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "colors": ("INT", {"default": 16, "min": 8, "max": 256, "step": 1}),
                "mode": (["Chart", "back_to_back"],),
            },
        }

    RETURN_TYPES = ("IMAGE","LIST")
    RETURN_NAMES = ("image","color_palettes")
    FUNCTION = "image_generate_palette"

    CATEGORY = "WAS Suite/Image/Analyze"

    def image_generate_palette(self, image, colors=16, mode="chart"):

        # WAS Filters
        WTools = WAS_Tools_Class()

        res_dir = os.path.join(WAS_SUITE_ROOT, 'res')
        font = os.path.join(res_dir, 'font.ttf')

        if not os.path.exists(font):
            font = None
        else:
            if mode == "Chart":
                cstr(f'Found font at `{font}`').msg.print()

        if len(image) > 1:
            palette_strings = []
            palette_images = []
            for img in image:
                img = tensor2pil(img)
                palette_image, palette = WTools.generate_palette(img, colors, 128, 10, font, 15, mode.lower())
                palette_images.append(pil2tensor(palette_image))
                palette_strings.append(palette)
            palette_images = torch.cat(palette_images, dim=0)
            return (palette_images, palette_strings)
        else:
            image = tensor2pil(image)
            palette_image, palette = WTools.generate_palette(image, colors, 128, 10, font, 15, mode.lower())
            return (pil2tensor(palette_image), [palette,])


# HEX TO HSL

class WAS_Hex_to_HSL:
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "hex_color": ("STRING", {"default": "#FF0000"}),
            },
            "optional": {
                "include_alpha": ("BOOLEAN", {"default": False})
            }
        }

    RETURN_TYPES = ("INT", "INT", "INT", "FLOAT", "STRING")
    RETURN_NAMES = ("hue", "saturation", "lightness", "alpha", "hsl")

    FUNCTION = "hex_to_hsl"
    CATEGORY = "WAS Suite/Utilities"

    @staticmethod
    def hex_to_hsl(hex_color, include_alpha=False):
        if hex_color.startswith("#"):
            hex_color = hex_color[1:]

        red = int(hex_color[0:2], 16) / 255.0
        green = int(hex_color[2:4], 16) / 255.0
        blue = int(hex_color[4:6], 16) / 255.0
        alpha = int(hex_color[6:8], 16) / 255.0 if include_alpha and len(hex_color) == 8 else 1.0
        max_val = max(red, green, blue)
        min_val = min(red, green, blue)
        delta = max_val - min_val
        luminance = (max_val + min_val) / 2.0

        if delta == 0:
            hue = 0
            saturation = 0
        else:
            saturation = delta / (1 - abs(2 * luminance - 1))
            if max_val == red:
                hue = ((green - blue) / delta) % 6
            elif max_val == green:
                hue = (blue - red) / delta + 2
            elif max_val == blue:
                hue = (red - green) / delta + 4
            hue *= 60
            if hue < 0:
                hue += 360

        luminance = luminance * 100
        saturation = saturation * 100

        hsl_string = f'hsl({round(hue)}, {round(saturation)}%, {round(luminance)}%)' if not include_alpha e
lse f'hsla({round(hue)}, {round(saturation)}%, {round(luminance)}%, {round(alpha, 2)})'
        output = (round(hue), round(saturation), round(luminance), round(alpha, 2), hsl_string)

        return output


# HSL TO HEX


class WAS_HSL_to_Hex:
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "hsl_color": ("STRING", {"default": "hsl(0, 100%, 50%)"}),
            }
        }

    RETURN_TYPES = ("STRING",)
    RETURN_NAMES = ("hex_color",)

    FUNCTION = "hsl_to_hex"
    CATEGORY = "WAS Suite/Utilities"

    @staticmethod
    def hsl_to_hex(hsl_color):
        import re
        hsl_pattern = re.compile(r'hsla?\(\s*(\d+),\s*(\d+)%?,\s*(\d+)%?(?:,\s*([\d.]+))?\s*\)')
        match = hsl_pattern.match(hsl_color)

        if not match:
            raise ValueError("Invalid HSL(A) color format")

        h, s, l = map(int, match.groups()[:3])
        a = float(match.groups()[3]) if match.groups()[3] else 1.0

        s /= 100
        l /= 100

        c = (1 - abs(2 * l - 1)) * s
        x = c * (1 - abs((h / 60) % 2 - 1))
        m = l - c/2

        if 0 <= h < 60:
            r, g, b = c, x, 0
        elif 60 <= h < 120:
            r, g, b = x, c, 0
        elif 120 <= h < 180:
            r, g, b = 0, c, x
        elif 180 <= h < 240:
            r, g, b = 0, x, c
        elif 240 <= h < 300:
            r, g, b = x, 0, c
        elif 300 <= h < 360:
            r, g, b = c, 0, x
        else:
            r, g, b = 0, 0, 0

        r = int((r + m) * 255)
        g = int((g + m) * 255)
        b = int((b + m) * 255)
        alpha = int(a * 255)

        hex_color = f'#{r:02X}{g:02X}{b:02X}'
        if a < 1:
            hex_color += f'{alpha:02X}'

        return (hex_color,)


# IMAGE ANALYZE


class WAS_Image_Analyze:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "mode": (["Black White Levels", "RGB Levels"],),
            },
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_analyze"

    CATEGORY = "WAS Suite/Image/Analyze"

    def image_analyze(self, image, mode='Black White Levels'):

        # Convert images to PIL
        image = tensor2pil(image)

        # WAS Filters
        WTools = WAS_Tools_Class()

        # Analye Image
        if mode:
            if mode == 'Black White Levels':
                image = WTools.black_white_levels(image)
            elif mode == 'RGB Levels':
                image = WTools.channel_frequency(image)
            else:
                image = image

        return (pil2tensor(image), )


# IMAGE GENERATE GRADIENT

class WAS_Image_Generate_Gradient:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        gradient_stops = '''0:255,0,0
25:255,255,255
50:0,255,0
75:0,0,255'''
        return {
            "required": {
                "width": ("INT", {"default":512, "max": 4096, "min": 64, "step":1}),
                "height": ("INT", {"default":512, "max": 4096, "min": 64, "step":1}),
                "direction": (["horizontal", "vertical"],),
                "tolerance": ("INT", {"default":0, "max": 255, "min": 0, "step":1}),
                "gradient_stops": ("STRING", {"default": gradient_stops, "multiline": True}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_gradient"

    CATEGORY = "WAS Suite/Image/Generate"

    def image_gradient(self, gradient_stops, width=512, height=512, direction='horizontal', tolerance=0):

        import io

        # WAS Filters
        WTools = WAS_Tools_Class()

        colors_dict = {}
        stops = io.StringIO(gradient_stops.strip().replace(' ',''))
        for stop in stops:
            parts = stop.split(':')
            colors = parts[1].replace('\n','').split(',')
            colors_dict[parts[0].replace('\n','')] = colors

        image = WTools.gradient((width, height), direction, colors_dict, tolerance)

        return (pil2tensor(image), )

# IMAGE GRADIENT MAP

class WAS_Image_Gradient_Map:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "gradient_image": ("IMAGE",),
                "flip_left_right": (["false", "true"],),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_gradient_map"

    CATEGORY = "WAS Suite/Image/Filter"

    def image_gradient_map(self, image, gradient_image, flip_left_right='false'):

        # Convert images to PIL
        image = tensor2pil(image)
        gradient_image = tensor2pil(gradient_image)

        # WAS Filters
        WTools = WAS_Tools_Class()

        image = WTools.gradient_map(image, gradient_image, (True if flip_left_right == 'true' else False))

        return (pil2tensor(image), )


# IMAGE TRANSPOSE

class WAS_Image_Transpose:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "image_overlay": ("IMAGE",),
                "width": ("INT", {"default": 512, "min": -48000, "max": 48000, "step": 1}),
                "height": ("INT", {"default": 512, "min": -48000, "max": 48000, "step": 1}),
                "X": ("INT", {"default": 0, "min": -48000, "max": 48000, "step": 1}),
                "Y": ("INT", {"default": 0, "min": -48000, "max": 48000, "step": 1}),
                "rotation": ("INT", {"default": 0, "min": -360, "max": 360, "step": 1}),
                "feathering": ("INT", {"default": 0, "min": 0, "max": 4096, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_transpose"

    CATEGORY = "WAS Suite/Image/Transform"

    def image_transpose(self, image: torch.Tensor, image_overlay: torch.Tensor, width: int, height: int, X: 
int, Y: int, rotation: int, feathering: int = 0):
        return (pil2tensor(self.apply_transpose_image(tensor2pil(image), tensor2pil(image_overlay), (widt
h, height), (X, Y), rotation, feathering)), )

    def apply_transpose_image(self, image_bg, image_element, size, loc, rotate=0, feathering=0):

        # Apply transformations to the element image
        image_element = image_element.rotate(rotate, expand=True)
        image_element = image_element.resize(size)

        # Create a mask for the image with the faded border
        if feathering > 0:
            mask = Image.new('L', image_element.size, 255)  # Initialize with 255 instead of 0
            draw = ImageDraw.Draw(mask)
            for i in range(feathering):
                alpha_value = int(255 * (i + 1) / feathering)  # Invert the calculation for alpha value
                draw.rectangle((i, i, image_element.size[0] - i, image_element.size[1] - i), fill=alpha_value)
            alpha_mask = Image.merge('RGBA', (mask, mask, mask, mask))
            image_element = Image.composite(image_element, Image.new('RGBA', image_element.size, (0, 
0, 0, 0)), alpha_mask)

        # Create a new image of the same size as the base image with an alpha channel
        new_image = Image.new('RGBA', image_bg.size, (0, 0, 0, 0))
        new_image.paste(image_element, loc)

        # Paste the new image onto the base image
        image_bg = image_bg.convert('RGBA')
        image_bg.paste(new_image, (0, 0), new_image)

        return image_bg



# IMAGE RESCALE

class WAS_Image_Rescale:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "mode": (["rescale", "resize"],),
                "supersample": (["true", "false"],),
                "resampling": (["lanczos", "nearest", "bilinear", "bicubic"],),
                "rescale_factor": ("FLOAT", {"default": 2, "min": 0.01, "max": 16.0, "step": 0.01}),
                "resize_width": ("INT", {"default": 1024, "min": 1, "max": 48000, "step": 1}),
                "resize_height": ("INT", {"default": 1536, "min": 1, "max": 48000, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_rescale"

    CATEGORY = "WAS Suite/Image/Transform"

    def image_rescale(self, image, mode="rescale", supersample='true', resampling="lanczos", rescale_
factor=2, resize_width=1024, resize_height=1024):
        scaled_images = []
        for img in image:
            scaled_images.append(pil2tensor(self.apply_resize_image(tensor2pil(img), mode, supersample,
rescale_factor, resize_width, resize_height, resampling)))
        scaled_images = torch.cat(scaled_images, dim=0)

        return (scaled_images, )

    def apply_resize_image(self, image: Image.Image, mode='scale', supersample='true', factor: int = 2,
width: int = 1024, height: int = 1024, resample='bicubic'):

        # Get the current width and height of the image
        current_width, current_height = image.size

        # Calculate the new width and height based on the given mode and parameters
        if mode == 'rescale':
            new_width, new_height = int(
                current_width * factor), int(current_height * factor)
        else:
            new_width = width if width % 8 == 0 else width + (8 - width % 8)
            new_height = height if height % 8 == 0 else height + \
                (8 - height % 8)

        # Define a dictionary of resampling filters
        resample_filters = {
            'nearest': 0,
            'bilinear': 2,
            'bicubic': 3,
            'lanczos': 1
        }

        # Apply supersample
        if supersample == 'true':
            image = image.resize((new_width * 8, new_height * 8), resample=Image.Resampling(resample_f
ilters[resample]))

        # Resize the image using the given resampling filter
        resized_image = image.resize((new_width, new_height), resample=Image.Resampling(resample_fil
ters[resample]))
```

```
        return resized_image


# LOAD IMAGE BATCH

class WAS_Load_Image_Batch:
    def __init__(self):
        self.HDB = WASDatabase(WAS_HISTORY_DATABASE)

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "mode": (["single_image", "incremental_image", "random"],),
                "index": ("INT", {"default": 0, "min": 0, "max": 150000, "step": 1}),
                "label": ("STRING", {"default": 'Batch 001', "multiline": False}),
                "path": ("STRING", {"default": '', "multiline": False}),
                "pattern": ("STRING", {"default": '*', "multiline": False}),
                "allow_RGBA_output": (["false","true"],),
            },
            "optional": {
                "filename_text_extension": (["true", "false"],),
            }
        }

    RETURN_TYPES = ("IMAGE",TEXT_TYPE)
    RETURN_NAMES = ("image","filename_text")
    FUNCTION = "load_batch_images"

    CATEGORY = "WAS Suite/IO"

    def load_batch_images(self, path, pattern='*', index=0, mode="single_image", label='Batch 001', allo
w_RGBA_output='false', filename_text_extension='true'):

        allow_RGBA_output = (allow_RGBA_output == 'true')

        if not os.path.exists(path):
            return (None, )
        fl = self.BatchImageLoader(path, label, pattern)
        new_paths = fl.image_paths
        if mode == 'single_image':
            image, filename = fl.get_image_by_id(index)
            if image == None:
                cstr(f"No valid image was found for the inded `{index}`").error.print()
                return (None, None)
        elif mode == 'incremental_image':
            image, filename = fl.get_next_image()
            if image == None:
                cstr(f"No valid image was found for the next ID. Did you remove images from the source direc
tory?").error.print()
                return (None, None)
        else:
            newindex = int(random.random() * len(fl.image_paths))
            image, filename = fl.get_image_by_id(newindex)
            if image == None:
                cstr(f"No valid image was found for the next ID. Did you remove images from the source direc
tory?").error.print()
```

```
            return (None, None)


        # Update history
        update_history_images(new_paths)

        if not allow_RGBA_output:
           image = image.convert("RGB")

        if filename_text_extension == "false":
            filename = os.path.splitext(filename)[0]

        return (pil2tensor(image), filename)

class BatchImageLoader:
    def __init__(self, directory_path, label, pattern):
        self.WDB = WDB
        self.image_paths = []
        self.load_images(directory_path, pattern)
        self.image_paths.sort()
        stored_directory_path = self.WDB.get('Batch Paths', label)
        stored_pattern = self.WDB.get('Batch Patterns', label)
        if stored_directory_path != directory_path or stored_pattern != pattern:
            self.index = 0
            self.WDB.insert('Batch Counters', label, 0)
            self.WDB.insert('Batch Paths', label, directory_path)
            self.WDB.insert('Batch Patterns', label, pattern)
        else:
            self.index = self.WDB.get('Batch Counters', label)
        self.label = label

    def load_images(self, directory_path, pattern):
        for file_name in glob.glob(os.path.join(glob.escape(directory_path), pattern), recursive=True):
            if file_name.lower().endswith(ALLOWED_EXT):
                abs_file_path = os.path.abspath(file_name)
                self.image_paths.append(abs_file_path)

    def get_image_by_id(self, image_id):
        if image_id < 0 or image_id >= len(self.image_paths):
            cstr(f"Invalid image index `{image_id}`").error.print()
            return
        i = Image.open(self.image_paths[image_id])
        i = ImageOps.exif_transpose(i)
        return (i, os.path.basename(self.image_paths[image_id]))

    def get_next_image(self):
        if self.index >= len(self.image_paths):
            self.index = 0
        image_path = self.image_paths[self.index]
        self.index += 1
        if self.index == len(self.image_paths):
            self.index = 0
        cstr(f'{cstr.color.YELLOW}{self.label}{cstr.color.END} Index: {self.index}').msg.print()
        self.WDB.insert('Batch Counters', self.label, self.index)
        i = Image.open(image_path)
        i = ImageOps.exif_transpose(i)
        return (i, os.path.basename(image_path))
        def get_current_image(self):
            if self.index >= len(self.image_paths):
                self.index = 0
            image_path = self.image_paths[self.index]
            return os.path.basename(image_path)

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        if kwargs['mode'] != 'single_image':
            return float("NaN")
        else:
            fl = WAS_Load_Image_Batch.BatchImageLoader(kwargs['path'], kwargs['label'], kwargs['patter
n'])
            filename = fl.get_current_image()
            image = os.path.join(kwargs['path'], filename)
            sha = get_sha256(image)
            return sha


# IMAGE HISTORY NODE

class WAS_Image_History:
    def __init__(self):
        self.HDB = WASDatabase(WAS_HISTORY_DATABASE)
        self.conf = getSuiteConfig()

    @classmethod
    def INPUT_TYPES(cls):
        HDB = WASDatabase(WAS_HISTORY_DATABASE)
        conf = getSuiteConfig()
        paths = ['No History']
        if HDB.catExists("History") and HDB.keyExists("History", "Images"):
            history_paths = HDB.get("History", "Images")
            if conf.__contains__('history_display_limit'):
                history_paths = history_paths[-conf['history_display_limit']:]
                paths = []
            for path_ in history_paths:
                paths.append(os.path.join('...'+os.sep+os.path.basename(os.path.dirname(path_)), os.path.b
asename(path_)))

        return {
            "required": {
                "image": (paths,),
            },
        }

    RETURN_TYPES = ("IMAGE",TEXT_TYPE)
    RETURN_NAMES = ("image","filename_text")
    FUNCTION = "image_history"

    CATEGORY = "WAS Suite/History"

    def image_history(self, image):
        self.HDB = WASDatabase(WAS_HISTORY_DATABASE)
        paths = {}
        if self.HDB.catExists("History") and self.HDB.keyExists("History", "Images"):
            history_paths = self.HDB.get("History", "Images")
            for path_ in history_paths:
```

```
            paths.update({os.path.join('...'+os.sep+os.path.basename(os.path.dirname(path_)), os.path.b
asename(path_)): path_})
        if os.path.exists(paths[image]) and paths.__contains__(image):
            return (pil2tensor(Image.open(paths[image]).convert('RGB')), os.path.basename(paths[image]))
        else:
            cstr(f"The image `{image}` does not exist!").error.print()
            return (pil2tensor(Image.new('RGB', (512,512), (0, 0, 0, 0))), 'null')

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

# IMAGE PADDING

class WAS_Image_Stitch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_a": ("IMAGE",),
                "image_b": ("IMAGE",),
                "stitch": (["top", "left", "bottom", "right"],),
                "feathering": ("INT", {"default": 50, "min": 0, "max": 2048, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_stitching"

    CATEGORY = "WAS Suite/Image/Transform"

    def image_stitching(self, image_a, image_b, stitch="right", feathering=50):

        valid_stitches = ["top", "left", "bottom", "right"]
        if stitch not in valid_stitches:
            cstr(f"The stitch mode `{stitch}` is not valid. Valid sitch modes are {', '.join(valid_stitches)}").erro
r.print()
        if feathering > 2048:
            cstr(f"The stitch feathering of `{feathering}` is too high. Please choose a value between `0` and
`2048`").error.print()

        WTools = WAS_Tools_Class();

        stitched_image = WTools.stitch_image(tensor2pil(image_a), tensor2pil(image_b), stitch, featherin
g)

        return (pil2tensor(stitched_image), )



# IMAGE PADDING

class WAS_Image_Padding:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "feathering": ("INT", {"default": 120, "min": 0, "max": 2048, "step": 1}),
                "feather_second_pass": (["true", "false"],),
                "left_padding": ("INT", {"default": 512, "min": 8, "max": 48000, "step": 1}),
                "right_padding": ("INT", {"default": 512, "min": 8, "max": 48000, "step": 1}),
                "top_padding": ("INT", {"default": 512, "min": 8, "max": 48000, "step": 1}),
                "bottom_padding": ("INT", {"default": 512, "min": 8, "max": 48000, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE", "IMAGE")
    FUNCTION = "image_padding"

    CATEGORY = "WAS Suite/Image/Transform"

    def image_padding(self, image, feathering, left_padding, right_padding, top_padding, bottom_paddin
g, feather_second_pass=True):
        padding = self.apply_image_padding(tensor2pil(
            image), left_padding, right_padding, top_padding, bottom_padding, feathering, second_pass=fe
ather_second_pass)
        return (pil2tensor(padding[0]), pil2tensor(padding[1]))

    def apply_image_padding(self, image, left_pad=100, right_pad=100, top_pad=100, bottom_pad=100, f
eather_radius=50, second_pass=True):
        # Create a mask for the feathered edge
        mask = Image.new('L', image.size, 255)
        draw = ImageDraw.Draw(mask)

        # Draw black rectangles at each edge of the image with the specified feather radius
        draw.rectangle((0, 0, feather_radius*2, image.height), fill=0)
        draw.rectangle((image.width-feather_radius*2, 0,
                       image.width, image.height), fill=0)
        draw.rectangle((0, 0, image.width, feather_radius*2), fill=0)
        draw.rectangle((0, image.height-feather_radius*2,
                       image.width, image.height), fill=0)

        # Blur the mask to create a smooth gradient between the black shapes and the white background
        mask = mask.filter(ImageFilter.GaussianBlur(radius=feather_radius))

        # Apply mask if second_pass is False, apply both masks if second_pass is True
        if second_pass:

            # Create a second mask for the additional feathering pass
            mask2 = Image.new('L', image.size, 255)
            draw2 = ImageDraw.Draw(mask2)

            # Draw black rectangles at each edge of the image with a smaller feather radius
            feather_radius2 = int(feather_radius / 4)
            draw2.rectangle((0, 0, feather_radius2*2, image.height), fill=0)
            draw2.rectangle((image.width-feather_radius2*2, 0,
                            image.width, image.height), fill=0)
            draw2.rectangle((0, 0, image.width, feather_radius2*2), fill=0)
            draw2.rectangle((0, image.height-feather_radius2*2,
                        image.width, image.height), fill=0)

            # Blur the mask to create a smooth gradient between the black shapes and the white backgroun
d
            mask2 = mask2.filter(
                ImageFilter.GaussianBlur(radius=feather_radius2))

            feathered_im = Image.new('RGBA', image.size, (0, 0, 0, 0))
            feathered_im.paste(image, (0, 0), mask)
            feathered_im.paste(image, (0, 0), mask)

            # Apply the second mask to the feathered image
            feathered_im.paste(image, (0, 0), mask2)
            feathered_im.paste(image, (0, 0), mask2)

        else:

            # Apply the fist maskk
            feathered_im = Image.new('RGBA', image.size, (0, 0, 0, 0))
            feathered_im.paste(image, (0, 0), mask)

        # Calculate the new size of the image with padding added
        new_size = (feathered_im.width + left_pad + right_pad,
                    feathered_im.height + top_pad + bottom_pad)

        # Create a new transparent image with the new size
        new_im = Image.new('RGBA', new_size, (0, 0, 0, 0))

        # Paste the feathered image onto the new image with the padding
        new_im.paste(feathered_im, (left_pad, top_pad))

        # Create Padding Mask
        padding_mask = Image.new('L', new_size, 0)

        # Create a mask where the transparent pixels have a gradient
        gradient = [(int(255 * (1 - p[3] / 255)) if p[3] != 0 else 255)
                    for p in new_im.getdata()]
        padding_mask.putdata(gradient)

        # Save the new image with alpha channel as a PNG file
        return (new_im, padding_mask.convert('RGB'))


# IMAGE THRESHOLD NODE

class WAS_Image_Threshold:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "threshold": ("FLOAT", {"default": 0.5, "min": 0.0, "max": 1.0, "step": 0.01}),
            },
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_threshold"

    CATEGORY = "WAS Suite/Image/Process"

    def image_threshold(self, image, threshold=0.5):
        return (pil2tensor(self.apply_threshold(tensor2pil(image), threshold)), )

    def apply_threshold(self, input_image, threshold=0.5):
        # Convert the input image to grayscale
        grayscale_image = input_image.convert('L')

        # Apply the threshold to the grayscale image
        threshold_value = int(threshold * 255)
        thresholded_image = grayscale_image.point(
            lambda x: 255 if x >= threshold_value else 0, mode='L')

        return thresholded_image


# IMAGE CHROMATIC ABERRATION NODE

class WAS_Image_Chromatic_Aberration:

    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "red_offset": ("INT", {"default": 2, "min": -255, "max": 255, "step": 1}),
                "green_offset": ("INT", {"default": -1, "min": -255, "max": 255, "step": 1}),
                "blue_offset": ("INT", {"default": 1, "min": -255, "max": 255, "step": 1}),
                "intensity": ("FLOAT", {"default": 0.5, "min": 0.0, "max": 1.0, "step": 0.01}),
                "fade_radius": ("INT", {"default": 12, "min": 0, "max": 1024, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_chromatic_aberration"

    CATEGORY = "WAS Suite/Image/Filter"

    def image_chromatic_aberration(self, image, red_offset=4, green_offset=2, blue_offset=0, intensity=
1, fade_radius=12):
        return (pil2tensor(self.apply_chromatic_aberration(tensor2pil(image), red_offset, green_offset, blue
_offset, intensity, fade_radius)), )

    def apply_chromatic_aberration(self, img, r_offset, g_offset, b_offset, intensity, fade_radius):

        def lingrad(size, direction, white_ratio):
            image = Image.new('RGB', size)
            draw = ImageDraw.Draw(image)
            if direction == 'vertical':
                black_end = size[1] - white_ratio
                range_start = 0
                range_end = size[1]
                range_step = 1
                for y in range(range_start, range_end, range_step):
                    color_ratio = y / size[1]
                    if y <= black_end:
                        color = (0, 0, 0)
                    else:
                        color_value = int(((y - black_end) / (size[1] - black_end)) * 255)
                        color = (color_value, color_value, color_value)
                    draw.line([(0, y), (size[0], y)], fill=color)
            elif direction == 'horizontal':
                black_end = size[0] - white_ratio
                range_start = 0
                range_end = size[0]
                range_step = 1
                for x in range(range_start, range_end, range_step):
                    color_ratio = x / size[0]
                    if x <= black_end:
                        color = (0, 0, 0)
                    else:
                        color_value = int(((x - black_end) / (size[0] - black_end)) * 255)
                        color = (color_value, color_value, color_value)
                    draw.line([(x, 0), (x, size[1])], fill=color)

            return image.convert("L")

        def create_fade_mask(size, fade_radius):
            mask = Image.new("L", size, 255)

            left = ImageOps.invert(lingrad(size, 'horizontal', int(fade_radius * 2)))
            right = left.copy().transpose(Image.FLIP_LEFT_RIGHT)
            top = ImageOps.invert(lingrad(size, 'vertical', int(fade_radius *2)))
            bottom = top.copy().transpose(Image.FLIP_TOP_BOTTOM)

            # Multiply masks with the original mask image
            mask = ImageChops.multiply(mask, left)
            mask = ImageChops.multiply(mask, right)
            mask = ImageChops.multiply(mask, top)
            mask = ImageChops.multiply(mask, bottom)
            mask = ImageChops.multiply(mask, mask)

            return mask

        # split the channels of the image
        r, g, b = img.split()

        # apply the offset to each channel
        r_offset_img = ImageChops.offset(r, r_offset, 0)
        g_offset_img = ImageChops.offset(g, 0, g_offset)
        b_offset_img = ImageChops.offset(b, 0, b_offset)

        # merge the channels with the offsets
        merged = Image.merge("RGB", (r_offset_img, g_offset_img, b_offset_img))

        # create fade masks for blending
        fade_mask = create_fade_mask(img.size, fade_radius)

        # merge the blended channels back into an RGB image
```

```
        result = Image.composite(merged, img, fade_mask).convert("RGB")

        return result


# IMAGE BLOOM FILTER

class WAS_Image_Bloom_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "radius": ("FLOAT", {"default": 10, "min": 0.0, "max": 1024, "step": 0.1}),
                "intensity": ("FLOAT", {"default": 1, "min": 0.0, "max": 1.0, "step": 0.1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_bloom"

    CATEGORY = "WAS Suite/Image/Filter"

    def image_bloom(self, image, radius=0.5, intensity=1.0):
        return (pil2tensor(self.apply_bloom_filter(tensor2pil(image), radius, intensity)), )

    def apply_bloom_filter(self, input_image, radius, bloom_factor):
        # Apply a blur filter to the input image
        blurred_image = input_image.filter(
            ImageFilter.GaussianBlur(radius=radius))

        # Subtract the blurred image from the input image to create a high-pass filter
        high_pass_filter = ImageChops.subtract(input_image, blurred_image)

        # Create a blurred version of the bloom filter
        bloom_filter = high_pass_filter.filter(
            ImageFilter.GaussianBlur(radius=radius*2))

        # Adjust brightness and levels of bloom filter
        bloom_filter = ImageEnhance.Brightness(bloom_filter).enhance(2.0)

        # Multiply the bloom image with the bloom factor
        bloom_filter = ImageChops.multiply(bloom_filter, Image.new('RGB', input_image.size, (int(
            255 * bloom_factor), int(255 * bloom_factor), int(255 * bloom_factor))))

        # Multiply the bloom filter with the original image using the bloom factor
        blended_image = ImageChops.screen(input_image, bloom_filter)

        return blended_image


# IMAGE ROTATE HUE

class WAS_Image_Rotate_Hue:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "hue_shift": ("FLOAT", {"default": 0.0, "min": 0.0, "max": 1.0, "step": 0.001}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "rotate_hue"

    CATEGORY = "WAS Suite/Image/Adjustment"

    def rotate_hue(self, image, hue_shift=0.0):
        if hue_shift > 1.0 or hue_shift < 0.0:
            cstr(f"The hue_shift `{cstr.color.LIGHTYELLOW}{hue_shift}{cstr.color.END}` is out of range. Vali
d range is {cstr.color.BOLD}0.0 - 1.0{cstr.color.END}").error.print()
            hue_shift = 0.0
        shifted_hue = pil2tensor(self.hue_rotation(image, hue_shift))
        return (shifted_hue, )

    def hue_rotation(self, image, hue_shift=0.0):
        import colorsys
        if hue_shift > 1.0 or hue_shift < 0.0:
            print(f"The hue_shift '{hue_shift}' is out of range. Valid range is 0.0 - 1.0")
            hue_shift = 0.0

        pil_image = tensor2pil(image)
        width, height = pil_image.size
        rotated_image = Image.new("RGB", (width, height))

        for x in range(width):
            for y in range(height):
                r, g, b = pil_image.getpixel((x, y))
                h, l, s = colorsys.rgb_to_hls(r / 255, g / 255, b / 255)
                h = (h + hue_shift) % 1.0
                r, g, b = colorsys.hls_to_rgb(h, l, s)
                r, g, b = int(r * 255), int(g * 255), int(b * 255)
                rotated_image.putpixel((x, y), (r, g, b))

        return rotated_image


# IMAGE REMOVE COLOR

class WAS_Image_Remove_Color:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "target_red": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "target_green": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "target_blue": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "replace_red": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "replace_green": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "replace_blue": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "clip_threshold": ("INT", {"default": 10, "min": 0, "max": 255, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_remove_color"

    CATEGORY = "WAS Suite/Image/Process"

    def image_remove_color(self, image, clip_threshold=10, target_red=255, target_green=255, target_bl
ue=255, replace_red=255, replace_green=255, replace_blue=255):
        return (pil2tensor(self.apply_remove_color(tensor2pil(image), clip_threshold, (target_red, target_gr
een, target_blue), (replace_red, replace_green, replace_blue))), )

    def apply_remove_color(self, image, threshold=10, color=(255, 255, 255), rep_color=(0, 0, 0)):
        # Create a color image with the same size as the input image
        color_image = Image.new('RGB', image.size, color)

        # Calculate the difference between the input image and the color image
        diff_image = ImageChops.difference(image, color_image)

        # Convert the difference image to grayscale
        gray_image = diff_image.convert('L')

        # Apply a threshold to the grayscale difference image
        mask_image = gray_image.point(lambda x: 255 if x > threshold else 0)

        # Invert the mask image
        mask_image = ImageOps.invert(mask_image)

        # Apply the mask to the original image
        result_image = Image.composite(
            Image.new('RGB', image.size, rep_color), image, mask_image)

        return result_image


# IMAGE REMOVE BACKGROUND

class WAS_Remove_Background:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "mode": (["background", "foreground"],),
                "threshold": ("INT", {"default": 127, "min": 0, "max": 255, "step": 1}),
                "threshold_tolerance": ("INT", {"default": 2, "min": 1, "max": 24, "step": 1}),
            },
        }
    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "image_remove_background"

    CATEGORY = "WAS Suite/Image/Process"

    def image_remove_background(self, images, mode='background', threshold=127, threshold_toleranc
e=2):
        return (self.remove_background(images, mode, threshold, threshold_tolerance), )

    def remove_background(self, image, mode, threshold, threshold_tolerance):
        images = []
        image = [tensor2pil(img) for img in image]
        for img in image:
            grayscale_image = img.convert('L')
            if mode == 'background':
                grayscale_image = ImageOps.invert(grayscale_image)
                threshold = 255 - threshold  # adjust the threshold for "background" mode
            blurred_image = grayscale_image.filter(
                ImageFilter.GaussianBlur(radius=threshold_tolerance))
            binary_image = blurred_image.point(
                lambda x: 0 if x < threshold else 255, '1')
            mask = binary_image.convert('L')
            inverted_mask = ImageOps.invert(mask)
            transparent_image = img.copy()
            transparent_image.putalpha(inverted_mask)
            images.append(pil2tensor(transparent_image))
        batch = torch.cat(images, dim=0)

        return batch

# IMAGE REMBG
# Sam model needs additional input, may need to be new node entirely
#   See: https://github.com/danielgatis/rembg/blob/main/USAGE.md#using-input-points
# u2net_cloth_seg model needs additional inputs, may create a new node
# An undocumented feature "putaplha" changes how alpha is applied, but does not appear to make a d
ifference

class WAS_Remove_Rembg:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "transparency": ("BOOLEAN", {"default": True},),
                "model": (["u2net", "u2netp", "u2net_human_seg", "silueta", "isnet-general-use", "isnet-anim
e"],),
                "post_processing": ("BOOLEAN", {"default": False}),
                "only_mask": ("BOOLEAN", {"default": False},),
                "alpha_matting": ("BOOLEAN", {"default": False},),
                "alpha_matting_foreground_threshold": ("INT", {"default": 240, "min": 0, "max": 255}),
                "alpha_matting_background_threshold": ("INT", {"default": 10, "min": 0, "max": 255}),
                "alpha_matting_erode_size": ("INT", {"default": 10, "min": 0, "max": 255}),
                "background_color": (["none", "black", "white", "magenta", "chroma green", "chroma blue"],),
                # "putalpha": ("BOOLEAN", {"default": True},),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "image_rembg"

    CATEGORY = "WAS Suite/Image/AI"

     # A helper function to convert from strings to logical boolean
     # Conforms to https://docs.python.org/3/library/stdtypes.html#truth-value-testing
     # With the addition of evaluating string representations of Falsey types
    def __convertToBool(self, x):

        # Evaluate string representation of False types
        if type(x) == str:
            x = x.strip()
            if (x.lower() == 'false'
                or x.lower() == 'none'
                or x == '0'
                or x == '0.0'
                or x == '0j'
                or x == "''"
                or x == '""'
                or x == "()"
                or x == "[]"
                or x == "{}"
                or x.lower() == "decimal(0)"
                or x.lower() == "fraction(0,1)"
                or x.lower() == "set()"
                or x.lower() == "range(0)"
            ):
                return False
            else:
                return True

        # Anything else will be evaluated by the bool function
        return bool(x)

    def image_rembg(
            self,
            images,
            transparency=True,
            model="u2net",
            alpha_matting=False,
            alpha_matting_foreground_threshold=240,
            alpha_matting_background_threshold=10,
            alpha_matting_erode_size=10,
            post_processing=False,
            only_mask=False,
            background_color="none",
            # putalpha = False,
    ):

        # ComfyUI will allow strings in place of booleans, validate the input.
        transparency = transparency if type(transparency) is bool else self.__convertToBool(transparency)
        alpha_matting = alpha_matting if type(alpha_matting) is bool else self.__convertToBool(alpha_matti
ng)
        post_processing = post_processing if type(post_processing) is bool else self.__convertToBool(pos
t_processing)
        only_mask = only_mask if type(only_mask) is bool else self.__convertToBool(only_mask)

        if "rembg" not in packages():
            install_package("rembg")

        from rembg import remove, new_session

        os.environ['U2NET_HOME'] = os.path.join(MODELS_DIR, 'rembg')
        os.makedirs(os.environ['U2NET_HOME'], exist_ok=True)

