ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 1: Build Your Renderer

Due: Sunday, September 15th, 2024 at 11:59pm EST on myCourses

Final weight: **20%**

Contents

1 Assignment Setup and Policies

Start by downloading the `Python` base code from *myCourses*. The assignment base code directory and file structure is as follows:

- `A1_codebase/`
    - `interactive/`
        - `A1.py`
    - `scene_data_dir/`
    - `taichi_tracer/`
        - `camera.py`
        - `environment.py`
        - `geometry.py`
        - `materials.py`
        - `ray_intersector.py`
        - `ray.py`
        - `renderer.py`
        - `sampler.py`
        - `scene_data_loader.py`
        - `scene_data.py`
    - `pyproject.toml`
    - `README.md`

All of the class assignments will rely on taichi, a high performance `Python` library which parallellizes your code for you.

1.1 Development Setup using `Conda`

We recommend using `conda` to manage your `python` environments for this class. You can find all the documentation and installation guides here.

ⓘ **Note for Windows users**: After installing `Anaconda`, a new shell called `Anaconda Prompt` will be available for use; we recommend using this shell instead of PowerShell to run all the commands that follow.

Once you have `conda` installed, create and name a new `conda` environment (here, we chose `taichi_env` as our environment name):

```
$ conda create --name taichi_env python=3.10.14
```

After creation and installation, you can activate your environment using:

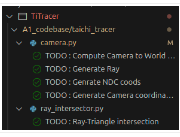
```
$ conda activate taichi_env
```

At this point, you can install the required libraries (run this command from the `A1_codebase/` root folder):

```
$ pip install -e .
```

1.2 Editing and Running Code

For this assignment, you will only be editing **camera.py** and **ray_intersector.py**. The `Python` functions that you will need to complete have been tagged with **`#TODO`** along with a small description. If you are using VSCode as your code editor, you can use extensions such as **TODO Tree** which will give you a concise list of your TODOs along with a link to jump to the code — this is only a suggestion, as you are free to use any code editor you like.


TiTracer
A1_codebase/taichi_tracer
camera.py M
TODO : Compute Camera to World ...
TODO : Generate Ray
TODO : Genrate NDC coods
TODO : Generate Camera coordina...
ray_intersector.py
TODO : Ray-Triangle intersection

***TODO Tree** in VSCode.*

In **`camera.py`** you'll find the `Camera` class, where you will implement **Eye Ray Generation** logic. Once you are able to correctly spawn eye/camera rays, you will need to implement **Ray-triangle Intersection** in the `RayIntersector` class in **`ray_intersector.py`**, in order for your ray tracer to interact with objects in the scene.

To run the code, simply run **A1.py** in the `interactive/` folder (from the root folder):

```
$ python ./interactive/A1.py
```

1.3 Assignment Submission

Gather all the `python` *source files* within the **`taichi_tracer`**`/` folder (i.e., everything *except* the `scene_data_dir/` folder) and compress them into a single `zip` file. Name your `zip` file according to your student ID, as:

`YourStudentID.zip`

For example, if your ID is **234567890**, your submission filename should be **234567890.zip**.

DO NOT ADD ANYTHING BEFORE OR AFTER THE MCGILL ID.

ⓘ Every time you submit a new file on *myCourses*, your previous submission will be overwritten. We will only grade the **final submitted file**, so feel free to submit often as you progress through the assignment.

In accordance with article 15 of the Charter of Students' Rights, students may submit any written or programming components in either French or English.

1.4 Late policy

All the assignments are to completed *individually*. You are expected to respect the late day policy and collaboration/plagiarism polices, discussed below.

☒ **Late Day Allotment and Late Policy**

Every student will be allowed a total of **six (6)** late days during the entire semester, without penalty. Specifically, failure to submit a (valid) assignment on time will result in a late day (rounded **up** to the nearest day) being deducted from the student's late day allotment. Once the late day allotment is exhausted, any further late submissions will obtain a score of **0%**. Exceptional circumstances will be treated as per McGill's Policies on Student Rights and Responsibilities.

If you require an accomodation, please advise McGill Student Accessibility and Achievement (514-398-6009) as early in the semester as possible. In the event of circumstances beyond our control, the evaluation scheme as detailed on the course website and on assignment handouts may require modification.

