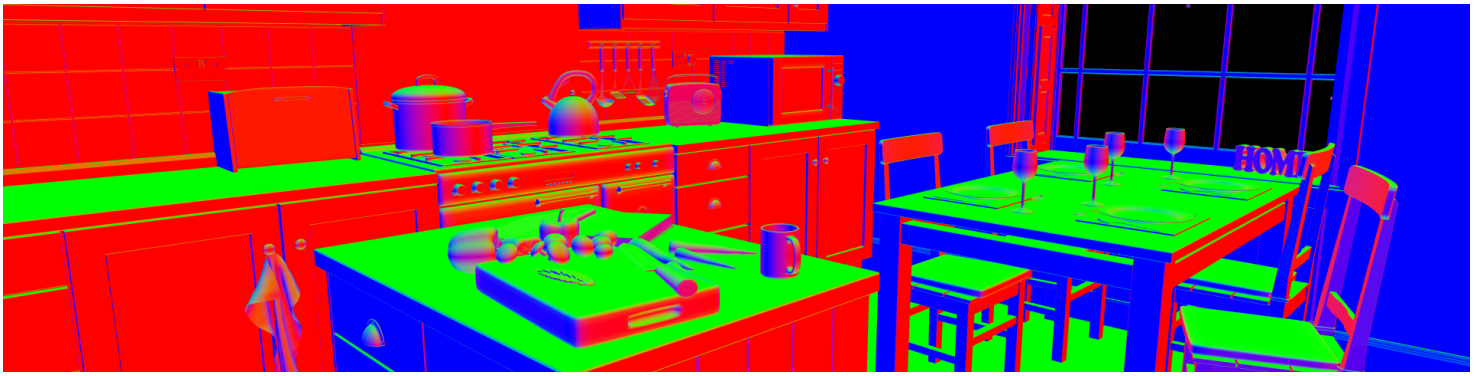
ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 1: Basic Render

Due Sunday, September 22nd, 2019 at 11:59pm EST on myCourses
10% (ECSE 446 / COMP 598) *or* **5%** (ECSE 546 / COMP 599)



In this assignment you will implement both an offline and online **rendering loop**, including algorithms to visualize the surface normal of different 3D meshes.

Contents

Preliminary coding tips

The codebase makes available a handful of convenience variables and objects. Most functions that we ask you to implement are often contained in C-like structures, and so you'll have direct access to the `struct`'s member variables from your implementation. You can find these variables by either exploring the structure declarations in `src/core/core.h` or using autocompletion functionality in your IDE. We briefly review the most important ones, here:

- The `Scene` structure stores all the information necessary to render a scene. Its encapsulates data about the scene geometry, light/emitters, materials, etc. This structure refers to a `Config` sub-structure, which contains configuration settings — such as a pointer to the TOML scene file, the camera position, the image/film dimension, etc. Parameters you may want to pass to your algorithms can be made available through, e.g., the `Config::IntegratorConfig` (in the case of offline rendering algorithms.)
- The `SurfaceInteraction` structure stores information that results from a ray-triangle intersection. It contains the intersection point's coordinates in *world space*, incoming/outgoing ray directions at the intersection in *local space*, the ID of the intersected triangle and the triangle's associated shape object, normal coordinate frames at the intersection point, and so on. This structure is populated and returned after tracing a ray into the scene.
- The `Frame` structure is a simple 3-axis coordinate frame, where *z* is chosen as our canonical up-vector. It contains two functions `toLocal()` and `toWorld()` which can be used to transform between coordinate systems. The `SurfaceInteraction` structure has two frames: `frameNg` is a coordinate frame based on the *geometric normal* and `frameNs` is one based on the *shading normal*. The former is the "true" normal for the triangle faces while the latter is an smoothed normal based on Phong Shading principles (as discussed in lectures).
- The `ThreadPool` structure contains implementations for sequential and parallel for-loops. The `ParallelFor()` function is implemented with `std::thread`, which should be supported on all major platforms. The main purpose of this structure is to allow you to more easily multithread your code — you should only enable multi-threading when you are *not* in *Debug* mode. You can use the following pre-processor trick to switch between single- and multi-threading at compile time:

```
#ifdef NDEBUG // Running in release mode - Use threads
  ThreadPool::ParallelFor(0, scene.config.height, [&] (int y) {
#else // Running in debug mode - Don't use threads
    ThreadPool::SequentialFor(0, scene.config.height, [&](int y) {
#endif
// Your code here
}
```

With these convenience structures in mind, you are ready to move on to actually coding your first task.

