ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 0: Vectorizing Python and Phenomenological Shading

Due: Thursday, September 14th, 2023 at 11:59pm EST on myCourses

Final weight: **0% + 10% bonus**

Contents

1 Assignment Policies and Submission Process

Download and modify the standalone Python script we provide on myCourses, renaming the file according to your student ID as

`YourStudentID.py`

For example, if your ID is **234567890**, your submission filename should be `234567890.py` and should include the entirety of your solution submission for this assignment, according to the instructions below.

> 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.

1.1 Late policy

All the assignments are to be completed individually, unless stated otherwise. You are expected to respect the late day policy and collaboration/plagiarism policies.

> **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 specified in McGill's Policies on Student Rights and Responsibilities.

1.2 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 necessarily preclude an environment where you can be comfortable discussing **ideas** with your colleagues. **When in doubt, some good rules to follow include:**

- fully understanding every step of every solution you submit,
- only submitting solution code that was written by you, and
- never referring to, nor looking at, another student's 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.

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

If you have a disability, please advise the Office for Students with Disabilities (514-398-6009) as early in the semester as possible. In the event of circumstances beyond our control, the evaluation scheme as set out above may require modification.

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

1.3 Installing Python, Libraries and Tools

This assignment will be completed in `Python 3.x` and using the latest versions of the `numpy` and `matplotlib` libraries.

You are free to install and configure your development environment, including your IDE of choice, as you wish. One popular, self-contained installation package that we recommend is the **Individual Edition** of the Anaconda software installer.

After following your platform specific installation instructions, the Anaconda Navigator provides a simple graphical interface that allows you to:

- define isolated development environments with an appropriate Python version (e.g., 3.7),
- download and install the required libraries (`numpy` and `matplotlib`), including their dependencies, into the environment, and
- [optionally] to pick between a variety of development IDEs.

> 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.

1.4 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 good example is that, if you need to sort values in a list before plotting them, 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 contact the TA.**

> 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 (valid!) 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.

To help, the (purposefully minimal) base code we provide you with includes a superset of all the library `imports` we could imagine you using for the assignment.

> Do NOT use 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. One notable exception — as highlighted in the base code — is code in the `__main__` here, code to test your implementation can include other imported libraries to help with debugging.

This course will rely heavily on `numpy` — in fact, you'll likely learn just as much about the power (and peculiarities) of the Python programming language as you will about the `numpy` library. This library not only provides convenience routines for matrix, vector and higher-order tensor operations, but also allows you to leverage high-performance vectorized operations if you're careful about restructuring your data/code in a vectorized fashion.

For numerical computation, `numpy` is a library that is implemented in highly-optimized machine code. When used appropriately, code that **carefully** leverages `numpy`'s ability to efficiently perform data-parallel operations over **higher-order tabular data** can be several orders of magnitude more efficient than its Python-only functional equivalent.

One of the key learning objectives of this optional assignment is to help you to start thinking about coding **vectorization-first** implementations. ECSE 446/546 will effectively **require** that your code be vectorized — i.e., it is unlikely that you will be able to complete future assignments (time-wise) with wholly unvectorized code.

1.5 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.

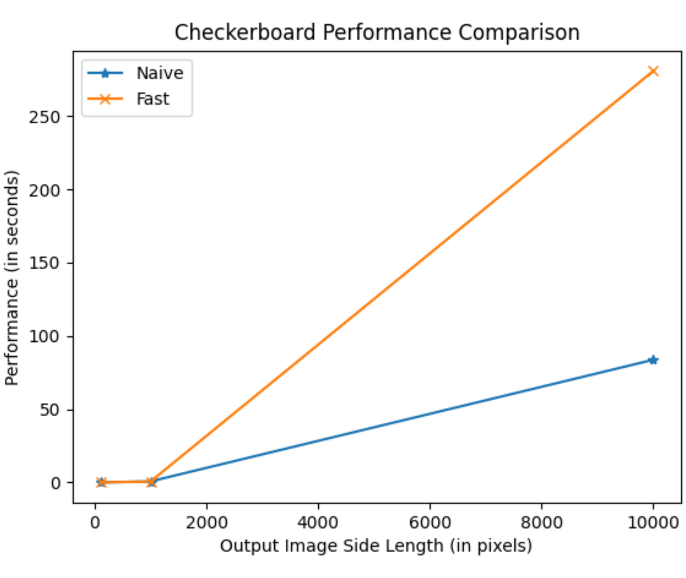
2 Your first image — a checkerboard

The first task is to implement a *vectorized* algorithm that creates, populates and displays a parameterizable checkerboard image. As reference, we include a non-vectorized (i.e., using `for` loops) implementation of the logic we expect your vectorized implementation to replicate.

