Render a video with Blender on many GPUs or CPUs in parallel

View on GitHub

This example shows how you can render an animated 3D scene using Blender's Python interface.

You can run it on CPUs to scale out on one hundred containers or run it on GPUs to get higher throughput per node. Even for this simple scene, GPUs render 10x faster than CPUs.

The final render looks something like this:

Defining a Modal app

```
from pathlib import Path

import modal
```

Modal runs your Python functions for you in the cloud. You organize your code into apps, collections of functions that work together.

```
app = modal.App("examples-blender-video")
```

We need to define the environment each function runs in — its container image. The block below defines a container image, starting from a basic Debian Linux image adding Blender's system-level dependencies and then installing the `bpy` package, which is Blender's Python API.

```
rendering_image = (
    modal.Image.debian_slim(python_version="3.11")
    .apt_install("xorg", "libxkbcommon0")  # X11 (Unix GUI) dependencies
    .pip_install("bpy==4.1.0")  # Blender as a Python package
)
```

Rendering a single frame

We define a function that renders a single frame. We'll scale this function out on Modal later.

Functions in Modal are defined along with their hardware and their dependencies. This function can be run with GPU acceleration or without it, and we'll use a global flag in the code to switch between the two.

```
WITH_GPU = True  # try changing this to False to run rendering massively in parallel on CP
```

We decorate the function with `@app.function` to define it as a Modal function. Note that in addition to defining the hardware requirements of the function, we also specify the container image that the function runs in (the one we defined above).

The details of the scene aren't too important for this example, but we'll load a .blend file that we created earlier. This scene contains a rotating Modal logo made of a transmissive ice-like material, with a generated displacement map. The animation keyframes were defined in Blender.

```
@app.function(
    gpu="A10G" if WITH_GPU else None,
    # default limits on Modal free tier
    concurrency_limit=10 if WITH_GPU else 100,
    image=rendering_image,
)
def render(blend_file: bytes, frame_number: int = 0) -> bytes:
    """Renders the n-th frame of a Blender file as a PNG."""
    import bpy

    input_path = "/tmp/input.blend"
    output_path = f"/tmp/output-{frame_number}.png"
```

```
    # Blender requires input as a file.
    Path(input_path).write_bytes(blend_file)

    bpy.ops.wm.open_mainfile(filepath=input_path)
    bpy.context.scene.frame_set(frame_number)
    bpy.context.scene.render.filepath = output_path
    configure_rendering(bpy.context, with_gpu=WITH_GPU)
    bpy.ops.render.render(write_still=True)

    # Blender renders image outputs to a file as well.
    return Path(output_path).read_bytes()
```

Rendering with acceleration

We can configure the rendering process to use GPU acceleration with NVIDIA CUDA. We select the Cycles rendering engine, which is compatible with CUDA, and then activate the GPU.

```
def configure_rendering(ctx, with_gpu: bool):
    # configure the rendering process
    ctx.scene.render.engine = "CYCLES"
    ctx.scene.render.resolution_x = 3000
    ctx.scene.render.resolution_y = 2000
    ctx.scene.render.resolution_percentage = 50
    ctx.scene.cycles.samples = 128

    cycles = ctx.preferences.addons["cycles"]

    # Use GPU acceleration if available.
    if with_gpu:
        cycles.preferences.compute_device_type = "CUDA"
        ctx.scene.cycles.device = "GPU"

        # reload the devices to update the configuration
        cycles.preferences.get_devices()
        for device in cycles.preferences.devices:
            device.use = True

    else:
        ctx.scene.cycles.device = "CPU"

    # report rendering devices -- a nice snippet for debugging and ensuring the accelerato
    for dev in cycles.preferences.devices:
        print(
            f"ID:{dev['id']} Name:{dev['name']} Type:{dev['type']} Use:{dev['use']}"
        )
```

Combining frames into a video

Rendering 3D images is fun, and GPUs can make it faster, but rendering 3D videos is better! We add another function to our app, running on a different, simpler container image and different hardware, to combine the frames into a video.

```
combination_image = modal.Image.debian_slim(python_version="3.11").apt_install(
    "ffmpeg"
)
```

The video has a few parameters, which we set here.

```
FPS = 60
FRAME_COUNT = 250
FRAME_SKIP = 1  # increase this to skip frames and speed up rendering
```

The function to combine the frames into a video takes a sequence of byte sequences, one for each rendered frame, and converts them into a single sequence of bytes, the MP4 file.

```
@app.function(image=combination_image)
def combine(frames_bytes: list[bytes], fps: int = FPS) -> bytes:
    import subprocess
    import tempfile

    with tempfile.TemporaryDirectory() as tmpdir:
        for i, frame_bytes in enumerate(frames_bytes):
            frame_path = Path(tmpdir) / f"frame_{i:05}.png"
            frame_path.write_bytes(frame_bytes)
        out_path = Path(tmpdir) / "output.mp4"
        subprocess.run(
            f"ffmpeg -framerate {fps} -pattern_type glob -i '{tmpdir}/*.png' -c:v libx264
            shell=True,
        )
        return out_path.read_bytes()
```

Rendering in parallel in the cloud from the comfort of the command line

With these two functions defined, we need only a few more lines to run our rendering at scale on Modal.

First, we need a function that coordinates our functions to `render` frames and `combine` them. We decorate that function with `@app.local_entrypoint` so that we can run it with `modal run blender_video.py`.

In that function, we use `render.map` to map the `render` function over the range of frames, so that the logo will spin in the final video.

We collect the bytes from each frame into a `list` locally and then send it to `combine` with `.remote`.

The bytes for the video come back to our local machine, and we write them to a file.

The whole rendering process (for 4 seconds of 1080p 60 FPS video) takes about five minutes to run on 10 A10G GPUs, with a per-frame latency of about 10 seconds, and about five minutes to run on 100 CPUs, with a per-frame latency of about one minute.

```
@app.local_entrypoint()
def main():
    output_directory = Path("/tmp") / "render"
    output_directory.mkdir(parents=True, exist_ok=True)

    input_path = Path(__file__).parent / "IceModal.blend"
    blend_bytes = input_path.read_bytes()
    args = [
        (blend_bytes, frame) for frame in range(1, FRAME_COUNT + 1, FRAME_SKIP)
    ]
    images = list(render.starmap(args))
    for i, image in enumerate(images):
        frame_path = output_directory / f"frame_{i + 1}.png"
        frame_path.write_bytes(image)
        print(f"Frame saved to {frame_path}")

    video_path = output_directory / "output.mp4"
    video_bytes = combine.remote(images)
    video_path.write_bytes(video_bytes)
    print(f"Video saved to {video_path}")
```