A Tiny World: Atom

CHUNG AN, CHEN, YANG, LIU, and LINXI, TAO

Additional Key Words and Phrases: Taichi, Atom, 3D, Rendering, Electron Cloud

ACM Reference Format:
Chung An, Chen, Yang, Liu, and Linxi, Tao. 2022. A Tiny World: Atom. 1, 1 (January 2022), 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Prior to the electron cloud model, there are a lot of attempts on building a theoretical model for an atom. Most of them, for example, the commonly known Rutherford-Bohr model, perceive the electrons as if they are planets orbiting the sun — a non-changing centripetal movement. The electron cloud, however, models an atom consisting of a small, yet massive when put aside to its electrons, nucleus surrounded by a non-deterministic cloud of electrons.

In the electron cloud model, electrons can theoretically be found anywhere in the space. However, they are generally found more often in some regions, which we usually call them orbitals despite no orbital motions of electrons are being held, than others. A Carbon atom has two orbitals with the inner layer contains two electrons and the outer layer contains four. Electrons are less likely to be found in between layers.

This project aims to simulate and render a Carbon atom's presence in a three-dimensional space. The simulation will be grounded in the more accurate electron cloud model, shown in Figure 1 below. An electron cloud model depicts the occurence of an atom's electrons by a density map containing numerous sparse dots. In any region, the density of the dots draws a directly proportional relationship to the probability of an electron being present. To illustrate this model, We leverage a recently-built parallel programming language, namely **Taichi** [Hu et al. 2019], as our main development framework.

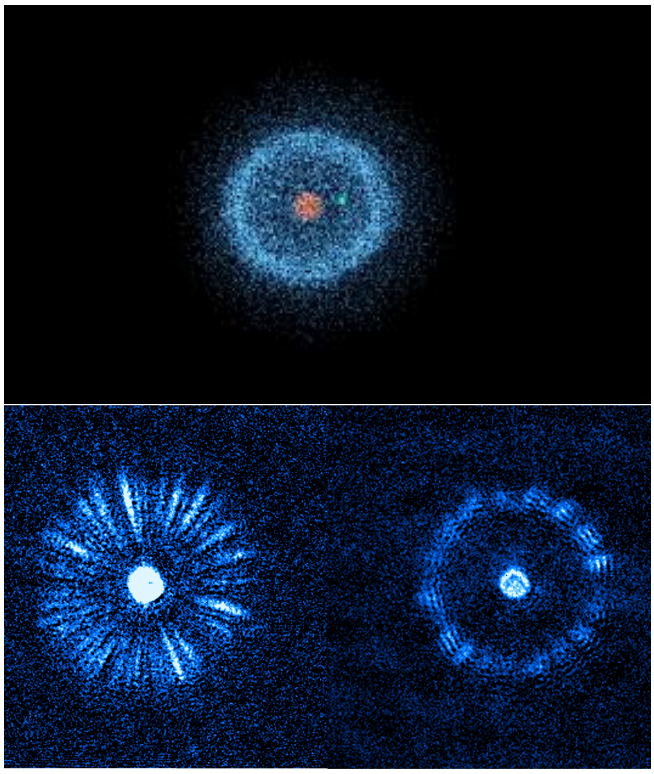

Fig. 1. Atoms and Their Nucleus and Electron Cloud

2 SOFTWARE

Taichi is a Python-based Domain-Specific Language designed for differentiable programming and speed up computer graphics development without compromising too much rendering performance. Despite inheriting most of its syntax from Python, **Taichi** relies on mostly parallel programming and does not carry over the downsides, for example, the slow computation speed, from Python.

Taichi is a higher level language such that it can be made to run on multiple mainstream rendering backends such as OpenGL, CUDA, and Metal. Using Pythonic decorators such as @ti.kernel and @ti.func brings the succeeding block of code into **Taichi**'s scope, where functions are naturally parallelized and differentiable on CPU or GPU devices. Two main data types in **Taichi** are Primitive Types and Compound Types. Primitive Types are numerical data types used by backends, while Compound Types are user-defined types composed of multiple members. To bridge across **Taichi**'s scope and Python's Scope, we use a global variable object called Field. They can correspond to OpenGL's vectors or matrices depending on how users define them.

