Mathematics and Computer Science
2025, Vol. 10, No. 3, pp. 43-51
https://doi.org/10.11648/j.mcs.20251003.11

Research Article

Interactive 3D Visualization of Bohr's Atomic Model: Enhancing Educational Tools with WebGL and Force-Directed Algorithms

Zaid Kraitem*, **Hamza Alhaj, Mohamad Taky**

Department of Information Engineering, Al-Wataniya Private University, Hama, Syria

Abstract

The study presents an interactive 3D web-based application designed to visualize atomic structures according to Bohr's Model using Three.js and WebGL. The primary aim of the project is to enhance educational tools in atomic physics by providing an interactive, real-time representation of atomic structures. This tool allows users to explore atomic models dynamically, offering a detailed view of electron orbits, nuclear structure, and electron movement. The visualization system is built around Three.js, a JavaScript library for 3D rendering, and incorporates force-directed algorithms for the realistic positioning of protons and neutrons within the nucleus. These particles are placed using Eades' 1984 force-directed graph algorithm, which simulates physical forces to arrange the particles in a minimal energy configuration. The electron orbits are generated procedurally using circular subdivision methods, ensuring that electrons appear to move around the nucleus in defined energy levels, as proposed by Bohr. The application also accounts for performance optimization and user interaction. It ensures frame rate independence by calculating delta time between render cycles, providing smooth motion even on devices with varying processing capabilities. The user can interact with the model, adjusting the camera view to zoom in or rotate the atomic structure, thus fostering a deeper understanding of atomic physics. The study also highlights the integration of TypeScript, which improves maintainability and type safety in the development process. The application's usability has been tested with engineering students, confirming its effectiveness as an educational tool. Future work includes expanding the model to incorporate quantum mechanical adaptations and potentially integrating augmented reality for more immersive learning experiences. In conclusion, this research contributes to the field of computer-aided education by providing an interactive 3D atomic visualization tool. It offers an engaging and effective method for learning about atomic structures and their behavior, making complex scientific concepts more accessible.

Keywords

Computer Graphics, Bohr Model, Atom Visualization, WebGL, THREE.js, Interactive Simulation

1. Introduction

The Bohr Model, introduced in 1913 by Niels Bohr, describes an atomic structure where electrons orbit the nucleus in defined energy levels [1]. As *Figure 1* shows.

While this model has been refined by quantum mechanics,

*Corresponding author: zaidkretem@gmail.com (Zaid Kraitem)

Received: 8 April 2025; **Accepted:** 19 April 2025; **Published:** 22 May 2025

it remains essential for conceptualizing atomic behavior, especially in education. Existing atomic visualization tools often rely on static 2D representations, which lack interactivity and depth perception. This study proposes an interactive 3D visualization tool that allows users to explore atomic structures dynamically.

The application is designed using Three.js, an industry-standard JavaScript library for 3D rendering, and incorporates force-directed algorithms for realistic nucleus particle arrangement [2, 3].

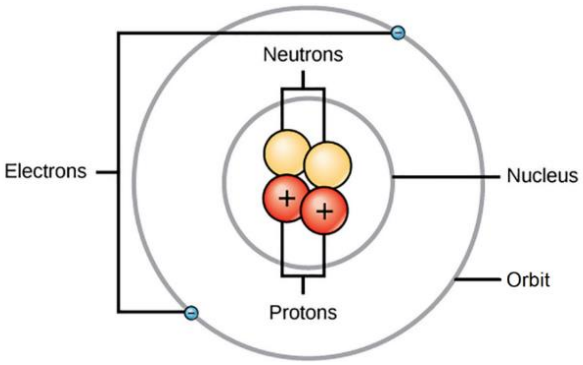

Figure 1. *Bohr-Model Representation.*

2. Related Work

Several studies have explored computer-based atomic visualizations. Foley et al. (2020) discussed the importance of real-time rendering in scientific visualization. Angel (2018) demonstrated interactive simulations using OpenGL, but lacked browser-based implementation. More recently, Hearn & Baker (2022) introduced WebGL-based atomic rendering, though without procedural nucleus generation. Our approach integrates real-time rendering, dynamic nucleus positioning, and adaptive electron motion in a fully interactive web environment [4, 5].

The integration of 3D visualization into education has been explored in systems like NPIPVis [6], which combines NBA data analysis with machine learning to create interactive educational tools. While NPIPVis focuses on sports analytics, our work shares its goal of enhancing comprehension through real-time interactivity but applies this principle to atomic physics. Similarly, PartLabeling [7] introduces a 3D label management framework for complex models, emphasizing user-centric design. Our application adopts a comparable philosophy by enabling users to interactively select and render atoms, though we prioritize simplicity and pedagogical clarity over multi-label systems.

