生物演化史 - 页面概览 - 总报告

目录

4-screen-biological-classification-sunburst-2D.html.md

实验报告：　物分类可视化系统

一、引言

随着生物科学的不断发展，生物分类学作为一门重要的学科，越来越受到重视。为了更好地理解生物的分类关系，我开发了一款基于 Web 的生物分类可视化系统。该系统利用 D3.js 库实现了生物分类的交互式可视化，用户可以通过旋转、缩放等操作深入探索生物分类的层次结构。

二、创新功能点

1. **交互式可视化**：用户可以通过鼠标拖拽、键盘操作等方式与可视化图形进行交互，增强了用户体验。
2. **放大镜功能**：用户可以在图形上方显示放大镜，实时查看细节，适合观察复杂的生物分类结构。
3. **搜索功能**：用户可以通过输入生物名称快速定位到相应的分类节点，提升了查找效率。
4. **信息面板**：当用户悬停在某个节点上时，右侧信息面板会显示该生物的详细信息，包括图片、描述等。
5. **全屏模式**：用户可以选择进入全屏模式，以便更好地查看可视化效果。

三、算法描述

3.1 数据结构

系统使用树形结构来表示生物分类数据。每个节点包含以下信息：`- name`：生物名称 `- src`：生物图片链接 `- description`：生物描述 `- children`：子节点数组

3.2 可视化算法

使用 D3.js 库的分区布局（partition layout）和弧生成器（arc generator）来创建生物分类的可视化图形。具体步骤如下：

1. **数据处理**：将原始数据转换为 D3.js 可以处理的层次结构。
2. **创建 SVG 元素**：使用 D3.js 创建 SVG 元素，并设置其宽高和中心点。
3. **生成弧形**：使用弧生成器根据节点的深度和宽度生成相应的弧形路径。
4. **绘制路径**：将生成的路径添加到 SVG 中，并根据节点的深度设置不同的颜色。
5. **添加文本标签**：在每个节点的中心位置添加文本标签，显示生物名称。

3.3 交互功能

- **拖拽旋转**：通过监听鼠标事件，计算当前鼠标位置与中心点的角度差，更新当前旋转角度。
- **键盘控制**：使用 WASD 和方向键控制同层节点的移动和旋转。
- **搜索功能**：通过输入框实时过滤节点，并高亮显示匹配的节点。

四、代码实现

4.1 主要代码片段

以下是实现生物分类可视化的关键代码片段：

```
// 创建 SVG 元素
const svg = d3.select("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${width/2},${height/2})`);

// 创建分区布局
const partition = data => {
    const root = d3.hierarchy(data)
        .sum(d => 1)
        .sort((a, b) => b.value - a.value);
    return d3.partition()
        .size([2 * Math.PI, radius])
        .padding(0.005)(root);
};

// 创建弧生成器
const arc = d3.arc()
    .startAngle(d => d.x0)
    .endAngle(d => d.x1)
    .padAngle(0.005)
    .padRadius(radius / 2)
    .innerRadius(d => d.y0)
    .outerRadius(d => d.y1 - 1;

// 处理数据并绘制路径
const root = partition(data);
const path = svg.append("g")
    .selectAll("path")
    .data(root.descendants().slice(1))
    .join("path")
    .attr("fill", d => color(d))
    .attr("d", d => arc(d.current));
```

4.2 关键库函数引用

- **D3.js**：用于数据驱动的文档操作，提供了强大的数据可视化功能。
  - `d3.hierarchy(data)`：将数据转换为层次结构。
  - `d3.partition()`：创建分区布局。
  - `d3.arc()`：生成弧形路径。
- **HTML2Canvas**：用于将 HTML 元素转换为 Canvas 图像，便于后续处理和下载。

4.3 事件处理

以下是处理用户交互的代码示例：

```
// 添加拖拽功能
d3.select("body")
    .style("cursor", "move")
    .on("mousedown", dragstarted)
    .on("mousemove", dragged)
    .on("mouseup", dragended);

function dragstarted(event) {
    isDragging = true;
    dragStartAngle = Math.atan2(event.pageY - height/2, event.pageX - w
idth/2) * 180 / Math.PI;
    dragStartRotation = currentRotation;
}

function dragged(event) {
    if (!isDragging) return;
    const currentAngle = Math.atan2(event.pageY - height/2, event.pageX
 - width/2) * 180 / Math.PI;
    const deltaAngle = currentAngle - dragStartAngle;
    currentRotation = dragStartRotation + deltaAngle;
    svg.attr("transform", `translate(${width/2},${height/2}) rotate(${c
urrentRotation})`);
}

function dragended() {
    isDragging = false;
}
```