3 CODE AND DEVELOPMENT

We decided to proceed with render a Carbon atom to represent the main topic of our work. However, we can easily scale up or down by editting the code to render other atoms as they only differ by the number of protons, neutrons, electrons, and the layers of the electron cloud.

For the prototype shown in Figure 2, we started off by using a simple ray-marching model and shattered beams of light of extremely small radius towards the protons and neutrons. Instead of directly specifying the layers of electron cloud, we manipulate shadow of the protons and neutrons and use them to form a relatively hollow region to represent the sparseness of the electrons. In the outer

Authors' address: Chung An, Chen; Yang, Liu; Linxi, Tao.

XXXX-XXXX/2022/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

region, the light beams are able to fully hit the back of the wall and thus generating a denser cloud of electrons.

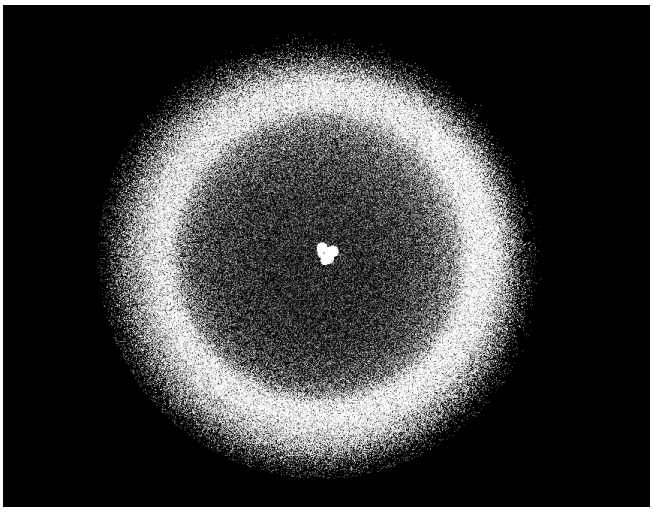

Fig. 2. Ray-marching Prototype

However, we did not proceed with the prototype as the layers of electrons cloud are not scalable.

For the next step, we decided to proceed with rendering protons, neutrons, and electron clouds using tiny particles. We abide the physics observations and define protons and neutrons to be roughly equal size of round particles, with neutrons being slightly larger, of size around 0.05. We do not render the electrons since they only possess mass and charge. We use particles that are ten times smaller just to depict the electron cloud.