1.5 Collaboration & Plagiarism

Plagiarism is an academic offense of misrepresenting authorship. This can result in penalties up to expulsion. It is also possible to plagiarise *your own work*, e.g., by submitting work from another course without proper attribution. **When in doubt, attribute!**

You are expected to submit your own work. Assignments are individual tasks. This does not need to preclude forming an environment where you can be comfortable discussing **ideas** with your classmates. **When in doubt, some good rules to follow include**:

- fully understand every step of every solution you submit,
- only submit solution code that was *written* (not copy/pasted/modified, not `ChatGPT`'ed, etc.) by you, and
- never refer to another student's code — if at all possible, we recommend that you avoid *looking* at another classmates code.

McGill values academic integrity and students should take the time to fully understand the meaning and consequences of cheating, plagiarism and other academic offenses (as defined in the Code of Student Conduct and Disciplinary Procedures — see these two links).

☑ Computational plagiarism detection tools are employed as part of the evaluation procedure in ECSE 446/546. Students may only be notified of potential infractions at the end of the semester.

Additional policies governing academic issues which affect students can be found in the Handbook on Student Rights and Responsibilities.

1.6 Installing `Python`, Libraries and Tools

This assignment will be completed in `Python` and using, primarily, the `numpy` and `taichi` libraries.

You are free to install and configure your development environment, including your IDE of choice, as you wish (although we provide a suggested setup, above.

ⓘ Some IDEs will automatically *add code* to your source files; it is your responsibility to review your code before submitting it to ensure it meets the submission specifications.

1.7 `Python` Language and Library Usage Rules

`Python` is a powerful language, with many built-in features. Feel free to explore the base language features and apply them as a convenience. A representative example is that, if you need to sort values in a list, you should feel free to use the built-in `sort` function rather than implementing your own sorting algorithm (although, that's perfectly fine, too!):

```
myFavouritePrimes = [11, 3, 7, 5, 2]

# In ECSE 446/546, learning how to sort a list is NOT a core learning objective
myFavouritePrimes.sort() # 100% OK to use this!

print(myFavouritePrimes) # Output: [2, 3, 5, 7, 11]
```

We will, however, draw exceptions when the use of (typically external) library routines allows you to shortcut through the *core learning objective(s)* of an assignment. When in doubt as to whether a library (or even a built-in) routine is “safe” to use in your solution, please **ask the TA** during the tutorial session.

ⓘ Python 3.x has a built-in convenience `breakpoint()` function which will break code execution into a debugger, where you can inspect variables in the debug REPL and even execute (stateful) code! This is a very powerful way to test your code *as it runs* and to tinker (e.g., inline in the REPL) with function calling conventions and input/output behaviour.

You can additionally/alternatively rely on a debugging framework, e.g., embedded in an IDE.

Be careful, as you can change the execution state (i.e., the debug environment is not isolated from your code's execution stack and heap), if you insert REPL code and then continue the execution of your script from the debugger.

The base code we provide you with includes a superset of all the library `imports` we could imagine you using for the assignment.

☒ Do not include any additional `imports` in your solution, other than those provided by us. Doing so will result in a score of **zero (0%)** on the assignment.

This course will rely *heavily* on `taichi` and, to a lesser extent, `numpy` — in fact, you'll likely learn just as much about the power (and peculiarities) of the `Python` programming language as you will about these libraries. The `taichi` and `numpy` libraries not only provide convenience routines for matrix, vector and higher-order tensor operations, but also allow you to leverage high-performance vectorized operations and parallelization, as well as providing lightweight windowing and user-input functionalities.

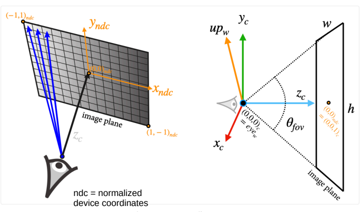
1.8 Let's get started... but let's also not forget...

With these preliminaries out of the way, we can dive into the assignment tasks. **Future assignment handouts will not include these preliminaries, although they will continue to hold true.** Should you forget, they will remain online in this handout for your reference throughout the semester.

2 Eye Ray Generation

Eye rays all originate at the camera location and traverse in directions towards the *center* of pixels on an image plane located one unit away (in world space) from the camera.

In order to form the complete geometric setup necessary to compute the eye ray directions, we require an appropriate coordinate system: i.e., one that is centered at the camera, that orients the **central viewing axis** and the vertical and horizontal **extents** of the (clipped) image viewing plane. The diagrams below complement our discussion of the parameterization of this coordinate system.



NDC and Camera Coordinate Systems.

