Simple Ray Casting

In this assignment we will implement the simplest form of the ray casting algorithm to display, for each pixel, the color of the surface closest to it along the camera ray's direction.

Prepare your home machine

You can work on the assignments exclusively in the Gralab, if you want to work at home you might have to install additional packages. To install them on a debian-based system you can use

```
$ sudo apt install build-essential pkg-config imagemagick libmagickwand-dev libassimp-dev libglm-dev libpng++-dev
```

Setting Up (Gralab)

The tarball `rtgi-2021-a01.tar.bz2`, available on GRIPS, holds the code for this assignment. Grab it and unpack it (e.g. via `tar xfj rtgi-2021-a01.tar.bz2`).

You can then enter the unpacked directory and configure the source code for your environment:

```
$ ./configure
```

If this should report any error, let us know and include the terminal output (as plain text). If you run into problems on your own machine, please make sure you followed the above instructions.

You can then compile

```
$ make -j9
```

(The number after `-j` is the number of files to compile in parallel, choose as it fits to your system).

The result of this will be an executable program, `./rtgi`, that you can run with a script file:

```
$ ./rtgi -s scripts/a1-tri
```

For more program options see `./rtgi --help`, note that `-l` ('ell') instead of `-s` will load the commands from the specified script but not terminate after executing them. This way you can enter additional commands. This provides a somewhat limited form of interactive working environment.

Compilation Flags

Configuration is also responsible for defining compilation flags to use later on, with nothing further specified we compile a somewhat optimized and somewhat debuggable variant. To get a really optimized build you can run `./configure CXXFLAGS="-O4"`, to get a version that is easy to analyze in a debugger you can run `./configure CXXFLAGS="-O0 -ggdb3"` (that is 'Oh zero'). For Debugging we suggest using `cgdb` (or regular `gdb` where you can type `C-x a` to also see your code).

Assignment 1

Have a look at `scripts/a1-tri`, you will see that the first few lines define the camera's position and orientation. Also note the `resolution` command. Follow these commands in `driver/interaction.cpp`.

Then, follow the commands `raytracer` and `commit` and see how the latter initializes the ray tracer, in our case `seq_tri_is`.

Finally, observe that `run` triggers `primary_hit_display::sample_pixel`. Read the documentation (also, cf. `gi/algorithm.h`) and start implementing `primary_hit_display::sample_pixel` by setting up a camera ray through the center of the current pixel. This also entails implementing the function `cam_ray` in `gi/camera.cpp` (cf. the respective header file as well). The `offset` parameter that defines a position inside the pixel for anti-aliasing, can, for now, be set to `glm::vec2(0)`.

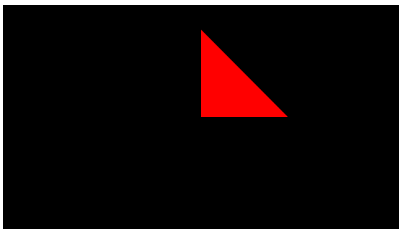
You can test your ray setup from within `main` or `run` (both in `driver/main.cpp` by calling `test_camrays` and examining the resulting `test.obj` file in Blender.

Note: 3D tools often have their own interpretation of what the coordinate system should look like (or make this configurable in the import process), if you are skeptical, pay close attention to those setting and later, when intersecting with a model, just import that model as well to be sure.

Assignment 2

The next step towards a functioning ray caster is then to implement ray/triangle intersection. To this end, extend the function `intersect` (in `libgi/intersect.h`) as described in the lecture (and in Shirley's book, it is available in the library (I think) and excerpts are online on the GRIPS page).

You can test this function manually by constructing a ray and a triangle yourself and print out the info (or stepping through it with `cgdb` et al). When you are fairly certain that it works you can go ahead and test your real camera rays with the single triangle that is loaded by `scripts/a1-tri`. If you do this in `primary_hit_display::sample_pixel` you can mark pixels that generate camera rays intersecting that triangle with a certain color to validate your implementation. This can be done adding a value to the `result` array, where a single `sample_result` constitutes a color (first part) and a sample location on the pixel (which is not in RTGI currently).
Your results should map to this example:



Assignment 3

You can now combine these two by fully implementing `primary_hit_display::sample_pixel`. To this end, iterate over the requested number of samples, and, for each sample, set up a camera ray. Pass this ray to the ray caster and, if it returns a valid hitpoint (see `triangle_intersection`) return its surface's color (see `diff_geom::diff_geom` and `diff_geom::albedo`, both in `libgi/rt.h`).

The only thing left now is to implement the sequential ray tracer. To this end, open `rt/seq/seq.cpp` and extend `seq_tri_is::closest_hit` such that it iterates over all triangles, intersecting them and keeping the intersection information of the one closest to the observer.