Efficient rendering is critical for web-based 3D applications. For instance, Depth of Field Rendering Using Multi-layer-Neighborhood Optimization [8] improves visual realism through advanced post-processing, whereas our work prioritizes frame rate independence to ensure smooth animations across devices. The latter aligns with the performance goals of Efficient Binocular Rendering of Volumetric Density Fields [9], which optimizes rendering for virtual reality (VR). While our tool does not target VR, we similarly leverage adaptive algorithms (e.g., Eades' force-directed method) to balance computational load and visual fidelity.

3. Discussion

In this research, we discuss various methods such as:

1. Development Environment: Implemented using Three.js, WebGL, and TypeScript for enhanced maintainability [10].
2. Nucleus Generation: Uses Eades' 1984 force-directed graph algorithm to position protons and neutrons dynamically [11].
3. Electron Orbits: Procedural orbit generation based on circular subdivision methods [12].
4. Rendering & Performance Optimization: Frame rate-independent electron movement using delta time calculations for smooth motion.
5. User Interaction: Camera controls via OrbitControls for zooming and exploration [13, 14].

3.1. Three.js

Three.js is a cross-browser JavaScript library and application programming interface used to create and display animated 3D and 2D computer graphics in a web browser using WebGL JavaScript API.

It provides a lot of features including scene creation, different camera types, materials, geometries, lighting, texturing, interactivity and much more.

A basic Three.js application consists of a scene, camera and renderer

1. Scene: represents a scene which contains other 3D objects.
2. Camera: mimics Camera point of view for the application which also describes the used rendering projection and other properties.
3. Renderer: an object that describes the rendering behavior of the application.
4. Object3D: the base class for most objects in three.js and provides a set of properties and methods for manipulating objects in 3D space.

A high-level overview of the application execution flow looks like this:

```
//main.ts
// atom generation
const { nucleus, maxOrbitRadius, nucleusRadius, orbits, shells } = generateAtom("oganesson");
// initial camera positioning and camera controls adjustments
controls.minDistance = nucleusRadius + 2;
controls.maxDistance = maxOrbitRadius * 2.5;
camera.position.multiplyScalar(maxOrbitRadius * 2.5);
// adding generated objects (meshes) to the scene
```

```
shells.forEach((shell) => shell.forEach((electron) => scene.add(electron)));
orbits.forEach((orbit) => scene.add(orbit));
nucleus.forEach((particle) => scene.add(particle));
// draw function
function draw(): void {
requestAnimationFrame(draw);
tick();
for (let i = 0; i < shells.length; i++)
for (let j = 0; j < shells[i].length; j++) {
shells[i][j].userData.dir.applyAxisAngle(UP, degToRad(-SPEED * deltaTime));
shells[i][j].position.copy(shells[i][j].userData.dir);
}
renderer.render(scene, camera);
}
draw();
```

We start the rendering loop by calling the draw function which handles scene updates for a single render cycle. Then, we pass the same draw function as a as a callback to *requestAnimationFrame* method which tells the browser to perform the callback before the next repaint. The frequency of the callback function calls generally matches device screen refresh rate. 60hz is the most common refresh rate (60 render cycles/frames per second) but higher refresh rates like 120hz and 144hz are also widely used [15].

This effectively creates an infinite render loop and syncs it with browser repaints while also matching device screen refresh rate to ensure smooth and proper rendering loop as opposed to other methods like intervals and timeouts which does not count for device refresh rate and can cause inconsistent results. We will dive deeper into other details of the application throughout this paper.

3.2. Nucleus Generation

The nucleus consists of protons and neutrons based on atomic mass and atomic number so we needed a way to position these particles in 3D space.

There are two main approaches to achieve this: using predefined positions or position the particles algorithmically.

Manual positioning of each particle in every atom is not an option especially for larger atoms that requires a lot of work to find proper positions which is nearly impossible to do so we went with the second approach. The proposed solution was using one of the Force directed graph algorithms which are used for graph drawing in an aesthetically pleasing way. The idea was first used by Eades (Ead84) using physical and attractive forces.

The Ead84 algorithm is succinctly summarized as follows:

"To embed a graph, we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system. the vertices are placed in some initial layout and let go so that the spring forces on the rings move the system to a minimal energy state".

Each node is affected by repulsive and attractive forces based on the graph edges and how nodes are connected [16].

(i). Repulsive Force

Applied between non-adjacent nodes based on the distance between them:

$$Frep(u,v) = \frac{Crep}{\|Pv-Pu\|^2}.\widehat{PuPv} \ldots. \quad (1)$$

1. $Crep$: repulsive constant (2 is usually a proper value as stated by Eades), higher values cause more repulsion and thus affecting the end result of the graph layout.
2. Pv, Pu: node positions.
3. repulsive force change is inversely proportional to the squared distance between the corresponding nodes.
4. the further the distance gets, the smaller the force will be.