The sole learning objective of this task is to start exploring and realizing non-trivial vectorized implementations using `numpy`.

> **Deliverable 1 [5 points]**
> Complete a vectorized `numpy` implementation of the function `render_checkerboard_fast` that returns a 2D `numpy.array` representing a binary monochromatic checkerboard image. Refer to the functionality of the implementation in `render_checkerboard_slow`, which your implementation should reproduce perfectly (albeit with much higher performance). You should tinker (i.e,. in the `__main__` routine) with the `render_checkerboard_slow` routine to fully understand the semantics of its `height`, `width`, and `stride` parameters.

Our base code includes basic test code in `__main__`, which you are (very) welcome to extend and/or modify, as you see fit. In the case of this deliverable, the test code visualizes one output of the functions and generates a plot that compares their performance as a function of increasing problem size (on a logarithmic scale, to highlight order-of-magnitude performance differences).



Performance comparison between vectorized and non-vectorized checkerboard implementations.

3 Your second image — simple 2D shapes

You may recall from calculus that we can implicitly define 2D curves as functions $f(x, y) = 0$. Here, you can similarly extend this definition into a simple inside-outside 2D test, where 2D points (x, y) that have $f < 0$ lie on the "inside" of the curve and those where $f > 0$ lie on the "outside".

The goal of the next few deliverables is to generate an image of two 2D shapes, a circle and a heart, where the pixels inside and outside of the shapes are colored using two different colors (e.g., black and white; so, no actual shading, per se).

You'll first have to implement the two shape functions for the circle and heart, based on their two implicit functions:

$$f_{circle}(x, y) = x^2 + y^2 - \left(\tfrac{1}{2}\right)^2 \text{ and } f_{heart}(x, y) = (x^2 + y^2 - 1)^3 - x^2\,y^3 .$$

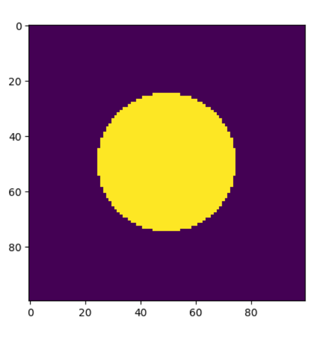
> **Deliverable 2a [2.5 points]**
> Complete a vectorized `numpy` implementation of the functions `circle` and `heart` which take in a set of `x` and `y` coordinates, evaluates the appropriate implicit shape function at those points, and returns a 1D `numpy.array` representing the appropriate implicit shape function evaluations.

The f_{circle} function corresponds to a $r = 1/2$ circle centered at the origin $(0, 0)$ in 2D Cartesian coordinates, and so you can immediately imagine testing (at least the sign of) the input/output behaviour of your `circle` routine. While you could (and should) debug/test these functions point-wise, the next deliverable will help you to *visually* test your shape functions.

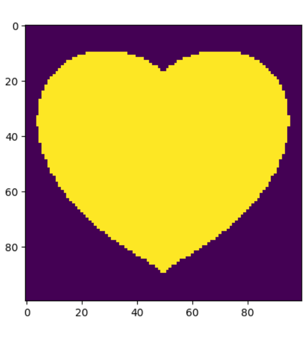
To do so, you will populate a 2D `numpy` array corresponding to an "inside-outside pixel grid" that evaluates your shape functions at discrete image pixel locations and either returns a 1 for points that are "inside" the shape (i.e., where f_{circle} or f_{heart} are ≤ 0) or 0 otherwise.

> **Deliverable 2b [10 points]**
> Complete a vectorized `numpy` implementation of the functions `visibility_2d` which takes a `scene_2d` parameter as input and outputs a 2D `numpy` array of either binary or floating-point inside-outside pixel outputs. The `scene_2d` input is a `Python` dictionary object that encapsulates all of the inputs required for this task: the image output resolution, the x and y evaluation bounds, and which shape function to use (see `__main__` for an example). Your implementation of `visibility_2d` should be vectorized over the pixel grid.

We provide some debug code in `__main__` that creates an example `scene_2d` instances using your `circle` and `heart` and `visibility_2d` routines, "rendering" your inside-outside shape images onto an image buffer and displaying the results, which we include below for your reference, below.



Rendering the 2D circle.



Rendering the 2D heart.