        # Set bgcolor
        bgrgba = None
        if background_color == "black":
            bgrgba = [0, 0, 0, 255]
        elif background_color == "white":
            bgrgba = [255, 255, 255, 255]
        elif background_color == "magenta":
            bgrgba = [255, 0, 255, 255]
        elif background_color == "chroma green":
            bgrgba = [0, 177, 64, 255]
        elif background_color == "chroma blue":
            bgrgba = [0, 71, 187, 255]
        else:
            bgrgba = None

        if transparency and bgrgba is not None:
            bgrgba[3] = 0

        batch_tensor = []
        for image in images:
            image = tensor2pil(image)
            batch_tensor.append(pil2tensor(
                remove(
                    image,
                    session=new_session(model),
                    post_process_mask=post_processing,
                    alpha_matting=alpha_matting,
                    alpha_matting_foreground_threshold=alpha_matting_foreground_threshold,
                    alpha_matting_background_threshold=alpha_matting_background_threshold,
                    alpha_matting_erode_size=alpha_matting_erode_size,
                    only_mask=only_mask,
                    bgcolor=bgrgba,
                    # putalpha = putalpha,
                )
                .convert(('RGBA' if transparency else 'RGB'))))
        batch_tensor = torch.cat(batch_tensor, dim=0)

        return (batch_tensor,)


# IMAGE BLEND MASK NODE

class WAS_Image_Blend_Mask:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_a": ("IMAGE",),
                "image_b": ("IMAGE",),
                "mask": ("IMAGE",),
                "blend_percentage": ("FLOAT", {"default": 0.5, "min": 0.0, "max": 1.0, "step": 0.01}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_blend_mask"

    CATEGORY = "WAS Suite/Image"

    def image_blend_mask(self, image_a, image_b, mask, blend_percentage):

        # Convert images to PIL
        img_a = tensor2pil(image_a)
        img_b = tensor2pil(image_b)
        mask = ImageOps.invert(tensor2pil(mask).convert('L'))

        # Mask image
        masked_img = Image.composite(img_a, img_b, mask.resize(img_a.size))

        # Blend image
        blend_mask = Image.new(mode="L", size=img_a.size,
                               color=(round(blend_percentage * 255)))
        blend_mask = ImageOps.invert(blend_mask)
        img_result = Image.composite(img_a, masked_img, blend_mask)

        del img_a, img_b, blend_mask, mask

        return (pil2tensor(img_result), )


# IMAGE BLANK NOE


class WAS_Image_Blank:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "width": ("INT", {"default": 512, "min": 8, "max": 4096, "step": 1}),
                "height": ("INT", {"default": 512, "min": 8, "max": 4096, "step": 1}),
                "red": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "green": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
                "blue": ("INT", {"default": 255, "min": 0, "max": 255, "step": 1}),
            }
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "blank_image"
    CATEGORY = "WAS Suite/Image"

    def blank_image(self, width, height, red, green, blue):

        # Ensure multiples
        width = (width // 8) * 8
        height = (height // 8) * 8

        # Blend image
        blank = Image.new(mode="RGB", size=(width, height),
                          color=(red, green, blue))

        return (pil2tensor(blank), )


# IMAGE HIGH PASS

class WAS_Image_High_Pass_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "radius": ("INT", {"default": 10, "min": 1, "max": 500, "step": 1}),
                "strength": ("FLOAT", {"default": 1.5, "min": 0.0, "max": 255.0, "step": 0.1}),
                "color_output": (["true", "false"],),
                "neutral_background": (["true", "false"],),
            }
        }
    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "high_pass"

    CATEGORY = "WAS Suite/Image/Filter"

    def high_pass(self, images, radius=10, strength=1.5, color_output="true", neutral_background="tru
e"):
        batch_tensor = []
        for image in images:
            transformed_image = self.apply_hpf(tensor2pil(image), radius, strength, color_output, neutral_b
ackground)
            batch_tensor.append(pil2tensor(transformed_image))
        batch_tensor = torch.cat(batch_tensor, dim=0)
        return (batch_tensor, )

    def apply_hpf(self, img, radius=10, strength=1.5, color_output="true", neutral_background="true"):
        img_arr = np.array(img).astype('float')
        blurred_arr = np.array(img.filter(ImageFilter.GaussianBlur(radius=radius))).astype('float')
        hpf_arr = img_arr - blurred_arr
        hpf_arr = np.clip(hpf_arr * strength, 0, 255).astype('uint8')

        if color_output == "true":
            high_pass = Image.fromarray(hpf_arr, mode='RGB')
        else:
            grayscale_arr = np.mean(hpf_arr, axis=2).astype('uint8')
            high_pass = Image.fromarray(grayscale_arr, mode='L')

        if neutral_background == "true":
            neutral_color = (128, 128, 128) if high_pass.mode == 'RGB' else 128
            neutral_bg = Image.new(high_pass.mode, high_pass.size, neutral_color)
            high_pass = ImageChops.screen(neutral_bg, high_pass)

        return high_pass.convert("RGB")


# IMAGE LEVELS NODE

class WAS_Image_Levels:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "black_level": ("FLOAT", {"default": 0.0, "min": 0.0, "max": 255.0, "step": 0.1}),
                "mid_level": ("FLOAT", {"default": 127.5, "min": 0.0, "max": 255.0, "step": 0.1}),
                "white_level": ("FLOAT", {"default": 255, "min": 0.0, "max": 255.0, "step": 0.1}),
            }
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "apply_image_levels"

    CATEGORY = "WAS Suite/Image/Adjustment"

    def apply_image_levels(self, image, black_level, mid_level, white_level):

        # Convert image to PIL
        tensor_images = []
        for img in image:
            img = tensor2pil(img)
            levels = self.AdjustLevels(black_level, mid_level, white_level)
            tensor_images.append(pil2tensor(levels.adjust(img)))
        tensor_images = torch.cat(tensor_images, dim=0)

        # Return adjust image tensor
        return (tensor_images, )


    class AdjustLevels:
        def __init__(self, min_level, mid_level, max_level):
            self.min_level = min_level
            self.mid_level = mid_level
            self.max_level = max_level

        def adjust(self, im):

            im_arr = np.array(im)
            im_arr[im_arr < self.min_level] = self.min_level
            im_arr = (im_arr - self.min_level) * \
                (255 / (self.max_level - self.min_level))
        im_arr[im_arr < 0] = 0
        im_arr[im_arr > 255] = 255
        im_arr = im_arr.astype(np.uint8)

        im = Image.fromarray(im_arr)
        im = ImageOps.autocontrast(im, cutoff=self.max_level)

        return im


# FILM GRAIN NODE

class WAS_Film_Grain:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "density": ("FLOAT", {"default": 1.0, "min": 0.01, "max": 1.0, "step": 0.01}),
                "intensity": ("FLOAT", {"default": 1.0, "min": 0.01, "max": 1.0, "step": 0.01}),
                "highlights": ("FLOAT", {"default": 1.0, "min": 0.01, "max": 255.0, "step": 0.01}),
                "supersample_factor": ("INT", {"default": 4, "min": 1, "max": 8, "step": 1})
            }
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "film_grain"

    CATEGORY = "WAS Suite/Image/Filter"

    def film_grain(self, image, density, intensity, highlights, supersample_factor):
        return (pil2tensor(self.apply_film_grain(tensor2pil(image), density, intensity, highlights, supersampl
e_factor)), )

    def apply_film_grain(self, img, density=0.1, intensity=1.0, highlights=1.0, supersample_factor=4):
        """
        Apply grayscale noise with specified density, intensity, and highlights to a PIL image.
        """
        img_gray = img.convert('L')
        original_size = img.size
        img_gray = img_gray.resize(
            ((img.size[0] * supersample_factor), (img.size[1] * supersample_factor)), Image.Resampling(2))
        num_pixels = int(density * img_gray.size[0] * img_gray.size[1])

        noise_pixels = []
        for i in range(num_pixels):
            x = random.randint(0, img_gray.size[0]-1)
            y = random.randint(0, img_gray.size[1]-1)
            noise_pixels.append((x, y))

        for x, y in noise_pixels:
            value = random.randint(0, 255)
            img_gray.putpixel((x, y), value)

        img_noise = img_gray.convert('RGB')
        img_noise = img_noise.filter(ImageFilter.GaussianBlur(radius=0.125))
        img_noise = img_noise.resize(original_size, Image.Resampling(1))
        img_noise = img_noise.filter(ImageFilter.EDGE_ENHANCE_MORE)
        img_final = Image.blend(img, img_noise, intensity)
        enhancer = ImageEnhance.Brightness(img_final)
        img_highlights = enhancer.enhance(highlights)

        # Return the final image
        return img_highlights


# IMAGE FLIP NODE

class WAS_Image_Flip:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "mode": (["horizontal", "vertical",],),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "image_flip"

    CATEGORY = "WAS Suite/Image/Transform"

    def image_flip(self, images, mode):

        batch_tensor = []
        for image in images:
            image = tensor2pil(image)
            if mode == 'horizontal':
                image = image.transpose(0)
            if mode == 'vertical':
                image = image.transpose(1)
            batch_tensor.append(pil2tensor(image))
        batch_tensor = torch.cat(batch_tensor, dim=0)

        return (batch_tensor, )


class WAS_Image_Rotate:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "mode": (["transpose", "internal",],),
                "rotation": ("INT", {"default": 0, "min": 0, "max": 360, "step": 90}),
                "sampler": (["nearest", "bilinear", "bicubic"],),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "image_rotate"

    CATEGORY = "WAS Suite/Image/Transform"

    def image_rotate(self, images, mode, rotation, sampler):

        batch_tensor = []
        for image in images:
            # PIL Image
            image = tensor2pil(image)

            # Check rotation
            if rotation > 360:
                rotation = int(360)
            if (rotation % 90 != 0):
                rotation = int((rotation//90)*90)

            # Set Sampler
            if sampler:
                if sampler == 'nearest':
                    sampler = Image.NEAREST
                elif sampler == 'bicubic':
                    sampler = Image.BICUBIC
                elif sampler == 'bilinear':
                    sampler = Image.BILINEAR
                else:
                    sampler == Image.BILINEAR

            # Rotate Image
            if mode == 'internal':
                image = image.rotate(rotation, sampler)
            else:
                rot = int(rotation / 90)
                for _ in range(rot):
                    image = image.transpose(2)

            batch_tensor.append(pil2tensor(image))

        batch_tensor = torch.cat(batch_tensor, dim=0)

        return (batch_tensor, )


# IMAGE NOVA SINE FILTER

class WAS_Image_Nova_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "amplitude": ("FLOAT", {"default": 0.1, "min": 0.0, "max": 1.0, "step": 0.001}),
                "frequency": ("FLOAT", {"default": 3.14, "min": 0.0, "max": 100.0, "step": 0.001}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "nova_sine"

    CATEGORY = "WAS Suite/Image/Filter"

    def nova_sine(self, image, amplitude, frequency):

        # Convert image to numpy
        img = tensor2pil(image)

        # Convert the image to a numpy array
        img_array = np.array(img)

        # Define a sine wave function
        def sine(x, freq, amp):
            return amp * np.sin(2 * np.pi * freq * x)

        # Calculate the sampling frequency of the image
        resolution = img.info.get('dpi')  # PPI
        physical_size = img.size  # pixels

        if resolution is not None:
            # Convert PPI to pixels per millimeter (PPM)
            ppm = 25.4 / resolution
            physical_size = tuple(int(pix * ppm) for pix in physical_size)

        # Set the maximum frequency for the sine wave
        max_freq = img.width / 2

        # Ensure frequency isn't outside visual representable range
        if frequency > max_freq:
            frequency = max_freq

        # Apply levels to the image using the sine function
        for i in range(img_array.shape[0]):
            for j in range(img_array.shape[1]):
                for k in range(img_array.shape[2]):
                    img_array[i, j, k] = int(
                        sine(img_array[i, j, k]/255, frequency, amplitude) * 255)

        return (torch.from_numpy(img_array.astype(np.float32) / 255.0).unsqueeze(0), )


# IMAGE CANNY FILTER


class WAS_Canny_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "enable_threshold": (['false', 'true'],),
                "threshold_low": ("FLOAT", {"default": 0.0, "min": 0.0, "max": 1.0, "step": 0.01}),
                "threshold_high": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 1.0, "step": 0.01}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "canny_filter"

    CATEGORY = "WAS Suite/Image/Filter"

    def canny_filter(self, images, threshold_low, threshold_high, enable_threshold):

        if enable_threshold == 'false':
            threshold_low = None
            threshold_high = None

        batch_tensor = []
        for image in images:

            image_canny = Image.fromarray(self.Canny_detector(
                255. * image.cpu().numpy().squeeze(), threshold_low, threshold_high)).convert('RGB')

            batch_tensor.append(pil2tensor(image_canny))

        batch_tensor = torch.cat(batch_tensor, dim=0)

        return (pil2tensor(image_canny), )

    def Canny_detector(self, img, weak_th=None, strong_th=None):

        import cv2

        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        img = cv2.GaussianBlur(img, (5, 5), 1.4)
        gx = cv2.Sobel(np.float32(img), cv2.CV_64F, 1, 0, 3)  # type: ignore
        gy = cv2.Sobel(np.float32(img), cv2.CV_64F, 0, 1, 3)  # type: ignore

        mag, ang = cv2.cartToPolar(gx, gy, angleInDegrees=True)

        mag_max = np.max(mag)
        if not weak_th:
            weak_th = mag_max * 0.1
        if not strong_th:
            strong_th = mag_max * 0.5

        height, width = img.shape

        for i_x in range(width):
            for i_y in range(height):

                grad_ang = ang[i_y, i_x]
                grad_ang = abs(
                    grad_ang-180) if abs(grad_ang) > 180 else abs(grad_ang)
                neighb_1_x, neighb_1_y = -1, -1
                neighb_2_x, neighb_2_y = -1, -1

                if grad_ang <= 22.5:
                    neighb_1_x, neighb_1_y = i_x-1, i_y
                    neighb_2_x, neighb_2_y = i_x + 1, i_y

                elif grad_ang > 22.5 and grad_ang <= (22.5 + 45):
                    neighb_1_x, neighb_1_y = i_x-1, i_y-1
                    neighb_2_x, neighb_2_y = i_x + 1, i_y + 1
                elif grad_ang > (22.5 + 45) and grad_ang <= (22.5 + 90):
                    neighb_1_x, neighb_1_y = i_x, i_y-1
                    neighb_2_x, neighb_2_y = i_x, i_y + 1
                elif grad_ang > (22.5 + 90) and grad_ang <= (22.5 + 135):
                    neighb_1_x, neighb_1_y = i_x-1, i_y + 1
                    neighb_2_x, neighb_2_y = i_x + 1, i_y-1
                elif grad_ang > (22.5 + 135) and grad_ang <= (22.5 + 180):
                    neighb_1_x, neighb_1_y = i_x-1, i_y
                    neighb_2_x, neighb_2_y = i_x + 1, i_y
                if width > neighb_1_x >= 0 and height > neighb_1_y >= 0:
                    if mag[i_y, i_x] < mag[neighb_1_y, neighb_1_x]:
                        mag[i_y, i_x] = 0
                        continue

                if width > neighb_2_x >= 0 and height > neighb_2_y >= 0:
                    if mag[i_y, i_x] < mag[neighb_2_y, neighb_2_x]:
                        mag[i_y, i_x] = 0

        weak_ids = np.zeros_like(img)
        strong_ids = np.zeros_like(img)
        ids = np.zeros_like(img)

        for i_x in range(width):
            for i_y in range(height):

                grad_mag = mag[i_y, i_x]

                if grad_mag < weak_th:
                    mag[i_y, i_x] = 0
                elif strong_th > grad_mag >= weak_th:
                    ids[i_y, i_x] = 1
                else:
                    ids[i_y, i_x] = 2

        return mag

# IMAGE EDGE DETECTION

class WAS_Image_Edge:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "mode": (["normal", "laplacian"],),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_edges"

    CATEGORY = "WAS Suite/Image/Filter"

    def image_edges(self, image, mode):

        # Convert image to PIL
        image = tensor2pil(image)

        # Detect edges
        if mode:
            if mode == "normal":
                image = image.filter(ImageFilter.FIND_EDGES)
            elif mode == "laplacian":
                image = image.filter(ImageFilter.Kernel((3, 3), (-1, -1, -1, -1, 8,
                                                        -1, -1, -1, -1), 1, 0))
            else:
                image = image

        return (torch.from_numpy(np.array(image).astype(np.float32) / 255.0).unsqueeze(0), )


# IMAGE FDOF NODE

class WAS_Image_fDOF:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "depth": ("IMAGE",),
                "mode": (["mock", "gaussian", "box"],),
                "radius": ("INT", {"default": 8, "min": 1, "max": 128, "step": 1}),
                "samples": ("INT", {"default": 1, "min": 1, "max": 3, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "fdof_composite"

    CATEGORY = "WAS Suite/Image/Filter"

    def fdof_composite(self, image, depth, radius, samples, mode):

        import cv2 as cv

        # Convert tensor to a PIL Image
        tensor_images = []
        for i in range(len(image)):
            if i <= len(image):
                img = tensor2pil(image[i])
            else:
                img = tensor2pil(image[-1])
            if i <= len(depth):
                dpth = tensor2pil(depth[i])
            else:
                dpth = tensor2pil(depth[-1])
            tensor_images.append(pil2tensor(self.portraitBlur(img, dpth, radius, samples, mode)))
        tensor_images = torch.cat(tensor_images, dim=0)

        return (tensor_images, )

    def portraitBlur(self, img, mask, radius, samples, mode='mock'):
        mask = mask.resize(img.size).convert('L')
        bimg: Optional[Image.Image] = None
        if mode == 'mock':
            bimg = medianFilter(img, radius, (radius * 1500), 75)
        elif mode == 'gaussian':
            bimg = img.filter(ImageFilter.GaussianBlur(radius=radius))
        elif mode == 'box':
            bimg = img.filter(ImageFilter.BoxBlur(radius))
        else:
            return
        bimg.convert(img.mode)
        rimg: Optional[Image.Image] = None
        if samples > 1:
            for i in range(samples):
                if not rimg:
                    rimg = Image.composite(img, bimg, mask)
                else:
                    rimg = Image.composite(rimg, bimg, mask)
        else:
            rimg = Image.composite(img, bimg, mask).convert('RGB')

        return rimg


# IMAGE DRAGAN PHOTOGRAPHY FILTER

class WAS_Dragon_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "saturation": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 16.0, "step": 0.01}),
                "contrast": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 16.0, "step": 0.01}),
                "brightness": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 16.0, "step": 0.01}),
                "sharpness": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 6.0, "step": 0.01}),
                "highpass_radius": ("FLOAT", {"default": 6.0, "min": 0.0, "max": 255.0, "step": 0.01}),
                "highpass_samples": ("INT", {"default": 1, "min": 0, "max": 6.0, "step": 1}),
                "highpass_strength": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 3.0, "step": 0.01}),
                "colorize": (["true","false"],),
            },
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "apply_dragan_filter"

    CATEGORY = "WAS Suite/Image/Filter"

    def apply_dragan_filter(self, image, saturation, contrast, sharpness, brightness, highpass_radius, hig
hpass_samples, highpass_strength, colorize):

        WTools = WAS_Tools_Class()

        tensor_images = []
        for img in image:
            tensor_images.append(pil2tensor(WTools.dragan_filter(tensor2pil(img), saturation, contrast, sha
rpness, brightness, highpass_radius, highpass_samples, highpass_strength, colorize)))
        tensor_images = torch.cat(tensor_images, dim=0)

        return (tensor_images, )



# IMAGE MEDIAN FILTER NODE

class WAS_Image_Median_Filter:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "diameter": ("INT", {"default": 2.0, "min": 0.1, "max": 255, "step": 1}),
                "sigma_color": ("FLOAT", {"default": 10.0, "min": -255.0, "max": 255.0, "step": 0.1}),
                "sigma_space": ("FLOAT", {"default": 10.0, "min": -255.0, "max": 255.0, "step": 0.1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "apply_median_filter"

    CATEGORY = "WAS Suite/Image/Filter"

    def apply_median_filter(self, image, diameter, sigma_color, sigma_space):

        tensor_images = []
        for img in image:
            img = tensor2pil(img)
            # Apply Median Filter effect
            tensor_images.append(pil2tensor(medianFilter(img, diameter, sigma_color, sigma_space)))
        tensor_images = torch.cat(tensor_images, dim=0)

        return (tensor_images, )

# IMAGE SELECT COLOR


class WAS_Image_Select_Color:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "red": ("INT", {"default": 255.0, "min": 0.0, "max": 255.0, "step": 0.1}),
                "green": ("INT", {"default": 255.0, "min": 0.0, "max": 255.0, "step": 0.1}),
                "blue": ("INT", {"default": 255.0, "min": 0.0, "max": 255.0, "step": 0.1}),
                "variance": ("INT", {"default": 10, "min": 0, "max": 255, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "select_color"

    CATEGORY = "WAS Suite/Image/Process"

    def select_color(self, image, red=255, green=255, blue=255, variance=10):

        image = self.color_pick(tensor2pil(image), red, green, blue, variance)

        return (pil2tensor(image), )

    def color_pick(self, image, red=255, green=255, blue=255, variance=10):
        # Convert image to RGB mode
        image = image.convert('RGB')

        # Create a new black image of the same size as the input image
        selected_color = Image.new('RGB', image.size, (0, 0, 0))

        # Get the width and height of the image
        width, height = image.size

        # Loop through every pixel in the image
        for x in range(width):
            for y in range(height):
                # Get the color of the pixel
                pixel = image.getpixel((x, y))
                r, g, b = pixel

                # Check if the pixel is within the specified color range
                if ((r >= red-variance) and (r <= red+variance) and
                    (g >= green-variance) and (g <= green+variance) and
                        (b >= blue-variance) and (b <= blue+variance)):
                    # Set the pixel in the selected_color image to the RGB value of the pixel
                    selected_color.putpixel((x, y), (r, g, b))

        # Return the selected color image
        return selected_color

# IMAGE CONVERT TO CHANNEL


class WAS_Image_Select_Channel:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "channel": (['red', 'green', 'blue'],),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "select_channel"

    CATEGORY = "WAS Suite/Image/Process"

    def select_channel(self, image, channel='red'):

        image = self.convert_to_single_channel(tensor2pil(image), channel)

        return (pil2tensor(image), )

    def convert_to_single_channel(self, image, channel='red'):

        # Convert to RGB mode to access individual channels
        image = image.convert('RGB')

        # Extract the desired channel and convert to greyscale
        if channel == 'red':
            channel_img = image.split()[0].convert('L')
        elif channel == 'green':
            channel_img = image.split()[1].convert('L')
        elif channel == 'blue':
            channel_img = image.split()[2].convert('L')
        else:
            raise ValueError(
                "Invalid channel option. Please choose 'red', 'green', or 'blue'.")

        # Convert the greyscale channel back to RGB mode
        channel_img = Image.merge(
            'RGB', (channel_img, channel_img, channel_img))

        return channel_img

# IMAGES TO RGB

class WAS_Images_To_RGB:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
            },
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_to_rgb"

    CATEGORY = "WAS Suite/Image"

    def image_to_rgb(self, images):

        if len(images) > 1:
            tensors = []
            for image in images:
                tensors.append(pil2tensor(tensor2pil(image).convert('RGB')))
            tensors = torch.cat(tensors, dim=0)
            return (tensors, )
        else:
            return (pil2tensor(tensor2pil(images).convert("RGB")), )

# IMAGES TO LINEAR

class WAS_Images_To_Linear:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_to_linear"

    CATEGORY = "WAS Suite/Image"

    def image_to_linear(self, images):

        if len(images) > 1:
            tensors = []
            for image in images:
                tensors.append(pil2tensor(tensor2pil(image).convert('L')))
            tensors = torch.cat(tensors, dim=0)
            return (tensors, )
        else:
            return (pil2tensor(tensor2pil(images).convert("L")), )


# IMAGE MERGE RGB CHANNELS

class WAS_Image_RGB_Merge:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "red_channel": ("IMAGE",),
                "green_channel": ("IMAGE",),
                "blue_channel": ("IMAGE",),
            },
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "merge_channels"

    CATEGORY = "WAS Suite/Image/Process"

    def merge_channels(self, red_channel, green_channel, blue_channel):

        # Apply mix rgb channels
        image = self.mix_rgb_channels(tensor2pil(red_channel).convert('L'), tensor2pil(
            green_channel).convert('L'), tensor2pil(blue_channel).convert('L'))

        return (pil2tensor(image), )

    def mix_rgb_channels(self, red, green, blue):
        # Create an empty image with the same size as the channels
        width, height = red.size
        merged_img = Image.new('RGB', (width, height))

        # Merge the channels into the new image
        merged_img = Image.merge('RGB', (red, green, blue))

        return merged_img

# IMAGE Ambient Occlusion

class WAS_Image_Ambient_Occlusion:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "depth_images": ("IMAGE",),
                "strength": ("FLOAT", {"min": 0.0, "max": 5.0, "default": 1.0, "step": 0.01}),
                "radius": ("FLOAT", {"min": 0.01, "max": 1024, "default": 30, "step": 0.01}),
                "ao_blur": ("FLOAT", {"min": 0.01, "max": 1024, "default": 2.5, "step": 0.01}),
                "specular_threshold": ("INT", {"min":0, "max": 255, "default": 25, "step": 1}),
                "enable_specular_masking": (["True", "False"],),
                "tile_size": ("INT", {"min": 1, "max": 512, "default": 1, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE","IMAGE","IMAGE")
    RETURN_NAMES = ("composited_images", "ssao_images", "specular_mask_images")
    FUNCTION = "ambient_occlusion"

    CATEGORY = "WAS Suite/Image/Filter"

    def ambient_occlusion(self, images, depth_images, strength, radius, ao_blur, specular_threshold, ena
ble_specular_masking, tile_size):
        enable_specular_masking = (enable_specular_masking == 'True')
        composited = []
        occlusions = []
        speculars = []
        for i, image in enumerate(images):
            cstr(f"Processing SSAO image {i+1}/{len(images)} ...").msg.print()
            composited_image, occlusion_image, specular_mask = self.create_ambient_occlusion(
                tensor2pil(image),
                tensor2pil(depth_images[(i if len(depth_images) >= i else -1)]),
                strength=strength,
                radius=radius,
                ao_blur=ao_blur,
                spec_threshold=specular_threshold,
                enable_specular_masking=enable_specular_masking,
                tile_size=tile_size
            )
            composited.append(pil2tensor(composited_image))
            occlusions.append(pil2tensor(occlusion_image))
            speculars.append(pil2tensor(specular_mask))

        composited = torch.cat(composited, dim=0)
        occlusions = torch.cat(occlusions, dim=0)
        speculars = torch.cat(speculars, dim=0)

        return ( composited, occlusions, speculars )

    def process_tile(self, tile_rgb, tile_depth, tile_x, tile_y, radius):
        tile_occlusion = calculate_ambient_occlusion_factor(tile_rgb, tile_depth, tile_rgb.shape[0], tile_rgb.
shape[1], radius)
        return tile_x, tile_y, tile_occlusion


    def create_ambient_occlusion(self, rgb_image, depth_image, strength=1.0, radius=30, ao_blur=5, spe
c_threshold=200, enable_specular_masking=False, tile_size=1):

        import concurrent.futures

        if depth_image.size != rgb_image.size:
            depth_image = depth_image.resize(rgb_image.size)
        rgb_normalized = np.array(rgb_image, dtype=np.float32) / 255.0
        depth_normalized = np.array(depth_image, dtype=np.float32) / 255.0

        height, width, _ = rgb_normalized.shape

        if tile_size <= 1:
            print("Processing single-threaded AO (highest quality) ...")
            occlusion_array = calculate_ambient_occlusion_factor(rgb_normalized, depth_normalized, heigh
t, width, radius)
        else:
            tile_size = ((tile_size if tile_size <= 8 else 8) if tile_size > 1 else 1)
            num_tiles_x = (width - 1) // tile_size + 1
            num_tiles_y = (height - 1) // tile_size + 1

            occlusion_array = np.zeros((height, width), dtype=np.uint8)

            with concurrent.futures.ThreadPoolExecutor() as executor:
                futures = []
            with tqdm(total=num_tiles_y * num_tiles_x) as pbar:
                for tile_y in range(num_tiles_y):
                    for tile_x in range(num_tiles_x):
                        tile_left = tile_x * tile_size
                        tile_upper = tile_y * tile_size
                        tile_right = min(tile_left + tile_size, width)
                        tile_lower = min(tile_upper + tile_size, height)

                        tile_rgb = rgb_normalized[tile_upper:tile_lower, tile_left:tile_right]
                        tile_depth = depth_normalized[tile_upper:tile_lower, tile_left:tile_right]

                        future = executor.submit(self.process_tile, tile_rgb, tile_depth, tile_x, tile_y, radius)
                        futures.append(future)

                for future in concurrent.futures.as_completed(futures):
                    tile_x, tile_y, tile_occlusion = future.result()
                    tile_left = tile_x * tile_size
                    tile_upper = tile_y * tile_size
                    tile_right = min(tile_left + tile_size, width)
                    tile_lower = min(tile_upper + tile_size, height)

                    occlusion_array[tile_upper:tile_lower, tile_left:tile_right] = tile_occlusion

                    pbar.update(1)

        occlusion_array = (occlusion_array * strength).clip(0, 255).astype(np.uint8)

        occlusion_image = Image.fromarray(occlusion_array, mode='L')
        occlusion_image = occlusion_image.filter(ImageFilter.GaussianBlur(radius=ao_blur))
        occlusion_image = occlusion_image.filter(ImageFilter.SMOOTH)
        occlusion_image = ImageChops.multiply(occlusion_image, ImageChops.multiply(occlusion_image,
occlusion_image))

        mask = rgb_image.convert('L')
        mask = mask.point(lambda x: x > spec_threshold, mode='1')
        mask = mask.convert("RGB")
        mask = mask.filter(ImageFilter.GaussianBlur(radius=2.5)).convert("L")

        if enable_specular_masking:
            occlusion_image = Image.composite(Image.new("L", rgb_image.size, 255), occlusion_image, ma
sk)
        occlsuion_result = ImageChops.multiply(rgb_image, occlusion_image.convert("RGB"))

        return occlsuion_result, occlusion_image, mask

# IMAGE Direct Occlusion

class WAS_Image_Direct_Occlusion:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "depth_images": ("IMAGE",),
                "strength": ("FLOAT", {"min": 0.0, "max": 5.0, "default": 1.0, "step": 0.01}),
                "radius": ("FLOAT", {"min": 0.01, "max": 1024, "default": 30, "step": 0.01}),
                "specular_threshold": ("INT", {"min":0, "max": 255, "default": 128, "step": 1}),
                "colored_occlusion": (["True", "False"],),
            },
        }

    RETURN_TYPES = ("IMAGE","IMAGE","IMAGE", "IMAGE")
    RETURN_NAMES = ("composited_images", "ssdo_images", "ssdo_image_masks", "light_source_imag
e_masks")
    FUNCTION = "direct_occlusion"

    CATEGORY = "WAS Suite/Image/Filter"

    def direct_occlusion(self, images, depth_images, strength, radius, specular_threshold, colored_occlu
sion):

        composited = []
        occlusions = []
        occlusion_masks = []
        light_sources = []
        for i, image in enumerate(images):
            cstr(f"Processing SSDO image {i+1}/{len(images)} ...").msg.print()
            composited_image, occlusion_image, occlusion_mask, light_source = self.create_direct_occlusi
on(
                tensor2pil(image),
                tensor2pil(depth_images[(i if len(depth_images) >= i else -1)]),
                strength=strength,
                radius=radius,
                threshold=specular_threshold,
                colored=True
            )
            composited.append(pil2tensor(composited_image))
            occlusions.append(pil2tensor(occlusion_image))
            occlusion_masks.append(pil2tensor(occlusion_mask))
            light_sources.append(pil2tensor(light_source))

        composited = torch.cat(composited, dim=0)
        occlusions = torch.cat(occlusions, dim=0)
        occlusion_masks = torch.cat(occlusion_masks, dim=0)
        light_sources = torch.cat(light_sources, dim=0)

        return ( composited, occlusions, occlusion_masks, light_sources )

    def find_light_source(self, rgb_normalized, threshold):
        from skimage.measure import regionprops
        from skimage import measure
        rgb_uint8 = (rgb_normalized * 255).astype(np.uint8)
        rgb_to_grey = Image.fromarray(rgb_uint8, mode="RGB")
        dominant = self.dominant_region(rgb_to_grey, threshold)
        grayscale_image = np.array(dominant.convert("L"), dtype=np.float32) / 255.0
        regions = measure.label(grayscale_image > 0)

        if np.max(regions) > 0:
            region_sums = measure.regionprops(regions, intensity_image=grayscale_image)
            brightest_region = max(region_sums, key=lambda r: r.mean_intensity)
            light_y, light_x = brightest_region.centroid
            light_mask = (regions == brightest_region.label).astype(np.uint8)
            light_mask_cluster = light_mask
        else:
            light_x, light_y = np.nan, np.nan
            light_mask_cluster = np.zeros_like(dominant, dtype=np.uint8)
        return light_mask_cluster, light_x, light_y


    def dominant_region(self, image, threshold=128):
        from scipy.ndimage import label
        image = ImageOps.invert(image.convert("L"))
        binary_image = image.point(lambda x: 255 if x > threshold else 0, mode="1")
        l, n = label(np.array(binary_image))
        sizes = np.bincount(l.flatten())
        dominant = 0
        try:
            dominant = np.argmax(sizes[1:]) + 1
        except ValueError:
            pass
        dominant_region_mask = (l == dominant).astype(np.uint8) * 255
        result = Image.fromarray(dominant_region_mask, mode="L")
        return result.convert("RGB")

    def create_direct_occlusion(self, rgb_image, depth_image, strength=1.0, radius=10, threshold=200, c
olored=False):
        rgb_normalized = np.array(rgb_image, dtype=np.float32) / 255.0
        depth_normalized = np.array(depth_image, dtype=np.float32) / 255.0
        height, width, _ = rgb_normalized.shape
        light_mask, light_x, light_y = self.find_light_source(rgb_normalized, threshold)
        occlusion_array = calculate_direct_occlusion_factor(rgb_normalized, depth_normalized, height, wi
dth, radius)
        #occlusion_scaled = (occlusion_array / np.max(occlusion_array) * 255).astype(np.uint8)
        occlusion_scaled = ((occlusion_array - np.min(occlusion_array)) / (np.max(occlusion_array) - np.m
in(occlusion_array)) * 255).astype(np.uint8)
        occlusion_image = Image.fromarray(occlusion_scaled, mode="L")
        occlusion_image = occlusion_image.filter(ImageFilter.GaussianBlur(radius=0.5))
        occlusion_image = occlusion_image.filter(ImageFilter.SMOOTH_MORE)

        if colored:
            occlusion_result = Image.composite(
                Image.new("RGB", rgb_image.size, (0, 0, 0)),
                rgb_image,
                occlusion_image
            )
            occlusion_result = ImageOps.autocontrast(occlusion_result, cutoff=(0, strength))
        else:
            occlusion_result = Image.blend(occlusion_image, occlusion_image, strength)

        light_image = ImageOps.invert(Image.fromarray(light_mask * 255, mode="L"))

        direct_occlusion_image = ImageChops.screen(rgb_image, occlusion_result.convert("RGB"))

        return direct_occlusion_image, occlusion_result, occlusion_image, light_image

# EXPORT API

class WAS_Export_API:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "save_prompt_api": (["true","true"],),
                "output_path": ("STRING", {"default": "./ComfyUI/output/", "multiline": False}),
                "filename_prefix": ("STRING", {"default": "ComfyUI_Prompt"}),
                "filename_delimiter": ("STRING", {"default":"_"}),
                "filename_number_padding": ("INT", {"default":4, "min":2, "max":9, "step":1}),
                "parse_text_tokens": ("BOOLEAN", {"default": False})
            },
            "hidden": {
                "prompt": "PROMPT"
            }
        }

    OUTPUT_NODE = True
    RETURN_TYPES = ()
    FUNCTION = "export_api"

    CATEGORY = "WAS Suite/Debug"

    def export_api(self, output_path=None, filename_prefix="ComfyUI", filename_number_padding=4,
                    filename_delimiter='_', prompt=None, save_prompt_api="true", parse_text_tokens=False):
        delimiter = filename_delimiter
        number_padding = filename_number_padding if filename_number_padding > 1 else 4

        tokens = TextTokens()

        if output_path in [None, '', "none", "."]:
            output_path = comfy_paths.output_directory
        else:
            output_path = tokens.parseTokens(output_path)

        pattern = f"{re.escape(filename_prefix)}{re.escape(filename_delimiter)}(\\d{{{number_paddin
g}}})"
        existing_counters = [
            int(re.search(pattern, filename).group(1))
            for filename in os.listdir(output_path)
            if re.match(pattern, filename)
        ]
        existing_counters.sort(reverse=True)

        if existing_counters:
            counter = existing_counters[0] + 1
        else:
            counter = 1

        file = f"{filename_prefix}{filename_delimiter}{counter:0{number_padding}}.json"
        output_file = os.path.abspath(os.path.join(output_path, file))

        if prompt:

            if parse_text_tokens:
                prompt = self.parse_prompt(prompt, tokens, keys_to_parse)

            prompt_json = json.dumps(prompt, indent=4)
            cstr("Prompt API JSON").msg.print()
            print(prompt_json)

            if save_prompt_api == "true":

                with open(output_file, 'w') as f:
                    f.write(prompt_json)

                cstr(f"Output file path: {output_file}").msg.print()

        return {"ui": {"string": prompt_json}}

    def parse_prompt(self, obj, tokens, keys_to_parse):
        if isinstance(obj, dict):
            return {
                key: self.parse_prompt(value, tokens, keys_to_parse)
                if key in keys_to_parse else value
                for key, value in obj.items()
            }
        elif isinstance(obj, list):
            return [self.parse_prompt(element, tokens, keys_to_parse) for element in obj]
        elif isinstance(obj, str):
            return tokens.parseTokens(obj)
        else:
            return obj