The following function takes $Crep$, Pv, Pu and return a vector representing the repulsive force between the past positions:

```
//calculateRepulsiveForce.ts
function calculateRepulsiveForce(repulsiveConstant: number, from: THREE.Vector3, to: THREE.Vector3) {
const displacement = new THREE.Vector3().subVectors(to, from);
const squaredDistance = displacement.lengthSq();
return displacement.normalize().multiplyScalar(repulsiveConstant / squaredDistance);
}
```

(ii). Attractive Force

Applied between adjacent nodes using a logarithmic strength spring.

$$Fspring(u,v) = Cspring.\log\left(\frac{\|Pv-Pu\|}{l}\right).\widehat{PuPv} \quad (2)$$

1. $Cspring$: spring constant (1 is usually a proper value as stated by Eades).
2. l: ideal spring length in equilibrium state.

if the distance is equal to l, no force will be applied:

$$\|Pv-Pu\| = l \xrightarrow{yields} \log\left(\frac{\|Pv-Pu\|}{l}\right) = 0 \xrightarrow{yields} Fspring = 0$$

if the distance is greater than l, the logarithm will be positive and the spring pulls:

$$\|Pv-Pu\| > l \xrightarrow{yields} \log\left(\frac{\|Pv-Pu\|}{l}\right) > 0 \xrightarrow{yields} Fspring > 0$$

This function takes parameters for $Cspring$ and l and return the spring force between two positions:

```
//calculateSpringForce.ts
function calculateSpringForce(springConstant:number, l:number, from:THREE.Vector3, to:THREE.Vector3) {
const displacement = new THREE.Vector3().subVectors(to, from);
const distance = displacement.length();
```

return displacement.normalize().multiplyScalar(springConstant * Math.log(distance / l));}

Finally, we have to subtract $Frep$ from $Fspring$ since pushing and pulling of adjacent nodes is already handled by $Fspring$ and we don't want to apply both forces:

$$Fattr(u,v) = Fspring(u,v) - Frep(u,v) \quad (3)$$

Calculating and applying these forces for multiple iterations will result a specific graph layout based on chosen values for $Crep$, $Cspring$ and l which can be tweaked to achieve the layout you want. In our case we have set every particle at a fixed distance and random direction from the world origin (0,0,0) and performed $Fspring$ for all particles (nodes) with the world origin while also applying $Frep$ for each particle with all other particles. We have chosen our constant values as follows:

1. $Crep = 0.02$
2. $Cspring = 0.5$
3. $l = 0.5$

To avoid shifts and wrong or inconsistent results for each iteration, we had to only apply forces after calculating them for each particle and after 100 iteration we achieve the spherical shape of the Nucleus.

The following function implements the algorithm by taking constant's values and an atom parameter which points to one of the properties of our JSON data source (Periodic-Table-JSON) that contains information about every element in the periodic table to calculate the layout and return the final meshes array:

```
//generateNucleus.ts
function generateNucleus(atom:keyof typeof atoms, iterations:number, l:number, repulsiveConstant:number, springConstant:number) {
const atomConfig = atoms[atom];
const nucleus: Particle[] = [];
for (let i = 0; i < atomConfig.atomic_number; i++) {
const randomPosition = new THREE.Vector3().randomDirection().multiplyScalar(10);
const proton = generateParticle("proton", randomPosition);
nucleus.push(proton);
}
for (let i = 0; i < Math.floor(atomConfig.atomic_mass - atomConfig.atomic_number); i++) {
const randomPosition = new THREE.Vector3().randomDirection().multiplyScalar(10);
const neutorn = generateParticle("neutorn", randomPosition);
nucleus.push(neutorn);
}
for (let i = 0; i < iterations; i++) {
const forces = [];
for (let j = 0; j < nucleus.length; j++) {
const attractiveForce = calculateSpringForce(springConstant, l, nucleus[j].position, ZERO);
for (let k = 0; k < nucleus.length; k++) {
if (j === k) continue;
const repulsiveForce = calculateRepulsiveForce(repulsiveConstant, nucleus[k].position, nucleus[j].position);
attractiveForce.add(repulsiveForce);
}
forces.push(attractiveForce);
}
for (let j = 0; j < nucleus.length; j++)
nucleus[j].position.add(forces[j]);
}
return nucleus;
}
```

Finally, we add these meshes to the scene for rendering:

```
currentAtom = generateAtom(element.getAttribute("data-element") as keyof typeof atoms);
currentAtom.shells.forEach((shell) => shell.forEach((electron) => scene.add(electron)));
currentAtom.orbits.forEach((orbit) => scene.add(orbit));
currentAtom.nucleus.forEach((particle) => scene.add(particle));
```

Repulsive and attractive forces are approximately represented with the following Figure 2.