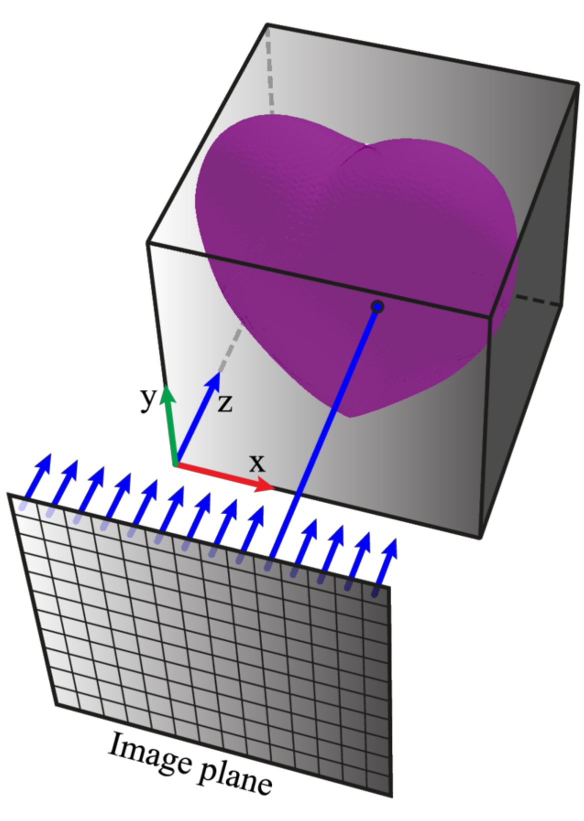
4 Your third image — simple 3D shading

We're going to extend the idea from the previous deliverable from 2D shape functions to 3D shape functions, as well as adding some **very basic** phenomenological shading. So now, instead of a 2D implicit shape function $f(x, y)$, we have a 3D implicit shape function $f(x, y, z)$ where $f(x, y, z) = 0$ defines the set of 3D points that lie on the *surface of the 3D shape* (i.e., instead of points that lie on the *curve of the 2D shape*, as seen earlier). As before, points inside the volume of the shape have $f \leq 0$ and those outside have $f > 0$.

We will consider a 3D heart shape, similar to the 2D heart from above, with an implicit shape function of

$f_{heart3D}(x, y, z) = (x^2 + \alpha\, z^2 + y^2 - 1)^3 - x^2\, y^3 - \beta\, z^2\, y^3$ where $\alpha = 9/4$ and $\beta = 9/200$.

When rendering this shape onto the pixel grid, we will implement an extremely simple camera model: the camera will be located at $(0, 0, -\infty)$ and the rendering will use an orthographic projection towards the $-z$ direction; effectively, this visualizes the projection of the 3D shape onto the xy-plane (see schematic, below.)



Schematic of camera setup and orthographic projection for the 3D heart scene.

ECSE 446 students will be provided with a `heart_3d` function that takes (a set of) x and y coordinates for 2D points (i.e., on the pixel grid), and returns a list of:

1. z coordinates corresponding to the z coordinate of the *nearest* surface point along the $+z$ viewing direction, such that (x, y, z) is a valid point on the surface — note: for those (x, y) pixel coordinates that don't overlap the shape, z will equal NaN (Not a Number) (i.e., `numpy.nan`),
2. function values for $f_{heart3D}(x, y, z)$ evaluated at points `(x,y,z)`, but only for those points where z≠ NaN; otherwise, it returns NaN (i.e., `numpy.nan`), and
3. unit length normals (represented as a `numpy` array of row vectors) at the (valid) surface points; again, for those points (x, y) where no valid surface point is visible, the outputted normals will be $(0, 0, 0)$.

ECSE 446 students can safely skip the Deliverable 3a below, and jump straight to Deliverable 3b.