# Image Save (NSP Compatible)
# Originally From ComfyUI/nodes.py

class WAS_Image_Save:
    def __init__(self):
        self.output_dir = comfy_paths.output_directory
        self.type = 'output'
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE", ),
                "output_path": ("STRING", {"default": '[time(%Y-%m-%d)]', "multiline": False}),
                "filename_prefix": ("STRING", {"default": "ComfyUI"}),
                "filename_delimiter": ("STRING", {"default":"_"}),
                "filename_number_padding": ("INT", {"default":4, "min":1, "max":9, "step":1}),
                "filename_number_start": (["false", "true"],),
                "extension": (['png', 'jpg', 'jpeg', 'gif', 'tiff', 'webp', 'bmp'], ),
                "dpi": ("INT", {"default": 300, "min": 1, "max": 2400, "step": 1}),
                "quality": ("INT", {"default": 100, "min": 1, "max": 100, "step": 1}),
                "optimize_image": (["true", "false"],),
                "lossless_webp": (["false", "true"],),
                "overwrite_mode": (["false", "prefix_as_filename"],),
                "show_history": (["false", "true"],),
                "show_history_by_prefix": (["true", "false"],),
                "embed_workflow": (["true", "false"],),
                "show_previews": (["true", "false"],),
            },
            "hidden": {
                "prompt": "PROMPT", "extra_pnginfo": "EXTRA_PNGINFO"
            },
        }
    RETURN_TYPES = ("IMAGE", "STRING",)
    RETURN_NAMES = ("images", "files",)

    FUNCTION = "was_save_images"

    OUTPUT_NODE = True

    CATEGORY = "WAS Suite/IO"

    def was_save_images(self, images, output_path='', filename_prefix="ComfyUI", filename_delimiter
='_',
                        extension='png', dpi=96, quality=100, optimize_image="true", lossless_webp="false", pr
ompt=None, extra_pnginfo=None,
                        overwrite_mode='false', filename_number_padding=4, filename_number_start='false',
                        show_history='false', show_history_by_prefix="true", embed_workflow="true",
                        show_previews="true"):

        delimiter = filename_delimiter
        number_padding = filename_number_padding
        lossless_webp = (lossless_webp == "true")
        optimize_image = (optimize_image == "true")

        # Define token system
        tokens = TextTokens()

        original_output = self.output_dir
        # Parse prefix tokens
        filename_prefix = tokens.parseTokens(filename_prefix)

        # Setup output path
        if output_path in [None, '', "none", "."]:
            output_path = self.output_dir
        else:
            output_path = tokens.parseTokens(output_path)
        if not os.path.isabs(output_path):
            output_path = os.path.join(self.output_dir, output_path)
        base_output = os.path.basename(output_path)
        if output_path.endswith("ComfyUI/output") or output_path.endswith("ComfyUI\output"):
            base_output = ""

        # Check output destination
        if output_path.strip() != '':
            if not os.path.isabs(output_path):
                output_path = os.path.join(comfy_paths.output_directory, output_path)
            if not os.path.exists(output_path.strip()):
                cstr(f'The path `{output_path.strip()}` specified doesn\'t exist! Creating directory.').warning.pr
int()
                os.makedirs(output_path, exist_ok=True)

        # Find existing counter values
        if filename_number_start == 'true':
            pattern = f"(\\d+){re.escape(delimiter)}{re.escape(filename_prefix)}"
        else:
            pattern = f"{re.escape(filename_prefix)}{re.escape(delimiter)}(\\d+)"
        existing_counters = [
            int(re.search(pattern, filename).group(1))
            for filename in os.listdir(output_path)
            if re.match(pattern, os.path.basename(filename))
        ]
        existing_counters.sort(reverse=True)

        # Set initial counter value
        if existing_counters:
            counter = existing_counters[0] + 1
        else:
            counter = 1

        # Set initial counter value
        if existing_counters:
            counter = existing_counters[0] + 1
        else:
            counter = 1

        # Set Extension
        file_extension = '.' + extension
        if file_extension not in ALLOWED_EXT:
            cstr(f"The extension `{extension}` is not valid. The valid formats are: {', '.join(sorted(ALLOWED_
EXT))}").error.print()
            file_extension = "png"

        results = list()
        output_files = list()
        for image in images:
            i = 255. * image.cpu().numpy()
            img = Image.fromarray(np.clip(i, 0, 255).astype(np.uint8))

            # Delegate metadata/pnginfo
            if extension == 'webp':
                img_exif = img.getexif()
                if embed_workflow == 'true':
                    workflow_metadata = ''
                    prompt_str = ''
                    if prompt is not None:
                        prompt_str = json.dumps(prompt)
                        img_exif[0x010f] = "Prompt:" + prompt_str
                    if extra_pnginfo is not None:
                        for x in extra_pnginfo:
                            workflow_metadata += json.dumps(extra_pnginfo[x])
                    img_exif[0x010e] = "Workflow:" + workflow_metadata
                exif_data = img_exif.tobytes()
            else:
                metadata = PngInfo()
                if embed_workflow == 'true':
                    if prompt is not None:
                        metadata.add_text("prompt", json.dumps(prompt))
                    if extra_pnginfo is not None:
                        for x in extra_pnginfo:
                            metadata.add_text(x, json.dumps(extra_pnginfo[x]))
                exif_data = metadata

            # Delegate the filename stuffs
            if overwrite_mode == 'prefix_as_filename':
                file = f"{filename_prefix}{file_extension}"
            else:
                if filename_number_start == 'true':
```

```
            file = f"{counter:0{number_padding}}{delimiter}{filename_prefix}{file_extension}"
        else:
            file = f"{filename_prefix}{delimiter}{counter:0{number_padding}}{file_extension}"
        if os.path.exists(os.path.join(output_path, file)):
            counter += 1

    # Save the images
    try:
        output_file = os.path.abspath(os.path.join(output_path, file))
        if extension in ["jpg", "jpeg"]:
            img.save(output_file,
                     quality=quality, optimize=optimize_image, dpi=(dpi, dpi))
        elif extension == 'webp':
            img.save(output_file,
                     quality=quality, lossless=lossless_webp, exif=exif_data)
        elif extension == 'png':
            img.save(output_file,
                     pnginfo=exif_data, optimize=optimize_image)
        elif extension == 'bmp':
            img.save(output_file)
        elif extension == 'tiff':
            img.save(output_file,
                     quality=quality, optimize=optimize_image)
        else:
            img.save(output_file,
                     pnginfo=exif_data, optimize=optimize_image)

        cstr(f"Image file saved to: {output_file}").msg.print()
        output_files.append(output_file)

        if show_history != 'true' and show_previews == 'true':
            subfolder = self.get_subfolder_path(output_file, original_output)
            results.append({
                "filename": file,
                "subfolder": subfolder,
                "type": self.type
            })

        # Update the output image history
        update_history_output_images(output_file)

    except OSError as e:
        cstr(f'Unable to save file to: {output_file}').error.print()
        print(e)
    except Exception as e:
        cstr('Unable to save file due to the to the following error:').error.print()
        print(e)

    if overwrite_mode == 'false':
        counter += 1

filtered_paths = []
if show_history == 'true' and show_previews == 'true':
    HDB = WASDatabase(WAS_HISTORY_DATABASE)
    conf = getSuiteConfig()
    if HDB.catExists("History") and HDB.keyExists("History", "Output_Images"):
        history_paths = HDB.get("History", "Output_Images")
    else:
```

```
                history_paths = None

            if history_paths:

                for image_path in history_paths:
                    image_subdir = self.get_subfolder_path(image_path, self.output_dir)
                    current_subdir = self.get_subfolder_path(output_file, self.output_dir)
                    if not os.path.exists(image_path):
                        continue
                    if show_history_by_prefix == 'true' and image_subdir != current_subdir:
                        continue
                    if show_history_by_prefix == 'true' and not os.path.basename(image_path).startswith(filena
me_prefix):
                        continue
                    filtered_paths.append(image_path)

                if conf.__contains__('history_display_limit'):
                    filtered_paths = filtered_paths[-conf['history_display_limit']:]

                filtered_paths.reverse()

        if filtered_paths:
            for image_path in filtered_paths:
                subfolder = self.get_subfolder_path(image_path, self.output_dir)
                image_data = {
                    "filename": os.path.basename(image_path),
                    "subfolder": subfolder,
                    "type": self.type
                }
                results.append(image_data)

        if show_previews == 'true':
            return {"ui": {"images": results, "files": output_files}, "result": (images, output_files,)}
        else:
            return {"ui": {"images": []}, "result": (images, output_files,)}

    def get_subfolder_path(self, image_path, output_path):
        output_parts = output_path.strip(os.sep).split(os.sep)
        image_parts = image_path.strip(os.sep).split(os.sep)
        common_parts = os.path.commonprefix([output_parts, image_parts])
        subfolder_parts = image_parts[len(common_parts):]
        subfolder_path = os.sep.join(subfolder_parts[:-1])
        return subfolder_path


# Image Send HTTP
# Sends images over http
class WAS_Image_Send_HTTP:
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "url": ("STRING", {"default": "example.com"}),
                "method_type": (["post", "put", "patch"], {"default": "post"}),
                "request_field_name": ("STRING", {"default": "image"}),
            },
            "optional": {
                "additional_request_headers": ("DICT",)
            }
        }

    RETURN_TYPES = ("INT", "STRING")
    RETURN_NAMES = ("status_code", "result_text")

    FUNCTION = "was_send_images_http"
    OUTPUT_NODE = True

    CATEGORY = "WAS Suite/IO"

    def was_send_images_http(self, images, url="example.com",
                             method_type="post",
                             request_field_name="image",
                             additional_request_headers=None):
        from io import BytesIO

        images_to_send = []
        for idx, image in enumerate(images):
            i = 255. * image.cpu().numpy()
            img = Image.fromarray(np.clip(i, 0, 255).astype(np.uint8))
            byte_io = BytesIO()
            img.save(byte_io, 'png')
            byte_io.seek(0)
            images_to_send.append(
                (request_field_name, (f"image_{idx}.png", byte_io, "image/png"))
            )
        request = requests.Request(url=url, method=method_type.upper(),
                                   headers=additional_request_headers,
                                   files=images_to_send)
        prepped = request.prepare()
        session = requests.Session()

        response = session.send(prepped)
        return (response.status_code, response.text,)


# LOAD IMAGE NODE
class WAS_Load_Image:

    def __init__(self):
        self.input_dir = comfy_paths.input_directory
        self.HDB = WASDatabase(WAS_HISTORY_DATABASE)

    @classmethod
    def INPUT_TYPES(cls):
        return {
                "required": {
                    "image_path": ("STRING", {"default": './ComfyUI/input/example.png', "multiline": False}),
                    "RGBA": (["false","true"],),
                },
                "optional": {
                    "filename_text_extension": (["true", "false"],),
                }
            }

    RETURN_TYPES = ("IMAGE", "MASK", TEXT_TYPE)
    RETURN_NAMES = ("image", "mask", "filename_text")
    FUNCTION = "load_image"

    CATEGORY = "WAS Suite/IO"

    def load_image(self, image_path, RGBA='false', filename_text_extension="true"):

        RGBA = (RGBA == 'true')

        if image_path.startswith('http'):
            from io import BytesIO
            i = self.download_image(image_path)
            i = ImageOps.exif_transpose(i)
        else:
            try:
                i = Image.open(image_path)
                i = ImageOps.exif_transpose(i)
            except OSError:
                cstr(f"The image `{image_path.strip()}` specified doesn't exist!").error.print()
                i = Image.new(mode='RGB', size=(512, 512), color=(0, 0, 0))
        if not i:
            return

        # Update history
        update_history_images(image_path)

        image = i
        if not RGBA:
            image = image.convert('RGB')
        image = np.array(image).astype(np.float32) / 255.0
        image = torch.from_numpy(image)[None,]

        if 'A' in i.getbands():
            mask = np.array(i.getchannel('A')).astype(np.float32) / 255.0
            mask = 1. - torch.from_numpy(mask)
        else:
            mask = torch.zeros((64, 64), dtype=torch.float32, device="cpu")

        if filename_text_extension == "true":
            filename = os.path.basename(image_path)
        else:
            filename = os.path.splitext(os.path.basename(image_path))[0]

        return (image, mask, filename)

    def download_image(self, url):
        try:
            response = requests.get(url)
            response.raise_for_status()
            img = Image.open(BytesIO(response.content))
            return img
        except requests.exceptions.HTTPError as errh:
            cstr(f"HTTP Error: ({url}): {errh}").error.print()
        except requests.exceptions.ConnectionError as errc:
            cstr(f"Connection Error: ({url}): {errc}").error.print()
        except requests.exceptions.Timeout as errt:
            cstr(f"Timeout Error: ({url}): {errt}").error.print()
        except requests.exceptions.RequestException as err:
            cstr(f"Request Exception: ({url}): {err}").error.print()

    @classmethod
    def IS_CHANGED(cls, image_path):
        if image_path.startswith('http'):
            return float("NaN")
        m = hashlib.sha256()
        with open(image_path, 'rb') as f:
            m.update(f.read())
        return m.digest().hex()

# MASK BATCH TO MASK

class WAS_Mask_Batch_to_Single_Mask:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "masks": ("MASK",),
                "batch_number": ("INT", {"default": 0, "min": 0, "max": 64, "step": 1}),
            },
        }

    RETURN_TYPES = ("MASK",)
    FUNCTION = "mask_batch_to_mask"

    CATEGORY = "WAS Suite/Image/Masking"

    def mask_batch_to_mask(self, masks=[], batch_number=0):
        count = 0
        for _ in masks:
            if batch_number == count:
                tensor = masks[batch_number][0]
                return (tensor,)
            count += 1

        cstr(f"Batch number `{batch_number}` is not defined, returning last image").error.print()
        last_tensor = masks[-1][0]
        return (last_tensor,)

# TENSOR BATCH TO IMAGE

class WAS_Tensor_Batch_to_Image:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images_batch": ("IMAGE",),
                "batch_image_number": ("INT", {"default": 0, "min": 0, "max": 64, "step": 1}),
            },
        }
    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "tensor_batch_to_image"

    CATEGORY = "WAS Suite/Latent/Transform"

    def tensor_batch_to_image(self, images_batch=[], batch_image_number=0):

        count = 0
        for _ in images_batch:
            if batch_image_number == count:
                return (images_batch[batch_image_number].unsqueeze(0), )
            count = count+1

        cstr(f"Batch number `{batch_image_number}` is not defined, returning last image").error.print()
        return (images_batch[-1].unsqueeze(0), )


#! LATENT NODES

# IMAGE TO MASK

class WAS_Image_To_Mask:

    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
                "required": {
                    "images": ("IMAGE",),
                    "channel": (["alpha", "red", "green", "blue"], ),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "image_to_mask"

    def image_to_mask(self, images, channel):
        mask_images = []
        for image in images:

            image = tensor2pil(image).convert("RGBA")
            r, g, b, a = image.split()
            if channel == "red":
                channel_image = r
            elif channel == "green":
                channel_image = g
            elif channel == "blue":
                channel_image = b
            elif channel == "alpha":
                channel_image = a

            mask = torch.from_numpy(np.array(channel_image.convert("L")).astype(np.float32) / 255.0)
            mask_images.append(mask)

        return (torch.cat(mask_images, dim=0), )


# MASK TO IMAGE

class WAS_Mask_To_Image:

    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("IMAGES",)

    FUNCTION = "mask_to_image"

    def mask_to_image(self, masks):
        if masks.ndim == 4:
            # If input has shape [N, C, H, W]
            tensor = masks.permute(0, 2, 3, 1)
            tensor_rgb = torch.cat([tensor] * 3, dim=-1)
            return (tensor_rgb,)
        elif masks.ndim == 3:
            # If Input has shape [N, H, W]
            tensor = masks.unsqueeze(-1)
            tensor_rgb = torch.cat([tensor] * 3, dim=-1)
            return (tensor_rgb, )
        elif masks.ndim == 2:
            # If input has shape [H, W]
            tensor = masks.unsqueeze(0).unsqueeze(-1)
            tensor_rgb = torch.cat([tensor] * 3, dim=-1)
            return (tensor_rgb,)
        else:
            cstr("Invalid input shape. Expected [N, C, H, W] or [H, W].").error.print()
            return masks


# MASK CROP DOMINANT REGION

class WAS_Mask_Crop_Dominant_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                "required": {
                    "masks": ("MASK",),
                    "padding": ("INT", {"default": 24, "min": 0, "max": 4096, "step": 1}),
                }
            }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "crop_dominant_region"

    def crop_dominant_region(self, masks, padding=24):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_pil = Image.fromarray(np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.
uint8))
                region_mask = self.WT.Masking.crop_dominant_region(mask_pil, padding)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_pil = Image.fromarray(np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.ui
nt8))
            region_mask = self.WT.Masking.crop_dominant_region(mask_pil, padding)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK CROP MINORITY REGION

class WAS_Mask_Crop_Minority_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                "required": {
                    "masks": ("MASK",),
                    "padding": ("INT", {"default": 24, "min": 0, "max": 4096, "step": 1}),
                }
            }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "crop_minority_region"

    def crop_minority_region(self, masks, padding=24):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_pil = Image.fromarray(np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.
uint8))
                region_mask = self.WT.Masking.crop_minority_region(mask_pil, padding)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_pil = Image.fromarray(np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.ui
nt8))
            region_mask = self.WT.Masking.crop_minority_region(mask_pil, padding)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK CROP REGION

class WAS_Mask_Crop_Region:
    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "mask": ("MASK",),
                "padding": ("INT",{"default": 24, "min": 0, "max": 4096, "step": 1}),
                "region_type": (["dominant", "minority"],),
            }
        }

    RETURN_TYPES = ("MASK", "CROP_DATA", "INT", "INT", "INT", "INT", "INT", "INT")
    RETURN_NAMES = ("cropped_mask", "crop_data", "top_int", "left_int", "right_int", "bottom_int", "widt
h_int", "height_int")
    FUNCTION = "mask_crop_region"

    CATEGORY = "WAS Suite/Image/Masking"

    def mask_crop_region(self, mask, padding=24, region_type="dominant"):

        mask_pil = Image.fromarray(np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint
8))
        region_mask, crop_data = self.WT.Masking.crop_region(mask_pil, region_type, padding)
        region_tensor = pil2mask(ImageOps.invert(region_mask)).unsqueeze(0).unsqueeze(1)

        (width, height), (left, top, right, bottom) = crop_data

        return (region_tensor, crop_data, top, left, right, bottom, width, height)


# IMAGE PASTE CROP

class WAS_Mask_Paste_Region:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "mask": ("MASK",),
                "crop_mask": ("MASK",),
                "crop_data": ("CROP_DATA",),
                "crop_blending": ("FLOAT", {"default": 0.25, "min": 0.0, "max": 1.0, "step": 0.01}),
                "crop_sharpening": ("INT", {"default": 0, "min": 0, "max": 3, "step": 1}),
            }
        }

    RETURN_TYPES = ("MASK", "MASK")
    FUNCTION = "mask_paste_region"

    CATEGORY = "WAS Suite/Image/Masking"

    def mask_paste_region(self, mask, crop_mask, crop_data=None, crop_blending=0.25, crop_sharpeni
ng=0):

        if crop_data == False:
            cstr("No valid crop data found!").error.print()
            return( pil2mask(Image.new("L", (512, 512), 0)).unsqueeze(0).unsqueeze(1),
                    pil2mask(Image.new("L", (512, 512), 0)).unsqueeze(0).unsqueeze(1) )

        mask_pil = Image.fromarray(np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint
8))
        mask_crop_pil = Image.fromarray(np.clip(255. * crop_mask.cpu().numpy().squeeze(), 0, 255).asty
pe(np.uint8))

        result_mask, result_crop_mask = self.paste_image(mask_pil, mask_crop_pil, crop_data, crop_blend
ing, crop_sharpening)

        return (pil2mask(result_mask).unsqueeze(0).unsqueeze(1), pil2mask(result_crop_mask).unsqueez
e(0).unsqueeze(1))

    def paste_image(self, image, crop_image, crop_data, blend_amount=0.25, sharpen_amount=1):

        def lingrad(size, direction, white_ratio):
            image = Image.new('RGB', size)
            draw = ImageDraw.Draw(image)
            if direction == 'vertical':
                black_end = int(size[1] * (1 - white_ratio))
                range_start = 0
                range_end = size[1]
                range_step = 1
                for y in range(range_start, range_end, range_step):
                    color_ratio = y / size[1]
                    if y <= black_end:
                        color = (0, 0, 0)
                    else:
                        color_value = int(((y - black_end) / (size[1] - black_end)) * 255)
                        color = (color_value, color_value, color_value)
                    draw.line([(0, y), (size[0], y)], fill=color)
            elif direction == 'horizontal':
                black_end = int(size[0] * (1 - white_ratio))
                range_start = 0
                range_end = size[0]
                range_step = 1
                for x in range(range_start, range_end, range_step):
                    color_ratio = x / size[0]
                    if x <= black_end:
                        color = (0, 0, 0)
                    else:
                        color_value = int(((x - black_end) / (size[0] - black_end)) * 255)
                        color = (color_value, color_value, color_value)
                    draw.line([(x, 0), (x, size[1])], fill=color)

            return image.convert("L")

        crop_size, (left, top, right, bottom) = crop_data
        crop_image = crop_image.resize(crop_size)

        if sharpen_amount > 0:
            for _ in range(int(sharpen_amount)):
                crop_image = crop_image.filter(ImageFilter.SHARPEN)

        blended_image = Image.new('RGBA', image.size, (0, 0, 0, 255))
        blended_mask = Image.new('L', image.size, 0)  # Update to 'L' mode for MASK image
        crop_padded = Image.new('RGBA', image.size, (0, 0, 0, 0))
        blended_image.paste(image, (0, 0))
        crop_padded.paste(crop_image, (left, top))
        crop_mask = Image.new('L', crop_image.size, 0)

        if top > 0:
            gradient_image = ImageOps.flip(lingrad(crop_image.size, 'vertical', blend_amount))
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        if left > 0:
            gradient_image = ImageOps.mirror(lingrad(crop_image.size, 'horizontal', blend_amount))
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        if right < image.width:
            gradient_image = lingrad(crop_image.size, 'horizontal', blend_amount)
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        if bottom < image.height:
            gradient_image = lingrad(crop_image.size, 'vertical', blend_amount)
            crop_mask = ImageChops.screen(crop_mask, gradient_image)

        crop_mask = ImageOps.invert(crop_mask)
        blended_mask.paste(crop_mask, (left, top))
        blended_mask = blended_mask.convert("L")
        blended_image.paste(crop_padded, (0, 0), blended_mask)

        return (ImageOps.invert(blended_image.convert("RGB")).convert("L"), ImageOps.invert(blended_
mask.convert("RGB")).convert("L"))




# MASK DOMINANT REGION

class WAS_Mask_Dominant_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()
    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                        "threshold": ("INT", {"default":128, "min":0, "max":255, "step":1}),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "dominant_region"

    def dominant_region(self, masks, threshold=128):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_pil = Image.fromarray(np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.
uint8))
                region_mask = self.WT.Masking.dominant_region(mask_pil, threshold)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_pil = Image.fromarray(np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.ui
nt8))
            region_mask = self.WT.Masking.dominant_region(mask_pil, threshold)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK MINORITY REGION

class WAS_Mask_Minority_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                        "threshold": ("INT", {"default":128, "min":0, "max":255, "step":1}),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "minority_region"
    def minority_region(self, masks, threshold=128):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.minority_region(pil_image, threshold)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.minority_region(pil_image, threshold)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)



# MASK ARBITRARY REGION

class WAS_Mask_Arbitrary_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                        "size": ("INT", {"default":256, "min":1, "max":4096, "step":1}),
                        "threshold": ("INT", {"default":128, "min":0, "max":255, "step":1}),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "arbitrary_region"

    def arbitrary_region(self, masks, size=256, threshold=128):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.arbitrary_region(pil_image, size, threshold)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.arbitrary_region(pil_image, size, threshold)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)

# MASK SMOOTH REGION

class WAS_Mask_Smooth_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                        "sigma": ("FLOAT", {"default":5.0, "min":0.0, "max":128.0, "step":0.1}),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "smooth_region"

    def smooth_region(self, masks, sigma=128):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.smooth_region(pil_image, sigma)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.smooth_region(pil_image, sigma)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK ERODE REGION

class WAS_Mask_Erode_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
```

```
                    "masks": ("MASK",),
                    "iterations": ("INT", {"default":5, "min":1, "max":64, "step":1}),
                }
            }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "erode_region"

    def erode_region(self, masks, iterations=5):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.erode_region(pil_image, iterations)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.erode_region(pil_image, iterations)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)

# MASKS SUBTRACT

class WAS_Mask_Subtract:

    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
                "required": {
                    "masks_a": ("MASK",),
                    "masks_b": ("MASK",),
                }
            }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "subtract_masks"

    def subtract_masks(self, masks_a, masks_b):
        subtracted_masks = torch.clamp(masks_a - masks_b, 0, 255)
        return (subtracted_masks,)

# MASKS ADD
```

```
class WAS_Mask_Add:

    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks_a": ("MASK",),
                        "masks_b": ("MASK",),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "add_masks"

    def add_masks(self, masks_a, masks_b):
        if masks_a.ndim > 2 and masks_b.ndim > 2:
            added_masks = masks_a + masks_b
        else:
            added_masks = torch.clamp(masks_a.unsqueeze(1) + masks_b.unsqueeze(1), 0, 255)
            added_masks = added_masks.squeeze(1)
        return (added_masks,)

# MASKS ADD

class WAS_Mask_Invert:

    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "add_masks"

    def add_masks(self, masks):
        return (1. - masks,)

# MASK DILATE REGION

class WAS_Mask_Dilate_Region:
    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                        "iterations": ("INT", {"default":5, "min":1, "max":64, "step":1}),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "dilate_region"

    def dilate_region(self, masks, iterations=5):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.dilate_region(pil_image, iterations)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.dilate_region(pil_image, iterations)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK FILL REGION

class WAS_Mask_Fill_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)
    FUNCTION = "fill_region"

    def fill_region(self, masks):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.fill_region(pil_image)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.fill_region(pil_image)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK THRESHOLD

class WAS_Mask_Threshold_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                        "black_threshold": ("INT",{"default":75, "min":0, "max": 255, "step": 1}),
                        "white_threshold": ("INT",{"default":175, "min":0, "max": 255, "step": 1}),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "threshold_region"

    def threshold_region(self, masks, black_threshold=75, white_threshold=255):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.threshold_region(pil_image, black_threshold, white_threshol
d)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.threshold_region(pil_image, black_threshold, white_threshold)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK FLOOR REGION

class WAS_Mask_Floor_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "masks": ("MASK",),
            }
        }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "floor_region"

    def floor_region(self, masks):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.floor_region(pil_image)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.floor_region(pil_image)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK CEILING REGION

class WAS_Mask_Ceiling_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "masks": ("MASK",),
            }
        }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "ceiling_region"

    def ceiling_region(self, masks):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.ceiling_region(pil_image)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.ceiling_region(pil_image)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK GAUSSIAN REGION

class WAS_Mask_Gaussian_Region:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "masks": ("MASK",),
                "radius": ("FLOAT", {"default": 5.0, "min": 0.0, "max": 1024, "step": 0.1}),
            }
        }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)
    RETURN_NAMES = ("MASKS",)

    FUNCTION = "gaussian_region"

    def gaussian_region(self, masks, radius=5.0):
        if masks.ndim > 3:
            regions = []
            for mask in masks:
                mask_np = np.clip(255. * mask.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
                pil_image = Image.fromarray(mask_np, mode="L")
                region_mask = self.WT.Masking.gaussian_region(pil_image, radius)
                region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
                regions.append(region_tensor)
            regions_tensor = torch.cat(regions, dim=0)
            return (regions_tensor,)
        else:
            mask_np = np.clip(255. * masks.cpu().numpy().squeeze(), 0, 255).astype(np.uint8)
            pil_image = Image.fromarray(mask_np, mode="L")
            region_mask = self.WT.Masking.gaussian_region(pil_image, radius)
            region_tensor = pil2mask(region_mask).unsqueeze(0).unsqueeze(1)
            return (region_tensor,)


# MASK COMBINE

class WAS_Mask_Combine:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "mask_a": ("MASK",),
                        "mask_b": ("MASK",),
                    },
                    "optional": {
                        "mask_c": ("MASK",),
                        "mask_d": ("MASK",),
                        "mask_e": ("MASK",),
                        "mask_f": ("MASK",),
                    }
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)

    FUNCTION = "combine_masks"

    def combine_masks(self, mask_a, mask_b, mask_c=None, mask_d=None, mask_e=None, mask_f=No
ne):
        masks = [mask_a, mask_b]
        if mask_c:
            masks.append(mask_c)
        if mask_d:
            masks.append(mask_d)
        if mask_e:
            masks.append(mask_e)
        if mask_f:
            masks.append(mask_f)
        combined_mask = torch.sum(torch.stack(masks, dim=0), dim=0)
        combined_mask = torch.clamp(combined_mask, 0, 1)  # Ensure values are between 0 and 1
        return (combined_mask, )
```

```
class WAS_Mask_Combine_Batch:

    def __init__(self):
        self.WT = WAS_Tools_Class()

    @classmethod
    def INPUT_TYPES(cls):
        return {
                    "required": {
                        "masks": ("MASK",),
                    },
                }

    CATEGORY = "WAS Suite/Image/Masking"

    RETURN_TYPES = ("MASK",)

    FUNCTION = "combine_masks"

    def combine_masks(self, masks):
        combined_mask = torch.sum(torch.stack([mask.unsqueeze(0) for mask in masks], dim=0), dim=0)
        combined_mask = torch.clamp(combined_mask, 0, 1)  # Ensure values are between 0 and 1
        return (combined_mask, )

# LATENT UPSCALE NODE

class WAS_Latent_Upscale:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {"required": {"samples": ("LATENT",), "mode": (["area", "bicubic", "bilinear", "nearest"],),
                             "factor": ("FLOAT", {"default": 2.0, "min": 0.1, "max": 8.0, "step": 0.01}),
                             "align": (["true", "false"], )}}
    RETURN_TYPES = ("LATENT",)
    FUNCTION = "latent_upscale"

    CATEGORY = "WAS Suite/Latent/Transform"

    def latent_upscale(self, samples, mode, factor, align):
        valid_modes = ["area", "bicubic", "bilinear", "nearest"]
        if mode not in valid_modes:
            cstr(f"Invalid interpolation mode `{mode}` selected. Valid modes are: {', '.join(valid_modes)}").er
ror.print()
            return (s, )
        align = True if align == 'true' else False
        if not isinstance(factor, float) or factor <= 0:
            cstr(f"The input `factor` is `{factor}`, but should be a positive or negative float.").error.print()
            return (s, )
        s = samples.copy()
        shape = s['samples'].shape
        size = tuple(int(round(dim * factor)) for dim in shape[-2:])
        if mode in ['linear', 'bilinear', 'bicubic', 'trilinear']:
            s["samples"] = torch.nn.functional.interpolate(
                s['samples'], size=size, mode=mode, align_corners=align)
        else:
            s["samples"] = torch.nn.functional.interpolate(s['samples'], size=size, mode=mode)
        return (s,)

# LATENT NOISE INJECTION NODE


class WAS_Latent_Noise:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "samples": ("LATENT",),
                "noise_std": ("FLOAT", {"default": 0.1, "min": 0.0, "max": 1.0, "step": 0.01}),
            }
        }

    RETURN_TYPES = ("LATENT",)
    FUNCTION = "inject_noise"

    CATEGORY = "WAS Suite/Latent/Generate"

    def inject_noise(self, samples, noise_std):
        s = samples.copy()
        noise = torch.randn_like(s["samples"]) * noise_std
        s["samples"] = s["samples"] + noise
        return (s,)



# MIDAS DEPTH APPROXIMATION NODE

class MiDaS_Model_Loader:
    def __init__(self):
        self.midas_dir = os.path.join(MODELS_DIR, 'midas')

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "midas_model": (["DPT_Large", "DPT_Hybrid"],),
            },
        }

    RETURN_TYPES = ("MIDAS_MODEL",)
    RETURN_NAMES = ("midas_model",)
    FUNCTION = "load_midas_model"

    CATEGORY = "WAS Suite/Loaders"

    def load_midas_model(self, midas_model):

        global MIDAS_INSTALLED

        if not MIDAS_INSTALLED:
            self.install_midas()
        if midas_model == 'DPT_Large':
            model_name = 'dpt_large_384.pt'
        elif midas_model == 'DPT_Hybrid':
            model_name = 'dpt_hybrid_384.pt'
        else:
            model_name = 'dpt_large_384.pt'

        model_path = os.path.join(self.midas_dir, 'checkpoints'+os.sep+model_name)

        torch.hub.set_dir(self.midas_dir)
        if os.path.exists(model_path):
            cstr(f"Loading MiDaS Model from `{model_path}`").msg.print()
            midas_type = model_path
        else:
            cstr("Downloading and loading MiDaS Model...").msg.print()
        midas = torch.hub.load("intel-isl/MiDaS", midas_model, trust_repo=True)
        device = torch.device("cpu")

        cstr(f"MiDaS is using passive device `{device}` until in use.").msg.print()

        midas.to(device)
        midas_transforms = torch.hub.load("intel-isl/MiDaS", "transforms")
        transform = midas_transforms.dpt_transform

        return ( (midas, transform), )

    def install_midas(self):
        global MIDAS_INSTALLED
        if 'timm' not in packages():
            install_package("timm")
        MIDAS_INSTALLED = True


# MIDAS DEPTH APPROXIMATION NODE

class MiDaS_Depth_Approx:
    def __init__(self):
        self.midas_dir = os.path.join(MODELS_DIR, 'midas')

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "use_cpu": (["false", "true"],),
                "midas_type": (["DPT_Large", "DPT_Hybrid"],),
                "invert_depth": (["false", "true"],),
            },
            "optional": {
                "midas_model": ("MIDAS_MODEL",),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    RETURN_NAMES = ("images",)
    FUNCTION = "midas_approx"
    CATEGORY = "WAS Suite/Image/AI"

    def midas_approx(self, image, use_cpu, midas_type, invert_depth, midas_model=None):

        global MIDAS_INSTALLED

        if not MIDAS_INSTALLED:
            self.install_midas()

        import cv2 as cv

        if midas_model:

            midas = midas_model[0]
            transform = midas_model[1]
            device = torch.device("cuda") if torch.cuda.is_available() and use_cpu == 'false' else torch.devi
ce("cpu")
            cstr(f"MiDaS is using device: {device}").msg.print()
            midas.to(device).eval()

        else:

            if midas_model == 'DPT_Large':
                model_name = 'dpt_large_384.pt'
            elif midas_model == 'DPT_Hybrid':
                model_name = 'dpt_hybrid_384.pt'
            else:
                model_name = 'dpt_large_384.pt'

            model_path = os.path.join(self.midas_dir, 'checkpoints'+os.sep+model_name)

            torch.hub.set_dir(self.midas_dir)
            if os.path.exists(model_path):
                cstr(f"Loading MiDaS Model from `{model_path}`").msg.print()
                midas_type = model_path
            else:
                cstr("Downloading and loading MiDaS Model...").msg.print()
            midas = torch.hub.load("intel-isl/MiDaS", midas_type, trust_repo=True)

            cstr(f"MiDaS is using device: {device}").msg.print()

            midas.to(device).eval()
            midas_transforms = torch.hub.load("intel-isl/MiDaS", "transforms")

            transform = midas_transforms.dpt_transform

        tensor_images = []
        for i, img in enumerate(image):

            img = np.array(tensor2pil(img))

            img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
            input_batch = transform(img).to(device)

            cstr(f"Approximating depth for image {i+1}/{len(image)}").msg.print()

            with torch.no_grad():
                prediction = midas(input_batch)
                prediction = torch.nn.functional.interpolate(
                    prediction.unsqueeze(1),
                    size=img.shape[:2],
                    mode="bicubic",
                    align_corners=False,
                ).squeeze()