1 Offline rendering (*50 pts*)

1.1 Rendering loop (*25 pts*)

Your first task is to implement an offline rendering loop. The function `Renderer::render()` in `src/renderer.cpp` is currently empty: complete it by looping over all pixels on the image plane and computing their color. At a high-level, this is done as follows:

1. calculate the camera perspective, the camera-to-world transformation matrix and the aspect ratio,
2. loop over all pixels on the image plane, and
3. generate a ray through each pixel and *splat* its contribution onto the image plane.

The last step can be broken down into five substeps:

1. retrieve the *pixel center* (x, y),
2. construct a ray with its origin at the camera's center and its direction pointing through the pixel center,
3. transform this ray to world coordinates — taking the FoV and aspect ratio into account,
4. call the integrator on your newly constructed ray, passing in the scene's random number sampler, and
5. collect the returned radiance contribution from the integrator (for the pixel) and output it the image buffer.

Each step is detailed in the tutorial slides for A1. To get started, you can type `scene.config` and let your IDE autocomplete to see what member functions/variables are at your disposal.

When completed correctly, your code should generate a `.exr` file of a solid green image — regardless of which TOML scene file you pass in to your renderer from the `data` folder. You'll also see a console message that's similar to:

```
Found 1 shape
Mesh 0: cube [12 primitives | Diffuse]
BVH built in 1.3e-05s
Saved EXR image to ../data/cube/cube_normal.exr

Process finished with exit code 0
```

Always open your rendered images with an HDR viewer! See tutorial slides for software recommendations.

The next part of this assignment (Part 1.2) will validate your code by rendering an actual scene.

1.2 Uniform pixel supersampling (*15 pts*)

Most physically-based renderers normally sample multiple rays through each pixel and average their contribution in order to, i.e., perform antialiasing. This means that rays don't *have* to always pass through the *centers* of a pixel: they can go through any point in the square pixel. Modify your rendering loop to support multiple samples per pixel:

0. for every samples:
    1. uniformly sample a location on the pixel area, and
    2. generate an eye ray that passes through this location, calling the integrator (as before) and outputting the *average* of the computed radiance contributions.

Retrieve the desired number of samples per pixel (spp) from `scene.config.spp`. Perform uniform sampling whenever this number is larger than one, otherwise fall back to sampling a single ray through the pixel centers.

The `sampler.next()` function is pseudorandom number generator that returns a canonical uniform random value in the interval (0, 1). To test your new implementation, render the cube scene (after having implemented multiple pixel samples) with a value of 16 spp and compare your result with the ground truth image we provide.

> You will need to complete (Part 1.2) in order to test whether uniform sampling works. It may make sense to complete (Part 1.2) before returning to this task.

1.3 Surface normal integrator (*10 pts*)

In TinyRender, the part of an offline rendering algorithm that computes the radiance/color for a ray is referred to as an **integrator** because, later on in the course, these algorithms generally solve a numerical integration problem. The second offline task of this assignment is to modify your *normal integrator* to display the surface normals (in world coordinates) of a mesh. Each triangle of a mesh has a unit normal vector; your goal is to retrieve it and return its value for each intersected triangle. Each of the (x, y, z) Cartesian coordinates of a normal vector can only take on values in the interval $(-1,1)$ which, after applying the absolute value, will allow you visualize normal vectors as RGB colors. You will have to modify `NormalIntegrator::render()`, which currently returns the color green regardless of the ray intersection, to do so:

Normal integrator (`integrators/normal.h`)

```
struct NormalIntegrator : Integrator {
  explicit NormalIntegrator(const Scene &scene) : Integrator(scene) { }

  v3f render(const Ray &ray, Sampler &sampler) const override {
      // TODO: Implement this
      return v3f(0.f, 1.f, 0.f);
  }
};
```

This function is called in your rendering loop, for each eye ray, after you generate eye rays through each pixel (Part 1.1): each of these eye rays calls the `render()` function of the integrator instantiated from the scene file. The ray parameter passed to `render()` is a camera ray through a given pixel; in your `NormalIntegrator`, you will

intersect the scene with this ray to find the closest surface to the ray origin and along its direction — solving the *primary-visibility problem*. If an intersection is found, simply return the component-wise *absolute value* of the **shading normal** (in world coordinates) at the intersection point, as a color. Refer to the `AcceleratorBVH` structure in `src/accel.h` to see how to trace rays against the scene geometry.