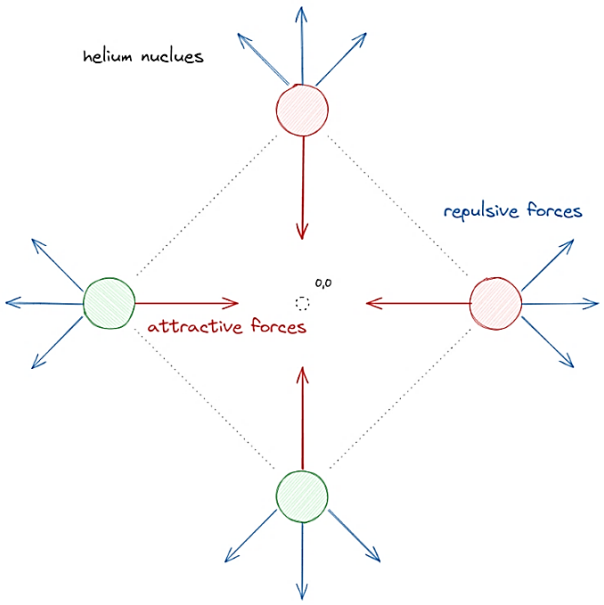

Figure 2. *Nucleus generation forces for Helium nucleus.*

The generation process looks like *Figure3*. shows.

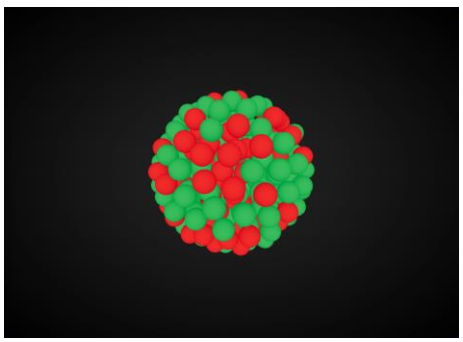

Figure 3. *Generation process of Oganesson nucleus.*

3.3. Electrons Visualization

Bohr model states that electrons are distributed based on their energy in predefined energy levels called shells. which plays a fundamental role in understanding atomic structure and electrons behavior within the atom. Simulating this behavior is achieved by generating orbits first, then placing electrons and moving them in circular motion each render cycle.

3.3.1. Drawing the Orbits

This is achieved by utilizing a procedural approach based on the concept of circle subdivision which involves generating a circle iteratively by equally distributing vertices around the center of the circle and then connecting them to form a circular shape.

First, since we know that a radian angle of 2π is exactly equal to one turn of a circle, we can divide this value by the number of subdivisions to get the required step between each two adjacent vertices. Then, we position the vertices away from the center at a distance equal to the radius of the circle and according to the calculated step. Finally, we connect these vertices together to form a circular line loop. This example demonstrates the process for 8 subdivisions as *Figure 4* illustrates.

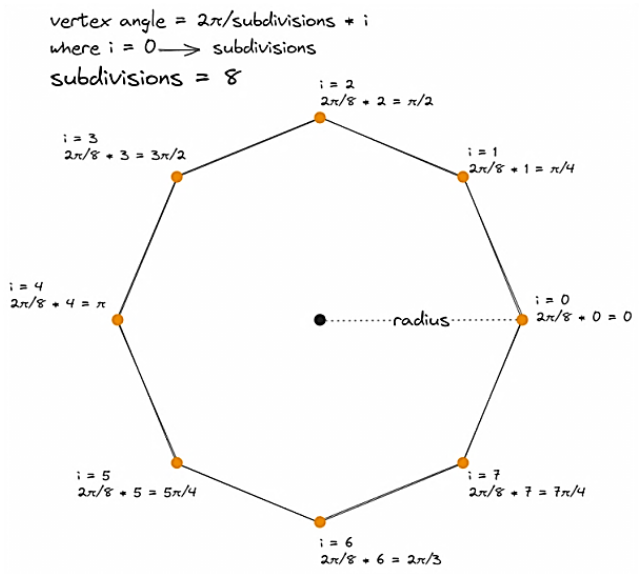

Figure 4. *Orbit generation for eight subdivisions.*

We can control the number of subdivisions to adjust the resolution of the generated mesh, Higher number of subdivisions creates a smoother mesh. The following function takes radius and resolution parameters to generate the orbit mesh using the built-in LineLoop class provided by Three.js to create a closed line from vertices based on their order in the buffer:

```
//generateOrbit.ts
export default function generateOrbit(distance: number, resloution: number) {
const points = [];
for (let i = 0; i < resloution; i++) {
const point = new THREE.Vector3()
.copy(RIGHT)
.applyAxisAngle(UP, degToRad((360 / resloution) * i))
.multiplyScalar(distance)
points.push(point);
}
const geomentry = new THREE.BufferGeometry().setFromPoints(points);
const material = new THREE.LineBasicMaterial({ color: 0x808080 });
return new THREE.LineLoop(geomentry, material);
}
```