            # Normalize and convert to uint8
            min_val = torch.min(prediction)
            max_val = torch.max(prediction)
            prediction = (prediction - min_val) / (max_val - min_val)
            prediction = (prediction * 255).clamp(0, 255).round().cpu().numpy().astype(np.uint8)

            depth = Image.fromarray(prediction)

            # Invert depth map
            if invert_depth == 'true':
                depth = ImageOps.invert(depth)

            tensor_images.append(pil2tensor(depth.convert("RGB")))

        tensor_images = torch.cat(tensor_images, dim=0)
        if not midas_model:
            del midas, device, midas_transforms
        del midas, transform, img, input_batch, prediction

        return (tensor_images, )

    def install_midas(self):
        global MIDAS_INSTALLED
        if 'timm' not in packages():
            install_package("timm")
        MIDAS_INSTALLED = True

# MIDAS REMOVE BACKGROUND/FOREGROUND NODE


class MiDaS_Background_Foreground_Removal:
    def __init__(self):
        self.midas_dir = os.path.join(MODELS_DIR, 'midas')

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "use_cpu": (["false", "true"],),
                "midas_model": (["DPT_Large", "DPT_Hybrid", "DPT_Small"],),
                "remove": (["background", "foregroud"],),
                "threshold": (["false", "true"],),
                "threshold_low": ("FLOAT", {"default": 10, "min": 0, "max": 255, "step": 1}),
                "threshold_mid": ("FLOAT", {"default": 200, "min": 0, "max": 255, "step": 1}),
                "threshold_high": ("FLOAT", {"default": 210, "min": 0, "max": 255, "step": 1}),
                "smoothing": ("FLOAT", {"default": 0.25, "min": 0.0, "max": 16.0, "step": 0.01}),
                "background_red": ("INT", {"default": 0, "min": 0, "max": 255, "step": 1}),
                "background_green": ("INT", {"default": 0, "min": 0, "max": 255, "step": 1}),
                "background_blue": ("INT", {"default": 0, "min": 0, "max": 255, "step": 1}),
            },
        }

    RETURN_TYPES = ("IMAGE", "IMAGE")
    FUNCTION = "midas_remove"

    CATEGORY = "WAS Suite/Image/AI"

    def midas_remove(self,
                     image,
                     midas_model,
                     use_cpu='false',
                     remove='background',
                     threshold='false',
                     threshold_low=0,
                     threshold_mid=127,
                     threshold_high=255,
                     smoothing=0.25,
                     background_red=0,
                     background_green=0,
                     background_blue=0):

        global MIDAS_INSTALLED

        if not MIDAS_INSTALLED:
            self.install_midas()

        import cv2 as cv

        # Convert the input image tensor to a numpy and PIL Image
        i = 255. * image.cpu().numpy().squeeze()
        img = i
        # Original image
        img_original = tensor2pil(image).convert('RGB')

        cstr("Downloading and loading MiDaS Model...").msg.print()
        torch.hub.set_dir(self.midas_dir)
        midas = torch.hub.load("intel-isl/MiDaS", midas_model, trust_repo=True)
        device = torch.device("cuda") if torch.cuda.is_available(
        ) and use_cpu == 'false' else torch.device("cpu")

        cstr(f"MiDaS is using device: {device}").msg.print()

        midas.to(device).eval()
        midas_transforms = torch.hub.load("intel-isl/MiDaS", "transforms")

        if midas_model == "DPT_Large" or midas_model == "DPT_Hybrid":
            transform = midas_transforms.dpt_transform
        else:
            transform = midas_transforms.small_transform

        img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
        input_batch = transform(img).to(device)

        cstr("Approximating depth from image.").msg.print()

        with torch.no_grad():
            prediction = midas(input_batch)
```

```
            prediction = torch.nn.functional.interpolate(
                prediction.unsqueeze(1),
                size=img.shape[:2],
                mode="bicubic",
                align_corners=False,
            ).squeeze()

        # Invert depth map
        if remove == 'foreground':
            depth = (255 - prediction.cpu().numpy().astype(np.uint8))
            depth = depth.astype(np.float32)
        else:
            depth = prediction.cpu().numpy().astype(np.float32)
        depth = depth * 255 / (np.max(depth)) / 255
        depth = Image.fromarray(np.uint8(depth * 255))

        # Threshold depth mask
        if threshold == 'true':
            levels = self.AdjustLevels(
                threshold_low, threshold_mid, threshold_high)
            depth = levels.adjust(depth.convert('RGB')).convert('L')
        if smoothing > 0:
            depth = depth.filter(ImageFilter.GaussianBlur(radius=smoothing))
        depth = depth.resize(img_original.size).convert('L')

        # Validate background color arguments
        background_red = int(background_red) if isinstance(
            background_red, (int, float)) else 0
        background_green = int(background_green) if isinstance(
            background_green, (int, float)) else 0
        background_blue = int(background_blue) if isinstance(
            background_blue, (int, float)) else 0

        # Create background color tuple
        background_color = (background_red, background_green, background_blue)

        # Create background image
        background = Image.new(
            mode="RGB", size=img_original.size, color=background_color)

        # Composite final image
        result_img = Image.composite(img_original, background, depth)

        del midas, device, midas_transforms
        del transform, img, img_original, input_batch, prediction

        return (pil2tensor(result_img), pil2tensor(depth.convert('RGB')))

    class AdjustLevels:
        def __init__(self, min_level, mid_level, max_level):
            self.min_level = min_level
            self.mid_level = mid_level
            self.max_level = max_level

        def adjust(self, im):
            # load the image

            # convert the image to a numpy array
            im_arr = np.array(im)

            # apply the min level adjustment
            im_arr[im_arr < self.min_level] = self.min_level

            # apply the mid level adjustment
            im_arr = (im_arr - self.min_level) * \
                (255 / (self.max_level - self.min_level))
            im_arr[im_arr < 0] = 0
            im_arr[im_arr > 255] = 255
            im_arr = im_arr.astype(np.uint8)

            # apply the max level adjustment
            im = Image.fromarray(im_arr)
            im = ImageOps.autocontrast(im, cutoff=self.max_level)

            return im

    def install_midas(self):
        global MIDAS_INSTALLED
        if 'timm' not in packages():
            install_package("timm")
        MIDAS_INSTALLED = True


#! CONDITIONING NODES


# NSP CLIPTextEncode NODE

class WAS_NSP_CLIPTextEncoder:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "mode": (["Noodle Soup Prompts", "Wildcards"],),
                "noodle_key": ("STRING", {"default": '__', "multiline": False}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
                "text": ("STRING", {"multiline": True}),
                "clip": ("CLIP",),
            }
        }

    OUTPUT_NODE = True
    RETURN_TYPES = ("CONDITIONING", TEXT_TYPE, TEXT_TYPE)
    RETURN_NAMES = ("conditioning", "parsed_text", "raw_text")
    FUNCTION = "nsp_encode"

    CATEGORY = "WAS Suite/Conditioning"

    def nsp_encode(self, clip, text, mode="Noodle Soup Prompts", noodle_key='__', seed=0):

        if mode == "Noodle Soup Prompts":
            new_text = nsp_parse(text, seed, noodle_key)
        else:
            new_text = replace_wildcards(text, (None if seed == 0 else seed), noodle_key)

        new_text = parse_dynamic_prompt(new_text, seed)
        new_text, text_vars = parse_prompt_vars(new_text)
        cstr(f"CLIPTextEncode Prased Prompt:\n {new_text}").msg.print()
        CLIPTextEncode = nodes.CLIPTextEncode()
        encoded = CLIPTextEncode.encode(clip=clip, text=new_text)

        return (encoded[0], new_text, text, { "ui": { "string": new_text } })


#! SAMPLING NODES

# KSAMPLER

class WAS_KSampler:
    @classmethod
    def INPUT_TYPES(cls):
        return {"required":

                {"model": ("MODEL", ),
                 "seed": ("SEED", ),
                 "steps": ("INT", {"default": 20, "min": 1, "max": 10000}),
                 "cfg": ("FLOAT", {"default": 8.0, "min": 0.0, "max": 100.0}),
                 "sampler_name": (comfy.samplers.KSampler.SAMPLERS, ),
                 "scheduler": (comfy.samplers.KSampler.SCHEDULERS, ),
                 "positive": ("CONDITIONING", ),
                 "negative": ("CONDITIONING", ),
                 "latent_image": ("LATENT", ),
                 "denoise": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 1.0, "step": 0.01}),
                 }}

    RETURN_TYPES = ("LATENT",)
    FUNCTION = "sample"

    CATEGORY = "WAS Suite/Sampling"

    def sample(self, model, seed, steps, cfg, sampler_name, scheduler, positive, negative, latent_image,
denoise=1.0):
        return nodes.common_ksampler(model, seed['seed'], steps, cfg, sampler_name, scheduler, positi
ve, negative, latent_image, denoise=denoise)

# KSampler Cycle

class WAS_KSampler_Cycle:
    @classmethod
    def INPUT_TYPES(s):
        return {
                "required": {
                    "model": ("MODEL",),
                    "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
                    "steps": ("INT", {"default": 20, "min": 1, "max": 10000}),
                    "cfg": ("FLOAT", {"default": 8.0, "min": 0.0, "max": 100.0}),
                    "sampler_name": (comfy.samplers.KSampler.SAMPLERS, ),
                    "scheduler": (comfy.samplers.KSampler.SCHEDULERS, ),
                    "positive": ("CONDITIONING", ),
                    "negative": ("CONDITIONING", ),
                    "latent_image": ("LATENT", ),
                    "tiled_vae": (["disable", "enable"], ),
                    "latent_upscale": (["disable","nearest-exact", "bilinear", "area", "bicubic", "bislerp"],),
                    "upscale_factor": ("FLOAT", {"default":2.0, "min": 0.1, "max": 8.0, "step": 0.1}),
                    "upscale_cycles": ("INT", {"default": 2, "min": 2, "max": 12, "step": 1}),
                    "starting_denoise": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 1.0, "step": 0.01}),
                    "cycle_denoise": ("FLOAT", {"default": 0.5, "min": 0.0, "max": 1.0, "step": 0.01}),
                    "scale_denoise": (["enable", "disable"],),
                    "scale_sampling": (["bilinear", "bicubic", "nearest", "lanczos"],),
                    "vae": ("VAE",),
                },
                "optional": {
                    "secondary_model": ("MODEL",),
                    "secondary_start_cycle": ("INT", {"default": 2, "min": 2, "max": 16, "step": 1}),
                    "upscale_model": ("UPSCALE_MODEL",),
                    "processor_model": ("UPSCALE_MODEL",),
                    "pos_additive": ("CONDITIONING",),
                    "neg_additive": ("CONDITIONING",),
                    "pos_add_mode": (["increment", "decrement"],),
                    "pos_add_strength": ("FLOAT", {"default": 0.25, "min": 0.01, "max": 1.0, "step": 0.01}),
                    "pos_add_strength_scaling": (["enable", "disable"],),
                    "pos_add_strength_cutoff": ("FLOAT", {"default": 2.0, "min": 0.01, "max": 10.0, "step": 0.0
1}),
                    "neg_add_mode": (["increment", "decrement"],),
                    "neg_add_strength": ("FLOAT", {"default": 0.25, "min": 0.01, "max": 1.0, "step": 0.01}),
                    "neg_add_strength_scaling": (["enable", "disable"],),
                    "neg_add_strength_cutoff": ("FLOAT", {"default": 2.0, "min": 0.01, "max": 10.0, "step": 0.0
1}),
                    "sharpen_strength": ("FLOAT", {"default": 0.0, "min": 0.0, "max": 10.0, "step": 0.01}),
                    "sharpen_radius": ("INT", {"default": 2, "min": 1, "max": 12, "step": 1}),
                    "steps_scaling": (["enable", "disable"],),
                    "steps_control": (["decrement", "increment"],),
                    "steps_scaling_value": ("INT", {"default": 10, "min": 1, "max": 20, "step": 1}),
                    "steps_cutoff": ("INT", {"default": 20, "min": 4, "max": 1000, "step": 1}),
                    "denoise_cutoff": ("FLOAT", {"default": 0.25, "min": 0.01, "max": 1.0, "step": 0.01}),
                }
            }

    RETURN_TYPES = ("LATENT",)
    RETURN_NAMES =  ("latent(s)",)
    FUNCTION = "sample"

    CATEGORY = "WAS Suite/Sampling"

    def sample(self, model, seed, steps, cfg, sampler_name, scheduler, positive, negative, latent_image,
tiled_vae, latent_upscale, upscale_factor,
                upscale_cycles, starting_denoise, cycle_denoise, scale_denoise, scale_sampling, vae, second
ary_model=None, secondary_start_cycle=None,
                pos_additive=None, pos_add_mode=None, pos_add_strength=None, pos_add_strength_scalin
g=None, pos_add_strength_cutoff=None,
                neg_additive=None, neg_add_mode=None, neg_add_strength=None, neg_add_strength_scali
ng=None, neg_add_strength_cutoff=None,
                upscale_model=None, processor_model=None, sharpen_strength=0, sharpen_radius=2, step
s_scaling=None, steps_control=None,
                steps_scaling_value=None, steps_cutoff=None, denoise_cutoff=0.25):

        upscale_steps = upscale_cycles
        division_factor = upscale_steps if steps >= upscale_steps else steps
        current_upscale_factor = upscale_factor ** (1 / (division_factor - 1))
```

```
tiled_vae = (tiled_vae == "enable")
scale_denoise = (scale_denoise == "enable")
pos_add_strength_scaling = (pos_add_strength_scaling == "enable")
neg_add_strength_scaling = (neg_add_strength_scaling == "enable")
steps_scaling = (steps_scaling == "enable")
run_model = model
secondary_switched = False

for i in range(division_factor):

    cstr(f"Cycle Pass {i+1}/{division_factor}").msg.print()

    if scale_denoise:
        denoise = (
            ( round(cycle_denoise * (2 ** (-(i-1))), 2) if i > 0 else cycle_denoise )
            if i > 0 else round(starting_denoise, 2)
        )
    else:
        denoise = round((cycle_denoise if i > 0 else starting_denoise), 2)

    if denoise < denoise_cutoff and scale_denoise:
        denoise = denoise_cutoff

    if i >= (secondary_start_cycle - 1) and secondary_model and not secondary_switched:
        run_model = secondary_model
        denoise = cycle_denoise
        model = None
        secondary_switched = True

    if steps_scaling and i > 0:

        steps = (
            steps + steps_scaling_value
            if steps_control == 'increment'
            else steps - steps_scaling_value
        )
        steps = (
            ( steps
            if steps <= steps_cutoff
            else steps_cutoff )
            if steps_control == 'increment'
            else ( steps
            if steps >= steps_cutoff
            else steps_cutoff )
        )

    print("Steps:", steps)
    print("Denoise:", denoise)

    if pos_additive:

        pos_strength = 0.0 if i <= 0 else pos_add_strength

        if pos_add_mode == 'increment':
            pos_strength = (
                ( round(pos_add_strength * (2 ** (i-1)), 2)
                if i > 0
                else pos_add_strength )
                if pos_add_strength_scaling
                else pos_add_strength
            )
            pos_strength = (
                pos_add_strength_cutoff
                if pos_strength > pos_add_strength_cutoff
                else pos_strength
            )
        else:
            pos_strength = (
                ( round(pos_add_strength / (2 ** (i-1)), 2)
                if i > 0
                else pos_add_strength )
                if pos_add_strength_scaling
                else pos_add_strength
            )
            pos_strength = (
                pos_add_strength_cutoff
                if pos_strength < pos_add_strength_cutoff
                else pos_strength
            )
        comb = nodes.ConditioningAverage()
        positive = comb.addWeighted(pos_additive, positive, pos_strength)[0]
        print("Positive Additive Strength:", pos_strength)

    if neg_additive:

        neg_strength = 0.0 if i <= 0 else pos_add_strength

        if neg_add_mode == 'increment':
            neg_strength = (
                ( round(neg_add_strength * (2 ** (i-1)), 2)
                if i > 0
                else neg_add_strength )
                if neg_add_strength_scaling
                else neg_add_strength
            )
            neg_strength = (
                neg_add_strength_cutoff
                if neg_strength > neg_add_strength_cutoff
                else neg_strength
            )
        else:
            neg_strength = (
                ( round(neg_add_strength / (2 ** (i-1)), 2)
                if i > 0
                else neg_add_strength )
                if neg_add_strength_scaling
                else neg_add_strength
            )
            neg_strength = (
                neg_add_strength_cutoff
                if neg_strength < neg_add_strength_cutoff
                else neg_strength
            )

        comb = nodes.ConditioningAverage()
        negative = comb.addWeighted(neg_additive, negative, neg_strength)[0]
```

```
                print("Negative Additive Strength:", neg_strength)

            if i != 0:
                latent_image = latent_image_result

            samples = nodes.common_ksampler(
                run_model,
                seed,
                steps,
                cfg,
                sampler_name,
                scheduler,
                positive,
                negative,
                latent_image,
                denoise=denoise,
            )

            # Upscale
            if i < division_factor - 1:

                tensors = None
                upscaler = None

                resample_filters = {
                    'nearest': 0,
                    'bilinear': 2,
                    'bicubic': 3,
                    'lanczos': 1
                }

                if latent_upscale == 'disable':

                    if tiled_vae:
                        tensors = vae.decode_tiled(samples[0]['samples'])
                    else:
                        tensors = vae.decode(samples[0]['samples'])

                    if processor_model or upscale_model:

                        from comfy_extras import nodes_upscale_model
                        upscaler = nodes_upscale_model.ImageUpscaleWithModel()

                    if processor_model:

                        original_size = tensor2pil(tensors[0]).size
                        upscaled_tensors = upscaler.upscale(processor_model, tensors)
                        tensor_images = []
                        for tensor in upscaled_tensors[0]:
                            pil = tensor2pil(tensor)
                            if pil.size[0] != original_size[0] or pil.size[1] != original_size[1]:
                                pil = pil.resize((original_size[0], original_size[1]), Image.Resampling(resample_filters
[scale_sampling]))
                            if sharpen_strength != 0.0:
                                pil = self.unsharp_filter(pil, sharpen_radius, sharpen_strength)
                            tensor_images.append(pil2tensor(pil))

                        tensor_images = torch.cat(tensor_images, dim=0)
                    if upscale_model:

                        if processor_model:
                            tensors = tensor_images
                            del tensor_images

                        original_size = tensor2pil(tensors[0]).size
                        new_width = round(original_size[0] * current_upscale_factor)
                        new_height = round(original_size[1] * current_upscale_factor)
                        new_width = int(round(new_width / 32) * 32)
                        new_height = int(round(new_height / 32) * 32)
                        upscaled_tensors = upscaler.upscale(upscale_model, tensors)
                        tensor_images = []
                        for tensor in upscaled_tensors[0]:
                            tensor = pil2tensor(tensor2pil(tensor).resize((new_width, new_height), Image.Resampling(resample_filters[scale_sampling])))
                            size = max(tensor2pil(tensor).size)
                            if sharpen_strength != 0.0:
                                tensor = pil2tensor(self.unsharp_filter(tensor2pil(tensor), sharpen_radius, sharpen_strength))
                            tensor_images.append(tensor)

                        tensor_images = torch.cat(tensor_images, dim=0)

                    else:

                        tensor_images = []
                        scale = WAS_Image_Rescale()
                        for tensor in tensors:
                            tensor = scale.image_rescale(tensor.unsqueeze(0), "rescale", "true", scale_sampling, current_upscale_factor, 0, 0)[0]
                            size = max(tensor2pil(tensor).size)
                            if sharpen_strength > 0.0:
                                tensor = pil2tensor(self.unsharp_filter(tensor2pil(tensor), sharpen_radius, sharpen_strength))
                            tensor_images.append(tensor)
                        tensor_images = torch.cat(tensor_images, dim=0)

                    if tiled_vae:
                        latent_image_result = {"samples": vae.encode_tiled(self.vae_encode_crop_pixels(tensor_images)[:,:,:,:3])}
                    else:
                        latent_image_result = {"samples": vae.encode(self.vae_encode_crop_pixels(tensor_images)[:,:,:,:3])}

                else:

                    upscaler = nodes.LatentUpscaleBy()
                    latent_image_result = upscaler.upscale(samples[0], latent_upscale, current_upscale_factor)[0]

            else:

                latent_image_result = samples[0]

        return (latent_image_result, )
```

```
    @staticmethod
    def vae_encode_crop_pixels(pixels):
        x = (pixels.shape[1] // 8) * 8
        y = (pixels.shape[2] // 8) * 8
        if pixels.shape[1] != x or pixels.shape[2] != y:
            x_offset = (pixels.shape[1] % 8) // 2
            y_offset = (pixels.shape[2] % 8) // 2
            pixels = pixels[:, x_offset:x + x_offset, y_offset:y + y_offset, :]
        return pixels

    @staticmethod
    def unsharp_filter(image, radius=2, amount=1.0):
        from skimage.filters import unsharp_mask
        img_array = np.array(image)
        img_array = img_array / 255.0
        sharpened = unsharp_mask(img_array, radius=radius, amount=amount, channel_axis=2)
        sharpened = (sharpened * 255.0).astype(np.uint8)
        sharpened_pil = Image.fromarray(sharpened)

        return sharpened_pil


# Latent Blend

class WAS_Blend_Latents:
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "latent_a": ("LATENT",),
                "latent_b": ("LATENT",),
                "operation": (["add", "multiply", "divide", "subtract", "overlay", "hard_light", "soft_light", "scre
en", "linear_dodge", "difference", "exclusion", "random"],),
                "blend": ("FLOAT", {"default": 0.5, "min": 0.01, "max": 1.0, "step": 0.01}),
            }
        }

    RETURN_TYPES = ("LATENT",)
    FUNCTION = "latent_blend"

    CATEGORY = "WAS Suite/Latent"

    def latent_blend(self, latent_a, latent_b, operation, blend):
        return ( {"samples": self.blend_latents(latent_a['samples'], latent_b['samples'], operation, blend)},
)

    def blend_latents(self, latent1, latent2, mode='add', blend_percentage=0.5):

        def overlay_blend(latent1, latent2, blend_factor):
            low = 2 * latent1 * latent2
            high = 1 - 2 * (1 - latent1) * (1 - latent2)
            blended_latent = (latent1 * blend_factor) * low + (latent2 * blend_factor) * high
            return blended_latent

        def screen_blend(latent1, latent2, blend_factor):
            inverted_latent1 = 1 - latent1
            inverted_latent2 = 1 - latent2
            blended_latent = 1 - (inverted_latent1 * inverted_latent2 * (1 - blend_factor))
            return blended_latent

        def difference_blend(latent1, latent2, blend_factor):
            blended_latent = abs(latent1 - latent2) * blend_factor
            return blended_latent

        def exclusion_blend(latent1, latent2, blend_factor):
            blended_latent = (latent1 + latent2 - 2 * latent1 * latent2) * blend_factor
            return blended_latent

        def hard_light_blend(latent1, latent2, blend_factor):
            blended_latent = torch.where(latent2 < 0.5, 2 * latent1 * latent2, 1 - 2 * (1 - latent1) * (1 - latent2))
* blend_factor
            return blended_latent

        def linear_dodge_blend(latent1, latent2, blend_factor):
            blended_latent = torch.clamp(latent1 + latent2, 0, 1) * blend_factor
            return blended_latent

        def soft_light_blend(latent1, latent2, blend_factor):
            low = 2 * latent1 * latent2 + latent1 ** 2 - 2 * latent1 * latent2 * latent1
            high = 2 * latent1 * (1 - latent2) + torch.sqrt(latent1) * (2 * latent2 - 1)
            blended_latent = (latent1 * blend_factor) * low + (latent2 * blend_factor) * high
            return blended_latent

        def random_noise(latent1, latent2, blend_factor):
            noise1 = torch.randn_like(latent1)
            noise2 = torch.randn_like(latent2)
            noise1 = (noise1 - noise1.min()) / (noise1.max() - noise1.min())
            noise2 = (noise2 - noise2.min()) / (noise2.max() - noise2.min())
            blended_noise = (latent1 * blend_factor) * noise1 + (latent2 * blend_factor) * noise2
            blended_noise = torch.clamp(blended_noise, 0, 1)
            return blended_noise

        blend_factor1 = blend_percentage
        blend_factor2 = 1 - blend_percentage

        if mode == 'add':
            blended_latent = (latent1 * blend_factor1) + (latent2 * blend_factor2)
        elif mode == 'multiply':
            blended_latent = (latent1 * blend_factor1) * (latent2 * blend_factor2)
        elif mode == 'divide':
            blended_latent = (latent1 * blend_factor1) / (latent2 * blend_factor2)
        elif mode == 'subtract':
            blended_latent = (latent1 * blend_factor1) - (latent2 * blend_factor2)
        elif mode == 'overlay':
            blended_latent = overlay_blend(latent1, latent2, blend_factor1)
        elif mode == 'screen':
            blended_latent = screen_blend(latent1, latent2, blend_factor1)
        elif mode == 'difference':
            blended_latent = difference_blend(latent1, latent2, blend_factor1)
        elif mode == 'exclusion':
            blended_latent = exclusion_blend(latent1, latent2, blend_factor1)
        elif mode == 'hard_light':
            blended_latent = hard_light_blend(latent1, latent2, blend_factor1)
        elif mode == 'linear_dodge':
            blended_latent = linear_dodge_blend(latent1, latent2, blend_factor1)
        elif mode == 'soft_light':
            blended_latent = soft_light_blend(latent1, latent2, blend_factor1)
        elif mode == 'random':
            blended_latent = random_noise(latent1, latent2, blend_factor1)
        else:
            raise ValueError("Unsupported blending mode. Please choose from 'add', 'multiply', 'divide', 'su
btract', 'overlay', 'screen', 'difference', 'exclusion', 'hard_light', 'linear_dodge', 'soft_light', 'custom_nois
e'.")

        blended_latent = self.normalize(blended_latent)
        return blended_latent

    def normalize(self, latent):
        return (latent - latent.min()) / (latent.max() - latent.min())



# SEED NODE

class WAS_Seed:
    @classmethod
    def INPUT_TYPES(cls):
        return {"required":
                {"seed": ("INT", {"default": 0, "min": 0,
                          "max": 0xffffffffffffffff})}
                }

    RETURN_TYPES = ("SEED", "NUMBER", "FLOAT", "INT")
    RETURN_NAMES = ("seed", "number", "float", "int")
    FUNCTION = "seed"

    CATEGORY = "WAS Suite/Number"

    def seed(self, seed):
        return ({"seed": seed, }, seed, float(seed), int(seed) )


# IMAGE SEED

class WAS_Image_To_Seed:
    @classmethod
    def INPUT_TYPES(cls):
        return {"required": {
                    "images": ("IMAGE",),
                }
            }

    RETURN_TYPES = ("INT",)
    OUTPUT_IS_LIST = (True,)

    FUNCTION = "image_to_seed"
    CATEGORY = "WAS Suite/Image/Analyze"

    def image_to_seed(self, images):

        seeds = []
        for image in images:
            image = tensor2pil(image)
            seeds.append(image2seed(image))
        return (seeds, )


#! TEXT NODES

class WAS_Prompt_Styles_Selector:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        style_list = []
        if os.path.exists(STYLES_PATH):
            with open(STYLES_PATH, "r") as f:
                if len(f.readlines()) != 0:
                    f.seek(0)
                    data = f.read()
                    styles = json.loads(data)
                    for style in styles.keys():
                        style_list.append(style)
        if not style_list:
            style_list.append("None")
        return {
            "required": {
                "style": (style_list,),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("positive_string", "negative_string")
    FUNCTION = "load_style"

    CATEGORY = "WAS Suite/Text"

    def load_style(self, style):

        styles = {}
        if os.path.exists(STYLES_PATH):
            with open(STYLES_PATH, 'r') as data:
                styles = json.load(data)
        else:
            cstr(f"The styles file does not exist at `{STYLES_PATH}`. Unable to load styles! Have you import
ed your AUTOMATIC1111 WebUI styles?").error.print()

        if styles and style != None or style != 'None':
            prompt = styles[style]['prompt']
            negative_prompt = styles[style]['negative_prompt']
        else:
            prompt = ''
            negative_prompt = ''

        return (prompt, negative_prompt)

class WAS_Prompt_Multiple_Styles_Selector:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(cls):
        style_list = []
        if os.path.exists(STYLES_PATH):
            with open(STYLES_PATH, "r") as f:
                if len(f.readlines()) != 0:
                    f.seek(0)
                    data = f.read()
                    styles = json.loads(data)
                    for style in styles.keys():
                        style_list.append(style)
        if not style_list:
            style_list.append("None")
        return {
            "required": {
                "style1": (style_list,),
                "style2": (style_list,),
                "style3": (style_list,),
                "style4": (style_list,),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("positive_string", "negative_string")
    FUNCTION = "load_style"

    CATEGORY = "WAS Suite/Text"

    def load_style(self, style1, style2, style3, style4):
        styles = {}
        if os.path.exists(STYLES_PATH):
            with open(STYLES_PATH, 'r') as data:
                styles = json.load(data)
        else:
            cstr(f"The styles file does not exist at `{STYLES_PATH}`. Unable to load styles! Have you import
ed your AUTOMATIC1111 WebUI styles?").error.print()
            return ('', '')

        # Check if the selected styles exist in the loaded styles dictionary
        selected_styles = [style1, style2, style3, style4]
        for style in selected_styles:
            if style not in styles:
                print(f"Style '{style}' was not found in the styles file.")
                return ('', '')

        prompt = ""
        negative_prompt = ""

        # Concatenate the prompts and negative prompts of the selected styles
        for style in selected_styles:
            prompt += styles[style]['prompt'] + " "
            negative_prompt += styles[style]['negative_prompt'] + " "

        return (prompt.strip(), negative_prompt.strip())

# Text Multiline Node

class WAS_Text_Multiline:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": ("STRING", {"default": '', "multiline": True, "dynamicPrompts": True}),
            }
        }
    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_multiline"

    CATEGORY = "WAS Suite/Text"

    def text_multiline(self, text):
        import io
        new_text = []
        for line in io.StringIO(text):
            if not line.strip().startswith('#'):
                new_text.append(line.replace("\n", ''))
        new_text = "\n".join(new_text)

        tokens = TextTokens()
        new_text = tokens.parseTokens(new_text)

        return (new_text, )


class WAS_Text_Multiline_Raw:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": ("STRING", {"default": '', "multiline": True, "dynamicPrompts": False}),
            }
        }
    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_multiline"

    CATEGORY = "WAS Suite/Text"

    def text_multiline(self, text):
        tokens = TextTokens()
        new_text = tokens.parseTokens(text)

        return (new_text, )


# Text List Concatenate Node

class WAS_Text_List_Concatenate:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
            },
            "optional": {
                "list_a": ("LIST", {"forceInput": True}),
                "list_b": ("LIST", {"forceInput": True}),
                "list_c": ("LIST", {"forceInput": True}),
                "list_d": ("LIST", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("LIST",)
    FUNCTION = "text_concatenate_list"

    CATEGORY = "WAS Suite/Text"

    def text_concatenate_list(self, **kwargs):
        merged_list: list[str] = []

        # Iterate over the received inputs in sorted order.
        for k in sorted(kwargs.keys()):
            v = kwargs[k]

            # Only process "list" input ports.
            if isinstance(v, list):
                merged_list += v

        return (merged_list,)


# Text List Node

class WAS_Text_List:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
            },
            "optional": {
                "text_a": ("STRING", {"forceInput": True}),
                "text_b": ("STRING", {"forceInput": True}),
                "text_c": ("STRING", {"forceInput": True}),
                "text_d": ("STRING", {"forceInput": True}),
                "text_e": ("STRING", {"forceInput": True}),
                "text_f": ("STRING", {"forceInput": True}),
                "text_g": ("STRING", {"forceInput": True}),
            }
        }
    RETURN_TYPES = ("LIST",)
    FUNCTION = "text_as_list"

    CATEGORY = "WAS Suite/Text"
    def text_as_list(self, **kwargs):
        text_list: list[str] = []

        # Iterate over the received inputs in sorted order.
        for k in sorted(kwargs.keys()):
            v = kwargs[k]

            # Only process string input ports.
            if isinstance(v, str):
                text_list.append(v)

        return (text_list,)


# Text List to Text Node

class WAS_Text_List_to_Text:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "delimiter": ("STRING", {"default": ", "}),
                "text_list": ("LIST", {"forceInput": True}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_list_to_text"

    CATEGORY = "WAS Suite/Text"

    def text_list_to_text(self, delimiter, text_list):
        # Handle special case where delimiter is "\n" (literal newline).
        if delimiter == "\\n":
            delimiter = "\n"

        merged_text = delimiter.join(text_list)

        return (merged_text,)


# Text Parse Embeddings

class WAS_Text_Parse_Embeddings_By_Name:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }
    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_parse_embeddings"

    CATEGORY = "WAS Suite/Text/Parse"

    def text_parse_embeddings(self, text):
        return (self.convert_a1111_embeddings(text), )

    def convert_a1111_embeddings(self, text):
        for embeddings_path in comfy_paths.folder_names_and_paths["embeddings"][0]:
            for filename in os.listdir(embeddings_path):
                basename, ext = os.path.splitext(filename)
                pattern = re.compile(r'\b(?<!embedding:){}\b'.format(re.escape(basename)))
                replacement = 'embedding:{}'.format(basename)
                text = re.sub(pattern, replacement, text)

        return text


# Text Dictionary Concatenate

class WAS_Dictionary_Update:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "dictionary_a": ("DICT", ),
                "dictionary_b": ("DICT", ),
            },
            "optional": {
                "dictionary_c": ("DICT", ),
                "dictionary_d": ("DICT", ),
            }
        }
    RETURN_TYPES = ("DICT",)
    FUNCTION = "dictionary_update"

    CATEGORY = "WAS Suite/Text"

    def dictionary_update(self, dictionary_a, dictionary_b, dictionary_c=None, dictionary_d=None):
        return_dictionary = {**dictionary_a, **dictionary_b}
        if dictionary_c is not None:
            return_dictionary = {**return_dictionary, **dictionary_c}
        if dictionary_d is not None:
            return_dictionary = {**return_dictionary, **dictionary_d}
        return (return_dictionary, )


class WAS_Dictionary_Convert:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "dictionary_text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)})
            },
        }
    RETURN_TYPES = ("DICT",)
    FUNCTION = "dictionary_convert"

    CATEGORY = "WAS Suite/Text"

    def dictionary_convert(self, dictionary_text):
        # using ast.literal_eval here because the string is not guaranteed to be json (using double quotes)
        # https://stackoverflow.com/questions/988228/convert-a-string-representation-of-a-dictionary-to
-a-dictionary
        return (ast.literal_eval(dictionary_text), )


# Text Dictionary Get

class WAS_Dictionary_Get:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "dictionary": ("DICT", ),
                "key": ("STRING", {"default":"", "multiline": False}),
            },
            "optional": {
                "default_value": ("STRING", {"default":"", "multiline": False}),
            }
        }
    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "dictionary_get"

    CATEGORY = "WAS Suite/Text"

    def dictionary_get(self, dictionary, key, default_value=""):
        return (str(dictionary.get(key, default_value)), )


# Text Dictionary New

class WAS_Dictionary_New:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "key_1": ("STRING", {"default":"", "multiline": False}),
                "value_1": ("STRING", {"default":"", "multiline": False}),
            },
            "optional": {
                "key_2": ("STRING", {"default":"", "multiline": False}),
                "value_2": ("STRING", {"default":"", "multiline": False}),
                "key_3": ("STRING", {"default":"", "multiline": False}),
                "value_3": ("STRING", {"default":"", "multiline": False}),
                "key_4": ("STRING", {"default":"", "multiline": False}),
                "value_4": ("STRING", {"default":"", "multiline": False}),
                "key_5": ("STRING", {"default":"", "multiline": False}),
                "value_5": ("STRING", {"default":"", "multiline": False}),
            }
        }
    RETURN_TYPES = ("DICT",)
    FUNCTION = "dictionary_new"

    CATEGORY = "WAS Suite/Text"

    def append_to_dictionary(self, dictionary, key, value):
        if key is not None and key != "":
            dictionary[key] = value
        return dictionary

    def dictionary_new(self, key_1, value_1, key_2, value_2, key_3, value_3, key_4, value_4, key_5, value_5):
        dictionary = {}
        dictionary = self.append_to_dictionary(dictionary, key_1, value_1)
        dictionary = self.append_to_dictionary(dictionary, key_2, value_2)
        dictionary = self.append_to_dictionary(dictionary, key_3, value_3)
        dictionary = self.append_to_dictionary(dictionary, key_4, value_4)
        dictionary = self.append_to_dictionary(dictionary, key_5, value_5)
        return (dictionary, )