This assignment will only consider the generation of a *single eye ray* through the **center** of each pixel. In order to obtain the world space origins and directions of each eye ray, we will perform a series of coordinate system transformations:

- Starting from *2D normalized device coordinates* (NDC) on the image plane, we specify the (continuous, per-pixel centered) pixel coordinates such that the corners of the NDC plane lie at $(\pm 1, \pm 1)_{ndc}$[1]. Be sure to shift the 2D pixel coordinate to lie in the center of each stratum, using the width (`self.width`) and height (`self.height`) in the `Camera` class to appropriately discretize the NDC plane. The 2D coordinate axes, x_{ndc} and y_{ndc}, for the NDC plane are illustrated on the left of the diagram, above.

- A perspective camera is parameterized by its:
    - location (position) in world space, eye_w (`self.eye` in `Camera`),
    - a point it is looking at in world space (`self.at` in `Camera`),
    - a direction specifying where "up" is for the camera, up_w, also expressed in world space (`self.up` in `Camera`), and
    - the camera's vertical field of view angle (fov) expressed in degrees, (`self.fov` in `Camera`); note that the horizontal fov can be obtained using the aspect ratio (`self.width/self.height`).

- While, given the aforementioned camera parameters, one can proceed to directly obtain the world space eye ray directions[2], it is often more convenient to first express these directions in a coordinate system centered at the camera, before finally transforming them into world space. The right side of the diagram below illustrates this *camera coordinate system*, with the camera at the origin looking down the (positive) z-axis (z_c); here, the x- and y-axes in the coordinate system are defined as $x_c = up_w \times z_c$, with z_c as the (normalized) vector from the eye to look-at point, and $y_c = z_c \times x_c$. In this coordinate system, the image plane is one unit away from the eye along z_c.
    - It can be useful to think about how the same points/directions are expressed across these three coordinate systems, for example:
        - the center of the (2D) NDC plane coincides with the (3D) point one unit along the camera coordinate system's z-axis *and* with the (3D) point from the eye towards the look-at direction in world space, i.e., $(0,0)_{ndc} = (0,0,1)_c = (eye_w + \texttt{self.at})_w$.

- In the camera coordinate system, you can use the trigonometric relationships formed by the vertical (and horizontal) fovs, and the unit distance between the origin and the center of the image plane, to transform the NDC-space (centered) pixel coordinates to camera-space eye ray points or directions.

- Finally, you can transform these points/directions from camera-space to world space using, e.g.,

$$\begin{bmatrix} | & & | & | \\ x_c & y_c & z_c & eye_w \\ | & & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

ⓘ Be mindful of how you treat homogeneous coordinates, especially for points post-transformation: their w component should be normalized to 1, and it's good practice to also normalize direction vectors.

[1] We will use the subscripts $\square_{ndc}$, $\square_c$ and $\square_w$ to distinguish between coordinates in the NDC-, camera- and world spaces.

[2] All the eye rays have the same origin in world space, eye_w (`self.eye` in `Camera`), for a perspective camera.

☑ **Deliverable 1 [10 points]**
Complete a `taichi` implementation of the `generate_ray` function in `Camera`. You will need to first call the `generate_ndc_coords` function, then `generate_camera_coords`. Finally, compute the world coordinates using the camera-to-world matrix outputed by the `compute_matrix` function. Ignore the `jitter` parameter for now, as this will be used in future assignments.

3 Ray-triangle Intersection

Consider a ray defined by its origin $\mathbf{o}$, direction $\mathbf{d}$ and parametric distance t. A point $\mathbf{x} = \mathbf{r}(t)$ along the ray can be expressed as

$$\mathbf{x} = \mathbf{r}(t) = \mathbf{o} + t\mathbf{d}\,.$$

In our case, the **Ray** struct is defined in `ray.py`, where **`Ray.origin`** corresponds to $\mathbf{o}$, and **`Ray.direction`** to $\mathbf{d}$:

```
@ti.dataclass
class Ray:
    origin: tm.vec3
    direction: tm.vec3
```

We define a triangle by its three vertices $\mathbf{v_0}$, $\mathbf{v_1}$, $\mathbf{v_2}$. To determine whether or not a ray intersects the triangle, we first compute edge vectors

$$\mathbf{e_1} = \mathbf{v_1} - \mathbf{v_0}$$
$$\mathbf{e_2} = \mathbf{v_2} - \mathbf{v_0}$$

before computing the **determinant** (i.e., "triple product") that defines the triangle's orientation: to do so, we perform a cross product between the ray direction and one of the edges, followed by the dot product of that vector with the remaining edge.

$$det = \mathbf{e_1} \cdot (\mathbf{d} \times \mathbf{e_2})$$

If $det = 0$, then the ray is parallel to the triangle, otherwise, it intersect's the plane of the triangle.