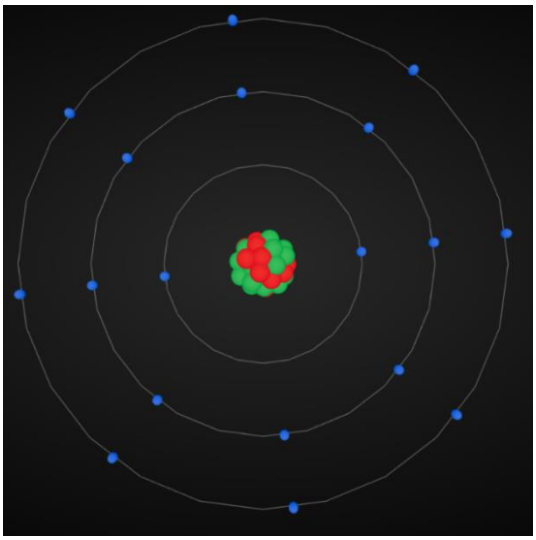

Figure 5. *Argon atom orbits generated using 24 subdivisions.*

A resolution of 24 is mostly suitable for small orbits since connected vertices are close to each other thus spanning relatively short distance compared to longer circumference of outer orbits which are more jagged as we can see in the *Figures 5, 6.*

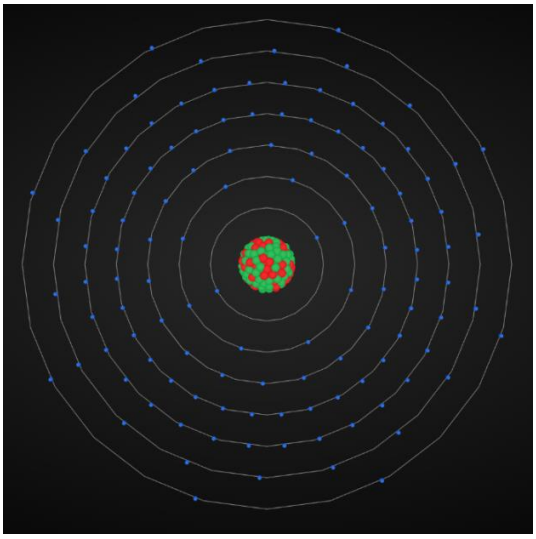

Figure 6. *Oganesson atom orbits generated using 24 subdivisions.*

As we can notice, the outermost orbit quality is lower compared to the *innermost* orbit because the distance between vertices is longer, we have set a default resolution value of 100 vertices per orbit since it suits all orbits.

3.3.2. Moving Electrons Around the Nucleus

This is achieved inside the draw function by updating electrons rotation on each render cycle. We are utilizing the *userData* object which is a predefined empty object (by default) for every mesh generated by THREE.js that can be used to store any information required for application logic.

In our case, we are storing a displacement vector from the center of the atom (world origin) and applying a small rotation to it on each draw cycle then, we apply the new displacement as the new position of the electron. And the movement of electrons will be appeared like *Figure 7* shows.

```
//main.ts
function draw(): void {
requestAnimationFrame(draw);
tick();
for (let i = 0; i < shells.length; i++) {
for (let j = 0; j < shells[i].length; j++) {
shells[i][j].userData.dir.applyAxisAngle(UP, degToRad(-SPEED * deltaTime));
shells[i][j].position.copy(shells[i][j].userData.dir);
}
}
renderer.render(scene, camera);
}
```

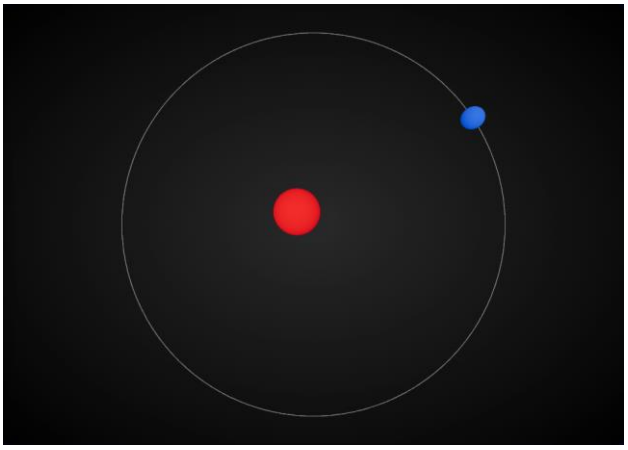

Figure 7. *Electron movement of Hydrogen atom.*

3.4. Framerate Independence & Time Scaling

Realtime applications render cycles must have consistent updates that are independent from the rendering framerate whether the application is running on a high-end device that is capable of pushing high number of frames per second to the screen or a lower end device that is less capable.

To achieve framerate independent updates, we calculate the time elapsed between each two render cycles (the current one and the one before it) which is commonly referred to as delta time in game engines such as Unity 3D.

This is very close to how a real clock ticks, we start by setting a *time* and *deltatime* variables equal to zero on initialization. Then, on each cycle we subtract *time* from *performace.now()* which returns the number of milliseconds passed since the start of the application to get the *deltatime* between cycles. After that, we update *time* to the current time and when the function is called again, the value will be updated and the next *deltatime* will be calculated:

```
//time.ts
let time = 0;
let deltaTime = 0;
let timeScale = 1;
const scales = [0.25, 0.5, 0.75, 1, 2, 4, 8];
function tick() {
// Milliseconds to seconds convertion
deltaTime = (performance.now() * 0.001 - time) * timeScale;
time = performance.now() * 0.001;
}
```

Simply multiply any value by *deltaTime* to make it framerate independent.

