Mystery App: OSPRay Studio

Introduction

Welcome to the IndySCC 2023 Mystery Application! You will be using Intel's open-source ray tracing engine, OSPRay, and its sibling application, OSPRay Studio, to produce high-fidelity renders of large volumetric datasets.

https://www.ospray.org/

https://www.ospray.org/ospray_studio/

What you're going to be doing

- Installing and compiling OSPRay Studio and its dependencies
- Configuring the run
- Run the render job with a timer
- Look at the pretty picture!

Things you can optimize

- Compilation options
- Runtime options
  - Samples per pixel
  - Image resolution
  - Denoising
- Number of processes and nodes

Overall goals

- Render speed
- Image quality
- Stretch goals

Setup

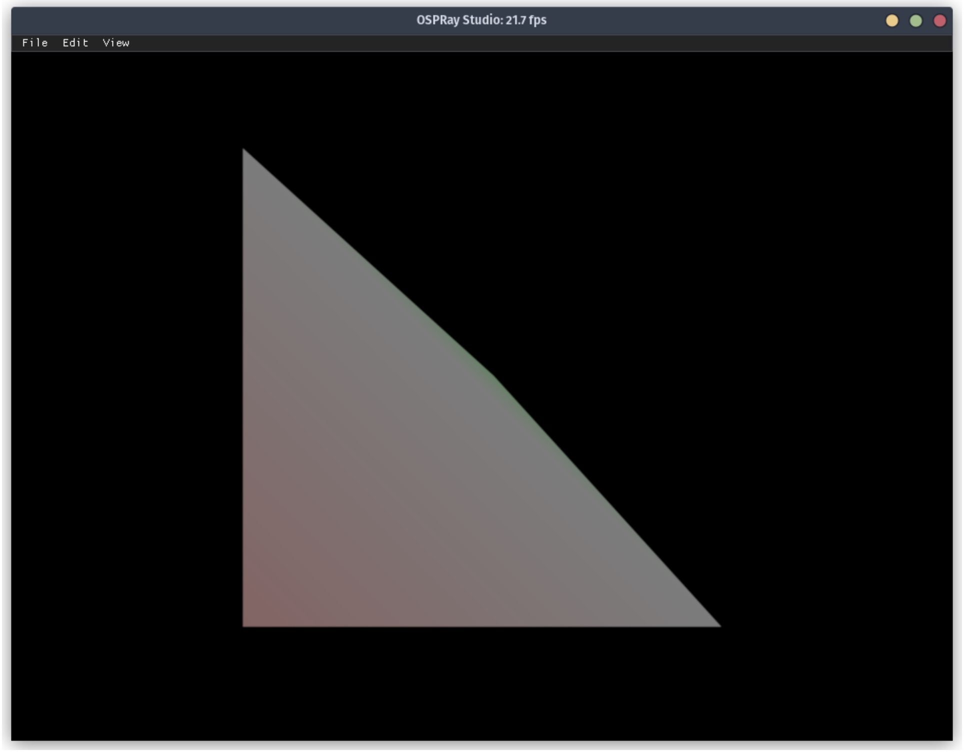
OSPRay Studio GUI

Download Studio from https://github.com/ospray/ospray_studio. You'll start with the binary release to become familiar with the application, but soon you will need to build your own copy.

To verify that things are working correctly, try to run the GUI application.

```
$ ./ospStudio
```

You will need to connect with X forwarding or VNC to see the graphical window. When it opens you can use the File menu to load a demo scene. You should see something like this:



Submit a screenshot of the GUI with one of the demo scenes loaded. Feel free to experiment with any options in the menus to change the render options.

Batch mode

Now you can try running Studio in batch mode. First, download a sample GLTF model from https://github.com/KhronosGroup/glTF-Sample-Models. Then run a command like this:

```
$ ./ospStudio batch --image ~/tmp/box ~/Downloads/Box.glb
```

This will load Box.glb, create a default render and save it to a file named box.00000.png in ~/tmp.

Submit your output image and a text file containing the command you used to create it and the command's output.

Measure render time

We will use the bash command "time" to measure render times. You can use the syntax below to redirect the output to a file.

```
$ { time my-command ; } 2> ~/tmp/timer.txt
```

Re-run your batch command from above with the timer and submit the timing file. **[Some number of points.] [Should we also capture the command and its output?]**

Build OSPRay Studio

Now that you're familiar with Studio's basic capabilities, you will now need to build it yourself. It is up to you whether you want to build its dependencies (OSPRay, OpenVDB, etc) or use binaries. Be sure that your build supports VDB files and MPI.