# Text Dictionary Keys

class WAS_Dictionary_Keys:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "dictionary": ("DICT",)
            },
            "optional": {}
        }
    RETURN_TYPES = ("LIST",)
    FUNCTION = "dictionary_keys"

    CATEGORY = "WAS Suite/Text"

    def dictionary_keys(self, dictionary):
        return (dictionary.keys(), )


class WAS_Dictionary_to_Text:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "dictionary": ("DICT",)
            },
            "optional": {}
        }
    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "dictionary_to_text"

    CATEGORY = "WAS Suite/Text"

    def dictionary_to_text(self, dictionary):
        return (str(dictionary), )


# Text String Node

class WAS_Text_String:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": ("STRING", {"default": '', "multiline": False}),
            },
            "optional": {
                "text_b": ("STRING", {"default": '', "multiline": False}),
                "text_c": ("STRING", {"default": '', "multiline": False}),
                "text_d": ("STRING", {"default": '', "multiline": False}),
            }
        }
    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE,TEXT_TYPE,TEXT_TYPE)
    FUNCTION = "text_string"

    CATEGORY = "WAS Suite/Text"

    def text_string(self, text='', text_b='', text_c='', text_d=''):

        tokens = TextTokens()

        text = tokens.parseTokens(text)
        text_b = tokens.parseTokens(text_b)
        text_c = tokens.parseTokens(text_c)
        text_d = tokens.parseTokens(text_d)

        return (text, text_b, text_c, text_d)


# Text String Truncation

class WAS_Text_String_Truncate:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": ("STRING", {"forceInput": True}),
                "truncate_by": (["characters", "words"],),
                "truncate_from": (["end", "beginning"],),
                "truncate_to": ("INT", {"default": 10, "min": -99999999, "max": 99999999, "step": 1}),
            },
            "optional": {
                "text_b": ("STRING", {"forceInput": True}),
                "text_c": ("STRING", {"forceInput": True}),
                "text_d": ("STRING", {"forceInput": True}),
            }
        }
    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE,TEXT_TYPE,TEXT_TYPE)
    FUNCTION = "truncate_string"

    CATEGORY = "WAS Suite/Text/Operations"

    def truncate_string(self, text, truncate_by, truncate_from, truncate_to, text_b='', text_c='', text_d=''):
        return (
            self.truncate(text, truncate_to, truncate_from, truncate_by),
            self.truncate(text_b, truncate_to, truncate_from, truncate_by),
            self.truncate(text_c, truncate_to, truncate_from, truncate_by),
            self.truncate(text_d, truncate_to, truncate_from, truncate_by),
        )

    def truncate(self, string, max_length, mode='end', truncate_by='characters'):
        if mode not in ['beginning', 'end']:
            cstr("Invalid mode. 'mode' must be either 'beginning' or 'end'.").error.print()
            mode = "end"
        if truncate_by not in ['characters', 'words']:
            cstr("Invalid truncate_by. 'truncate_by' must be either 'characters' or 'words'.").error.print()
        if truncate_by == 'characters':
            if mode == 'beginning':
                return string[:max_length] if max_length >= 0 else string[max_length:]
            else:
                return string[-max_length:] if max_length >= 0 else string[:max_length]
        words = string.split()
        if mode == 'beginning':
            return ' '.join(words[:max_length]) if max_length >= 0 else ' '.join(words[max_length:])
        else:
            return ' '.join(words[-max_length:]) if max_length >= 0 else ' '.join(words[:max_length])




# Text Compare Strings

class WAS_Text_Compare:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text_a": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "text_b": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "mode": (["similarity","difference"],),
                "tolerance": ("FLOAT", {"default":0.0,"min":0.0,"max":1.0,"step":0.01}),
        }
    }
    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE,"BOOLEAN","NUMBER",TEXT_TYPE)
    RETURN_NAMES = ("TEXT_A_PASS","TEXT_B_PASS","BOOLEAN","SCORE_NUMBER","COMPARISON
_TEXT")
    FUNCTION = "text_compare"

    CATEGORY = "WAS Suite/Text/Search"

    def text_compare(self, text_a='', text_b='', mode='similarity', tolerance=0.0):

        boolean = ( 1 if text_a == text_b else 0 )
        sim = self.string_compare(text_a, text_b, tolerance, ( True if mode == 'difference' else False ))
        score = float(sim[0])
        sim_result = ' '.join(sim[1][::-1])
        sim_result = ' '.join(sim_result.split())

        return (text_a, text_b, bool(boolean), score, sim_result)

    def string_compare(self, str1, str2, threshold=1.0, difference_mode=False):
        m = len(str1)
        n = len(str2)
        if difference_mode:
            dp = [[0 for x in range(n+1)] for x in range(m+1)]
            for i in range(m+1):
                for j in range(n+1):
                    if i == 0:
                        dp[i][j] = j
                    elif j == 0:
                        dp[i][j] = i
                    elif str1[i-1] == str2[j-1]:
                        dp[i][j] = dp[i-1][j-1]
                    else:
                        dp[i][j] = 1 + min(dp[i][j-1],      # Insert
                                           dp[i-1][j],      # Remove
                                           dp[i-1][j-1])    # Replace
            diff_indices = []
            i, j = m, n
            while i > 0 and j > 0:
                if str1[i-1] == str2[j-1]:
                    i -= 1
                    j -= 1
                else:
                    diff_indices.append(i-1)
                    i, j = min((i, j-1), (i-1, j))
            diff_indices.reverse()
            words = []
            start_idx = 0
            for i in diff_indices:
                if str1[i] == " ":
                    words.append(str1[start_idx:i])
                    start_idx = i+1
            words.append(str1[start_idx:m])
            difference_score = 1 - ((dp[m][n] - len(words)) / max(m, n))
            return (difference_score, words[::-1])
        else:
            dp = [[0 for x in range(n+1)] for x in range(m+1)]
            similar_words = set()
            for i in range(m+1):
                for j in range(n+1):
                    if i == 0:
                        dp[i][j] = j
                    elif j == 0:
                        dp[i][j] = i
                    elif str1[i-1] == str2[j-1]:
                        dp[i][j] = dp[i-1][j-1]
                        if i > 1 and j > 1 and str1[i-2] == ' ' and str2[j-2] == ' ':
                            word1_start = i-2
                            word2_start = j-2
                            while word1_start > 0 and str1[word1_start-1] != " ":
                                word1_start -= 1
                            while word2_start > 0 and str2[word2_start-1] != " ":
                                word2_start -= 1
                            word1 = str1[word1_start:i-1]
                            word2 = str2[word2_start:j-1]
                            if word1 in str2 or word2 in str1:
                                if word1 not in similar_words:
                                    similar_words.add(word1)
                                if word2 not in similar_words:
                                    similar_words.add(word2)
                    else:
                        dp[i][j] = 1 + min(dp[i][j-1],      # Insert
                                           dp[i-1][j],      # Remove
                                           dp[i-1][j-1])    # Replace
                    if dp[i][j] <= threshold and i > 0 and j > 0:
                        word1_start = max(0, i-dp[i][j])
                        word2_start = max(0, j-dp[i][j])
                        word1_end = i
                        word2_end = j
                        while word1_start > 0 and str1[word1_start-1] != " ":
                            word1_start -= 1
                        while word2_start > 0 and str2[word2_start-1] != " ":
                            word2_start -= 1
                        while word1_end < m and str1[word1_end] != " ":
                            word1_end += 1
                        while word2_end < n and str2[word2_end] != " ":
                            word2_end += 1
                        word1 = str1[word1_start:word1_end]
                        word2 = str2[word2_start:word2_end]
                        if word1 in str2 or word2 in str1:
                            if word1 not in similar_words:
                                similar_words.add(word1)
                            if word2 not in similar_words:
                                similar_words.add(word2)
            if(max(m,n) == 0):
                similarity_score = 1
            else:
                similarity_score = 1 - (dp[m][n]/max(m,n))
            return (similarity_score, list(similar_words))


# Text Random Line

class WAS_Text_Random_Line:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_random_line"

    CATEGORY = "WAS Suite/Text"

    def text_random_line(self, text, seed):
        lines = text.split("\n")
        random.seed(seed)
        choice = random.choice(lines)
        return (choice, )


# Text Concatenate

class WAS_Text_Concatenate:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "delimiter": ("STRING", {"default": ", "}),
                "clean_whitespace": (["true", "false"],),
            },
            "optional": {
                "text_a": ("STRING", {"forceInput": True}),
                "text_b": ("STRING", {"forceInput": True}),
                "text_c": ("STRING", {"forceInput": True}),
                "text_d": ("STRING", {"forceInput": True}),
                "text_e": ("STRING", {"forceInput": True}),
                "text_f": ("STRING", {"forceInput": True}),
                "text_g": ("STRING", {"forceInput": True}),
                "text_h": ("STRING", {"forceInput": True}),
                "text_i": ("STRING", {"forceInput": True}),
                "text_j": ("STRING", {"forceInput": True}),
                "text_k": ("STRING", {"forceInput": True}),
                "text_l": ("STRING", {"forceInput": True}),
                "text_m": ("STRING", {"forceInput": True}),
                "text_n": ("STRING", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("STRING",)
    FUNCTION = "text_concatenate"

    CATEGORY = "WAS Suite/Text"
    def text_concatenate(self, delimiter, clean_whitespace, **kwargs):
        text_inputs = []

        # Handle special case where delimiter is "\n" (literal newline).
        if delimiter in ("\n", "\\n"):
            delimiter = "\n"

        # Iterate over the received inputs in sorted order.
        for k in sorted(kwargs.keys()):
            v = kwargs[k]

            # Only process string input ports.
            if isinstance(v, str):
                if clean_whitespace == "true":
                    # Remove leading and trailing whitespace around this input.
                    v = v.strip()

                # Only use this input if it's a non-empty string.
                if v != "":
                    text_inputs.append(v)

        # Merge the inputs. Will always generate an output, even if empty.
        merged_text = delimiter.join(text_inputs)

        return (merged_text,)

# Text Find


# Note that these nodes are exposed as "Find", not "Search". This is the first class that follows the nam
ing convention of the node itself.
class WAS_Find:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "substring": ("STRING", {"default": '', "multiline": False}),
                "pattern": ("STRING", {"default": '', "multiline": False}),
            }
        }

    RETURN_TYPES = ("BOOLEAN",)
    RETURN_NAMES = ("found",)
    FUNCTION = "execute"

    CATEGORY = "WAS Suite/Text/Search"

    def execute(self, text, substring, pattern):
        if substring:
            return substring in text

        return bool(re.search(pattern, text))
```

```
# Text Search and Replace

class WAS_Search_and_Replace:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "find": ("STRING", {"default": '', "multiline": False}),
                "replace": ("STRING", {"default": '', "multiline": False}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE, "NUMBER", "FLOAT", "INT")
    RETURN_NAMES = ("result_text", "replacement_count_number", "replacement_count_float", "replace
ment_count_int")
    FUNCTION = "text_search_and_replace"

    CATEGORY = "WAS Suite/Text/Search"

    def text_search_and_replace(self, text, find, replace):
        modified_text, count = self.replace_substring(text, find, replace)
        return (modified_text, count, float(count), int(count))

    def replace_substring(self, text, find, replace):
        modified_text, count = re.subn(find, replace, text)
        return (modified_text, count)


# Text Shuffle

class WAS_Text_Shuffle:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "separator": ("STRING", {"default": ',', "multiline": False}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "shuffle"

    CATEGORY = "WAS Suite/Text/Operations"

    def shuffle(self, text, separator, seed):

        if seed is not None:
            random.seed(seed)
        text_list = text.split(separator)
        random.shuffle(text_list)
        new_text = separator.join(text_list)

        return (new_text, )


# Text Sort

class WAS_Text_Sort:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "separator": ("STRING", {"default": ', ', "multiline": False}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "sort"

    CATEGORY = "WAS Suite/Text/Operations"

    def sort(self, text, separator):
        tokens = WAS_Text_Sort.split_using_protected_groups(text.strip(separator + " \t\n\r"), separator.st
rip())
        sorted_tokens = sorted(tokens, key=WAS_Text_Sort.token_without_leading_brackets)
        return (separator.join(sorted_tokens), )

    @staticmethod
    def token_without_leading_brackets(token):
        return token.replace("\\(", "\0\1").replace("(", "").replace("\0\1", "(").strip()

    @staticmethod
    def split_using_protected_groups(text, separator):
        protected_groups = ""
        nesting_level = 0
        for char in text:
            if char == "(": nesting_level += 1
            if char == ")": nesting_level -= 1

            if char == separator and nesting_level > 0:
                protected_groups += "\0"
            else:
                protected_groups += char

        return list(map(lambda t: t.replace("\0", separator).strip(), protected_groups.split(separator)))



# Text Search and Replace

class WAS_Search_and_Replace_Input:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "find": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "replace": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE, "NUMBER", "FLOAT", "INT")
    RETURN_NAMES = ("result_text", "replacement_count_number", "replacement_count_float", "replace
ment_count_int")
    FUNCTION = "text_search_and_replace"

    CATEGORY = "WAS Suite/Text/Search"

    def text_search_and_replace(self, text, find, replace):
        count = 0
        new_text = text
        while find in new_text:
            new_text = new_text.replace(find, replace, 1)
            count += 1
        return (new_text, count, float(count), int(count))

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")


# Text Search and Replace By Dictionary

class WAS_Search_and_Replace_Dictionary:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "dictionary": ("DICT",),
                "replacement_key": ("STRING", {"default": "__", "multiline": False}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_search_and_replace_dict"

    CATEGORY = "WAS Suite/Text/Search"

    def text_search_and_replace_dict(self, text, dictionary, replacement_key, seed):

        random.seed(seed)
        # Parse Text
        new_text = text

        for term in dictionary.keys():
            tkey = f'{replacement_key}{term}{replacement_key}'
            tcount = new_text.count(tkey)
            for _ in range(tcount):
                new_text = new_text.replace(tkey, random.choice(dictionary[term]), 1)
                if seed > 0 or seed < 0:
                    seed = seed + 1
                    random.seed(seed)

        return (new_text, )

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")


# Text Parse NSP

class WAS_Text_Parse_NSP:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "mode": (["Noodle Soup Prompts", "Wildcards"],),
                "noodle_key": ("STRING", {"default": '__', "multiline": False}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }

    OUTPUT_NODE = True
    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_parse_nsp"

    CATEGORY = "WAS Suite/Text/Parse"

    def text_parse_nsp(self, text, mode="Noodle Soup Prompts", noodle_key='__', seed=0):

        if mode == "Noodle Soup Prompts":

            new_text = nsp_parse(text, seed, noodle_key)
            cstr(f"Text Parse NSP:\n{new_text}").msg.print()

        else:

            new_text = replace_wildcards(text, (None if seed == 0 else seed), noodle_key)
            cstr(f"CLIPTextEncode Wildcards:\n{new_text}").msg.print()

        return (new_text, )
```

```
# TEXT SAVE

class WAS_Text_Save:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": ("STRING", {"forceInput": True}),
                "path": ("STRING", {"default": './ComfyUI/output/[time(%Y-%m-%d)]', "multiline": False}),
                "filename_prefix": ("STRING", {"default": "ComfyUI"}),
                "filename_delimiter": ("STRING", {"default": "_"}),
                "filename_number_padding": ("INT", {"default": 4, "min": 0, "max": 9, "step": 1}),
            },
            "optional": {
                "file_extension": ("STRING", {"default": ".txt"}),
                "encoding": ("STRING", {"default": "utf-8"})
            }
        }

    OUTPUT_NODE = True
    RETURN_TYPES = ()
    FUNCTION = "save_text_file"
    CATEGORY = "WAS Suite/IO"

    def save_text_file(self, text, path, filename_prefix='ComfyUI', filename_delimiter='_', filename_numbe
r_padding=4, file_extension='.txt', encoding='utf-8'):
        tokens = TextTokens()
        path = tokens.parseTokens(path)
        filename_prefix = tokens.parseTokens(filename_prefix)

        if not os.path.exists(path):
            cstr(f"The path `{path}` doesn't exist! Creating it...").warning.print()
            try:
                os.makedirs(path, exist_ok=True)
            except OSError as e:
                cstr(f"The path `{path}` could not be created! Is there write access?\n{e}").error.print()

        if text.strip() == '':
            cstr(f"There is no text specified to save! Text is empty.").error.print()

        delimiter = filename_delimiter
        number_padding = int(filename_number_padding)
        filename = self.generate_filename(path, filename_prefix, delimiter, number_padding, file_extensio
n)
        file_path = os.path.join(path, filename)
        self.write_text_file(file_path, text, encoding)
        update_history_text_files(file_path)
        return (text, {"ui": {"string": text}})

    def generate_filename(self, path, prefix, delimiter, number_padding, extension):
        if number_padding == 0:
            # If number_padding is 0, don't use a numerical suffix
            filename = f"{prefix}{extension}"
        else:
            pattern = f"{re.escape(prefix)}{re.escape(delimiter)}(\\d{{{number_padding}}})"
            existing_counters = [
                int(re.search(pattern, filename).group(1))
                for filename in os.listdir(path)
                if re.match(pattern, filename)
            ]
            existing_counters.sort(reverse=True)
            if existing_counters:
                counter = existing_counters[0] + 1
            else:
                counter = 1
            filename = f"{prefix}{delimiter}{counter:0{number_padding}}{extension}"
            while os.path.exists(os.path.join(path, filename)):
                counter += 1
                filename = f"{prefix}{delimiter}{counter:0{number_padding}}{extension}"
        return filename

    def write_text_file(self, file, content, encoding):
        try:
            with open(file, 'w', encoding=encoding, newline='\n') as f:
                f.write(content)
        except OSError:
            cstr(f"Unable to save file `{file}`").error.print()


# TEXT FILE HISTORY NODE

class WAS_Text_File_History:
    def __init__(self):
        self.HDB = WASDatabase(WAS_HISTORY_DATABASE)
        self.conf = getSuiteConfig()

    @classmethod
    def INPUT_TYPES(cls):
        HDB = WASDatabase(WAS_HISTORY_DATABASE)
        conf = getSuiteConfig()
        paths = ['No History',]
        if HDB.catExists("History") and HDB.keyExists("History", "TextFiles"):
            history_paths = HDB.get("History", "TextFiles")
            if conf.__contains__('history_display_limit'):
                history_paths = history_paths[-conf['history_display_limit']:]
                paths = []
            for path_ in history_paths:
                paths.append(os.path.join('...'+os.sep+os.path.basename(os.path.dirname(path_)), os.path.b
asename(path_)))

        return {
            "required": {
                "file": (paths,),
                "dictionary_name": ("STRING", {"default": '[filename]', "multiline": True}),
            },
        }

    RETURN_TYPES = (TEXT_TYPE,"DICT")
    FUNCTION = "text_file_history"

    CATEGORY = "WAS Suite/History"

    def text_file_history(self, file=None, dictionary_name='[filename]]'):
        file_path = file.strip()
        filename = ( os.path.basename(file_path).split('.', 1)[0]
            if '.' in os.path.basename(file_path) else os.path.basename(file_path) )
        if dictionary_name != '[filename]' or dictionary_name not in [' ', '']:
            filename = dictionary_name
        if not os.path.exists(file_path):
            cstr(f"The path `{file_path}` specified cannot be found.").error.print()
            return ('', {filename: []})
        with open(file_path, 'r', encoding="utf-8", newline='\n') as file:
            text = file.read()

        # Write to file history
        update_history_text_files(file_path)

        import io
        lines = []
        for line in io.StringIO(text):
            if not line.strip().startswith('#'):
                lines.append(line.replace("\n",''))
        dictionary = {filename: lines}

        return ("\n".join(lines), dictionary)

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

# TEXT TO CONDITIONIONG

class WAS_Text_to_Conditioning:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "clip": ("CLIP",),
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }

    RETURN_TYPES = ("CONDITIONING",)
    FUNCTION = "text_to_conditioning"

    CATEGORY = "WAS Suite/Text/Operations"

    def text_to_conditioning(self, clip, text):
        encoder = nodes.CLIPTextEncode()
        encoded = encoder.encode(clip=clip, text=text)
        return (encoded[0], { "ui": { "string": text } })



# TEXT PARSE TOKENS

class WAS_Text_Parse_Tokens:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_parse_tokens"

    CATEGORY = "WAS Suite/Text/Tokens"

    def text_parse_tokens(self, text):
        # Token Parser
        tokens = TextTokens()
        return (tokens.parseTokens(text), )

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

# TEXT ADD TOKENS


class WAS_Text_Add_Tokens:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "tokens": ("STRING", {"default": "[hello]: world", "multiline": True}),
                "print_current_tokens": (["false", "true"],),
            }
        }

    RETURN_TYPES = ()
    FUNCTION = "text_add_tokens"
    OUTPUT_NODE = True
    CATEGORY = "WAS Suite/Text/Tokens"

    def text_add_tokens(self, tokens, print_current_tokens="false"):

        import io

        # Token Parser
        tk = TextTokens()

        # Parse out Tokens
        for line in io.StringIO(tokens):
            parts = line.split(':')
            token = parts[0].strip()
            token_value = parts[1].strip()
            tk.addToken(token, token_value)
        # Current Tokens
        if print_current_tokens == "true":
            cstr(f'Current Custom Tokens:').msg.print()
            print(json.dumps(tk.custom_tokens, indent=4))

        return tokens

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")


# TEXT ADD TOKEN BY INPUT


class WAS_Text_Add_Token_Input:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "token_name": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "token_value": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "print_current_tokens": (["false", "true"],),
            }
        }

    RETURN_TYPES = ()
    FUNCTION = "text_add_token"
    OUTPUT_NODE = True
    CATEGORY = "WAS Suite/Text/Tokens"

    def text_add_token(self, token_name, token_value, print_current_tokens="false"):

        if token_name.strip() == '':
            cstr(f'A `token_name` is required for a token; token name provided is empty.').error.print()
            pass

        # Token Parser
        tk = TextTokens()

        # Add Tokens
        tk.addToken(token_name, token_value)

        # Current Tokens
        if print_current_tokens == "true":
            cstr(f'Current Custom Tokens:').msg.print()
            print(json.dumps(tk.custom_tokens, indent=4))

        return (token_name, token_value)

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")
```

```
# TEXT TO CONSOLE

class WAS_Text_to_Console:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "label": ("STRING", {"default": f'Text Output', "multiline": False}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    OUTPUT_NODE = True
    FUNCTION = "text_to_console"

    CATEGORY = "WAS Suite/Debug"

    def text_to_console(self, text, label):
        if label.strip() != '':
            cstr(f'\033[33m{label}\033[0m:\n{text}\n').msg.print()
        else:
            cstr(f"\033[33mText to Console\033[0m:\n{text}\n").msg.print()
        return (text, )

# DICT TO CONSOLE

class WAS_Dictionary_To_Console:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "dictionary": ("DICT",),
                "label": ("STRING", {"default": f'Dictionary Output', "multiline": False}),
            }
        }

    RETURN_TYPES = ("DICT",)
    OUTPUT_NODE = True
    FUNCTION = "text_to_console"

    CATEGORY = "WAS Suite/Debug"

    def text_to_console(self, dictionary, label):
        if label.strip() != '':
            print(f'\033[34mWAS Node Suite \033[33m{label}\033[0m:\n')
            from pprint import pprint
            pprint(dictionary, indent=4)
            print('')
        else:
            cstr(f"\033[33mText to Console\033[0m:\n")
            pprint(dictionary, indent=4)
            print('')
        return (dictionary, )


# LOAD TEXT FILE

class WAS_Text_Load_From_File:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "file_path": ("STRING", {"default": '', "multiline": False}),
                "dictionary_name": ("STRING", {"default": '[filename]', "multiline": False}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,"DICT")
    FUNCTION = "load_file"

    CATEGORY = "WAS Suite/IO"

    def load_file(self, file_path='', dictionary_name='[filename]]'):

        filename = ( os.path.basename(file_path).split('.', 1)[0]
            if '.' in os.path.basename(file_path) else os.path.basename(file_path) )
        if dictionary_name != '[filename]':
            filename = dictionary_name
        if not os.path.exists(file_path):
            cstr(f"The path `{file_path}` specified cannot be found.").error.print()
            return ('', {filename: []})
        with open(file_path, 'r', encoding="utf-8", newline='\n') as file:
            text = file.read()

        # Write to file history
        update_history_text_files(file_path)

        import io
        lines = []
        for line in io.StringIO(text):
            if not line.strip().startswith('#'):
                lines.append(line.replace("\n",'').replace("\r",''))
        dictionary = {filename: lines}

        return ("\n".join(lines), dictionary)

# TEXT LOAD FROM FILE

class WAS_Text_Load_Line_From_File:
    def __init__(self):
        self.HDB = WASDatabase(WAS_HISTORY_DATABASE)

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "file_path": ("STRING", {"default": '', "multiline": False}),
                "dictionary_name": ("STRING", {"default": '[filename]', "multiline": False}),
                "label": ("STRING", {"default": 'TextBatch', "multiline": False}),
                "mode": (["automatic", "index"],),
                "index": ("INT", {"default": 0, "min": 0, "step": 1}),
            },
            "optional": {
                "multiline_text": (TEXT_TYPE, {"forceInput": True}),
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        if kwargs['mode'] != 'index':
            return float("NaN")
        else:
            m = hashlib.sha256()
            if os.path.exists(kwargs['file_path']):
                with open(kwargs['file_path'], 'rb') as f:
                    m.update(f.read())
                return m.digest().hex()
            else:
                return False

    RETURN_TYPES = (TEXT_TYPE, "DICT")
    RETURN_NAMES = ("line_text", "dictionary")
    FUNCTION = "load_file"

    CATEGORY = "WAS Suite/Text"

    def load_file(self, file_path='', dictionary_name='[filename]', label='TextBatch',
                  mode='automatic', index=0, multiline_text=None):
        if multiline_text is not None:
            lines = multiline_text.strip().split('\n')
            if mode == 'index':
                if index < 0 or index >= len(lines):
                    cstr(f"Invalid line index `{index}`").error.print()
                    return ('', {dictionary_name: []})
                line = lines[index]
            else:
                line_index = self.HDB.get('TextBatch Counters', label)
                if line_index is None:
                    line_index = 0
                line = lines[line_index % len(lines)]
                self.HDB.insert('TextBatch Counters', label, line_index + 1)
            return (line, {dictionary_name: lines})

        if file_path == '':
            cstr("No file path specified.").error.print()
            return ('', {dictionary_name: []})

        if not os.path.exists(file_path):
            cstr(f"The path `{file_path}` specified cannot be found.").error.print()
            return ('', {dictionary_name: []})

        file_list = self.TextFileLoader(file_path, label)
        line, lines = None, []
        if mode == 'automatic':
            line, lines = file_list.get_next_line()
        elif mode == 'index':
            if index >= len(file_list.lines):
                index = index % len(file_list.lines)
            line, lines = file_list.get_line_by_index(index)
        if line is None:
            cstr("No valid line was found. The file may be empty or all lines have been read.").error.print()
            return ('', {dictionary_name: []})
        file_list.store_index()
        update_history_text_files(file_path)

        return (line, {dictionary_name: lines})

    class TextFileLoader:
        def __init__(self, file_path, label):
            self.WDB = WDB
            self.file_path = file_path
            self.lines = []
            self.index = 0
            self.label = label
            self.load_file(file_path)

        def load_file(self, file_path):
            stored_file_path = self.WDB.get('TextBatch Paths', self.label)
            stored_index = self.WDB.get('TextBatch Counters', self.label)
            if stored_file_path != file_path:
                self.index = 0
                self.WDB.insert('TextBatch Counters', self.label, 0)
                self.WDB.insert('TextBatch Paths', self.label, file_path)
            else:
                self.index = stored_index
            with open(file_path, 'r', encoding="utf-8", newline='\n') as file:
                self.lines = [line.strip() for line in file]

        def get_line_index(self):
            return self.index

        def set_line_index(self, index):
            self.index = index
            self.WDB.insert('TextBatch Counters', self.label, self.index)

        def get_next_line(self):
            if self.index >= len(self.lines):
                self.index = 0
            line = self.lines[self.index]
            self.index += 1
            if self.index == len(self.lines):
                self.index = 0
            cstr(f'{cstr.color.YELLOW}TextBatch{cstr.color.END} Index: {self.index}').msg.print()
            return line, self.lines

        def get_line_by_index(self, index):
            if index < 0 or index >= len(self.lines):
                cstr(f"Invalid line index `{index}`").error.print()
                return None, []
            self.index = index
```

```
            line = self.lines[self.index]
            cstr(f'{cstr.color.YELLOW}TextBatch{cstr.color.END} Index: {self.index}').msg.print()
            return line, self.lines

        def store_index(self):
            self.WDB.insert('TextBatch Counters', self.label, self.index)


class WAS_Text_To_String:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }

    RETURN_TYPES = ("STRING",)
    FUNCTION = "text_to_string"

    CATEGORY = "WAS Suite/Text/Operations"

    def text_to_string(self, text):
        return (text, )

class WAS_Text_To_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }

    RETURN_TYPES = ("NUMBER",)
    FUNCTION = "text_to_number"

    CATEGORY = "WAS Suite/Text/Operations"

    def text_to_number(self, text):
        if "." in text:
            number = float(text)
        else:
            number = int(text)
        return (number, )


class WAS_String_To_Text:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "string": ("STRING", {}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "string_to_text"

    CATEGORY = "WAS Suite/Text/Operations"

    def string_to_text(self, string):
        return (string, )

# Random Prompt

class WAS_Text_Random_Prompt:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "search_seed": ("STRING", {"multiline": False}),
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "random_prompt"

    CATEGORY = "WAS Suite/Text"

    def random_prompt(self, search_seed=None):
        if search_seed in ['', ' ']:
            search_seed = None
        return (self.search_lexica_art(search_seed), )

    def search_lexica_art(self, query=None):
        if not query:
            query = random.choice(["portrait","landscape","anime","superhero","animal","nature","scener
y"])
        url = f"https://lexica.art/api/v1/search?q={query}"
        try:
            response = requests.get(url)
            data = response.json()
            images = data.get("images", [])
            if not images:
                return "404 not found error"
            random_image = random.choice(images)
            prompt = random_image.get("prompt")
        except Exception:
            cstr("Unable to establish connection to Lexica API.").error.print()
            prompt = "404 not found error"

        return prompt

# BLIP Model Loader

class WAS_BLIP_Model_Loader:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "blip_model": ("STRING", {"default": "Salesforce/blip-image-captioning-base"}),
                "vqa_model_id": ("STRING", {"default": "Salesforce/blip-vqa-base"}),
                "device": (["cuda", "cpu"],),
            }
        }

    RETURN_TYPES = ("BLIP_MODEL",)
    FUNCTION = "blip_model"

    CATEGORY = "WAS Suite/Loaders"

    def blip_model(self, blip_model, vqa_model_id, device):

        blip_dir = os.path.join(comfy_paths.models_dir, "blip")

        # Attempt legacy support
        if blip_model in ("caption", "interrogate"):
            blip_model = "Salesforce/blip-image-captioning-base"

        blip_model = BlipWrapper(caption_model_id=blip_model, vqa_model_id=vqa_model_id, device=de
vice, cache_dir=blip_dir)

        return ( blip_model, )


# BLIP CAPTION IMAGE

class WAS_BLIP_Analyze_Image:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "images": ("IMAGE",),
                "mode": (["caption", "interrogate"], ),
                "question": ("STRING", {"default": "What does the background consist of?", "multiline": True,
"dynamicPrompts": False}),
                "blip_model": ("BLIP_MODEL",),
            },
            "optional": {
                "min_length": ("INT", {"min": 1, "max": 1024, "default": 24}),
                "max_length": ("INT", {"min": 2, "max": 1024, "default": 64}),
                "num_beams": ("INT", {"min": 1, "max": 12, "default": 5}),
                "no_repeat_ngram_size": ("INT", {"min": 1, "max": 12, "default": 3}),
                "early_stopping": ("BOOLEAN", {"default": False})
            }
        }

    RETURN_TYPES = (TEXT_TYPE, TEXT_TYPE)
    OUTPUT_IS_LIST = (False, True)

    FUNCTION = "blip_caption_image"
    CATEGORY = "WAS Suite/Text/AI"

    def blip_caption_image(self, images, mode, question, blip_model, min_length=24, max_length=64, n
um_beams=5, no_repeat_ngram_size=3, early_stopping=False):

        captions = []
        for image in images:
            pil_image = tensor2pil(image).convert("RGB")
            if mode == "caption":
                cap = blip_model.generate_caption(image=pil_image, min_length=min_length, max_length=m
ax_length, num_beams=num_beams, no_repeat_ngram_size=no_repeat_ngram_size, early_stopping=e
arly_stopping)
                captions.append(cap)
                cstr(f"\033[33mBLIP Caption:\033[0m {cap}").msg.print()
            else:
                cap = blip_model.answer_question(image=pil_image, question=question, min_length=min_len
gth, max_length=max_length, num_beams=num_beams, no_repeat_ngram_size=no_repeat_ngram_siz
e, early_stopping=early_stopping)
                captions.append(cap)
                cstr(f"\033[33m BLIP Answer:\033[0m {cap}").msg.print()

            full_captions = ""
            for i, caption in enumerate(captions):
                full_captions += caption + ("\n\n" if i < len(captions) else "")

        return (full_captions, captions)


# CLIPSeg Model Loader

class WAS_CLIPSeg_Model_Loader:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "model": ("STRING", {"default": "CIDAS/clipseg-rd64-refined", "multiline": False}),
            },
        }

    RETURN_TYPES = ("CLIPSEG_MODEL",)
    RETURN_NAMES = ("clipseg_model",)
    FUNCTION = "clipseg_model"

    CATEGORY = "WAS Suite/Loaders"
    def clipseg_model(self, model):
        from transformers import CLIPSegProcessor, CLIPSegForImageSegmentation

        cache = os.path.join(MODELS_DIR, 'clipseg')

        inputs = CLIPSegProcessor.from_pretrained(model, cache_dir=cache)
        model = CLIPSegForImageSegmentation.from_pretrained(model, cache_dir=cache)

        return ( (inputs, model), )

# CLIPSeg Node

class WAS_CLIPSeg:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "text": ("STRING", {"default": "", "multiline": False}),
            },
            "optional": {
                "clipseg_model": ("CLIPSEG_MODEL",),
            }
        }

    RETURN_TYPES = ("MASK", "IMAGE")
    RETURN_NAMES = ("MASK", "MASK_IMAGE")
    FUNCTION = "CLIPSeg_image"

    CATEGORY = "WAS Suite/Image/Masking"

    def CLIPSeg_image(self, image, text=None, clipseg_model=None):
        from transformers import CLIPSegProcessor, CLIPSegForImageSegmentation

        B, H, W, C = image.shape

        cache = os.path.join(MODELS_DIR, 'clipseg')

        if clipseg_model:
            inputs = clipseg_model[0]
            model = clipseg_model[1]
        else:
            inputs = CLIPSegProcessor.from_pretrained("CIDAS/clipseg-rd64-refined", cache_dir=cache)
            model = CLIPSegForImageSegmentation.from_pretrained("CIDAS/clipseg-rd64-refined", cache_
dir=cache)

        if B == 1:
            image = tensor2pil(image)
            with torch.no_grad():
                result = model(**inputs(text=text, images=image, padding=True, return_tensors="pt"))

            tensor = torch.sigmoid(result[0])
            mask = 1. - (tensor - tensor.min()) / tensor.max()
            mask = mask.unsqueeze(0)
            mask = tensor2pil(mask).convert("L")
            mask = mask.resize(image.size)

            return (pil2mask(mask), pil2tensor(ImageOps.invert(mask.convert("RGB"))))
        else:
            import torchvision
            with torch.no_grad():
                image = image.permute(0, 3, 1, 2)
                image = image * 255
                result = model(**inputs(text=[text] * B, images=image, padding=True, return_tensors="pt"))
                t = torch.sigmoid(result[0])
                mask = (t - t.min()) / t.max()
                mask = torchvision.transforms.functional.resize(mask, (H, W))
                mask: torch.tensor = mask.unsqueeze(-1)
                mask_img = mask.repeat(1, 1, 1, 3)
            return (mask, mask_img,)
class CLIPSeg2:
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
                "text": ("STRING", {"default": "", "multiline": False}),
                "use_cuda": ("BOOLEAN", {"default": False}),
            },
            "optional": {
                "clipseg_model": ("CLIPSEG_MODEL",),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "apply_transform"

    CATEGORY = "image/transformation"

    def apply_transform(self, image, text, use_cuda, clipseg_model):
        import torch
        import torch.nn.functional as F
        from transformers import CLIPSegProcessor, CLIPSegForImageSegmentation