To run your renderer and (hopefully) see the result of your work, invoke `tinyrender` on the files

- `data/a1/cube/tinyrender/cube_normal_offline.toml`
- `data/a1/bunny/tinyrender/bunny_normal_offline.toml`

to obtain the images below.

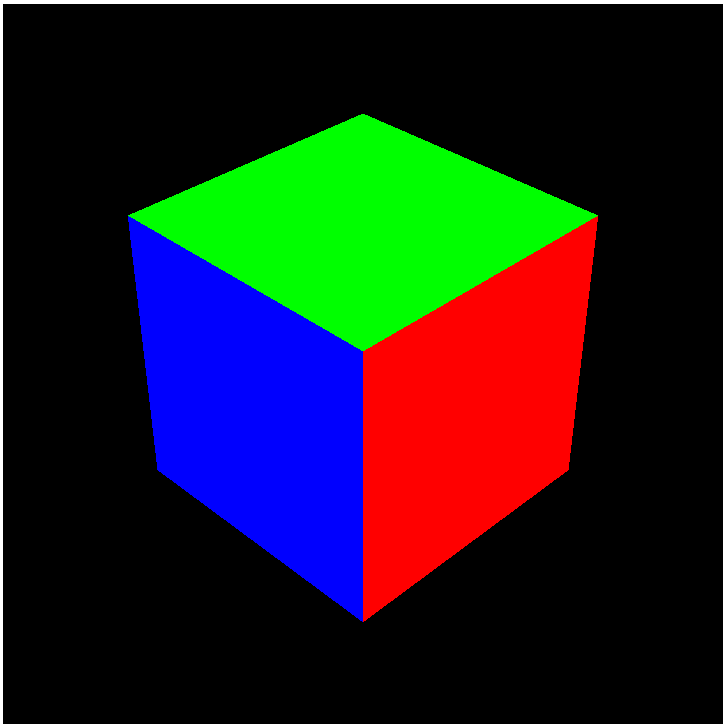


Figure 1: *Cube scene with no background.*

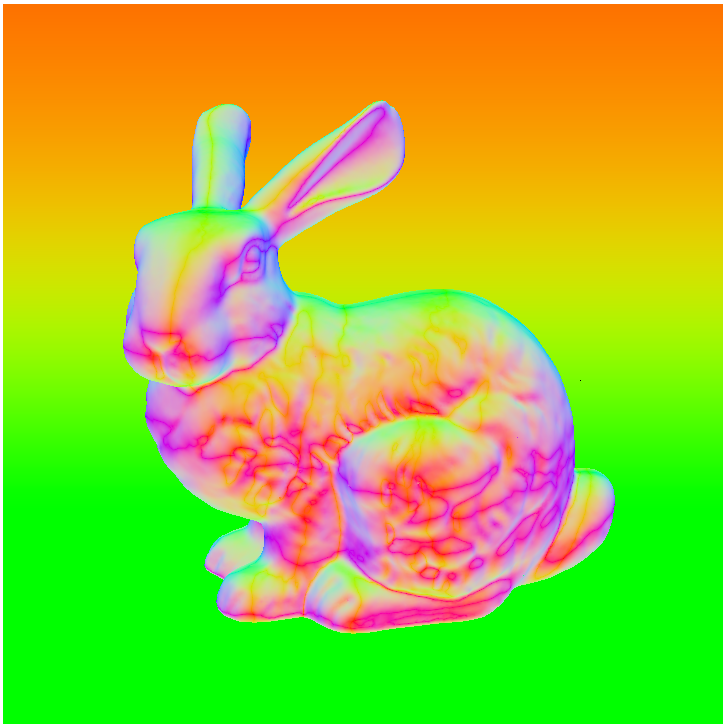


Figure 2: *Stanford bunny scene with a studio backdrop as background.*

2 Real-time rendering (*50 pts*)

2.1 Rendering loop (*30 pts*)

Your second task is to implement an online/interactive rendering loop in the function `Renderer::render()` within `src/renderer.cpp`. You must render the scene interactively, meaning fast enough that movement is fluid, and handle keyboard and mouse events to control the camera. In practice we render at least 60 FPS (frames per second) to give the illusion of a continuous image sequence.