Tips for building

For the competition, we suggest you use v0.12.1 of OSPRay Studio. Version 0.13 was recently released but has not been extensively tested by your app experts.

OSPRay Studio builds with CMake, as documented in the online instructions for the build at https://github.com/ospray/ospray_studio. However, MPI and OpenVDB support are not enabled by default. Further, there is some code in the MPI-supported codebase that seems to require C++17 standards. So, when running "cmake" at the command line, you will need to enable these features with the "-D" option as follows. The instructions will suggest that you download a copy of the code base from GitHub, change directory into the directory that makes, make a "build" directory, and change into that directory. It is there that one uses "cmake" to set up Makefiles to do the actual compile and linking steps. This is where you'll need to add to the "cmake .." command. The steps will look like this:

```
wget https://github.com/ospray/ospray_studio/archive/refs/tags/v0.12.1.tar.gz
tar -zxvf v0.12.1.tar.gz
ln –s ospray_studio-0.12.1 ospray_studio
cd ospray_studio
mkdir build && cd build
cmake -DCMAKE_CXX_STANDARD:STRING=17 -DUSE_MPI:BOOL=ON -
DENABLE_OPENVDB:BOOL=ON ..
```

From there you can issue a "make" command to build the source. Some external codes will be downloaded to be built as part of this process, such as OSPRay itself. This is expected.

You may run into a situation where errors are emitted for two source files (likely one at a time), where the compiler will be confused about which of two versions of the overloaded "`clamp()`" command it should use. Since we've had to use C++17 for other reasons, and this function in this the **std** namespace for C++17, you can edit these two files to specify which namespace to use. The files and lines on which to make the change from "`clamp`" to "`std::clamp`" are:

- ospray_studio/sg/importer/OBJ.cpp
    - Lines 68-70: 3 instances
- ospray_studio/app/MainWindow.cpp
    - Lines 528-531: 4 instances
    - Lines 534-535: 2 instances

After making these changes, the build should proceed smoothly. Also note, you can speed up the make process with the "`–j`" option followed by the number of concurrent tasks to run. A suggested make command would be "``make –j `nprocs` ``", where "`` `nprocs` ``" results in the number of processor cores available on your build system. Note the backquotes. They are important there.

MPI implementation

Your choice of MPI implementation may be of particular interest. OSPRay is an Intel product, and its documentation consequently recommends using Intel MPI. Is that really better in your case than something like OpenMPI, or is it just a bit of corporate synergy?

Submit a document discussing your choices here. Making the absolute best choices is less important than being able to justify the choices you do make. Also submit the logs generated by your build commands.

VDB support

Let's test your build's VDB support by loading a file. We will use a set of volumetric clouds provided by Intel in the VDB format. They can be downloaded from https://dpel.aswf.io/intel-cloud-library/.

You can now test your build's VDB support by loading one of the small cloud files.

```
$ ./ospStudio batch --image ~/tmp/cloud
~/Downloads/IntelVolumetricClouds/intelCloudLib_sparse.0.S.vdb
```

It will render an image of a cloud using a rainbow color map to show the density.

MPI support

Let's test your build's MPI support. Studio can use OSPRay's offload rendering to distribute rendering tasks across a cluster with MPI. You can do so using the "mpirun" command and by adding the "--mpi" parameter to your Studio command.

```
$ mpirun –n <N> ./ospStudio batch --mpi --image ~/tmp/cloud_mpi
~/Downloads/IntelVolumetricClouds/intelCloudLib_sparse.0.S.vdb
```

Try this for a few values of <N>. Save and submit the console output of one such MPI job.

If everything works, then you're ready to move on to the bigger tasks!

Testing render performance

Creating images with a high-performance renderer is all about balancing image quality and computational effort. In our case, the image quality is mostly dependent on the following things:

- The input dataset
- The output image resolution
- The samples per pixel

Input

For your renders, you will use the largest of the Intel clouds, intelCloudLib_dense.0.L.vdb. It defines a grid of about 8.5 billion cells, which occupies a file size of about 7 GB. Instead of loading the file directly, as you have done so far, you will load a .sg file which contains many settings for the 3D scene. This file should be available for download from the same place you downloaded these instructions. You will need to open this file in a text editor and update the path of the VDB file to match its location on your machine.

Output

Raytracing performance is highly dependent on the number of pixels in the render view. We will use the size of the highest-resolution TVs commonly available today: 8k or 7680px by 4320px. You can specify the image resolution in your command by using the "--resolution" parameter. In our case, you should use "--resolution 8k" or "--resolution 7680x4320".