        B, H, W, C = image.shape

        if B != 1:
            raise NotImplementedError("Batch size must be 1")

        # Desired slice size and overlap
        slice_size = 352
        overlap = slice_size // 2

        # Calculate the number of slices needed along each dimension
        num_slices_h = (H - overlap) // (slice_size - overlap) + 1
        num_slices_w = (W - overlap) // (slice_size - overlap) + 1

        # Prepare a list to store the slices
        slices = []

        # Generate the slices
        for i in range(num_slices_h):
            for j in range(num_slices_w):
                start_h = i * (slice_size - overlap)
                start_w = j * (slice_size - overlap)

                end_h = min(start_h + slice_size, H)
                end_w = min(start_w + slice_size, W)

                start_h = max(0, end_h - slice_size)
                start_w = max(0, end_w - slice_size)

                slice_ = image[:, start_h:end_h, start_w:end_w, :]
                slices.append(slice_)

        # Initialize CLIPSeg model and processor
        if clipseg_model:
            processor = clipseg_model[0]
            model = clipseg_model[1]
        else:
            processor = CLIPSegProcessor.from_pretrained("CIDAS/clipseg-rd64-refined")
            model = CLIPSegForImageSegmentation.from_pretrained("CIDAS/clipseg-rd64-refined")
        # Move model to CUDA if requested
        if use_cuda and torch.cuda.is_available():
            model = model.to('cuda')

        processor.image_processor.do_rescale = True
        processor.image_processor.do_resize = False

        image_global = image.permute(0, 3, 1, 2)
        image_global = F.interpolate(image_global, size=(slice_size, slice_size), mode='bilinear', align_cor
ners=False)
        image_global = image_global.permute(0, 2, 3, 1)
        _, image_global = self.CLIPSeg_image(image_global.float(), text, processor, model, use_cuda)
        image_global = image_global.permute(0, 3, 1, 2)
        image_global = F.interpolate(image_global, size=(H, W), mode='bilinear', align_corners=False)
        image_global = image_global.permute(0, 2, 3, 1)

        # Apply the transformation to each slice
        transformed_slices = []
        for slice_ in slices:
            transformed_mask, transformed_slice = self.CLIPSeg_image(slice_, text, processor, model, use_
cuda)
            transformed_slices.append(transformed_slice)

        transformed_slices = torch.cat(transformed_slices)

        # Initialize tensors for reconstruction
        reconstructed_image = torch.zeros((B, H, W, C))
        count_map = torch.zeros((B, H, W, C))

        # Create a blending mask
        mask = np.ones((slice_size, slice_size))
        mask[:overlap, :] *= np.linspace(0, 1, overlap)[:, None]
        mask[-overlap:, :] *= np.linspace(1, 0, overlap)[:, None]
        mask[:, :overlap] *= np.linspace(0, 1, overlap)[None, :]
        mask[:, -overlap:] *= np.linspace(1, 0, overlap)[None, :]
        mask = torch.tensor(mask, dtype=torch.float32).unsqueeze(0).unsqueeze(-1)

        # Place the transformed slices back into the original image dimensions
        for idx in range(transformed_slices.shape[0]):
            i = idx // num_slices_w
            j = idx % num_slices_w

            start_h = i * (slice_size - overlap)
            start_w = j * (slice_size - overlap)

            end_h = min(start_h + slice_size, H)
            end_w = min(start_w + slice_size, W)

            start_h = max(0, end_h - slice_size)
            start_w = max(0, end_w - slice_size)

            reconstructed_image[:, start_h:end_h, start_w:end_w, :] += transformed_slices[idx] * mask
            count_map[:, start_h:end_h, start_w:end_w, :] += mask

        # Avoid division by zero
        count_map[count_map == 0] = 1

        # Average the overlapping regions
        y = reconstructed_image / count_map

        total_power = (y + image_global) / 2
        just_black = image_global < 0.01

        p1 = total_power > .5
        p2 = y > .5
        p3 = image_global > .5

        condition = p1 | p2 | p3
        condition = condition & ~just_black
        y = torch.where(condition, 1.0, 0.0)

        return (y,)

    def CLIPSeg_image(self, image, text, processor, model, use_cuda):
        import torch
        import torchvision.transforms.functional as TF
        B, H, W, C = image.shape

        import torchvision
        with torch.no_grad():
            image = image.permute(0, 3, 1, 2).to(torch.float32) * 255

            inputs = processor(text=[text] * B, images=image, padding=True, return_tensors="pt")

            # Move model and image tensors to CUDA if requested
            if use_cuda and torch.cuda.is_available():
                model = model.to('cuda')
                inputs = {k: v.to('cuda') if isinstance(v, torch.Tensor) else v for k, v in inputs.items()}

            result = model(**inputs)
            t = torch.sigmoid(result[0])
            mask = (t - t.min()) / t.max()
            mask = torchvision.transforms.functional.resize(mask, (H, W))
            mask = mask.unsqueeze(-1)
            mask_img = mask.repeat(1, 1, 1, 3)

            # Move mask and mask_img back to CPU if they were moved to CUDA
            if use_cuda and torch.cuda.is_available():
                mask = mask.cpu()
                mask_img = mask_img.cpu()

        return (mask, mask_img,)

# CLIPSeg Node

class WAS_CLIPSeg_Batch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_a": ("IMAGE",),
                "image_b": ("IMAGE",),
                "text_a": ("STRING", {"default":"", "multiline": False}),
                "text_b": ("STRING", {"default":"", "multiline": False}),
            },
            "optional": {
                "image_c": ("IMAGE",),
                "image_d": ("IMAGE",),
                "image_e": ("IMAGE",),
                "image_f": ("IMAGE",),
                "text_c": ("STRING", {"default":"", "multiline": False}),
                "text_d": ("STRING", {"default":"", "multiline": False}),
                "text_e": ("STRING", {"default":"", "multiline": False}),
                "text_f": ("STRING", {"default":"", "multiline": False}),
            }
        }

    RETURN_TYPES = ("IMAGE", "MASK", "IMAGE")
    RETURN_NAMES = ("IMAGES_BATCH", "MASKS_BATCH", "MASK_IMAGES_BATCH")
    FUNCTION = "CLIPSeg_images"

    CATEGORY = "WAS Suite/Image/Masking"

    def CLIPSeg_images(self, image_a, image_b, text_a, text_b, image_c=None, image_d=None,
                    image_e=None, image_f=None, text_c=None, text_d=None, text_e=None, text_f=None):
        from transformers import CLIPSegProcessor, CLIPSegForImageSegmentation
        import torch.nn.functional as F

        images_pil = [tensor2pil(image_a), tensor2pil(image_b)]

        if image_c is not None:
            if image_c.shape[-2:] != image_a.shape[-2:]:
                cstr("Size of image_c is different from image_a.").error.print()
                return
            images_pil.append(tensor2pil(image_c))
        if image_d is not None:
            if image_d.shape[-2:] != image_a.shape[-2:]:
                cstr("Size of image_d is different from image_a.").error.print()
                return
            images_pil.append(tensor2pil(image_d))
        if image_e is not None:
            if image_e.shape[-2:] != image_a.shape[-2:]:
                cstr("Size of image_e is different from image_a.").error.print()
                return
            images_pil.append(tensor2pil(image_e))
        if image_f is not None:
            if image_f.shape[-2:] != image_a.shape[-2:]:
                cstr("Size of image_f is different from image_a.").error.print()
                return
            images_pil.append(tensor2pil(image_f))

        images_tensor = [torch.from_numpy(np.array(img.convert("RGB")).astype(np.float32) / 255.0).uns
queeze(0) for img in images_pil]
        images_tensor = torch.cat(images_tensor, dim=0)

        prompts = [text_a, text_b]
        if text_c:
            prompts.append(text_c)
        if text_d:
            prompts.append(text_d)
        if text_e:
            prompts.append(text_e)
        if text_f:
            prompts.append(text_f)

        cache = os.path.join(MODELS_DIR, 'clipseg')

        inputs = CLIPSegProcessor.from_pretrained("CIDAS/clipseg-rd64-refined", cache_dir=cache)
        model = CLIPSegForImageSegmentation.from_pretrained("CIDAS/clipseg-rd64-refined", cache_di
r=cache)

        with torch.no_grad():
            result = model(**inputs(text=prompts, images=images_pil, padding=True, return_tensors="pt"))

        masks = []
        mask_images = []
        for i, res in enumerate(result.logits):
            tensor = torch.sigmoid(res)
            mask = 1. - (tensor - tensor.min()) / tensor.max()
            mask = mask.unsqueeze(0)
            mask = tensor2pil(mask).convert("L")
            mask = mask.resize(images_pil[0].size)
            mask_batch = pil2mask(mask)

            masks.append(mask_batch.unsqueeze(0).unsqueeze(1))
            mask_images.append(pil2tensor(ImageOps.invert(mask.convert("RGB"))).squeeze(0))

        masks_tensor = torch.cat(masks, dim=0)
        mask_images_tensor = torch.stack(mask_images, dim=0)

        del inputs, model, result, tensor, masks, mask_images, images_pil

        return (images_tensor, masks_tensor, mask_images_tensor)


# SAM MODEL LOADER

class WAS_SAM_Model_Loader:
    def __init__(self):
        pass
    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "model_size": (["ViT-H", "ViT-L", "ViT-B"], ),
            }
        }

    RETURN_TYPES = ("SAM_MODEL",)
    FUNCTION = "sam_load_model"

    CATEGORY = "WAS Suite/Image/Masking"

    def sam_load_model(self, model_size):
        conf = getSuiteConfig()

        model_filename_mapping = {
            "ViT-H": "sam_vit_h_4b8939.pth",
            "ViT-L": "sam_vit_l_0b3195.pth",
            "ViT-B": "sam_vit_b_01ec64.pth",
        }

        model_url_mapping = {
            "ViT-H": conf['sam_model_vith_url'] if conf.__contains__('sam_model_vith_url') else r"https://dl.f
baipublicfiles.com/segment_anything/sam_vit_h_4b8939.pth",
            "ViT-L": conf['sam_model_vitl_url'] if conf.__contains__('sam_model_vitl_url') else r"https://dl.fbai
publicfiles.com/segment_anything/sam_vit_l_0b3195.pth",
            "ViT-B": conf['sam_model_vitb_url'] if conf.__contains__('sam_model_vitb_url') else r"https://dl.f
baipublicfiles.com/segment_anything/sam_vit_b_01ec64.pth",
        }

        model_url = model_url_mapping[model_size]
        model_filename = model_filename_mapping[model_size]

        if 'GitPython' not in packages():
            install_package("gitpython")

        if not os.path.exists(os.path.join(WAS_SUITE_ROOT, 'repos'+os.sep+'SAM')):
            from git.repo.base import Repo
            cstr("Installing SAM...").msg.print()
            Repo.clone_from('https://github.com/facebookresearch/segment-anything', os.path.join(WAS_S
UITE_ROOT, 'repos'+os.sep+'SAM'))

        sys.path.append(os.path.join(WAS_SUITE_ROOT, 'repos'+os.sep+'SAM'))

        sam_dir = os.path.join(MODELS_DIR, 'sam')
        if not os.path.exists(sam_dir):
            os.makedirs(sam_dir, exist_ok=True)

        sam_file = os.path.join(sam_dir, model_filename)
        if not os.path.exists(sam_file):
            cstr("Selected SAM model not found. Downloading...").msg.print()
            r = requests.get(model_url, allow_redirects=True)
            open(sam_file, 'wb').write(r.content)

        from segment_anything import build_sam_vit_h, build_sam_vit_l, build_sam_vit_b
        if model_size == 'ViT-H':
            sam_model = build_sam_vit_h(sam_file)
        elif model_size == 'ViT-L':
            sam_model = build_sam_vit_l(sam_file)
        elif model_size == 'ViT-B':
            sam_model = build_sam_vit_b(sam_file)
        else:
            raise ValueError(f"SAM model does not match the model_size: '{model_size}'.")

        return (sam_model, )


# SAM PARAMETERS
class WAS_SAM_Parameters:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "points": ("STRING", {"default": "[128, 128]; [0, 0]", "multiline": False}),
                "labels": ("STRING", {"default": "[1, 0]", "multiline": False}),
            }
        }

    RETURN_TYPES = ("SAM_PARAMETERS",)
    FUNCTION = "sam_parameters"

    CATEGORY = "WAS Suite/Image/Masking"

    def sam_parameters(self, points, labels):
        parameters = {
            "points": np.asarray(np.matrix(points)),
            "labels": np.array(np.matrix(labels))[0]
        }

        return (parameters,)


# SAM COMBINE PARAMETERS
class WAS_SAM_Combine_Parameters:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "sam_parameters_a": ("SAM_PARAMETERS",),
                "sam_parameters_b": ("SAM_PARAMETERS",),
            }
        }

    RETURN_TYPES = ("SAM_PARAMETERS",)
    FUNCTION = "sam_combine_parameters"

    CATEGORY = "WAS Suite/Image/Masking"
    def sam_combine_parameters(self, sam_parameters_a, sam_parameters_b):
        parameters = {
            "points": np.concatenate(
                (sam_parameters_a["points"],
                sam_parameters_b["points"]),
                axis=0
            ),
            "labels": np.concatenate(
                (sam_parameters_a["labels"],
                sam_parameters_b["labels"])
            )
        }

        return (parameters,)


# SAM IMAGE MASK
class WAS_SAM_Image_Mask:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "sam_model": ("SAM_MODEL",),
                "sam_parameters": ("SAM_PARAMETERS",),
                "image": ("IMAGE",),
            }
        }

    RETURN_TYPES = ("IMAGE", "MASK",)
    FUNCTION = "sam_image_mask"

    CATEGORY = "WAS Suite/Image/Masking"

    def sam_image_mask(self, sam_model, sam_parameters, image):
        image = tensor2sam(image)
        points = sam_parameters["points"]
        labels = sam_parameters["labels"]

        from segment_anything import SamPredictor

        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        sam_model.to(device=device)

        predictor = SamPredictor(sam_model)
        predictor.set_image(image)

        masks, scores, logits = predictor.predict(
            point_coords=points,
            point_labels=labels,
            multimask_output=False
        )

        sam_model.to(device='cpu')
        mask = np.expand_dims(masks, axis=-1)

        image = np.repeat(mask, 3, axis=-1)
        image = torch.from_numpy(image)

        mask = torch.from_numpy(mask)
        mask = mask.squeeze(2)
        mask = mask.squeeze().to(torch.float32)

        return (image, mask, )

#! BOUNDED IMAGES

# IMAGE BOUNDS

class WAS_Image_Bounds:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "image": ("IMAGE",),
            }
        }

    RETURN_TYPES = ("IMAGE_BOUNDS",)
    FUNCTION = "image_bounds"

    CATEGORY = "WAS Suite/Image/Bound"

    def image_bounds(self, image):
        # Ensure we are working with batches
        image = image.unsqueeze(0) if image.dim() == 3 else image

        return([(0, img.shape[0]-1 , 0, img.shape[1]-1) for img in image],)

# INSET IMAGE BOUNDS

class WAS_Inset_Image_Bounds:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "image_bounds": ("IMAGE_BOUNDS",),
                "inset_left": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
                "inset_right": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
                "inset_top": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
                "inset_bottom": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

    RETURN_TYPES = ("IMAGE_BOUNDS",)
    FUNCTION = "inset_image_bounds"
    CATEGORY = "WAS Suite/Image/Bound"

    def inset_image_bounds(self, image_bounds, inset_left, inset_right, inset_top, inset_bottom):
        inset_bounds = []
        for rmin, rmax, cmin, cmax in image_bounds:
            rmin += inset_top
            rmax -= inset_bottom
            cmin += inset_left
            cmax -= inset_right

            if rmin > rmax or cmin > cmax:
                raise ValueError("Invalid insets provided. Please make sure the insets do not exceed the imag
e bounds.")

            inset_bounds.append((rmin, rmax, cmin, cmax))
        return (inset_bounds,)

# BOUNDED IMAGE BLEND

class WAS_Bounded_Image_Blend:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "target": ("IMAGE",),
                "target_bounds": ("IMAGE_BOUNDS",),
                "source": ("IMAGE",),
                "blend_factor": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 1.0}),
                "feathering": ("INT", {"default": 16, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "bounded_image_blend"

    CATEGORY = "WAS Suite/Image/Bound"

    def bounded_image_blend(self, target, target_bounds, source, blend_factor, feathering):
        # Ensure we are working with batches
        target = target.unsqueeze(0) if target.dim() == 3 else target
        source = source.unsqueeze(0) if source.dim() == 3 else source

        # If number of target images and source images don't match then all source images
        # will be applied only to the first target image, otherwise they will be applied
        # 1 to 1
        # If the number of target bounds and source images don't match then all sourcess will
        # use the first target bounds for scaling and placing the source images, otherwise they
        # will be applied 1 to 1
        tgt_len = 1 if len(target) != len(source) else len(source)
        bounds_len = 1 if len(target_bounds) != len(source) else len(source)

        # Convert target PyTorch tensors to PIL images
        tgt_arr = [tensor2pil(tgt) for tgt in target[:tgt_len]]
        src_arr = [tensor2pil(src) for src in source]
        result_tensors = []
        for idx in range(len(src_arr)):
            src = src_arr[idx]
            # If only one target image, then ensure it is the only one used
            if (tgt_len == 1 and idx == 0) or tgt_len > 1:
                tgt = tgt_arr[idx]

            # If only one bounds object, no need to extract and calculate more than once.
            #   Additionally, if only one bounds obuect, then the mask only needs created once
            if (bounds_len == 1 and idx == 0) or bounds_len > 1:
                # Extract the target bounds
                rmin, rmax, cmin, cmax = target_bounds[idx]

                # Calculate the dimensions of the target bounds
                height, width = (rmax - rmin + 1, cmax - cmin + 1)

                # Create the feathered mask portion the size of the target bounds
                if feathering > 0:
                    inner_mask = Image.new('L', (width - (2 * feathering), height - (2 * feathering)), 255)
                    inner_mask = ImageOps.expand(inner_mask, border=feathering, fill=0)
                    inner_mask = inner_mask.filter(ImageFilter.GaussianBlur(radius=feathering))
                else:
                    inner_mask = Image.new('L', (width, height), 255)

                # Create a blend mask using the inner_mask and blend factor
                inner_mask = inner_mask.point(lambda p: p * blend_factor)

                # Create the blend mask with the same size as the target image
                tgt_mask = Image.new('L', tgt.size, 0)
                # Paste the feathered mask portion into the blend mask at the target bounds position
                tgt_mask.paste(inner_mask, (cmin, rmin))

            # Resize the source image to match the dimensions of the target bounds
            src_resized = src.resize((width, height), Image.Resampling.LANCZOS)

            # Create a blank image with the same size and mode as the target
            src_positioned = Image.new(tgt.mode, tgt.size)

            # Paste the source image onto the blank image using the target bounds
            src_positioned.paste(src_resized, (cmin, rmin))

            # Blend the source and target images using the blend mask
            result = Image.composite(src_positioned, tgt, tgt_mask)

            # Convert the result back to a PyTorch tensor
            result_tensors.append(pil2tensor(result))

        return (torch.cat(result_tensors, dim=0),)

# BOUNDED IMAGE CROP

class WAS_Bounded_Image_Crop:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "image": ("IMAGE",),
                "image_bounds": ("IMAGE_BOUNDS",),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "bounded_image_crop"

    CATEGORY = "WAS Suite/Image/Bound"

    def bounded_image_crop(self, image, image_bounds):
        # Ensure we are working with batches
        image = image.unsqueeze(0) if image.dim() == 3 else image

        # If number of images and bounds don't match, then only the first bounds will be used
        # to crop the images, otherwise, each bounds will be used for each image 1 to 1
        bounds_len = 1 if len(image_bounds) != len(image) else len(image)

        cropped_images = []
        for idx in range(len(image)):
            # If only one bounds object, no need to extract and calculate more than once.
            if (bounds_len == 1 and idx == 0) or bounds_len > 1:
                rmin, rmax, cmin, cmax = image_bounds[idx]

                # Check if the provided bounds are valid
                if rmin > rmax or cmin > cmax:
                    raise ValueError("Invalid bounds provided. Please make sure the bounds are within the ima
ge dimensions.")

            cropped_images.append(image[idx][rmin:rmax+1, cmin:cmax+1, :])

        return (torch.stack(cropped_images, dim=0),)


# BOUNDED IMAGE BLEND WITH MASK

class WAS_Bounded_Image_Blend_With_Mask:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(self):
        return {
            "required": {
                "target": ("IMAGE",),
                "target_mask": ("MASK",),
                "target_bounds": ("IMAGE_BOUNDS",),
                "source": ("IMAGE",),
                "blend_factor": ("FLOAT", {"default": 1.0, "min": 0.0, "max": 1.0}),
                "feathering": ("INT", {"default": 16, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "bounded_image_blend_with_mask"
    CATEGORY = "WAS Suite/Image/Bound"

    def bounded_image_blend_with_mask(self, target, target_mask, target_bounds, source, blend_factor,
feathering):
        # Ensure we are working with batches
        target = target.unsqueeze(0) if target.dim() == 3 else target
        source = source.unsqueeze(0) if source.dim() == 3 else source
        target_mask = target_mask.unsqueeze(0) if target_mask.dim() == 2 else target_mask

        # If number of target masks and source images don't match, then only the first mask will be used
on
        # the source images, otherwise, each mask will be used for each source image 1 to 1
        # Simarly, if the number of target images and source images don't match then
        # all source images will be applied only to the first target, otherwise they will be applied
        # 1 to 1
        tgt_mask_len = 1 if len(target_mask) != len(source) else len(source)
        tgt_len = 1 if len(target) != len(source) else len(source)
        bounds_len = 1 if len(target_bounds) != len(source) else len(source)

        tgt_arr = [tensor2pil(tgt) for tgt in target[:tgt_len]]
        src_arr = [tensor2pil(src) for src in source]
        tgt_mask_arr=[]

        # Convert Target Mask(s) to grayscale image format
        for m_idx in range(tgt_mask_len):
            np_array = np.clip((target_mask[m_idx].cpu().numpy().squeeze() * 255.0), 0, 255)
            tgt_mask_arr.append(Image.fromarray((np_array).astype(np.uint8), mode='L'))

        result_tensors = []
        for idx in range(len(src_arr)):
            src = src_arr[idx]
            # If only one target image, then ensure it is the only one used
            if (tgt_len == 1 and idx == 0) or tgt_len > 1:
                tgt = tgt_arr[idx]

            # If only one bounds, no need to extract and calculate more than once
            if (bounds_len == 1 and idx == 0) or bounds_len > 1:
                # Extract the target bounds
                rmin, rmax, cmin, cmax = target_bounds[idx]

                # Calculate the dimensions of the target bounds
                height, width = (rmax - rmin + 1, cmax - cmin + 1)

            # If only one mask, then ensure that is the only the first is used
            if (tgt_mask_len == 1 and idx == 0) or tgt_mask_len > 1:
                tgt_mask = tgt_mask_arr[idx]

            # If only one mask and one bounds, then mask only needs to
            #   be extended once because all targets will be the same size
            if (tgt_mask_len == 1 and bounds_len == 1 and idx == 0) or \
                (tgt_mask_len > 1 or bounds_len > 1):

                # This is an imperfect, but easy way to determine if  the mask based on the
                #   target image or source image. If not target, assume source. If neither,
                #   then it's not going to look right regardless
                if (tgt_mask.size != tgt.size):
                    # Create the blend mask with the same size as the target image
                    mask_extended_canvas = Image.new('L', tgt.size, 0)
                    # Paste the mask portion into the extended mask at the target bounds position
                    mask_extended_canvas.paste(tgt_mask, (cmin, rmin))

                    tgt_mask = mask_extended_canvas

                # Apply feathering (Gaussian blur) to the blend mask if feather_amount is greater than 0
                if feathering > 0:
                    tgt_mask = tgt_mask.filter(ImageFilter.GaussianBlur(radius=feathering))

                # Apply blending factor to the tgt mask now that it has been extended
                tgt_mask = tgt_mask.point(lambda p: p * blend_factor)

            # Resize the source image to match the dimensions of the target bounds
            src_resized = src.resize((width, height), Image.Resampling.LANCZOS)

            # Create a blank image with the same size and mode as the target
            src_positioned = Image.new(tgt.mode, tgt.size)

            # Paste the source image onto the blank image using the target
            src_positioned.paste(src_resized, (cmin, rmin))

            # Blend the source and target images using the blend mask
            result = Image.composite(src_positioned, tgt, tgt_mask)

            # Convert the result back to a PyTorch tensor
            result_tensors.append(pil2tensor(result))

        return (torch.cat(result_tensors, dim=0),)

# BOUNDED IMAGE CROP WITH MASK

class WAS_Bounded_Image_Crop_With_Mask:
        def __init__(self):
        pass

        @classmethod
        def INPUT_TYPES(self):
        return {
            "required": {
                "image": ("IMAGE",),
                "mask": ("MASK",),
                "padding_left": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
                "padding_right": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
                "padding_top": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
                "padding_bottom": ("INT", {"default": 64, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

        RETURN_TYPES = ("IMAGE", "IMAGE_BOUNDS",)
        FUNCTION = "bounded_image_crop_with_mask"

        CATEGORY = "WAS Suite/Image/Bound"

        def bounded_image_crop_with_mask(self, image, mask, padding_left, padding_right, padding_top, pa
dding_bottom):
        # Ensure we are working with batches
        image = image.unsqueeze(0) if image.dim() == 3 else image
        mask = mask.unsqueeze(0) if mask.dim() == 2 else mask

        # If number of masks and images don't match, then only the first mask will be used on
        # the images, otherwise, each mask will be used for each image 1 to 1
        mask_len = 1 if len(image) != len(mask) else len(image)

        cropped_images = []
        all_bounds = []
        for i in range(len(image)):
            # Single mask or multiple?
            if (mask_len == 1 and i == 0) or mask_len > 0:
                rows = torch.any(mask[i], dim=1)
                cols = torch.any(mask[i], dim=0)
                rmin, rmax = torch.where(rows)[0][[0, -1]]
                cmin, cmax = torch.where(cols)[0][[0, -1]]

                rmin = max(rmin - padding_top, 0)
                rmax = min(rmax + padding_bottom, mask[i].shape[0] - 1)
                cmin = max(cmin - padding_left, 0)
                cmax = min(cmax + padding_right, mask[i].shape[1] - 1)

            # Even if only a single mask, create a bounds for each cropped image
            all_bounds.append([rmin, rmax, cmin, cmax])
            cropped_images.append(image[i][rmin:rmax+1, cmin:cmax+1, :])

            return torch.stack(cropped_images), all_bounds

# DEBUG IMAGE BOUNDS TO CONSOLE

class WAS_Image_Bounds_to_Console:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_bounds": ("IMAGE_BOUNDS",),
                "label": ("STRING", {"default": 'Debug to Console', "multiline": False}),
            }
        }

    RETURN_TYPES = ("IMAGE_BOUNDS",)
    OUTPUT_NODE = True
    FUNCTION = "debug_to_console"

    CATEGORY = "WAS Suite/Debug"

    def debug_to_console(self, image_bounds, label):
        label_out = 'Debug to Console'
        if label.strip() != '':
            label_out = label

        bounds_out = 'Empty'
        if len(bounds_out) > 0:
            bounds_out = ', \n    '.join('\t(rmin={}, rmax={}, cmin={}, cmax={})'
                                .format(a, b, c, d) for a, b, c, d in image_bounds)
        cstr(f'\033[33m{label_out}\033[0m:\n[\n{bounds_out}\n]\n').msg.print()
        return (image_bounds, )

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

#! NUMBERS

# RANDOM NUMBER

class WAS_Random_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number_type": (["integer", "float", "bool"],),
                "minimum": ("FLOAT", {"default": 0, "min": -18446744073709551615, "max": 1844674407370
9551615}),
                "maximum": ("FLOAT", {"default": 0, "min": -18446744073709551615, "max": 184467440737
09551615}),
                "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
            }
        }

    RETURN_TYPES = ("NUMBER", "FLOAT", "INT")
    FUNCTION = "return_randm_number"

    CATEGORY = "WAS Suite/Number"

    def return_randm_number(self, minimum, maximum, seed, number_type='integer'):

        # Set Generator Seed
        random.seed(seed)

        # Return random number
        if number_type:
            if number_type == 'integer':
                number = random.randint(minimum, maximum)
            elif number_type == 'float':
                number = random.uniform(minimum, maximum)
            elif number_type == 'bool':
                number = random.random()
            else:
                return

        # Return number
        return (number, float(number), round(number))

    @classmethod
    def IS_CHANGED(cls, seed, **kwargs):
        m = hashlib.sha256()
        m.update(seed)
        return m.digest().hex()
```

```
# TRUE RANDOM NUMBER

class WAS_True_Random_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "api_key": ("STRING",{"default":"00000000-0000-0000-0000-000000000000", "multiline":
False}),
                "minimum": ("FLOAT", {"default": 0, "min": -18446744073709551615, "max": 1844674407370
9551615}),
                "maximum": ("FLOAT", {"default": 10000000, "min": -18446744073709551615, "max": 18446
744073709551615}),
                "mode": (["random", "fixed"],),
            }
        }

    RETURN_TYPES = ("NUMBER", "FLOAT", "INT")
    FUNCTION = "return_true_randm_number"

    CATEGORY = "WAS Suite/Number"

    def return_true_randm_number(self, api_key=None, minimum=0, maximum=10):

        # Get Random Number
        number = self.get_random_numbers(api_key=api_key, minimum=minimum, maximum=maximum)
[0]

        # Return number
        return (number, )

    def get_random_numbers(self, api_key=None, amount=1, minimum=0, maximum=10, mode="rando
m"):
        '''Get random number(s) from random.org'''
        if api_key in [None, '00000000-0000-0000-0000-000000000000', '']:
            cstr("No API key provided! A valid RANDOM.ORG API key is required to use `True Random.org N
umber Generator`").error.print()
            return [0]

        url = "https://api.random.org/json-rpc/2/invoke"
        headers = {"Content-Type": "application/json"}
        payload = {
            "jsonrpc": "2.0",
            "method": "generateIntegers",
            "params": {
                "apiKey": api_key,
                "n": amount,
                "min": minimum,
                "max": maximum,
                "replacement": True,
                "base": 10
            },
            "id": 1
        }
        response = requests.post(url, headers=headers, data=json.dumps(payload))
        if response.status_code == 200:
            data = response.json()
            if "result" in data:
                return data["result"]["random"]["data"], float(data["result"]["random"]["data"]), int(data["res
ult"]["random"]["data"])

        return [0]

    @classmethod
    def IS_CHANGED(cls, api_key, mode, **kwargs):
        m = hashlib.sha256()
        m.update(api_key)
        if mode == 'fixed':
            return m.digest().hex()
        return float("NaN")


# CONSTANT NUMBER

class WAS_Constant_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number_type": (["integer", "float", "bool"],),
                "number": ("FLOAT", {"default": 0, "min": -18446744073709551615, "max": 18446744073709
551615, "step": 0.01}),
            },
            "optional": {
                "number_as_text": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
            }
        }

    RETURN_TYPES = ("NUMBER", "FLOAT", "INT")
    FUNCTION = "return_constant_number"

    CATEGORY = "WAS Suite/Number"

    def return_constant_number(self, number_type, number, number_as_text=None):

        if number_as_text:
            if number_type == "integer":
                number = int(number_as_text)
            elif number_type == "float":
                number = float(number_as_text)
            else:
                number = bool(number_as_text)

        # Return number
        if number_type:
            if number_type == 'integer':
                return (int(number), float(number), int(number) )
            elif number_type == 'float':
                return (float(number), float(number), int(number) )
        elif number_type == 'bool':
            boolean = (1 if float(number) > 0.5 else 0)
            return (int(boolean), float(boolean), int(boolean) )
        else:
            return (number, float(number), int(number) )

# INCREMENT NUMBER

class WAS_Number_Counter:
    def __init__(self):
        self.counters = {}

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number_type": (["integer", "float"],),
                "mode": (["increment", "decrement", "increment_to_stop", "decrement_to_stop"],),
                "start": ("FLOAT", {"default": 0, "min": -18446744073709551615, "max": 18446744073709551
615, "step": 0.01}),
                "stop": ("FLOAT", {"default": 0, "min": -18446744073709551615, "max": 18446744073709551
615, "step": 0.01}),
                "step": ("FLOAT", {"default": 1, "min": 0, "max": 99999, "step": 0.01}),
            },
            "optional": {
                "reset_bool": ("NUMBER",),
            },
            "hidden": {
                "unique_id": "UNIQUE_ID",
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

    RETURN_TYPES = ("NUMBER", "FLOAT", "INT")
    RETURN_NAMES = ("number", "float", "int")
    FUNCTION = "increment_number"

    CATEGORY = "WAS Suite/Number"

    def increment_number(self, number_type, mode, start, stop, step, unique_id, reset_bool=0):

        counter = int(start) if mode == 'integer' else start
        if self.counters.__contains__(unique_id):
            counter = self.counters[unique_id]

        if round(reset_bool) >= 1:
            counter = start

        if mode == 'increment':
            counter += step
        elif mode == 'deccrement':
            counter -= step
        elif mode == 'increment_to_stop':
            counter = counter + step if counter < stop else counter
        elif mode == 'decrement_to_stop':
            counter = counter - step if counter > stop else counter

        self.counters[unique_id] = counter

        result = int(counter) if number_type == 'integer' else float(counter)

        return ( result, float(counter), int(counter) )


# NUMBER TO SEED

class WAS_Number_To_Seed:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number": ("NUMBER",),
            }
        }

    RETURN_TYPES = ("SEED",)
    FUNCTION = "number_to_seed"

    CATEGORY = "WAS Suite/Number/Operations"

    def number_to_seed(self, number):
        return ({"seed": number, }, )


# NUMBER TO INT

class WAS_Number_To_Int:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number": ("NUMBER",),
            }
        }

    RETURN_TYPES = ("INT",)
    FUNCTION = "number_to_int"

    CATEGORY = "WAS Suite/Number/Operations"

    def number_to_int(self, number):
        return (int(number), )



# NUMBER TO FLOAT
```

```
class WAS_Number_To_Float:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number": ("NUMBER",),
            }
        }

    RETURN_TYPES = ("FLOAT",)
    FUNCTION = "number_to_float"

    CATEGORY = "WAS Suite/Number/Operations"

    def number_to_float(self, number):
        return (float(number), )



# INT TO NUMBER

class WAS_Int_To_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "int_input": ("INT",),
            }
        }

    RETURN_TYPES = ("NUMBER",)
    FUNCTION = "int_to_number"

    CATEGORY = "WAS Suite/Number/Operations"

    def int_to_number(self, int_input):
        return (int(int_input), )



# NUMBER TO FLOAT

class WAS_Float_To_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "float_input": ("FLOAT",),
            }
        }

    RETURN_TYPES = ("NUMBER",)
    FUNCTION = "float_to_number"

    CATEGORY = "WAS Suite/Number/Operations"

    def float_to_number(self, float_input):
        return ( float(float_input), )


# NUMBER TO STRING

class WAS_Number_To_String:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number": ("NUMBER",),
            }
        }

    RETURN_TYPES = ("STRING",)
    FUNCTION = "number_to_string"

    CATEGORY = "WAS Suite/Number/Operations"

    def number_to_string(self, number):
        return ( str(number), )

# NUMBER TO STRING

class WAS_Number_To_Text:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number": ("NUMBER",),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "number_to_text"

    CATEGORY = "WAS Suite/Number/Operations"

    def number_to_text(self, number):
        return ( str(number), )


# NUMBER PI
```

```
class WAS_Number_PI:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {}
        }

    RETURN_TYPES = ("NUMBER", "FLOAT")
    FUNCTION = "number_pi"

    CATEGORY = "WAS Suite/Number"

    def number_pi(self):
        return (math.pi, math.pi)

# Boolean

class WAS_Boolean:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "boolean": ("FLOAT", {"default": 1, "min": 0.0, "max": 1.0, "step": 0.01}),
            }
        }

    RETURN_TYPES = ("BOOLEAN", "NUMBER", "INT", "FLOAT")
    FUNCTION = "return_boolean"

    CATEGORY = "WAS Suite/Logic"

    def return_boolean(self, boolean=1.0):
        boolean_bool = bool(int(round(boolean)))
        int_bool = int(round(boolean))
        return (boolean_bool, int_bool, int_bool, boolean)


# Logical Comparisons Base Class

class WAS_Logical_Comparisons:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "boolean_a": ("BOOLEAN", {"default": False}),
                "boolean_b": ("BOOLEAN", {"default": False}),
            }
        }
    RETURN_TYPES = ("BOOLEAN",)
    FUNCTION = "do"