ⓘ To account for numerics, rather than testing for $det = 0$, you should test against a sufficiently small numerical ϵ (which we provide as `self.EPSILON`).

Next, we need to determine if an intersect occurs inside or outside the triangle bounds. To do so, we will compute the three barycentric coordinates, u, v, and w as

$$u = \frac{1}{det}\Big((\mathbf{o} - \mathbf{v_0}) \cdot (\mathbf{d} \times \mathbf{e_2})\Big)\,,$$
$$v = \frac{1}{det}\Big(\mathbf{d} \cdot \big((\mathbf{o} - \mathbf{v_0}) \times \mathbf{e_1}\big)\Big)\,, \text{ and}$$
$$w = 1 - u - v\,.$$

We then verify whether the barycentric coordinates fall within the sheared triangle frame bounds, i.e., we test whether

$$0 \leq u \leq 1\,,$$
$$0 \leq v \leq 1\,, \text{ and}$$
$$u + v \leq 1\,.$$

If all these tests pass, then the ray intersects the triangle at parametric distance t determined as,

$$t = \frac{1}{det}\Big(\mathbf{e_2} \cdot \big((\mathbf{o} - \mathbf{v_0}) \times \mathbf{e_1}\big)\Big)\,.$$

ⓘ To account for numerics, only consider intersections for $t > \epsilon$

Finally, you will need to populate and return the **`HitData`** struct, which is defined in `ray.py` as:

```
@ti.dataclass
class HitData:
    is_hit: bool
    is_backfacing: bool
    triangle_id: int
    distance: float
    barycentric_coords: tm.vec2
    normal: tm.vec3
    material_id : int
```

with the following member variables:

- **`is_hit`**: a flag that is true if there is a successful hit,
- **`is_backfacing`**: a flag that is true if the triangle is backfacing (**hint**: check the sign of the determinant),
- **`triangle_id`**: the id of the triangle we've intersected,
- **`distance`**: the distance from the ray origin to the intersection point,
- **`barycentric_coords`**: the $[u, v]$ barycenters, populated as a `taichi.math vec2` type,
- **`normal`**: the per-pixel interpolated normal, returned as a `taichi.math vec3` type (**hint**: you will need to flip your normal if your triangle is backfacing, as well as performing the per-vertex to per-pixel interpolation and renormalization), and
- **`material_id`** : the material id of the intersected triangle.

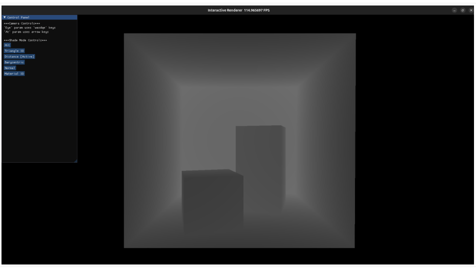
The aforementioned ray-triangle intersection algorithm is referred to as the Möller-Trumbore intersection algorithm. It remains among the most commonly implemented approaches in industry, and is based on a modification of one of the approaches discussed in lecture.

☑ **Deliverable 2 [10 points]**
Complete a `taichi` implementation of the `intersect_triangle` function in `ray_intersector.py`.

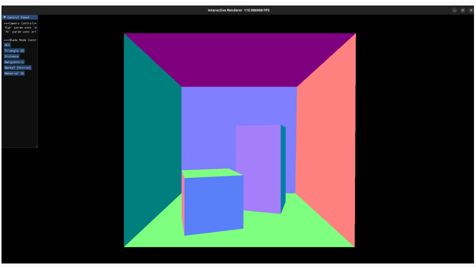
4 Your First Rendered Image

After implementing eye ray generation and ray-triangle intersection correctly, you can run the `A1.py` mainline script in the `interactive/` folder, which will deploy an interactive renderer that will display your results.

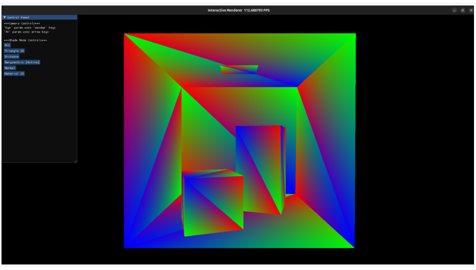
We provide some expected outputs for reference, below:



Intersection Distances



Intersection Normals



Intersection Barycenters

5 You're Done!

Congratulations, you've completed the 1st assignment. Review the submission procedures and guidelines at the start of this handout before submitting your assignment solution.