The upcoming examples compares speed change for 1 second performed on different framerates to show the effect of delta timing as *Figure 8* illustrates.

We are setting a consistent linear framerate in each case for demonstration purposes but in real applications the framerate can be changing all the time based on various performance factors (processing power, memory, rendering complexity …etc.).

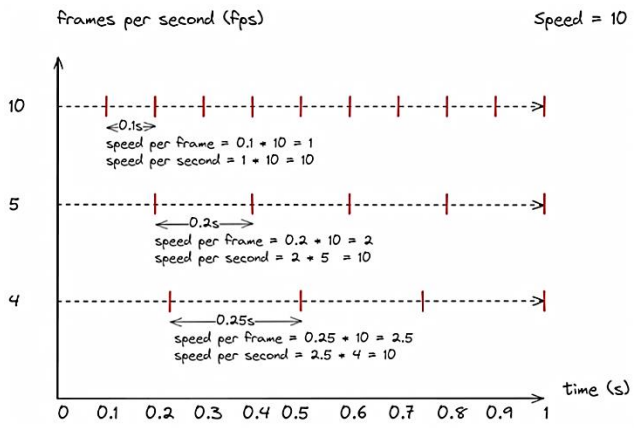

Figure 8. *Delta timing effect for different framerates.*

We can see that:

1. Speed per second is consistent regardless of framerate.
2. Higher framerate results in smoother updates and better user experience without interfering with the application consistency.

Similarly, *timescale* is used to control visualization speed

1. timescale = 1: normal simulation speed.
2. timescale > 1: faster simulation.

3. timescale < 1: slower simulation (slow motion).

The *Figure 9* shows the effective of choosing different time scales values.

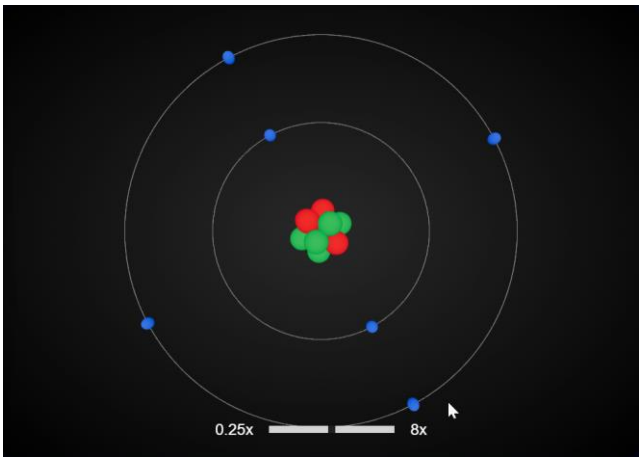

Figure 9. *Carbon atom rendering at different time scales.*

3.5. Leveraging TypeScript to Enhance Type Safety and Maintainability in Three.js Development

TypeScript has emerged as a popular choice for developing JavaScript applications, particularly in projects where type safety and code maintainability are paramount. TypeScript introduces static type checking, which involves annotating variables and functions with their expected data types. These annotations are checked by the TypeScript compiler, preventing type errors and ensuring type safety during development.

Three.js offers a rich ecosystem of components and functionalities for building 3D scenes and interactions. However, the dynamic nature of JavaScript can lead to type-related issues, such as type mismatches and unexpected behavior. These issues can make it difficult to debug and maintain code, particularly in large and complex projects.

3.5.1. Type Annotations

TypeScript type annotations are applied throughout the project, explicitly defining the types of variables, functions, and objects. This ensures type safety and helps prevent type-related errors during development.

3.5.2. Type Guards and Type Inference

TypeScript provides type guards and type inference mechanisms, allowing developers to write more concise and expressive code while maintaining type safety. These features enable dynamic checks based on variable values and improve code readability. The integration of TypeScript into the project involves applying type annotations to various components, including:

Geometry Definitions: Geometry objects, such as spheres, cubes, and meshes, are annotated with their corresponding data types, ensuring type safety in operations involving geometry creation, manipulation, and rendering.

Material Properties: Material properties, such as color, texture, and lighting parameters, are explicitly typed, preventing type mismatches and ensuring consistent usage.

Using TypeScript's static type checking has significantly reduced the occurrence of type errors while improving code reliability and reducing debugging time. Type annotations have made the code more self-documenting, making it easier for developers to understand and maintain the codebase.

TypeScript's type system has facilitated code refactoring, allowing us to make changes more confidently without introducing type errors. TypeScript's comprehensive type system and debugging tools provide a more robust and efficient development experience.

3.6. User-Driven Zoom and Exploration

The built-in *OrbitControls* object handles camera controls to adjust the camera's position and orientation according to user input which enables users to freely explore the scene and zoom in on specific parts of the atom.