    CATEGORY = "WAS Suite/Logic"

    def do(self, boolean_a, boolean_b):
        pass


# Logical OR

class WAS_Logical_OR(WAS_Logical_Comparisons):
    def do(self, boolean_a, boolean_b):
        return (boolean_a or boolean_b,)


# Logical AND

class WAS_Logical_AND(WAS_Logical_Comparisons):
    def do(self, boolean_a, boolean_b):
        return (boolean_a and boolean_b,)


# Logical XOR

class WAS_Logical_XOR(WAS_Logical_Comparisons):
    def do(self, boolean_a, boolean_b):
        return (boolean_a != boolean_b,)




# Boolean

class WAS_Boolean_Primitive:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "boolean": ("BOOLEAN", {"default": False}),
            }
        }

    RETURN_TYPES = ("BOOLEAN",)
    FUNCTION = "do"

    CATEGORY = "WAS Suite/Logic"

    def do(self, boolean):
        return (boolean,)


# Boolean

class WAS_Boolean_To_Text:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "boolean": ("BOOLEAN", {"default": False}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "do"

    CATEGORY = "WAS Suite/Logic"

    def do(self, boolean):
        if boolean:
            return ("True",)
        return ("False",)


# Logical NOT

class WAS_Logical_NOT:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "boolean": ("BOOLEAN", {"default": False}),
            }
        }

    RETURN_TYPES = ("BOOLEAN",)
    FUNCTION = "do"

    CATEGORY = "WAS Suite/Logic"

    def do(self, boolean):
        return (not boolean,)


# NUMBER OPERATIONS


class WAS_Number_Operation:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number_a": ("NUMBER",),
                "number_b": ("NUMBER",),
                "operation": (["addition", "subtraction", "division", "floor division", "multiplication", "exponenti
ation", "modulus", "greater-than", "greater-than or equals", "less-than", "less-than or equals", "equals",
"does not equal"],),
            }
        }

    RETURN_TYPES = ("NUMBER", "FLOAT", "INT")
    FUNCTION = "math_operations"

    CATEGORY = "WAS Suite/Number/Operations"

    def math_operations(self, number_a, number_b, operation="addition"):

        # Return random number
        if operation:
            if operation == 'addition':
                result = (number_a + number_b)
                return result, result, int(result)
            elif operation == 'subtraction':
                result = (number_a - number_b)
                return result, result, int(result)
            elif operation == 'division':
                result = (number_a / number_b)
                return result, result, int(result)
            elif operation == 'floor division':
                result = (number_a // number_b)
                return result, result, int(result)
            elif operation == 'multiplication':
                result = (number_a * number_b)
                return result, result, int(result)
            elif operation == 'exponentiation':
                result = (number_a ** number_b)
                return result, result, int(result)
            elif operation == 'modulus':
                result = (number_a % number_b)
                return result, result, int(result)
            elif operation == 'greater-than':
                result = +(number_a > number_b)
                return result, result, int(result)
            elif operation == 'greater-than or equals':
                result = +(number_a >= number_b)
                return result, result, int(result)
            elif operation == 'less-than':
                result = +(number_a < number_b)
                return result, result, int(result)
            elif operation == 'less-than or equals':
                result = +(number_a <= number_b)
                return result, result, int(result)
            elif operation == 'equals':
                result = +(number_a == number_b)
                return result, result, int(result)
            elif operation == 'does not equal':
                result = +(number_a != number_b)
                return result, result, int(result)
            else:
                cstr("Invalid number operation selected.").error.print()
                return (number_a, number_a, int(number_a))

# NUMBER MULTIPLE OF
```

```
class WAS_Number_Multiple_Of:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number": ("NUMBER",),
                "multiple": ("INT", {"default": 8, "min": -18446744073709551615, "max": 18446744073709551
615}),
            }
        }

    RETURN_TYPES =("NUMBER", "FLOAT", "INT")
    FUNCTION = "number_multiple_of"

    CATEGORY = "WAS Suite/Number/Functions"

    def number_multiple_of(self, number, multiple=8):
        if number % multiple != 0:
            return ((number // multiple) * multiple + multiple, )
        return (number, number, int(number))


#! MISC


# Bus.  Converts the 5 main connectors into one, and back again.  You can provide a bus as input
#       or the 5 separate inputs, or a combination.  If you provide a bus input and a separate
#       input (e.g. a model), the model will take precedence.
#
#       The term 'bus' comes from computer hardware, see https://en.wikipedia.org/wiki/Bus_(computin
g)
class WAS_Bus:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required":{},
            "optional": {
                "bus" : ("BUS",),
                "model": ("MODEL",),
                "clip": ("CLIP",),
                "vae": ("VAE",),
                "positive": ("CONDITIONING",),
                "negative": ("CONDITIONING",),
            }
        }
    RETURN_TYPES = ("BUS", "MODEL", "CLIP", "VAE", "CONDITIONING", "CONDITIONING",)
    RETURN_NAMES = ("bus", "model", "clip", "vae", "positive",     "negative")
    FUNCTION = "bus_fn"
    CATEGORY = "WAS Suite/Utilities"

    def bus_fn(self, bus=(None,None,None,None,None), model=None, clip=None, vae=None, positive=
None, negative=None):

        # Unpack the 5 constituents of the bus from the bus tuple.
        (bus_model, bus_clip, bus_vae, bus_positive, bus_negative) = bus

        # If you pass in specific inputs, they override what comes from the bus.
        out_model       = model     or bus_model
        out_clip        = clip      or bus_clip
        out_vae         = vae       or bus_vae
        out_positive    = positive  or bus_positive
        out_negative    = negative  or bus_negative

        # Squash all 5 inputs into the output bus tuple.
        out_bus = (out_model, out_clip, out_vae, out_positive, out_negative)

        if not out_model:
            raise ValueError('Either model or bus containing a model should be supplied')
        if not out_clip:
            raise ValueError('Either clip or bus containing a clip should be supplied')
        if not out_vae:
            raise ValueError('Either vae or bus containing a vae should be supplied')
        # We don't insist that a bus contains conditioning.

        return (out_bus, out_model, out_clip, out_vae, out_positive, out_negative)


# Image Width and Height to Number

class WAS_Image_Size_To_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image": ("IMAGE",),
            }
        }

    RETURN_TYPES = ("NUMBER", "NUMBER", "FLOAT", "FLOAT", "INT", "INT")
    RETURN_NAMES = ("width_num", "height_num", "width_float", "height_float", "width_int", "height_in
t")
    FUNCTION = "image_width_height"

    CATEGORY = "WAS Suite/Number/Operations"

    def image_width_height(self, image):
        image = tensor2pil(image)
        if image.size:
            return( image.size[0], image.size[1], float(image.size[0]), float(image.size[1]), image.size[0], ima
ge.size[1] )
        return ( 0, 0, 0, 0, 0, 0)


# Latent Width and Height to Number

class WAS_Latent_Size_To_Number:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "samples": ("LATENT",),
            }
        }

    RETURN_TYPES = ("NUMBER", "NUMBER", "FLOAT", "FLOAT", "INT", "INT")
    RETURN_NAMES = ("tensor_w_num","tensor_h_num")
    FUNCTION = "latent_width_height"

    CATEGORY = "WAS Suite/Number/Operations"

    def latent_width_height(self, samples):
        size_dict = {}
        i = 0
        for tensor in samples['samples'][0]:
            if not isinstance(tensor, torch.Tensor):
                cstr(f'Input should be a torch.Tensor').error.print()
            shape = tensor.shape
            tensor_height = shape[-2]
            tensor_width = shape[-1]
            size_dict.update({i:[tensor_width, tensor_height]})
        return ( size_dict[0][0], size_dict[0][1], float(size_dict[0][0]), float(size_dict[0][1]), size_dict[0][0], s
ize_dict[0][1] )


# LATENT INPUT SWITCH

class WAS_Latent_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "latent_a": ("LATENT",),
                "latent_b": ("LATENT",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("LATENT",)
    FUNCTION = "latent_input_switch"

    CATEGORY = "WAS Suite/Logic"

    def latent_input_switch(self, latent_a, latent_b, boolean=True):

        if boolean:
            return (latent_a, )
        else:
            return (latent_b, )
```

```
# NUMBER INPUT CONDITION

class WAS_Number_Input_Condition:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number_a": ("NUMBER",),
                "number_b": ("NUMBER",),
                "return_boolean": (["false", "true"],),
                "comparison": (["and", "or", "greater-than", "greater-than or equals", "less-than", "less-than o
r equals", "equals", "does not equal", "divisible by", "if A odd", "if A even", "if A prime", "factor of"],),
            }
        }

    RETURN_TYPES = ("NUMBER", "FLOAT", "INT")
    FUNCTION = "number_input_condition"

    CATEGORY = "WAS Suite/Logic"

    def number_input_condition(self, number_a, number_b, return_boolean="false", comparison="greate
r-than"):

        if comparison:
            if return_boolean == 'true':
                if comparison == 'and':
                    result = 1 if number_a != 0 and number_b != 0 else 0
                elif comparison == 'or':
                    result = 1 if number_a != 0 or number_b != 0 else 0
                elif comparison == 'greater-than':
                    result = 1 if number_a > number_b else 0
                elif comparison == 'greater-than or equals':
                    result = 1 if number_a >= number_b else 0
                elif comparison == 'less-than':
                    result = 1 if number_a < number_b else 0
                elif comparison == 'less-than or equals':
                    result = 1 if number_a <= number_b else 0
                elif comparison == 'equals':
                    result = 1 if number_a == number_b else 0
                elif comparison == 'does not equal':
                    result = 1 if number_a != number_b else 0
                elif comparison == 'divisible by':
                    result = 1 if number_b % number_a == 0 else 0
                elif comparison == 'if A odd':
                    result = 1 if number_a % 2 != 0 else 0
                elif comparison == 'if A even':
                    result = 1 if number_a % 2 == 0 else 0
                elif comparison == 'if A prime':
                    result = 1 if self.is_prime(number_a) else 0
                elif comparison == 'factor of':
                    result = 1 if number_b % number_a == 0 else 0
                else:
                    result = 0
            else:
```

```
            if comparison == 'and':
                result = number_a if number_a != 0 and number_b != 0 else number_b
            elif comparison == 'or':
                result = number_a if number_a != 0 or number_b != 0 else number_b
            elif comparison == 'greater-than':
                result = number_a if number_a > number_b else number_b
            elif comparison == 'greater-than or equals':
                result = number_a if number_a >= number_b else number_b
            elif comparison == 'less-than':
                result = number_a if number_a < number_b else number_b
            elif comparison == 'less-than or equals':
                result = number_a if number_a <= number_b else number_b
            elif comparison == 'equals':
                result = number_a if number_a == number_b else number_b
            elif comparison == 'does not equal':
                result = number_a if number_a != number_b else number_b
            elif comparison == 'divisible by':
                result = number_a if number_b % number_a == 0 else number_b
            elif comparison == 'if A odd':
                result = number_a if number_a % 2 != 0 else number_b
            elif comparison == 'if A even':
                result = number_a if number_a % 2 == 0 else number_b
            elif comparison == 'if A prime':
                result = number_a if self.is_prime(number_a) else number_b
            elif comparison == 'factor of':
                result = number_a if number_b % number_a == 0 else number_b
            else:
                result = number_a

        return (result, float(result), int(result))

    def is_prime(self, n):
        if n <= 1:
            return False
        elif n <= 3:
            return True
        elif n % 2 == 0 or n % 3 == 0:
            return False
        i = 5
        while i * i <= n:
            if n % i == 0 or n % (i + 2) == 0:
                return False
            i += 6
        return True

# ASPECT RATIO

class WAS_Image_Aspect_Ratio:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {},
            "optional": {
                "image": ("IMAGE",),
                "width": ("NUMBER",),
                "height": ("NUMBER",),
            }
        }

    RETURN_TYPES = ("NUMBER", "FLOAT", "NUMBER", TEXT_TYPE, TEXT_TYPE)
    RETURN_NAMES = ("aspect_number", "aspect_float", "is_landscape_bool", "aspect_ratio_common",
"aspect_type")
    FUNCTION = "aspect"

    CATEGORY = "WAS Suite/Logic"

    def aspect(self, boolean=True, image=None, width=None, height=None):

        if width and height:
            width = width; height = height
        elif image is not None:
            width, height = tensor2pil(image).size
        else:
            raise Exception("WAS_Image_Aspect_Ratio must have width and height provided if no image ten
sori supplied.")

        aspect_ratio = width / height
        aspect_type = "landscape" if aspect_ratio > 1 else "portrait" if aspect_ratio < 1 else "square"

        landscape_bool = 0
        if aspect_type == "landscape":
            landscape_bool = 1

        gcd = math.gcd(width, height)
        gcd_w = width // gcd
        gcd_h = height // gcd
        aspect_ratio_common = f"{gcd_w}:{gcd_h}"

        return aspect_ratio, aspect_ratio, landscape_bool, aspect_ratio_common, aspect_type


# NUMBER INPUT SWITCH

class WAS_Number_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number_a": ("NUMBER",),
                "number_b": ("NUMBER",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("NUMBER", "FLOAT", "INT")
    FUNCTION = "number_input_switch"

    CATEGORY = "WAS Suite/Logic"

    def number_input_switch(self, number_a, number_b, boolean=True):
        if boolean:
            return (number_a, float(number_a), int(number_a))
        else:
            return (number_b, float(number_b), int(number_b))


# IMAGE INPUT SWITCH

class WAS_Image_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "image_a": ("IMAGE",),
                "image_b": ("IMAGE",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("IMAGE",)
    FUNCTION = "image_input_switch"

    CATEGORY = "WAS Suite/Logic"

    def image_input_switch(self, image_a, image_b, boolean=True):

        if boolean:
            return (image_a, )
        else:
            return (image_b, )

# CONDITIONING INPUT SWITCH

class WAS_Conditioning_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "conditioning_a": ("CONDITIONING",),
                "conditioning_b": ("CONDITIONING",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("CONDITIONING",)
    FUNCTION = "conditioning_input_switch"

    CATEGORY = "WAS Suite/Logic"

    def conditioning_input_switch(self, conditioning_a, conditioning_b, boolean=True):
        if boolean:
            return (conditioning_a, )
        else:
            return (conditioning_b, )

# MODEL INPUT SWITCH

class WAS_Model_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "model_a": ("MODEL",),
                "model_b": ("MODEL",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("MODEL",)
    FUNCTION = "model_switch"

    CATEGORY = "WAS Suite/Logic"

    def model_switch(self, model_a, model_b, boolean=True):

        if boolean:
            return (model_a, )
        else:
            return (model_b, )

# VAE INPUT SWITCH

class WAS_VAE_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "vae_a": ("VAE",),
                "vae_b": ("VAE",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("VAE",)
    FUNCTION = "vae_switch"

    CATEGORY = "WAS Suite/Logic"

    def vae_switch(self, vae_a, vae_b, boolean=True):

        if boolean:
            return (vae_a, )
        else:
            return (vae_b, )

# CLIP INPUT SWITCH

class WAS_CLIP_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "clip_a": ("CLIP",),
                "clip_b": ("CLIP",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("CLIP",)
    FUNCTION = "clip_switch"

    CATEGORY = "WAS Suite/Logic"

    def clip_switch(self, clip_a, clip_b, boolean=True):

        if boolean:
            return (clip_a, )
        else:
            return (clip_b, )

# UPSCALE MODEL INPUT SWITCH

class WAS_Upscale_Model_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "upscale_model_a": ("UPSCALE_MODEL",),
                "upscale_model_b": ("UPSCALE_MODEL",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("UPSCALE_MODEL",)
    FUNCTION = "upscale_model_switch"

    CATEGORY = "WAS Suite/Logic"

    def upscale_model_switch(self, upscale_model_a, upscale_model_b, boolean=True):

        if boolean:
            return (upscale_model_a, )
        else:
            return (upscale_model_b, )
```

```
# CONTROL NET INPUT SWITCH

class WAS_Control_Net_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "control_net_a": ("CONTROL_NET",),
                "control_net_b": ("CONTROL_NET",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("CONTROL_NET",)
    FUNCTION = "control_net_switch"

    CATEGORY = "WAS Suite/Logic"

    def control_net_switch(self, control_net_a, control_net_b, boolean=True):

        if boolean:
            return (control_net_a, )
        else:
            return (control_net_b, )

# CLIP VISION INPUT SWITCH

class WAS_CLIP_Vision_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "clip_vision_a": ("CLIP_VISION",),
                "clip_vision_b": ("CLIP_VISION",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = ("CLIP_VISION",)
    FUNCTION = "clip_vision_switch"

    CATEGORY = "WAS Suite/Logic"

    def clip_vision_switch(self, clip_vision_a, clip_vision_b, boolean=True):

        if boolean:
            return (clip_vision_a, )
        else:
            return (clip_vision_b)
```

```
# TEXT INPUT SWITCH

class WAS_Text_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text_a": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "text_b": (TEXT_TYPE, {"forceInput": (True if TEXT_TYPE == 'STRING' else False)}),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,)
    FUNCTION = "text_input_switch"

    CATEGORY = "WAS Suite/Logic"

    def text_input_switch(self, text_a, text_b, boolean=True):

        if boolean:
            return (text_a, )
        else:
            return (text_b, )


# TEXT CONTAINS

class WAS_Text_Contains:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "text": ("STRING", {"default": '', "multiline": False}),
                "sub_text": ("STRING", {"default": '', "multiline": False}),
            },
            "optional": {
                "case_insensitive": ("BOOLEAN", {"default": True}),
            }
        }

    RETURN_TYPES = ("BOOLEAN",)
    FUNCTION = "text_contains"

    CATEGORY = "WAS Suite/Logic"

    def text_contains(self, text, sub_text, case_insensitive):
        if case_insensitive:
            sub_text = sub_text.lower()
            text = text.lower()

        return (sub_text in text,)
```

```
# DEBUG INPUT TO CONSOLE


class WAS_Debug_Number_to_Console:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "number": ("NUMBER",),
                "label": ("STRING", {"default": 'Debug to Console', "multiline": False}),
            }
        }

    RETURN_TYPES = ("NUMBER",)
    OUTPUT_NODE = True
    FUNCTION = "debug_to_console"

    CATEGORY = "WAS Suite/Debug"

    def debug_to_console(self, number, label):
        if label.strip() != '':
            cstr(f'\033[33m{label}\033[0m:\n{number}\n').msg.print()
        else:
            cstr(f'\033[33mDebug to Console\033[0m:\n{number}\n').msg.print()
        return (number, )

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")


# CUSTOM COMFYUI NODES

class WAS_Checkpoint_Loader:
    @classmethod
    def INPUT_TYPES(s):
        return {"required": { "config_name": (comfy_paths.get_filename_list("configs"), ),
                              "ckpt_name": (comfy_paths.get_filename_list("checkpoints"), )}}
    RETURN_TYPES = ("MODEL", "CLIP", "VAE", TEXT_TYPE)
    RETURN_NAMES = ("MODEL", "CLIP", "VAE", "NAME_STRING")
    FUNCTION = "load_checkpoint"

    CATEGORY = "WAS Suite/Loaders/Advanced"

    def load_checkpoint(self, config_name, ckpt_name, output_vae=True, output_clip=True):
        config_path = comfy_paths.get_full_path("configs", config_name)
        ckpt_path = comfy_paths.get_full_path("checkpoints", ckpt_name)
        out = comfy.sd.load_checkpoint(config_path, ckpt_path, output_vae=True, output_clip=True, embe
dding_directory=comfy_paths.get_folder_paths("embeddings"))
        return (out[0], out[1], out[2], os.path.splitext(os.path.basename(ckpt_name))[0])

class WAS_Checkpoint_Loader:
    @classmethod
    def INPUT_TYPES(s):
        return {"required": { "config_name": (comfy_paths.get_filename_list("configs"), ),
                              "ckpt_name": (comfy_paths.get_filename_list("checkpoints"), )}}
    RETURN_TYPES = ("MODEL", "CLIP", "VAE", TEXT_TYPE)
    RETURN_NAMES = ("MODEL", "CLIP", "VAE", "NAME_STRING")
    FUNCTION = "load_checkpoint"

    CATEGORY = "WAS Suite/Loaders/Advanced"

    def load_checkpoint(self, config_name, ckpt_name, output_vae=True, output_clip=True):
        config_path = comfy_paths.get_full_path("configs", config_name)
        ckpt_path = comfy_paths.get_full_path("checkpoints", ckpt_name)
        out = comfy.sd.load_checkpoint(config_path, ckpt_path, output_vae=True, output_clip=True, embe
dding_directory=comfy_paths.get_folder_paths("embeddings"))
        return (out[0], out[1], out[2], os.path.splitext(os.path.basename(ckpt_name))[0])

class WAS_Diffusers_Hub_Model_Loader:
    @classmethod
    def INPUT_TYPES(s):
        return {"required": { "repo_id": ("STRING", {"multiline":False}),
                              "revision": ("STRING", {"default": "None", "multiline":False})}}
    RETURN_TYPES = ("MODEL", "CLIP", "VAE", TEXT_TYPE)
    RETURN_NAMES = ("MODEL", "CLIP", "VAE", "NAME_STRING")
    FUNCTION = "load_hub_checkpoint"

    CATEGORY = "WAS Suite/Loaders/Advanced"

    def load_hub_checkpoint(self, repo_id=None, revision=None):
        if revision in ["", "None", "none", None]:
            revision = None
        model_path = comfy_paths.get_folder_paths("diffusers")[0]
        self.download_diffusers_model(repo_id, model_path, revision)
        diffusersLoader = nodes.DiffusersLoader()
        model, clip, vae = diffusersLoader.load_checkpoint(os.path.join(model_path, repo_id))
        return (model, clip, vae, repo_id)

    def download_diffusers_model(self, repo_id, local_dir, revision=None):
        if 'huggingface-hub' not in packages():
            install_package("huggingface_hub")

        from huggingface_hub import snapshot_download
        model_path = os.path.join(local_dir, repo_id)
        ignore_patterns = ["*.ckpt","*.safetensors","*.onnx"]
        snapshot_download(repo_id=repo_id, repo_type="model", local_dir=model_path, revision=revisio
n, use_auth_token=False, ignore_patterns=ignore_patterns)

class WAS_Checkpoint_Loader_Simple:
    @classmethod
    def INPUT_TYPES(s):
        return {"required": { "ckpt_name": (comfy_paths.get_filename_list("checkpoints"), ),
                             }}
    RETURN_TYPES = ("MODEL", "CLIP", "VAE", TEXT_TYPE)
    RETURN_NAMES = ("MODEL", "CLIP", "VAE", "NAME_STRING")
    FUNCTION = "load_checkpoint"

    CATEGORY = "WAS Suite/Loaders"

    def load_checkpoint(self, ckpt_name, output_vae=True, output_clip=True):
        ckpt_path = comfy_paths.get_full_path("checkpoints", ckpt_name)
        out = comfy.sd.load_checkpoint_guess_config(ckpt_path, output_vae=True, output_clip=True, emb
edding_directory=comfy_paths.get_folder_paths("embeddings"))
        return (out[0], out[1], out[2], os.path.splitext(os.path.basename(ckpt_name))[0])

class WAS_Diffusers_Loader:
    @classmethod
    def INPUT_TYPES(cls):
        paths = []
        for search_path in comfy_paths.get_folder_paths("diffusers"):
            if os.path.exists(search_path):
                paths += next(os.walk(search_path))[1]
        return {"required": {"model_path": (paths,), }}
    RETURN_TYPES = ("MODEL", "CLIP", "VAE", TEXT_TYPE)
    RETURN_NAMES = ("MODEL", "CLIP", "VAE", "NAME_STRING")
    FUNCTION = "load_checkpoint"

    CATEGORY = "WAS Suite/Loaders/Advanced"

    def load_checkpoint(self, model_path, output_vae=True, output_clip=True):
        for search_path in comfy_paths.get_folder_paths("diffusers"):
            if os.path.exists(search_path):
                paths = next(os.walk(search_path))[1]
                if model_path in paths:
                    model_path = os.path.join(search_path, model_path)
                    break

        out = comfy.diffusers_convert.load_diffusers(model_path, fp16=comfy.model_management.should
_use_fp16(), output_vae=output_vae, output_clip=output_clip, embedding_directory=comfy_paths.get_f
older_paths("embeddings"))
        return (out[0], out[1], out[2], os.path.basename(model_path))


class WAS_unCLIP_Checkpoint_Loader:
    @classmethod
    def INPUT_TYPES(s):
        return {"required": { "ckpt_name": (comfy_paths.get_filename_list("checkpoints"), ),
                             }}
    RETURN_TYPES = ("MODEL", "CLIP", "VAE", "CLIP_VISION", "STRING")
    RETURN_NAMES = ("MODEL", "CLIP", "VAE", "CLIP_VISION", "NAME_STRING")
    FUNCTION = "load_checkpoint"

    CATEGORY = "WAS Suite/Loaders"

    def load_checkpoint(self, ckpt_name, output_vae=True, output_clip=True):
        ckpt_path = comfy_paths.get_full_path("checkpoints", ckpt_name)
        out = comfy.sd.load_checkpoint_guess_config(ckpt_path, output_vae=True, output_clip=True, outp
ut_clipvision=True, embedding_directory=comfy_paths.get_folder_paths("embeddings"))
        return (out[0], out[1], out[2], out[3], os.path.splitext(os.path.basename(ckpt_name))[0])


class WAS_Lora_Input_Switch:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "model_a": ("MODEL",),
                "clip_a": ("CLIP",),
                "model_b": ("MODEL",),
                "clip_b": ("CLIP",),
                "boolean": ("BOOLEAN", {"forceInput": True}),
            }
        }
    RETURN_TYPES = ("MODEL", "CLIP")
    FUNCTION = "lora_input_switch"

    CATEGORY = "WAS Suite/Logic"

    def lora_input_switch(self, model_a, clip_a, model_b, clip_b, boolean=True):
        if boolean:
            return (model_a, clip_a)
        else:
            return (model_b, clip_b)


class WAS_Lora_Loader:
    def __init__(self):
        self.loaded_lora = None;

    @classmethod
    def INPUT_TYPES(s):
        file_list = comfy_paths.get_filename_list("loras")
        file_list.insert(0, "None")
        return {"required": { "model": ("MODEL",),
                              "clip": ("CLIP", ),
                              "lora_name": (file_list, ),
                              "strength_model": ("FLOAT", {"default": 1.0, "min": -10.0, "max": 10.0, "step": 0.01}),
                              "strength_clip": ("FLOAT", {"default": 1.0, "min": -10.0, "max": 10.0, "step": 0.01}),
                              }}
    RETURN_TYPES = ("MODEL", "CLIP", TEXT_TYPE)
    RETURN_NAMES = ("MODEL", "CLIP", "NAME_STRING")
    FUNCTION = "load_lora"

    CATEGORY = "WAS Suite/Loaders"

    def load_lora(self, model, clip, lora_name, strength_model, strength_clip):
        if strength_model == 0 and strength_clip == 0:
            return (model, clip)

        lora_path = comfy_paths.get_full_path("loras", lora_name)
        lora = None
        if self.loaded_lora is not None:
            if self.loaded_lora[0] == lora_path:
                lora = self.loaded_lora[1]
            else:
                temp = self.loaded_lora
                self.loaded_lora = None
                del temp

        if lora is None:
            lora = comfy.utils.load_torch_file(lora_path, safe_load=True)
            self.loaded_lora = (lora_path, lora)
        model_lora, clip_lora = comfy.sd.load_lora_for_models(model, clip, lora, strength_model, strength_
clip)
        return (model_lora, clip_lora, os.path.splitext(os.path.basename(lora_name))[0])

class WAS_Upscale_Model_Loader:
    @classmethod
    def INPUT_TYPES(s):
        return {"required": { "model_name": (comfy_paths.get_filename_list("upscale_models"), ),
                             }}
    RETURN_TYPES = ("UPSCALE_MODEL",TEXT_TYPE)
    RETURN_NAMES = ("UPSCALE_MODEL","MODEL_NAME_TEXT")
    FUNCTION = "load_model"

    CATEGORY = "WAS Suite/Loaders"

    def load_model(self, model_name):
        model_path = comfy_paths.get_full_path("upscale_models", model_name)
        sd = comfy.utils.load_torch_file(model_path)
        out = model_loading.load_state_dict(sd).eval()
        return (out,model_name)

# VIDEO WRITER

class WAS_Video_Writer:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        WTools = WAS_Tools_Class()
        v = WTools.VideoWriter()
        codecs = []
        for codec in v.get_codecs():
            codecs.append(codec.upper())
        codecs = sorted(codecs)
        return {
            "required": {
                "image": ("IMAGE",),
                "transition_frames": ("INT", {"default":30, "min":0, "max":120, "step":1}),
                "image_delay_sec": ("FLOAT", {"default":2.5, "min":0.1, "max":60000.0, "step":0.1}),
                "fps": ("INT", {"default":30, "min":1, "max":60.0, "step":1}),
                "max_size": ("INT", {"default":512, "min":128, "max":1920, "step":1}),
                "output_path": ("STRING", {"default": "./ComfyUI/output", "multiline": False}),
                "filename": ("STRING", {"default": "comfy_writer", "multiline": False}),
                "codec": (codecs,),
            }
        }

    #@classmethod
    #def IS_CHANGED(cls, **kwargs):
    #    return float("NaN")

    RETURN_TYPES = ("IMAGE",TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("IMAGE_PASS","filepath_text","filename_text")
    FUNCTION = "write_video"

    CATEGORY = "WAS Suite/Animation/Writer"
    def write_video(self, image, transition_frames=10, image_delay_sec=10, fps=30, max_size=512,
                            output_path="./ComfyUI/output", filename="morph", codec="H264"):

        conf = getSuiteConfig()
        if not conf.__contains__('ffmpeg_bin_path'):
            cstr(f"Unable to use MP4 Writer because the `ffmpeg_bin_path` is not set in `{WAS_CONFIG_FIL
E}`").error.print()
            return (image,"","")

        if conf.__contains__('ffmpeg_bin_path'):
            if conf['ffmpeg_bin_path'] != "/path/to/ffmpeg":
                sys.path.append(conf['ffmpeg_bin_path'])
                os.environ["OPENCV_FFMPEG_CAPTURE_OPTIONS"] = "rtsp_transport;udp"
                os.environ['OPENCV_FFMPEG_BINARY'] = conf['ffmpeg_bin_path']

        if output_path.strip() in [None, "", "."]:
            output_path = "./ComfyUI/output"

        if image == None:
            image = pil2tensor(Image.new("RGB", (512,512), (0,0,0)))

        if transition_frames < 0:
            transition_frames = 0
        elif transition_frames > 60:
            transition_frames = 60

        if fps < 1:
            fps = 1
        elif fps > 60:
            fps = 60

        results = []
        for img in image:
            print(img.shape)
            new_image = self.rescale_image(tensor2pil(img), max_size)
            print(new_image.size)

            tokens = TextTokens()
            output_path = os.path.abspath(os.path.join(*tokens.parseTokens(output_path).split('/')))
            output_file = os.path.join(output_path, tokens.parseTokens(filename))

            if not os.path.exists(output_path):
                os.makedirs(output_path, exist_ok=True)

            WTools = WAS_Tools_Class()
            MP4Writer = WTools.VideoWriter(int(transition_frames), int(fps), int(image_delay_sec), max_size
=max_size, codec=codec)
            path = MP4Writer.write(new_image, output_file)

            results.append(img)

        return (torch.cat(results, dim=0), path, filename)

    def rescale_image(self, image, max_dimension):
        width, height = image.size
        if width > max_dimension or height > max_dimension:
            scaling_factor = max(width, height) / max_dimension
            new_width = int(width / scaling_factor)
            new_height = int(height / scaling_factor)
            image = image.resize((new_width, new_height), Image.Resampling(1))
        return image

# VIDEO CREATOR

class WAS_Create_Video_From_Path:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        WTools = WAS_Tools_Class()
        v = WTools.VideoWriter()
        codecs = []
        for codec in v.get_codecs():
            codecs.append(codec.upper())
        codecs = sorted(codecs)
        return {
            "required": {
                "transition_frames": ("INT", {"default":30, "min":0, "max":120, "step":1}),
                "image_delay_sec": ("FLOAT", {"default":2.5, "min":0.01, "max":60000.0, "step":0.01}),
                "fps": ("INT", {"default":30, "min":1, "max":60.0, "step":1}),
                "max_size": ("INT", {"default":512, "min":128, "max":1920, "step":1}),
                "input_path": ("STRING", {"default": "./ComfyUI/input", "multiline": False}),
                "output_path": ("STRING", {"default": "./ComfyUI/output", "multiline": False}),
                "filename": ("STRING", {"default": "comfy_video", "multiline": False}),
                "codec": (codecs,),
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("filepath_text","filename_text")
    FUNCTION = "create_video_from_path"

    CATEGORY = "WAS Suite/Animation"

    def create_video_from_path(self, transition_frames=10, image_delay_sec=10, fps=30, max_size=512,
                        input_path="./ComfyUI/input", output_path="./ComfyUI/output", filename="morph", co
dec="H264"):

        conf = getSuiteConfig()
        if not conf.__contains__('ffmpeg_bin_path'):
            cstr(f"Unable to use MP4 Writer because the `ffmpeg_bin_path` is not set in `{WAS_CONFIG_FIL
E}`").error.print()
            return ("","")

        if conf.__contains__('ffmpeg_bin_path'):
            if conf['ffmpeg_bin_path'] != "/path/to/ffmpeg":
                sys.path.append(conf['ffmpeg_bin_path'])
                os.environ["OPENCV_FFMPEG_CAPTURE_OPTIONS"] = "rtsp_transport;udp"
                os.environ['OPENCV_FFMPEG_BINARY'] = conf['ffmpeg_bin_path']

        if output_path.strip() in [None, "", "."]:
            output_path = "./ComfyUI/output"

        if transition_frames < 0:
            transition_frames = 0
        elif transition_frames > 60:
            transition_frames = 60

        if fps < 1:
            fps = 1
        elif fps > 60:
            fps = 60

        tokens = TextTokens()

        # Check if output_path is an absolute path
        if not os.path.isabs(output_path):
            output_path = os.path.abspath(os.path.join(*tokens.parseTokens(output_path).split('/')))

        output_file = os.path.join(output_path, tokens.parseTokens(filename))

        if not os.path.exists(output_path):
            os.makedirs(output_path, exist_ok=True)

        WTools = WAS_Tools_Class()
        MP4Writer = WTools.VideoWriter(int(transition_frames), int(fps), int(image_delay_sec), max_size, c
odec)
        path = MP4Writer.create_video(input_path, output_file)

        return (path, filename)

# VIDEO FRAME DUMP

class WAS_Video_Frame_Dump:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "video_path": ("STRING", {"default":"./ComfyUI/input/MyVideo.mp4", "multiline":False}),
                "output_path": ("STRING", {"default": "./ComfyUI/input/MyVideo", "multiline": False}),
                "prefix": ("STRING", {"default": "frame_", "multiline": False}),
                "filenumber_digits": ("INT", {"default":4, "min":-1, "max":8, "step":1}),
                "extension": (["png","jpg","gif","tiff"],),
            }
        }

    @classmethod
    def IS_CHANGED(cls, **kwargs):
        return float("NaN")

    RETURN_TYPES = (TEXT_TYPE,"NUMBER")
    RETURN_NAMES = ("output_path","processed_count")
    FUNCTION = "dump_video_frames"

    CATEGORY = "WAS Suite/Animation"
    def dump_video_frames(self, video_path, output_path, prefix="fame_", extension="png",filenumber_
digits=-1):

        conf = getSuiteConfig()
        if not conf.__contains__('ffmpeg_bin_path'):
            cstr(f"Unable to use dump frames because the `ffmpeg_bin_path` is not set in `{WAS_CONFIG_F
ILE}`").error.print()
            return ("",0)

        if conf.__contains__('ffmpeg_bin_path'):
            if conf['ffmpeg_bin_path'] != "/path/to/ffmpeg":
                sys.path.append(conf['ffmpeg_bin_path'])
                os.environ["OPENCV_FFMPEG_CAPTURE_OPTIONS"] = "rtsp_transport;udp"
                os.environ['OPENCV_FFMPEG_BINARY'] = conf['ffmpeg_bin_path']

        if output_path.strip() in [None, "", "."]:
            output_path = "./ComfyUI/input/frames"

        tokens = TextTokens()
        output_path = os.path.abspath(os.path.join(*tokens.parseTokens(output_path).split('/')))
        prefix = tokens.parseTokens(prefix)

        WTools = WAS_Tools_Class()
        MP4Writer = WTools.VideoWriter()
        processed = MP4Writer.extract(video_path, output_path, prefix, extension,filenumber_digits)

        return (output_path, processed)