With this all done, running

```
$ ./driver/rtgi -s scripts/a1-tri
```

should generate the image displayed above, and running

```
$ ./driver/rtgi -s scripts/a1-sibenik
```

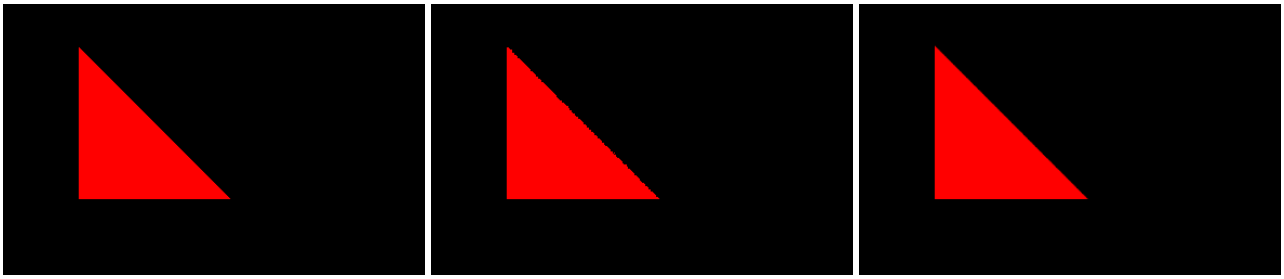
should generate the following image:

Assignment 4

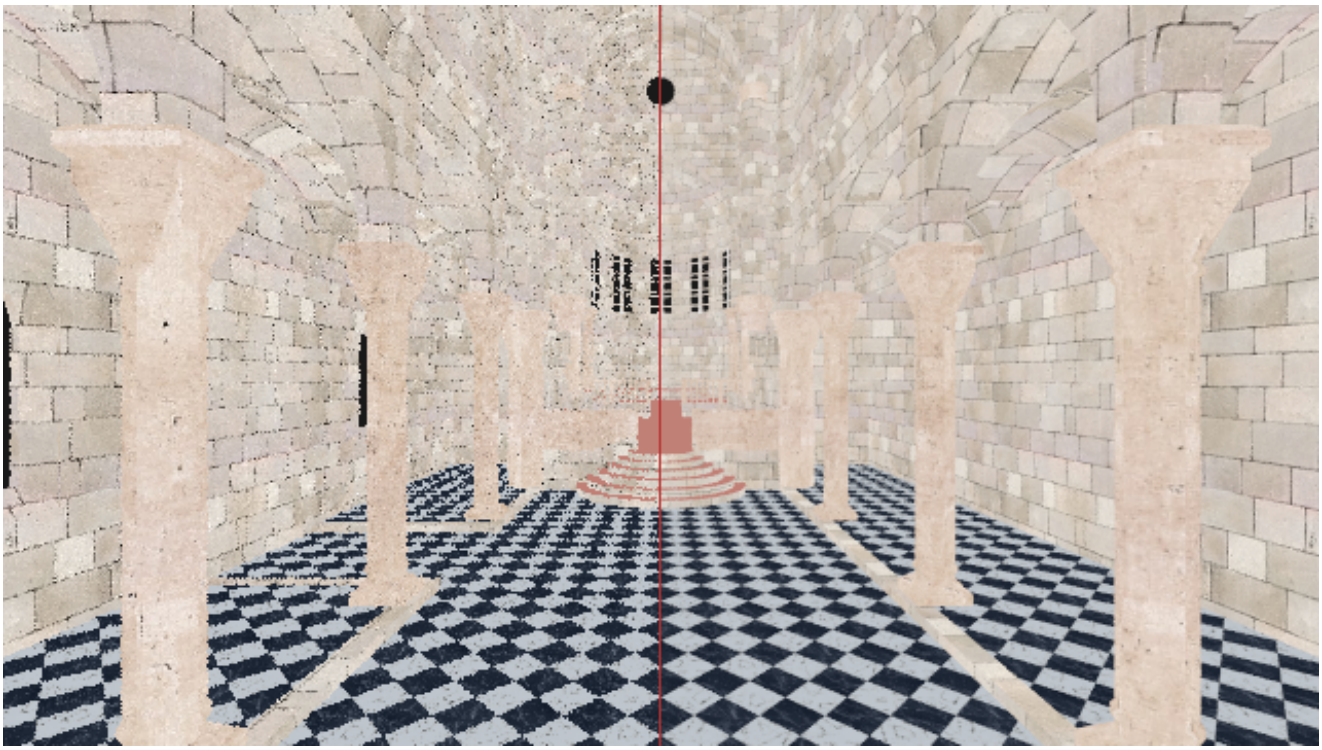
To improve the quality of the renderered images, add anti aliasing by computing multiple samples. The infrastructure for this is already in place, there are only two places to change for this: first, set the `sppx parameter` in the script and, second, fill in the `offset` parameter in the call to `cam_ray`.

Regarding the latter, you can compute uniform random values in $[0, 1)$ from the random number generator that is held in the rendering context.

The following images show the difference in the triangle scene, note how the left triangle has and evenly stepped edge while the center one, also computed at 1 sppx has an irregular diagonal edge. The effect of computing multiple such irregular samples per pixel (in this case 64) is shown in the right-most image. The images are in high resolution, i.e. you can zoom in. The lecture slides show those exact images considerably magnified as well.



The following image shows a comparison in the sibenik cathedral (left 1 sppx, right 16 sppx, also available on the lecture slides):



Note: This took 4 minutes with 16 Threads on my 2019 Ryzen machine.

The results using a single sample are worst for the more distant parts, why could that be? If you heard CG last term, how can we improve this?

Happy Hacking :)