For quick tests, you may want to reduce the resolution. If left unspecified, the resolution will default to 1024x768. You can use an even smaller resolution if you wish with something like “--resolution 640x480”.

Samples per pixel

Due to its non-deterministic algorithm, the pathtracing renderer can create images containing a lot of noise. To reduce the noise, it is necessary to increase the number of rays cast into the scene. This can be done using the “--spp” command-line parameter, which stands for “samples per pixel”.

Task 1: Scale up and optimize

The goal here will be to optimize the execution of a single render job. The job will have the following parameters:

- Input: studio_scene.sg
- Output: An 8k resolution image
- SPP: 64

The major avenues available to you to optimize this job are:

- Compilation options for Studio and its dependencies
- MPI implementation choice and options
- Number of nodes and processes

Record and submit timing files for your attempts. If you perform new builds, save and submit the build logs. Submit one of the images.

Create a table of your methods and their render times. Create a bar chart of this table using the visualization tool of your choice (Excel, Python, web tools like https://www.rawgraphs.io/, etc.).

Submit a document describing your strategy to optimize the render speed and the ultimate winning method. Be sure to include any changes to your original build strategy and why you made them. Is there a point at which increasing the number of nodes and processes no longer improves render times?

Task 2: Increase samples per pixel

Now that you have settled on a good way to run render jobs, let’s apply it to increasingly heavy workloads. Starting with 1 sample per pixel, produce a run of images where you double the SPP each time, up to 8192 samples. Other than SPP, use the same settings as in Task 1.

Record and submit timing files for these jobs, plus the images they create. Create a table showing the render times for each value of SPP. Create a line chart showing how render time changes with SPP using the visualization tool of your choice.

Review your images and submit a document discussing what you think the right balance is between SPP and render time. At what point is the perceived image quality not improved by increasing the SPP? Include comparison images to justify your opinion.

Stretch goals

These are extra tasks to perform if you have time. They aren't worth as many points as the main tasks, but they might make the difference for the winning team!

Denoising

Another way to reduce noise is to use OSPRay's support for Open Image Denoise. This postprocessing technique can compensate for low numbers of samples per pixel. You can enable the denoiser with the "--denoiser" command-line parameter.

Experiment with enabling the denoiser for low SPP values and compare the output to your images with much higher SPP. Are the denoised images good enough to skip high-SPP renders in the future? Are they faster to render? Submit a document with your assessment of low SPP + denoiser versus high SPP. Include comparison images in your document to justify your opinion. Also submit your denoised renders.

Data visualization

Each of the tasks asks you to create a simple data visualization. Are there any other forms of visualization which might be useful for making decisions about these render job strategies? Use the tool of your choice and get creative! Submit one or more visualizations and a document discussing why they are helpful.

Render settings

Studio has many settings for changing how the render looks. Use the GUI to experiment with things like Lights, Transfer Functions, Framebuffer settings, Renderer settings, and Camera settings. Save and submit your favorite image from the GUI. If you want to render at even higher quality, you can save a .sg Scene file to capture those settings and render it in batch mode on the cluster.

Submit a document describing your image and the choices required to achieve it.

Submission

Your submission should include the following items. Compress all your files into a .zip or .tar.gz file. Include a document describing your filenames and file hierarchy so that we can figure out which files go with which tasks.

1) Setup **[25 points]**
    a) Running Studio **[5 points]**
        i) A screenshot of Studio running in GUI mode
        ii) An image produced by Studio in batch mode and a text file showing the command and its output.
        iii) The output file from running a batch job with the time command.
    b) Building Studio **[20 points]**
        i) The logs from building Studio (and possibly its dependencies), plus a document describing your choices when producing a build.
        ii) An image of a cloud demonstrating that your build allows you to load VDB files.

        iii) A document containing the console output demonstrating that your build supports MPI.

2) Task 1 **[35 points]**
    a) Timing files for your attempts
    b) A document describing your attempts and optimization strategy
    c) Logs for any new builds
    d) An image from one of your attempts
    e) A table of your timings
    f) A bar chart visualization of that table
3) Task 2 **[20 points]**
    a) Timing files and images
    b) A table of the timings
    c) A line graph visualization of the table
    d) A document discussing the balance between image quality and computation time.
4) Stretch goal: denoising **[10 points]**
    a) Denoised images
    b) A document discussing low SPP + denoiser versus high SPP.
5) Stretch goal: data visualization **[5 points]**
    a) Alternative visualizations
    b) A document discussing why they may be helpful
6) Stretch goal: render settings **[5 points]**
    a) Your favorite image
    b) A document describing how it was created