# CACHING

class WAS_Cache:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "latent_suffix": ("STRING", {"default": str(random.randint(999999, 99999999))+"_cache", "m
ultiline":False}),
                "image_suffix": ("STRING", {"default": str(random.randint(999999, 99999999))+"_cache", "m
ultiline":False}),
                "conditioning_suffix": ("STRING", {"default": str(random.randint(999999, 99999999))+"_cach
e", "multiline":False}),
            },
            "optional": {
                "output_path": ("STRING", {"default": os.path.join(WAS_SUITE_ROOT, 'cache'), "multiline": Fal
se}),
                "latent": ("LATENT",),
                "image": ("IMAGE",),
                "conditioning": ("CONDITIONING",),
            }
        }

    RETURN_TYPES = (TEXT_TYPE,TEXT_TYPE,TEXT_TYPE)
    RETURN_NAMES = ("latent_filename","image_filename","conditioning_filename")
    FUNCTION = "cache_input"
    OUTPUT_NODE = True
    CATEGORY = "WAS Suite/IO"

    def cache_input(self, latent_suffix="_cache", image_suffix="_cache", conditioning_suffix="_cache", o
utput_path=None, latent=None, image=None, conditioning=None):

        if 'joblib' not in packages():
            install_package('joblib')

        import joblib

        output = os.path.join(WAS_SUITE_ROOT, 'cache')
        if output_path:
            if output_path.strip() not in ['', 'none', 'None']:
                output = output_path
        if not os.path.isabs(output):
            output = os.path.abspath(output)
        if not os.path.exists(output):
            os.makedirs(output, exist_ok=True)

        l_filename = ""
        i_filename = ""
        c_filename = ""

        tokens = TextTokens()
        output = tokens.parseTokens(output)

        if latent != None:
            l_filename = f'{tokens.parseTokens(latent_suffix)}.latent'
            out_file = os.path.join(output, l_filename)
            joblib.dump(latent, out_file)
            cstr(f"Latent saved to: {out_file}").msg.print()

        if image != None:
            i_filename = f'{tokens.parseTokens(image_suffix)}.image'
            out_file = os.path.join(output, i_filename)
            joblib.dump(image, out_file)
            cstr(f"Tensor batch saved to: {out_file}").msg.print()

        if conditioning != None:
            c_filename = f'{tokens.parseTokens(conditioning_suffix)}.conditioning'
            out_file = os.path.join(output, c_filename)
            joblib.dump(conditioning, os.path.join(output, out_file))
            cstr(f"Conditioning saved to: {out_file}").msg.print()

        return (l_filename, i_filename, c_filename)


class WAS_Load_Cache:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "latent_path": ("STRING", {"default": "", "multiline":False}),
                "image_path": ("STRING", {"default": "", "multiline":False}),
                "conditioning_path": ("STRING", {"default": "", "multiline":False}),
            }
        }

    RETURN_TYPES = ("LATENT","IMAGE","CONDITIONING")
    RETURN_NAMES = ("LATENT","IMAGE","CONDITIONING")
    FUNCTION = "load_cache"

    CATEGORY = "WAS Suite/IO"

    def load_cache(self, latent_path=None, image_path=None, conditioning_path=None):

        if 'joblib' not in packages():
            install_package('joblib')

        import joblib

        input_path = os.path.join(WAS_SUITE_ROOT, 'cache')

        latent = None
        image = None
        conditioning = None

        if latent_path not in ["",None]:
            if os.path.exists(latent_path):
                latent = joblib.load(latent_path)
            else:
                cstr(f"Unable to locate cache file {latent_path}").error.print()

        if image_path not in ["",None]:
            if os.path.exists(image_path):
                image = joblib.load(image_path)
            else:
                cstr(f"Unable to locate cache file {image_path}").msg.print()

        if conditioning_path not in ["",None]:
            if os.path.exists(conditioning_path):
                conditioning = joblib.load(conditioning_path)
            else:
                cstr(f"Unable to locate cache file {conditioning_path}").error.print()

        return (latent, image, conditioning)


# SAMPLES PASS STAT SYSTEM

class WAS_Samples_Passthrough_Stat_System:
    def __init__(self):
        pass

    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "samples": ("LATENT",),
            }
        }
    RETURN_TYPES = ("LATENT",)
    RETURN_NAMES = ("samples",)
    FUNCTION = "stat_system"

    CATEGORY = "WAS Suite/Debug"

    def stat_system(self, samples):

        log = ""
        for stat in self.get_system_stats():
            log += stat + "\n"

        cstr("\n"+log).msg.print()

        return (samples,)

    def get_system_stats(self):

        import psutil

        # RAM
        ram = psutil.virtual_memory()
        ram_used = ram.used / (1024 ** 3)
        ram_total = ram.total / (1024 ** 3)
        ram_stats = f"Used RAM: {ram_used:.2f} GB / Total RAM: {ram_total:.2f} GB"

        # VRAM (with PyTorch)
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        vram_used = torch.cuda.memory_allocated(device) / (1024 ** 3)
        vram_total = torch.cuda.get_device_properties(device).total_memory / (1024 ** 3)
        vram_stats = f"Used VRAM: {vram_used:.2f} GB / Total VRAM: {vram_total:.2f} GB"

        # Hard Drive Space
        hard_drive = psutil.disk_usage("/")
        used_space = hard_drive.used / (1024 ** 3)
        total_space = hard_drive.total / (1024 ** 3)
        hard_drive_stats = f"Used Space: {used_space:.2f} GB / Total Space: {total_space:.2f} GB"

        return [ram_stats, vram_stats, hard_drive_stats]

# Class to count the number of places on an integer

class WAS_Integer_Place_Counter:
    @classmethod
    def INPUT_TYPES(cls):
        return {
            "required": {
                "int_input": ("INT", {"default": 0, "min": 0, "max": 10000000, "step": 1}),
            }
        }
    RETURN_TYPES = ("INT",)
    RETURN_NAMES = ("INT_PLACES",)
    FUNCTION = "count_places"

    CATEGORY = "WAS Suite/Integer"

    def count_places(self, int_input):
        output = len(str(int_input))
        cstr("\nInteger Places Count: "+str(output)).msg.print()
        return (output,)


# NODE MAPPING
NODE_CLASS_MAPPINGS = {
    "BLIP Model Loader": WAS_BLIP_Model_Loader,
    "Blend Latents": WAS_Blend_Latents,
    "Bus Node": WAS_Bus,
    "Cache Node": WAS_Cache,
    "Checkpoint Loader": WAS_Checkpoint_Loader,
    "Checkpoint Loader (Simple)": WAS_Checkpoint_Loader_Simple,
    "CLIPTextEncode (NSP)": WAS_NSP_CLIPTextEncoder,
    "CLIP Input Switch": WAS_CLIP_Input_Switch,
    "CLIP Vision Input Switch": WAS_CLIP_Vision_Input_Switch,
    "Conditioning Input Switch": WAS_Conditioning_Input_Switch,
    "Constant Number": WAS_Constant_Number,
    "Create Grid Image": WAS_Image_Grid_Image,
    "Create Grid Image from Batch": WAS_Image_Grid_Image_Batch,
    "Create Morph Image": WAS_Image_Morph_GIF,
    "Create Morph Image from Path": WAS_Image_Morph_GIF_By_Path,
    "Create Video from Path": WAS_Create_Video_From_Path,
    "CLIPSeg Masking": WAS_CLIPSeg,
    "CLIPSeg Model Loader": WAS_CLIPSeg_Model_Loader,
    "CLIPSeg Batch Masking": WAS_CLIPSeg_Batch,
    "Convert Masks to Images": WAS_Mask_To_Image,
    "Control Net Model Input Switch": WAS_Control_Net_Input_Switch,
    "Debug Number to Console": WAS_Debug_Number_to_Console,
    "Dictionary to Console": WAS_Dictionary_To_Console,
    "Diffusers Model Loader": WAS_Diffusers_Loader,
    "Diffusers Hub Model Down-Loader": WAS_Diffusers_Hub_Model_Loader,
    "Export API": WAS_Export_API,
    "Latent Input Switch": WAS_Latent_Input_Switch,
    "Load Cache": WAS_Load_Cache,
    "Logic Boolean": WAS_Boolean,
    "Logic Boolean Primitive": WAS_Boolean_Primitive,
    "Logic Comparison OR": WAS_Logical_OR,
    "Logic Comparison AND": WAS_Logical_AND,
    "Logic Comparison XOR": WAS_Logical_XOR,
    "Logic NOT": WAS_Logical_NOT,
    "Lora Loader": WAS_Lora_Loader,
    "Hex to HSL": WAS_Hex_to_HSL,
    "HSL to Hex": WAS_HSL_to_Hex,
    "Image SSAO (Ambient Occlusion)": WAS_Image_Ambient_Occlusion,
    "Image SSDO (Direct Occlusion)": WAS_Image_Direct_Occlusion,
    "Image Analyze": WAS_Image_Analyze,
    "Image Aspect Ratio": WAS_Image_Aspect_Ratio,
    "Image Batch": WAS_Image_Batch,
    "Image Blank": WAS_Image_Blank,
    "Image Blend by Mask": WAS_Image_Blend_Mask,
    "Image Blend": WAS_Image_Blend,
    "Image Blending Mode": WAS_Image_Blending_Mode,
    "Image Bloom Filter": WAS_Image_Bloom_Filter,
    "Image Canny Filter": WAS_Canny_Filter,
    "Image Chromatic Aberration": WAS_Image_Chromatic_Aberration,
    "Image Color Palette": WAS_Image_Color_Palette,
    "Image Crop Face": WAS_Image_Crop_Face,
    "Image Crop Location": WAS_Image_Crop_Location,
    "Image Crop Square Location": WAS_Image_Crop_Square_Location,
    "Image Displacement Warp": WAS_Image_Displacement_Warp,
    "Image Lucy Sharpen": WAS_Lucy_Sharpen,
    "Image Paste Face": WAS_Image_Paste_Face_Crop,
    "Image Paste Crop": WAS_Image_Paste_Crop,
    "Image Paste Crop by Location": WAS_Image_Paste_Crop_Location,
    "Image Pixelate": WAS_Image_Pixelate,
    "Image Power Noise": WAS_Image_Power_Noise,
    "Image Dragan Photography Filter": WAS_Dragon_Filter,
    "Image Edge Detection Filter": WAS_Image_Edge,
    "Image Film Grain": WAS_Film_Grain,
    "Image Filter Adjustments": WAS_Image_Filters,
    "Image Flip": WAS_Image_Flip,
    "Image Gradient Map": WAS_Image_Gradient_Map,
    "Image Generate Gradient": WAS_Image_Generate_Gradient,
    "Image High Pass Filter": WAS_Image_High_Pass_Filter,
    "Image History Loader": WAS_Image_History,
    "Image Input Switch": WAS_Image_Input_Switch,
    "Image Levels Adjustment": WAS_Image_Levels,
    "Image Load": WAS_Load_Image,
    "Image Median Filter": WAS_Image_Median_Filter,
    "Image Mix RGB Channels": WAS_Image_RGB_Merge,
    "Image Monitor Effects Filter": WAS_Image_Monitor_Distortion_Filter,
    "Image Nova Filter": WAS_Image_Nova_Filter,
    "Image Padding": WAS_Image_Padding,
    "Image Perlin Noise": WAS_Image_Perlin_Noise,
    "Image Rembg (Remove Background)": WAS_Remove_Rembg,
    "Image Perlin Power Fractal": WAS_Image_Perlin_Power_Fractal,
    "Image Remove Background (Alpha)": WAS_Remove_Background,
    "Image Remove Color": WAS_Image_Remove_Color,
    "Image Resize": WAS_Image_Rescale,
    "Image Rotate": WAS_Image_Rotate,
    "Image Rotate Hue": WAS_Image_Rotate_Hue,
    "Image Send HTTP": WAS_Image_Send_HTTP,
    "Image Save": WAS_Image_Save,
    "Image Seamless Texture": WAS_Image_Make_Seamless,
    "Image Select Channel": WAS_Image_Select_Channel,
    "Image Select Color": WAS_Image_Select_Color,
    "Image Shadows and Highlights": WAS_Shadow_And_Highlight_Adjustment,
    "Image Size to Number": WAS_Image_Size_To_Number,
    "Image Stitch": WAS_Image_Stitch,
    "Image Style Filter": WAS_Image_Style_Filter,
    "Image Threshold": WAS_Image_Threshold,
    "Image Tiled": WAS_Image_Tile_Batch,
    "Image Transpose": WAS_Image_Transpose,
    "Image fDOF Filter": WAS_Image_fDOF,
    "Image to Latent Mask": WAS_Image_To_Mask,
    "Image to Noise": WAS_Image_To_Noise,
    "Image to Seed": WAS_Image_To_Seed,
    "Images to RGB": WAS_Images_To_RGB,
    "Images to Linear": WAS_Images_To_Linear,
    "Integer place counter": WAS_Integer_Place_Counter,
    "Image Voronoi Noise Filter": WAS_Image_Voronoi_Noise_Filter,
    "KSampler (WAS)": WAS_KSampler,
    "KSampler Cycle": WAS_KSampler_Cycle,
    "Latent Batch": WAS_Latent_Batch,
    "Latent Noise Injection": WAS_Latent_Noise,
    "Latent Size to Number": WAS_Latent_Size_To_Number,
    "Latent Upscale by Factor (WAS)": WAS_Latent_Upscale,
    "Load Image Batch": WAS_Load_Image_Batch,
    "Load Text File": WAS_Text_Load_From_File,
    "Load Lora": WAS_Lora_Loader,
    "Lora Input Switch": WAS_Lora_Input_Switch,
    "Masks Add": WAS_Mask_Add,
    "Masks Subtract": WAS_Mask_Subtract,
    "Mask Arbitrary Region": WAS_Mask_Arbitrary_Region,
    "Mask Batch to Mask": WAS_Mask_Batch_to_Single_Mask,
    "Mask Batch": WAS_Mask_Batch,
    "Mask Ceiling Region": WAS_Mask_Ceiling_Region,
    "Mask Crop Dominant Region": WAS_Mask_Crop_Dominant_Region,
    "Mask Crop Minority Region": WAS_Mask_Crop_Minority_Region,
    "Mask Crop Region": WAS_Mask_Crop_Region,
    "Mask Paste Region": WAS_Mask_Paste_Region,
    "Mask Dilate Region": WAS_Mask_Dilate_Region,
    "Mask Dominant Region": WAS_Mask_Dominant_Region,
    "Mask Erode Region": WAS_Mask_Erode_Region,
    "Mask Fill Holes": WAS_Mask_Fill_Region,
    "Mask Floor Region": WAS_Mask_Floor_Region,
    "Mask Gaussian Region": WAS_Mask_Gaussian_Region,
    "Mask Invert": WAS_Mask_Invert,
    "Mask Minority Region": WAS_Mask_Minority_Region,
    "Mask Smooth Region": WAS_Mask_Smooth_Region,
    "Mask Threshold Region": WAS_Mask_Threshold_Region,
    "Masks Combine Regions": WAS_Mask_Combine,
    "Masks Combine Batch": WAS_Mask_Combine_Batch,
    "MiDaS Model Loader": MiDaS_Model_Loader,
    "MiDaS Depth Approximation": MiDaS_Depth_Approx,
    "MiDaS Mask Image": MiDaS_Background_Foreground_Removal,
    "Model Input Switch": WAS_Model_Input_Switch,
    "Number Counter": WAS_Number_Counter,
    "Number Operation": WAS_Number_Operation,
    "Number to Float": WAS_Number_To_Float,
    "Number Input Switch": WAS_Number_Input_Switch,
    "Number Input Condition": WAS_Number_Input_Condition,
    "Number Multiple Of": WAS_Number_Multiple_Of,
    "Number PI": WAS_Number_PI,
    "Number to Int": WAS_Number_To_Int,
    "Number to Seed": WAS_Number_To_Seed,
    "Number to String": WAS_Number_To_String,
    "Number to Text": WAS_Number_To_Text,
    "Boolean To Text": WAS_Boolean_To_Text,
    "Prompt Styles Selector": WAS_Prompt_Styles_Selector,
    "Prompt Multiple Styles Selector": WAS_Prompt_Multiple_Styles_Selector,
    "Random Number": WAS_Random_Number,
    "Save Text File": WAS_Text_Save,
    "Seed": WAS_Seed,
    "Tensor Batch to Image": WAS_Tensor_Batch_to_Image,
    "BLIP Analyze Image": WAS_BLIP_Analyze_Image,
    "SAM Model Loader": WAS_SAM_Model_Loader,
    "SAM Parameters": WAS_SAM_Parameters,
    "SAM Parameters Combine": WAS_SAM_Combine_Parameters,
    "SAM Image Mask": WAS_SAM_Image_Mask,
    "Samples Passthrough (Stat System)": WAS_Samples_Passthrough_Stat_System,
    "String to Text": WAS_String_To_Text,
    "Image Bounds": WAS_Image_Bounds,
    "Inset Image Bounds": WAS_Inset_Image_Bounds,
    "Bounded Image Blend": WAS_Bounded_Image_Blend,
    "Bounded Image Blend with Mask": WAS_Bounded_Image_Blend_With_Mask,
    "Bounded Image Crop": WAS_Bounded_Image_Crop,
    "Bounded Image Crop with Mask": WAS_Bounded_Image_Crop_With_Mask,
    "Image Bounds to Console": WAS_Image_Bounds_to_Console,
    "Text Dictionary Update": WAS_Dictionary_Update,
    "Text Dictionary Get": WAS_Dictionary_Get,
    "Text Dictionary Convert": WAS_Dictionary_Convert,
    "Text Dictionary New": WAS_Dictionary_New,
    "Text Dictionary Keys": WAS_Dictionary_Keys,
    "Text Dictionary To Text": WAS_Dictionary_to_Text,
    "Text Add Tokens": WAS_Text_Add_Tokens,
    "Text Add Token by Input": WAS_Text_Add_Token_Input,
    "Text Compare": WAS_Text_Compare,
    "Text Concatenate": WAS_Text_Concatenate,
    "Text File History Loader": WAS_Text_File_History,
    "Text Find and Replace by Dictionary": WAS_Search_and_Replace_Dictionary,
    "Text Find and Replace Input": WAS_Search_and_Replace_Input,
    "Text Find and Replace": WAS_Search_and_Replace,
    "Text Find": WAS_Find,
    "Text Input Switch": WAS_Text_Input_Switch,
    "Text List": WAS_Text_List,
    "Text List Concatenate": WAS_Text_List_Concatenate,
    "Text List to Text": WAS_Text_List_to_Text,
    "Text Load Line From File": WAS_Text_Load_Line_From_File,
    "Text Multiline": WAS_Text_Multiline,
    "Text Multiline (Code Compatible)": WAS_Text_Multiline_Raw,
    "Text Parse A1111 Embeddings": WAS_Text_Parse_Embeddings_By_Name,
    "Text Parse Noodle Soup Prompts": WAS_Text_Parse_NSP,
    "Text Parse Tokens": WAS_Text_Parse_Tokens,
    "Text Random Line": WAS_Text_Random_Line,
    "Text Random Prompt": WAS_Text_Random_Prompt,
    "Text String": WAS_Text_String,
    "Text Contains": WAS_Text_Contains,
    "Text Shuffle": WAS_Text_Shuffle,
    "Text Sort": WAS_Text_Sort,
    "Text to Conditioning": WAS_Text_to_Conditioning,
    "Text to Console": WAS_Text_to_Console,
    "Text to Number": WAS_Text_To_Number,
    "Text to String": WAS_Text_To_String,
    "Text String Truncate": WAS_Text_String_Truncate,
    "True Random.org Number Generator": WAS_True_Random_Number,
    "unCLIP Checkpoint Loader": WAS_unCLIP_Checkpoint_Loader,
    "Upscale Model Loader": WAS_Upscale_Model_Loader,
    "Upscale Model Switch": WAS_Upscale_Model_Input_Switch,
    "Write to GIF": WAS_Image_Morph_GIF_Writer,
    "Write to Video": WAS_Video_Writer,
    "VAE Input Switch": WAS_VAE_Input_Switch,
    "Video Dump Frames": WAS_Video_Frame_Dump,
    "CLIPSEG2": CLIPSeg2
}

#! EXTRA NODES

# Check for BlenderNeko's Advanced CLIP Text Encode repo
BKAdvCLIP_dir = os.path.join(CUSTOM_NODES_DIR, "ComfyUI_ADV_CLIP_emb")
if os.path.exists(BKAdvCLIP_dir):
    cstr(f"BlenderNeko\'s Advanced CLIP Text Encode found, attempting to enable `CLIPTextEncode` su
pport.").msg.print()

    class WAS_AdvancedCLIPTextEncode:
        @classmethod
        def INPUT_TYPES(s):
            return {
                "required": {
                    "mode": (["Noodle Soup Prompts", "Wildcards"],),
                    "noodle_key": ("STRING", {"default": '__', "multiline": False}),
                    "seed": ("INT", {"default": 0, "min": 0, "max": 0xffffffffffffffff}),
                    "clip": ("CLIP", ),
                    "token_normalization": (["none", "mean", "length", "length+mean"],),
                    "weight_interpretation": (["comfy", "A1111", "compel", "comfy++"],),
                    "text": ("STRING", {"multiline": True}),
                    }
                }

        RETURN_TYPES = ("CONDITIONING", TEXT_TYPE, TEXT_TYPE)
        RETURN_NAMES = ("conditioning", "parsed_text", "raw_text")
        OUTPUT_NODE = True
        FUNCTION = "encode"
        CATEGORY = "WAS Suite/Conditioning"

        DESCRIPTION = "A node based on Blenderneko's <a href='https://github.com/BlenderNeko/Comf
yUI_ADV_CLIP_embw' target='_blank'>Advanced CLIP Text Encode</a>. This version adds the ability t
o use Noodle Soup Prompts and Wildcards. Wildcards are stored in WAS Node Suite root under the fol
der 'wildcards'. You can create the folder if it doesn't exist and move your wildcards into it."
        URL = {
            "Example Workflow": "https://github.com/WASasquatch/was-node-suite-comfyui",
        }
        IMAGES = [
            "https://i.postimg.cc/Jh4N2h5r/CLIPText-Encode-BLK-plus-NSP.png",
        ]

        def encode(self, clip, text, token_normalization, weight_interpretation, seed=0, mode="Noodle So
up Prompts", noodle_key="__"):

            BKAdvCLIP_dir = os.path.join(CUSTOM_NODES_DIR, "ComfyUI_ADV_CLIP_emb")
            sys.path.append(BKAdvCLIP_dir)

            from ComfyUI_ADV_CLIP_emb.nodes import AdvancedCLIPTextEncode

            if mode == "Noodle Soup Prompts":
                new_text = nsp_parse(text, int(seed), noodle_key)
            else:
                new_text = replace_wildcards(text, (None if seed == 0 else seed), noodle_key)

            new_text = parse_dynamic_prompt(new_text, seed)
            new_text, text_vars = parse_prompt_vars(new_text)
            cstr(f"CLIPTextEncode Prased Prompt:\n {new_text}").msg.print()

            encode = AdvancedCLIPTextEncode().encode(clip, new_text, token_normalization, weight_inter
pretation)

            sys.path.remove(BKAdvCLIP_dir)

            return ([[encode[0][0][0], encode[0][0][1]]], new_text, text, { "ui": { "string": new_text } } )
    NODE_CLASS_MAPPINGS.update({"CLIPTextEncode (BlenderNeko Advanced + NSP)": WAS_Advan
cedCLIPTextEncode})

    if NODE_CLASS_MAPPINGS.__contains__("CLIPTextEncode (BlenderNeko Advanced + NSP)"):
        cstr('`CLIPTextEncode (BlenderNeko Advanced + NSP)` node enabled under `WAS Suite/Condition
ing` menu.').msg.print()

# opencv-python-headless handling
if 'opencv-python' in packages() or 'opencv-python-headless' in packages():
    try:
        import cv2
        build_info = ' '.join(cv2.getBuildInformation().split())
        if "FFMPEG: YES" in build_info:
            if was_config.__contains__('show_startup_junk'):
                if was_config['show_startup_junk']:
                    cstr("OpenCV Python FFMPEG support is enabled").msg.print()
            if was_config.__contains__('ffmpeg_bin_path'):
                if was_config['ffmpeg_bin_path'] == "/path/to/ffmpeg":
                    cstr(f"`ffmpeg_bin_path` is not set in `{WAS_CONFIG_FILE}` config file. Will attempt to use
system ffmpeg binaries if available.").warning.print()
                else:
                    if was_config.__contains__('show_startup_junk'):
                        if was_config['show_startup_junk']:
                            cstr(f"`ffmpeg_bin_path` is set to: {was_config['ffmpeg_bin_path']}").msg.print()
        else:
            cstr(f"OpenCV Python FFMPEG support is not enabled\033[0m. OpenCV Python FFMPEG supp
ort, and FFMPEG binaries is required for video writing.").warning.print()
    except ImportError:
        cstr("OpenCV Python module cannot be found. Attempting install...").warning.print()
        install_package(
            package='opencv-python-headless[ffmpeg]',
            uninstall_first=['opencv-python', 'opencv-python-headless[ffmpeg]']
        )
        try:
            import cv2
            cstr("OpenCV Python installed.").msg.print()
        except ImportError:
            cstr("OpenCV Python module still cannot be imported. There is a system conflict.").error.print()
else:
    install_package('opencv-python-headless[ffmpeg]')
    try:
        import cv2
        cstr("OpenCV Python installed.").msg.print()
    except ImportError:
        cstr("OpenCV Python module still cannot be imported. There is a system conflict.").error.print()

# scipy handling
if 'scipy' not in packages():
    install_package('scipy')
    try:
        import scipy
    except ImportError as e:
        cstr("Unable to import tools for certain masking procedures.").msg.print()
        print(e)

# scikit-image handling
```

```
try:
    import skimage
except ImportError as e:
    install_package(
        package='scikit-image',
        uninstall_first=['scikit-image']
    )
    import skimage

was_conf = getSuiteConfig()

# Suppress warnings
if was_conf.__contains__('suppress_uncomfy_warnings'):
    if was_conf['suppress_uncomfy_warnings']:
        import warnings
        warnings.filterwarnings("ignore", category=UserWarning, module="safetensors")
        warnings.filterwarnings("ignore", category=UserWarning, module="torch")
        warnings.filterwarnings("ignore", category=UserWarning, module="transformers")

# Well we got here, we're as loaded as we're gonna get.
print(" ".join([cstr("Finished.").msg, cstr("Loaded").green, cstr(len(NODE_CLASS_MAPPINGS.keys())).e
nd, cstr("nodes successfully.").green]))

show_quotes = True
if was_conf.__contains__('show_inspiration_quote'):
    if was_conf['show_inspiration_quote'] == False:
        show_quotes = False
if show_quotes:
    art_quotes = [
        # ARTISTIC INSPIRATION QUOTES
        '\033[93m"Every artist was first an amateur."\033[0m\033[3m - Ralph Waldo Emerson',
        '\033[93m"Art is not freedom from discipline, but disciplined freedom."\033[0m\033[3m - John F.
Kennedy',
        '\033[93m"Art enables us to find ourselves and lose ourselves at the same time."\033[0m\033[3m
- Thomas Merton',
        '\033[93m"Art is the most intense mode of individualism that the world has known."\033[0m\033
[3m - Oscar Wilde',
        '\033[93m"The purpose of art is washing the dust of daily life off our souls."\033[0m\033[3m - Pa
blo Picasso',
        '\033[93m"Art is the lie that enables us to realize the truth."\033[0m\033[3m - Pablo Picasso',
        '\033[93m"Art is not what you see, but what you make others see."\033[0m\033[3m - Edgar Dega
s',
        '\033[93m"Every artist dips his brush in his own soul, and paints his own nature into his picture
s."\033[0m\033[3m - Henry Ward Beecher',
        '\033[93m"Art is the stored honey of the human soul."\033[0m\033[3m - Theodore Dreiser',
        '\033[93m"Creativity takes courage."\033[0m\033[3m - Henri Matisse',
        '\033[93m"Art should disturb the comfortable and comfort the disturbed." - Cesar Cruz',
        '\033[93m"Art is the most beautiful of all lies."\033[0m\033[3m - Claude Debussy',
        '\033[93m"Art is the journey of a free soul."\033[0m\033[3m - Alev Oguz',
        '\033[93m"The artist\'s world is limitless. It can be found anywhere, far from where he lives or a fe
w feet away. It is always on his doorstep."\033[0m\033[3m - Paul Strand',
        '\033[93m"Art is not a thing; it is a way."\033[0m\033[3m - Elbert Hubbard',
        '\033[93m"Art is the lie that enables us to recognize the truth."\033[0m\033[3m - Friedrich Nietzs
che',
        '\033[93m"Art is the triumph over chaos."\033[0m\033[3m - John Cheever',
        '\033[93m"Art is the lie that enables us to realize the truth."\033[0m\033[3m - Pablo Picasso',
        '\033[93m"Art is the only way to run away without leaving home."\033[0m\033[3m - Twyla Tharp',
        '\033[93m"Art is the most powerful tool we have to connect with the world and express our individ
```

```
uality."\033[0m\033[3m - Unknown',
        '\033[93m"Art is not about making something perfect, it\'s about making something meaningfu
l."\033[0m\033[3m - Unknown',
        '\033[93m"Art is the voice of the soul, expressing what words cannot."\033[0m\033[3m - Unknow
n',
        '\033[93m"Art is the bridge that connects imagination to reality."\033[0m\033[3m - Unknown',
        '\033[93m"Art is the language of the heart and the window to the soul."\033[0m\033[3m - Unkno
wn',
        '\033[93m"Art is the magic that brings beauty into the world."\033[0m\033[3m - Unknown',
        '\033[93m"Art is the freedom to create, explore, and inspire."\033[0m\033[3m - Unknown',
        '\033[93m"Art is the mirror that reflects the beauty within us."\033[0m\033[3m - Unknown',
        '\033[93m"Art is the universal language that transcends boundaries and speaks to all."\033[0m\0
33[3m - Unknown',
        '\033[93m"Art is the light that shines even in the darkest corners."\033[0m\033[3m - Unknown',
        '\033[93m"Art is the soul made visible."\033[0m\033[3m - George Crook',
        '\033[93m"Art is the breath of life."\033[0m\033[3m - Liza Donnelly',
        '\033[93m"Art is a harmony parallel with nature."\033[0m\033[3m - Paul Cézanne',
        '\033[93m"Art is the daughter of freedom."\033[0m\033[3m - Friedrich Schiller',
        # GENERAL INSPIRATION QUOTES
        '\033[93m"Believe you can and you\'re halfway there."\033[0m\033[3m - Theodore Roosevelt',
        '\033[93m"The only way to do great work is to love what you do."\033[0m\033[3m - Steve Jobs',
        '\033[93m"Success is not final, failure is not fatal: It is the courage to continue that counts."\033[0
m\033[3m - Winston Churchill',
        '\033[93m"Your time is limited, don\'t waste it living someone else\'s life."\033[0m\033[3m - Steve
Jobs',
        '\033[93m"The future belongs to those who believe in the beauty of their dreams."\033[0m\033[3
m - Eleanor Roosevelt',
        '\033[93m"Success is not the key to happiness. Happiness is the key to success."\033[0m\033[3
m - Albert Schweitzer',
        '\033[93m"The best way to predict the future is to create it."\033[0m\033[3m - Peter Drucker',
        '\033[93m"Don\'t watch the clock; do what it does. Keep going."\033[0m\033[3m - Sam Levenso
n',
        '\033[93m"Believe in yourself, take on your challenges, and dig deep within yourself to conquer fe
ars."\033[0m\033[3m - Chantal Sutherland',
        '\033[93m"Challenges are what make life interesting and overcoming them is what makes life mea
ningful."\033[0m\033[3m - Joshua J. Marine',
        '\033[93m"Opportunities don\'t happen. You create them."\033[0m\033[3m - Chris Grosser',
        '\033[93m"Your work is going to fill a large part of your life, and the only way to be truly satisfied i
s to do what you believe is great work."\033[0m\033[3m - Steve Jobs',
        '\033[93m"The harder I work, the luckier I get."\033[0m\033[3m - Samuel Goldwyn',
        '\033[93m"Don\'t be pushed around by the fears in your mind. Be led by the dreams in your hear
t."\033[0m\033[3m - Roy T. Bennett',
        '\033[93m"Believe in yourself, and the rest will fall into place."\033[0m\033[3m - Unknown',
        '\033[93m"Life is 10% what happens to us and 90% how we react to it."\033[0m\033[3m - Charle
s R. Swindoll',
        '\033[93m"Success is not just about making money. It\'s about making a difference."\033[0m\033
[3m - Unknown',
        '\033[93m"The only limit to our realization of tomorrow will be our doubts of today."\033[0m\033
[3m - Franklin D. Roosevelt',
        '\033[93m"Great minds discuss ideas; average minds discuss events; small minds discuss peopl
e."\033[0m\033[3m - Eleanor Roosevelt',
        '\033[93m"The future depends on what you do today."\033[0m\033[3m - Mahatma Gandhi',
        '\033[93m"Don\'t be afraid to give up the good to go for the great."\033[0m\033[3m - John D. Roc
kefeller',
        '\033[93m"Success usually comes to those who are too busy to be looking for it."\033[0m\033[3
m - Henry David Thoreau',
        '\033[93m"The secret to getting ahead is getting started."\033[0m\033[3m - Mark Twain',
        '\033[93m"Every great dream begins with a dreamer."\033[0m\033[3m - Harriet Tubman',
        '\033[93m"Do not wait for the opportunity. Create it."\033[0m\033[3m - George Bernard Shaw',
        '\033[93m"Your time is now. Start where you are and never stop."\033[0m\033[3m - Roy T. Benne
tt',
        '\033[93m"The only person you should try to be better than is the person you were yesterday."\03
3[0m\033[3m - Unknown',
        '\033[93m"Success is not in what you have, but who you are."\033[0m\033[3m - Bo Bennett',
        '\033[93m"Do one thing every day that scares you."\033[0m\033[3m - Eleanor Roosevelt',
        '\033[93m"Failure is the opportunity to begin again more intelligently."\033[0m\033[3m - Henry F
ord',
        '\033[93m"Dream big and dare to fail."\033[0m\033[3m - Norman Vaughan',
        '\033[93m"Everything you\'ve ever wanted is on the other side of fear."\033[0m\033[3m - George
Addair',
        '\033[93m"Believe you deserve it and the universe will serve it."\033[0m\033[3m - Unknown',
        '\033[93m"Don\'t wait. The time will never be just right."\033[0m\033[3m - Napoleon Hill',
        '\033[93m"The distance between insanity and genius is measured only by success."\033[0m\033
[3m - Bruce Feirstein',
        '\033[93m"Be the change that you wish to see in the world."\033[0m\033[3m - Mahatma Gandhi',
        '\033[93m"Success is not about being better than someone else. It\'s about being better than you
used to be."\033[0m\033[3m - Unknown',
        '\033[93m"The best revenge is massive success."\033[0m\033[3m - Frank Sinatra',
        '\033[93m"You have within you right now, everything you need to deal with whatever the world ca
n throw at you."\033[0m\033[3m - Brian Tracy',
        '\033[93m"Don\'t let yesterday take up too much of today."\033[0m\033[3m - Will Rogers',
        '\033[93m"The biggest risk is not taking any risk. In a world that is changing quickly, the only strat
egy that is guaranteed to fail is not taking risks."\033[0m\033[3m - Mark Zuckerberg',
        '\033[93m"The journey of a thousand miles begins with one step."\033[0m\033[3m - Lao Tzu',
        '\033[93m"Every strike brings me closer to the next home run."\033[0m\033[3m - Babe Ruth',
    ]
    print(f'\n\t\033[3m{random.choice(art_quotes)}\033[0m\n')
```

MeshGraphormer-DepthMapPreprocessor (HandFix)

```
pip uninstall transformers
pip install transformers==4.47
```

```
pip install yacs
```

간소화 스크립트

llama cpp 설치와 poetry 설치 및 경로 설정 스크립트

```
#!/bin/bash

# CMAKE_ARGS 설정하여 llama-cpp-python 설치
echo "Installing llama-cpp-python with CUDA support..."
CMAKE_ARGS="-DGGML_CUDA=on" pip install llama-cpp-python

# Poetry 설치
echo "Installing Poetry..."
curl -sSL https://install.python-poetry.org | python3 -

# PATH 설정
```

```
echo "Updating PATH for Poetry..."
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc

echo "Installation complete. You may need to restart your shell."
```

```
chmod +x script.sh
./script.sh
```

4. 빌드 및 실행

- FE/dev에 머지 후 승인이 되면, test-server에 Front-end가 배포됩니다.
- BE/dev에 머지 후 승인이 되면, test-server에 Back-end가 배포됩니다.
- main에 머지 후 승인이 되면, main-server에 main의 Front-end와 Back-end가 배포됩니다.