We are setting a minimum and maximum distance constraints based on viewed atom properties to eliminate unexpected and annoying behaviors such as zooming too far away from the atom or zooming in to a point where the camera gets inside the nucleus mesh.

```
//main.ts
controls.minDistance = nucleusRadius + 2;
controls.maxDistance = maxOrbitRadius * 2.5;
```

We also initialize the camera at a distance from the world origin proportional to current atom radius and thus keeping the whole atom inside camera view frustum and avoid the confusion of going outside screen borders when the application starts. The aspect ratio is based on browser window to avoid distortion and providing a responsive experience on various screen sizes and aspect ratios.

The following piece of code initializes a new *PerespectiveCamera* and *OrbitControls* objects with the correct properties and aspect ratio and listens to the browser window resize events to make required adjustments:

```
//MainCamera.ts
export const camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 100);
export const controls = new OrbitControls(camera, canvas);
camera.position.copy(UP)
camera.lookAt(ZERO);
window.addEventListener("resize", () => {
camera.aspect = window.innerWidth / window.innerHeight;
camera.updateProjectionMatrix();
});
//MainScene.ts
renderer = new THREE.WebGLRenderer({ canvas, antialias: true, alpha: true });
renderer.toneMapping = THREE.ACESFilmicToneMapping;
```

```
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.setPixelRatio(window.devicePixelRatio);
window.addEventListener("resize", () => {
renderer.setSize(window.innerWidth, window.innerHeight);
});
```

Let us go through the oxygen atom as *Figure 10* shows.

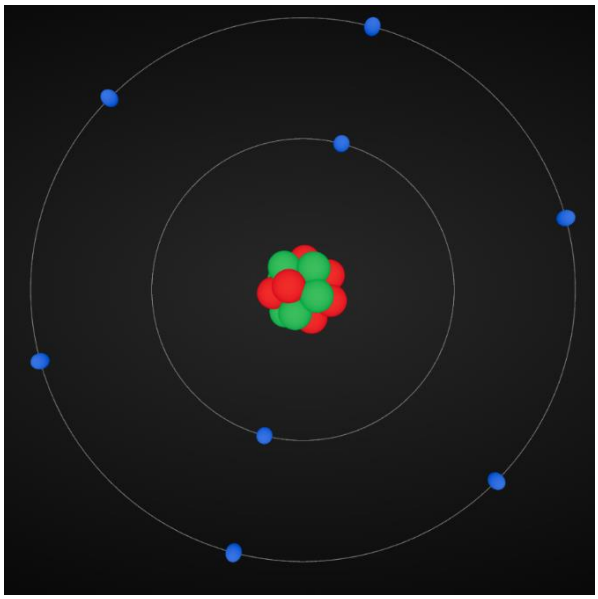

Figure 10. *Automatic height adjustment for a low radius atom (Oxygen).*

The camera always maintains a reasonable distance from the rendered atom even when rendering a larger atom such as Oganesson atom in *Figure 11*.

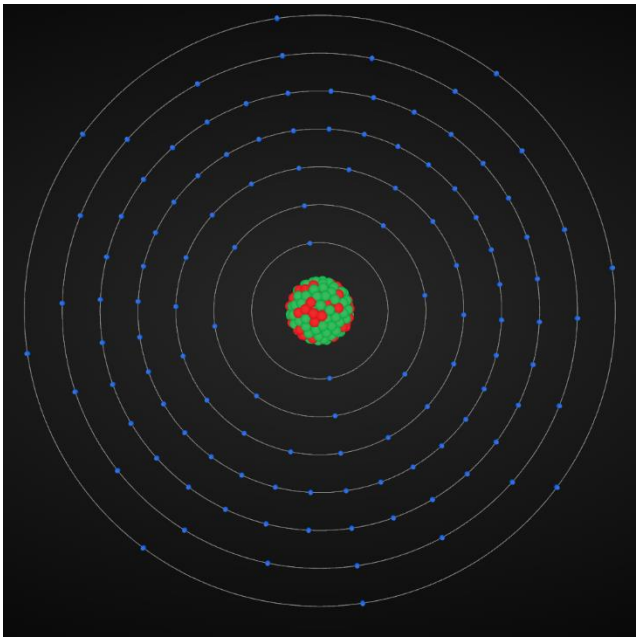

Figure 11. *Automatic height adjustment for a low radius atom (Oganesson).*

This perspective projection mode used by the *PerspectiveCamera* is designed to mimic the way the human eye sees. It is the most common projection mode used for rendering a 3D scene.

The camera view frustum is defined by its field of view, aspect ratio, near and far clipping planes which maps points from 3D space to 2D screen coordinates, determining how objects appear in the rendered scene.

4. Results

The implemented system provides smooth and accurate atomic representations. Performance evaluations show that using delta time calculations maintains consistent animation across devices. A usability study with engineering students confirmed that the interactive features significantly enhanced comprehension of atomic structures.