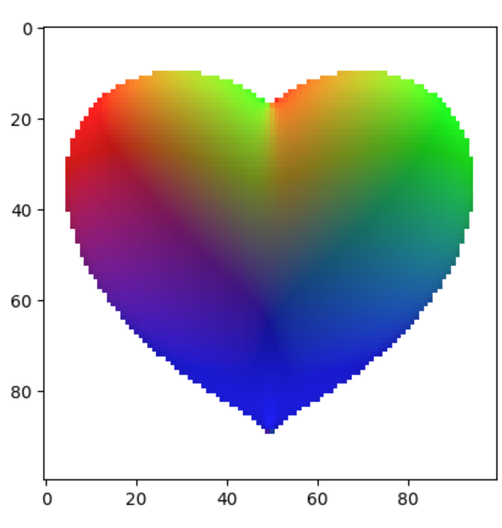
ECSE 546 Students Only

> **Deliverable 3a (ECSE 546 only!) [10 points]**
> Complete a vectorized `numpy` implementation of the `heart_3d` function by implementing the `get_heart_xyz`, `get_heart_values` and `get_heart_normals` subroutines, as described below. Some notes:
>
> - $f_{heart3D}$ is 6th-order polynomial in (x, y, z) and so, when solving for its roots in z (i.e., in order to find surface points $f_{heart3D} = 0$), you may obtain up to six unique roots! If none of them are real, there is no valid surface point (i.e., (x, y) on the pixel grid does not overlap with the shape); if some/any of the roots are real, you should retain the smallest one (i.e., the nearest z value, given that the camera is located at $(0, 0, -\infty)$).
> - the heart shape function has a nasty discontinuity at $y = 0$ (among other places, but this is the only one you'll need to deal with) — you can simply avoid problems here by , e.g., offsetting any $y \approx 0$ grid points.
> - you can obtain the (unnormalized) normal at a valid surface point (x, y, z) as $\nabla f_{heart3D}(x, y, z)$.
>
> 
>
> ```
> def get_heart_xyz(x, y, alpha, beta):
>     ### TO REPLACE FOR ECSE 546 STUDENTS ###
>     file_path = Path(__file__).parent / 'heart_xyz.npy'
>     x,y,z = np.load(str(file_path))
>     return x,y,z
>     ### TO REPLACE FOR ECSE 546 STUDENTS ###
>
> def get_heart_values(x, y, z, alpha, beta):
>     ### TO REPLACE FOR ECSE 546 STUDENTS ###
>     file_path = Path(__file__).parent / 'heart_values.npy'
>     values = np.load(str(file_path))
>     return values
>     ### TO REPLACE FOR ECSE 546 STUDENTS ###
>
> def get_heart_normals(x, y, z, alpha, beta):
>     ### TO REPLACE FOR ECSE 546 STUDENTS ###
>     file_path = Path(__file__).parent / 'heart_normals.npy'
>     normals = np.load(str(file_path))
>     return normals
>     ### TO REPLACE FOR ECSE 546 STUDENTS ###
> ```
>
> *Code to replace with your solution.*
>
> - you should modify the code in the `get_heart_xyz` routine to find the roots of $f_{heart3D}$ for given x, y, α, β values.
> - you should modify the code in the `get_heart_values` routine to compute the values of $f_{heart3D}$ for given x, y, z, α, β values.
> - you should modify the code in the `get_heart_normals` routine to compute the normals of $f_{heart3D}$ for given x, y, z, α, β values.
>
> If you are unable to complete Deliverable 3a with your own solution, leave the routines `get_heart_xyz`, `get_heart_values` and `get_heart_normals` untouched such that they load the provided `.npy` values. This will get you 0 points for Deliverable 3a but you will be able to tackle Deliverable 3b.

Given the `heart_3d` function, we will visualize this 3D shape a little more elaborately than we did earlier with the 2D shapes. Specifically, you will implement a simple diffuse shading model with directional light sources. A `scene_3d` `Python` dictionary will similarly encapsulate the rendering parameters for this task, now including additional information for each of (potentially many) lights in the scene: each light i has a direction $\mathbf{l}_i$ (and resides "at infinity") and a color $\mathbf{c}_i$.

Your visualization routine will shade every pixel that contains a valid surface point according to the following (simplified) diffuse shading equation: $s(x, y) = \sum_i \mathbf{c}_i \, \max(0, \mathbf{n}_{x,y} \cdot \mathbf{l}_i)$, where $\mathbf{n}_{x,y}$ are the unit normals for all the pixel coordinates `(x,y)` as returned in the third element of the list output of `heart_3d`.

> **Deliverable 3b [12.5 points]**
> Complete a vectorized `numpy` implementation of the aptly-names `render` routine, which takes a `scene_3d` parameter as input and outputs a 2D `numpy` array of floating-point pixel colors corresponding to the $s(x, y)$ equation, above. The outputted values should be clamped to 1 to avoid a `matplotlib` image display warning. As with Deliverable 2b, the `scene_3d` input is a `Python` dictionary object that encapsulates all of the inputs required for this task, and we provide an example in `__main__` (the scene that was used to generate the image, below). Your implementation of `render` should be vectorized over the pixel grid and over lights.



Rendering the 3D heart with simple diffuse shading and three colored directional lights.

5 You're Done!

Congratulations, you've completed the zero[th] assignment. Review the submission procedures and guidelines at the start of the handout before submitting the `Python` script file with your assignment solution.