To do so, create an infinite loop in which you:

1. detect and handle the `quit` event to close the window when you click on the `X` button (see SDL documentation),
2. pass any other event to the function `renderpass->updateCamera(SDL_Event)`, in order to handle user control of the camera for you,
3. call the render function using `renderpass->render()`, and
4. output the first image rendered into the GUI window using `SDL_GL_SwapWindow(renderpass->window)`.

The file `src/renderpasses/normal.h` sets up the different stages of the render pass for your interactive shaders. Of note, it compiles and links the shader files, and it creates and binds GL uniforms for the camera settings to these shaders. One part is missing however: a loop that actually performs the draw calls for every object in the scene, with the shaders bound to them:

```
for (auto &object : objects) {
  // TODO: Implement this
}
```

Implement this missing part along with the real-time rendering loop, before moving on to the second part. Try moving the camera around using the `wasd` keys and dragging the mouse cursor.

2.2 Surface normal shader (*20 pts*)

In TinyRender, real-time rendering algorithms are implemented as shaders written in GLSL. GLSL is a high-level shader programming language very similar to C.

A scene is composed of many objects, each with a *vertex buffer object* (VBO) and *vertex array object* (VAO). The VBO and VAO are populated for you during scene initialization, and so you need only worry about implementing drawing and shading logic in this course.

A Vertex Buffer Object is an OpenGL data buffer (using GPU's memory) that is typically used to store vertex attributes, such as the position, normal vector and color of each vertex (of each triangle and of each object) in your scene. The vertex positions are typically prescribed in *object space*. A Vertex Array Object allows us to define the memory layout of the attributes in the VBO. One VBO is assigned to each VAO.

Binding a VBO is the way we pass vertex attributes to a vertex shader. These vertex attributes will then be accessible in the vertex shader, according to a corresponding variable definition:

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
```

You can query the ID assigned to the VBO and VAO of each object using their `GLObject` variables.

As with the offline render loop, complete the implementation of a vertex shader `shaders/normal.vs` and fragment/pixel shader `shaders/normal.fs` in order to display the absolute value of the surface normals (in world coordinates) of the mesh. Most of the heavy lifting necessary to run shaders has already been done, and you need only worry about the actual shading algorithms (implemented in the shaders).

Vertex shader (`shaders/normal.vs`)

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat4 normalMat;

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
out vec3 vNormal;

void main() {
  // TODO: Implement this
}
```

Fragment shader (`shaders/normal.fs`)

```
in vec3 vNormal;
out vec3 color;

void main() {
  // TODO: Implement this
}
```

To run your renderer and (hopefully) see the result of your work, invoke `tinyrender` on the files `data/a1/cube/tinyrender/cube_normal_realtime.toml` and `data/a1/bunny/tinyrender/bunny_normal_realtime.toml`. The resulting images should be exactly the same as the one generated by the offline renderer. Note that if your shader has errors, the renderer won't start and the errors will appear in the console.

OpenGL docmentation is available here.

What to submit

Render all the scenes in the `data/a1` directory. When you're done, edit the `config.json` file (which can be found in the same directory) with your credentials and submit all your code in a `.zip` or `.tar` archive file. Please create a *separate* directory with **only** the files required for submission, and compress this directory — this is different than simply selecting all the files to submit and right-clicking compress. Include your raw `.exr` files in separated folders, according to the structure below:

```
a1_first_last.zip
   config.json
   src/
   offline/
      sphere_normal_offline.exr
      sphere_normal_offline_8spp.exr
      dragon_normal_offline.exr
      dragon_normal_offline_8spp.exr
   realtime/
      sphere_normal_realtime.exr
      dragon_normal_realtime.exr
```

Make sure your code compiles and runs for both online and offline parts before submitting! **Code that does not compile or does not follow this exact tree structure may result in a score of zero**. If your assignment does not compile at the moment of submitting but you do have some correct images from previous iterations, please submit these and explain where you think the problem may lie through the myCourses submission textbox or a `readme.txt`. You can use the `tree` command to verify your tree structure.