The current version of the application is accessible at bohr-model-visualization.vercel.app, serving as a valuable resource for learning and demonstrations. Its user-friendly interface enables users to select any atom from the periodic table and renders it with a simple click of a button, much like the way Bohr diagrams are presented to students. The source code is available on our GitHub repository github.com/mohamadtaky/bohr-model-Visualization, Please be aware that the current status of the project is subject to modifications, as additional enhancements and features are still planned to be added. The existing state should not be considered as the final version, as further refinements and updates are anticipated.

5. Conclusion

This research introduces a real-time interactive 3D visualization tool for Bohr's atomic model. By leveraging WebGL and force-directed algorithms, the project enhances educational tools for atomic physics. Future improvements include quantum mechanical adaptations and augmented reality integrations.

Abbreviations

VR	Virtual Reality
3D	Three Dimensional
JSON	JavaScript Object Notation
Three.js	JavaScript Library Used for Creating and Displaying 3D Graphics and Animations in Web Browsers

Acknowledgments

We thank Al-Wataniya Private University for technical support and for providing access to research facilities.

Author Contributions

Zaid Kraitem: Conceptualization, Data curation, Formal Analysis, Methodology, Project administration, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing

Hamza Alhaj: Conceptualization, Data curation, Formal Analysis, Methodology, Project administration, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing

Mohamad Taky: Conceptualization, Data curation, Formal Analysis, Methodology, Project administration, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing

Funding

No external funding was received for this research.

Data Availability Statement

The application is written in TypeScript for enhanced development experience to make it easier to develop and add more features to the project, which can be found at bohr-model-visualization.vercel.app. For those interested in viewing the source code, you can find it under our GitHub repository:
github.com/mohamadtaky/bohr-model-visualization.

Conflicts of Interest

The authors declare no conflicts of interest.

References

[1] Bohr, N. (1913). "On the Constitution of Atoms and Molecules." Philosophical Magazine, 26(151), 1-25.

[2] Foley, J., van Dam, A., Feiner, S., & Hughes, J. (2020). *Computer Graphics: Principles and Practice*. Addison-Wesley.

[3] Angel, E. (2018). *Interactive Computer Graphics: A Top-Down Approach with OpenGL*. Pearson.

[4] Hearn, D., & Baker, M. P. (2022). *Computer Graphics with WebGL*. Prentice Hall.

[5] Kobourov, S. (2013). "Force-Directed Drawing Algorithms." Handbook of Graph Drawing and Visualization, 385-386.

[6] Y. Li et al., "NPIPVis: A Visualization System Involving NBA Visual Analysis and Integrated Learning Model Prediction," *Virtual Reality & Intelligent Hardware*, vol. 4, no. 5, pp. 444–458, 2022.

[7] X. Chen et al., "PartLabeling: A Label Management Framework in 3D Space," *Virtual Reality & Intelligent Hardware*, vol. 5, no. 6, pp. 490–508, 2023.

[8] K. Zhang et al., "Depth of Field Rendering Using Multilayer-Neighborhood Optimization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 8, pp. 2546–2559, 2019.

[9] L. Wang et al., "Efficient Binocular Rendering of Volumetric Density Fields with Coupled Adaptive Cube-Map Ray Marching for Virtual Reality," *IEEE Transactions on Visualization & Computer Graphics*, vol. 30, no. 10, pp. 6625–6638, 2024.

[10] Mozilla Developers. (2024). "Window: requestAnimationFrame method." MDN Web Docs.

[11] Coleman, D. (2019). "Understanding Delta Time." Medium.

[12] Adachi, S., & Tanimura, Y. (2023). "Quantum Trajectories in Atomic Visualization." *Journal of Computational Physics*, 312(2), 554-571.

[13] Lin, X., & Zhang, Y. (2022). "Web-Based Scientific Simulations Using WebGL and Three.js." *Advances in Visualization Science*, 9(3), 201-215.

[14] Li, W., & Chen, Z. (2021). "Enhancing Interactive Learning with 3D Atom Models." *Educational Technology & Society*, 24(4), 102-119.

[15] Jizhe Xia, Qunying Huang, Zhipeng Gui & Wei Tu (2024). “Web-Based Mapping and Visualization Packages”. The Visual Computer. pp283-314.

[16] Xiaokun Wang, Yanrui Xu, Sinuo Liu, Bo Ren, Jiří Kosinka, Alexandru C. Telea, Jiamin Wang, Chongming Song, Jian Chang, Chenfeng Li, Jian Jun Zhang & Xiaojuan Ban. (2024). “Physics-based fluid simulation in computer graphics: Survey, research trends, and challenges”. The Visual Computer. Volume 10, pages 803–858.

Biography

Zaid Kraitem PhD in Computer Science, graduated from Latakia University in 2018. Lecturer at Al-Wataniya Private University in Hama. Over five years of teaching experience. More than 10 published research papers in the field of biomedical engineering. Expert in digital marketing, artificial intelligence, and deep learning.

Research Field

Zaid Kraitem: Computer Graphics, Computer Science, Computer Vision, Image Representation and Visualization, Digital Image Processing