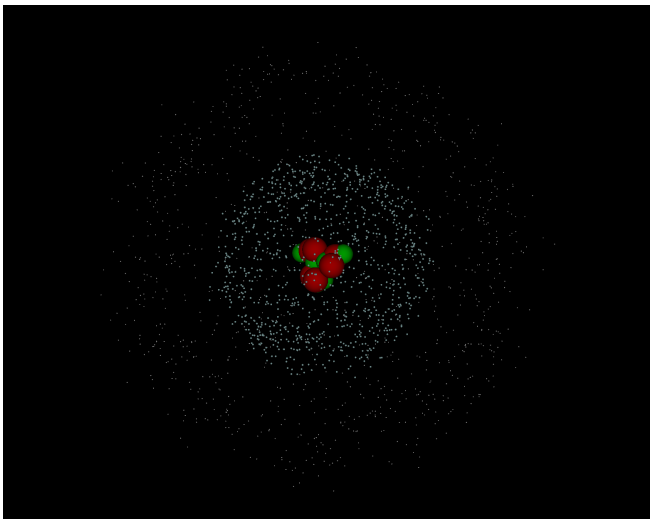

Fig. 3. A (Rotating) Carbon Atom

We set the electron clouds to two different colors just to make the image more friendly to human eyes. In the center we have six protons and six neutrons rendered green and red respectively. There are two hollow spaces in the image with one located in between the inner electron cloud layer and the neucleus and the other located between the two layers of electron cloud. This is because the energy that an electron can emit or absorbed is quantized, and thus electrons can only appear in specific levels of orbit.

```
class Atom:
    def __init__(self, radius, dim=3):
        self.radius = radius
        self.dim = dim
        self.color = ti.Vector.field(dim, ti.f32, shape=1)
        self.pos = ti.Vector.field(dim, ti.f32, shape=1)

    def display(self, scene):
        scene.particles(self.pos, self.radius, per_vertex_color=self.color)
```

```
@ti.data_oriented
class Proton(Atom):
    @ti.kernel
    def initialize(self, color: ti.template(), pos: ti.template()):
        self.color[0] = color
        self.pos[0] = pos
```

Fig. 4. Particle Implementation

Figure 4 shows the general implementation of a single proton, neutron, and electron particle. We simply render them as spherical objects with different sizes and colors. In reality, protons tend to repel each other and attract neutrons. Therefore, in our implementation, we try to pair protons to neutrons so that protons are not clustered and are some what distant apart.

```
def electron(pos, color, n, ratio, color_vec):
    for i in range(n):
        u1 = ti.acos(2 * random.random() - 1) - pi / 2
        u2 = 2 * pi * random.random()
        x = ti.cos(u1) * ti.cos(u2) + (random.random() - random.random()) / 5.0
        y = ti.cos(u1) * ti.sin(u2) + (random.random() - random.random()) / 5.0
        z = ti.sin(u1) + (random.random() - random.random()) / 5.0
        p = ratio * ti.Vector([x, y, z])
        c = ti.Vector(color_vec)
        pos[i] = p
        color[i] = c
```

Fig. 5. Electron Cloud

Figure 5 shows the formation of shells of electron clouds. Randomness is added to not only distribute them across the shell but also to level electrons on the same shell unevenly.

```
def rotate(angle, speed):
    angle += speed
    return angle
```

```
angle = rotate(angle, speed)
x = ti.sin(angle) * 10.0
y = ti.cos(angle) * 10.0
camera.position(x, y, 5.0)
```

Fig. 6. Rotation Implementation

Figure 6 shows how we rotate the camera to mimic the rotation of the atom instead of giving each partical a velocity and acceleration, which is extremely computationally expensive.

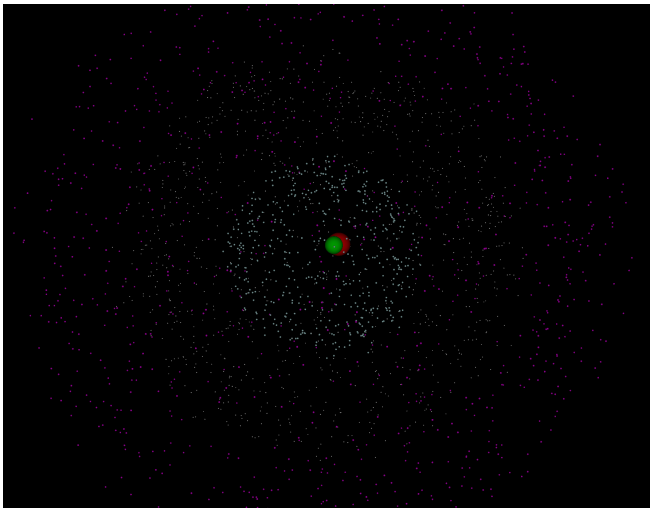

Fig. 7. A Different Example

Figure 7 is an example another implementation with differnt hyperparamters, namely an imaginary atom containing a neucleus of one proton and one neutron with three layers of electron clouds.

4 SOURCE

All of our work can be found here https://github.com/andy971022/acg-project.

REFERENCES

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 201.