五、总结

本实验通过 D3.js 库实现了一个生物分类可视化系统，提供了丰富的交互功能和用户体验。系统不仅能够直观地展示生物分类的层次结构，还通过搜索和放大镜功能增强了用户的探索能力。未来，我们计划进一步优化系统性能，增加更多生物分类数据，并探索使用 Three.js 实现 3D 可视化效果，以提升用户体验。

六、参考文献

1. D3.js 官方文档：https://d3js.org/
2. HTML2Canvas 文档：https://html2canvas.hertzen.com/
3. 生物分类学相关文献和资料。

4-screen-biological-evolution-dna-3D.html.md

实验报告：生物演化史-DNA 双螺旋-3D 可视化

一、引言

本实验旨在通过 3D 可视化技术展示生物演化史中的 DNA 双螺旋结构。利用 Three.js 库实现 3D 图形渲染，并结合 D3.js 库处理数据，提供交互式的用户体验。该项目不仅展示了生物演化的过程，还通过时间旅行模式让用户能够直观地理解生物演化的时间线。

二、创新功能点

1. **时间旅行模式**：用户可以通过控制按钮在生物演化的时间线上前进或倒退，直观地观察不同时间节点的生物演化事件。
2. **动态交互**：通过鼠标悬停事件，用户可以查看每个生物演化事件的详细信息，包括名称、时间和描述。
3. **全屏显示功能**：用户可以选择全屏模式，以更好地体验 3D 可视化效果。
4. **播放速度控制**：用户可以调整播放速度，以适应不同的观察需求。

三、算法描述

3.1 数据处理

使用 D3.js 库处理生物演化数据，数据结构为树形结构，包含每个事件的名称、时间、描述和子事件。通过递归函数将数据扁平化，便于在 3D 场景中展示。

3.2 3D 场景构建

使用 Three.js 库构建 3D 场景，包括相机、光源、物体等。通过创建 DNA 双螺旋结构和星空背景，增强视觉效果。

3.3 动画与交互

使用 TWEEN.js 库实现平滑动画效果，结合键盘事件和鼠标事件实现用户交互。用户可以通过按键控制相机的旋转，或通过鼠标悬停查看事件详情。

四、代码实现

4.1 主要库引用

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/three@0.128.0/examples/js/controls/OrbitControls.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/tween.js/18.9.5/Tween.min.js"></script>
```

4.2 3D 场景初始化

```
function init() {
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x000000);

    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight, 0.1, 10000);
    camera.position.set(100, 0, 100);
    camera.lookAt(0, 0, 0);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.maxDistance = 500;
    controls.minDistance = 50;

    createStarField();
    createDNASpiral();
    createTimelineEvents();

    animate();
}
```

4.3 创建 DNA 双螺旋

```
function createDNASpiral() {
    const curve1Points = [];
    const curve2Points = [];
    const radius = 20;
    const height = 500;
    const turns = 25;

    for(let i = 0; i <= 360 * turns; i++) {
        const angle = (i * Math.PI) / 180;
        const y = (i / (360 * turns)) * height - height/2;

        curve1Points.push(new THREE.Vector3(radius * Math.cos(angle), y,
```

```
 radius * Math.sin(angle)));
        curve2Points.push(new THREE.Vector3(radius * Math.cos(angle + M
ath.PI), y, radius * Math.sin(angle + Math.PI)));
    }

    const curve1Geometry = new THREE.BufferGeometry().setFromPoints(cur
ve1Points);
    const curve2Geometry = new THREE.BufferGeometry().setFromPoints(cur
ve2Points);

    const material = new THREE.LineBasicMaterial({ color: 0x00ff88, opa
city: 0.8, transparent: true });

    const dnaStrand1 = new THREE.Line(curve1Geometry, material);
    const dnaStrand2 = new THREE.Line(curve2Geometry, material);

    scene.add(dnaStrand1);
    scene.add(dnaStrand2);
}
```

4.4 创建时间线事件

```
function createTimelineEvents() {
    function processEvents(events, startY = -250) {
        const sortedEvents = flattenEvents(events);
        const totalHeight = 500;
        const heightPerEvent = totalHeight / sortedEvents.length;

        sortedEvents.forEach((event, index) => {
            const geometry = new THREE.SphereGeometry(1.2, 32, 32);
            const material = new THREE.MeshPhongMaterial({ color: getEv
entColor(event.timeValue), emissive: getEventColor(event.timeValue), em
issiveIntensity: 0.5 });
            const sphere = new THREE.Mesh(geometry, material);
            const y = startY + (index * heightPerEvent);
            sphere.position.set(20 * Math.cos(index), y, 20 * Math.sin
(index));
            sphere.userData = { name: event.name, time: event.time, des
cription: event.description };
            timelineObjects.add(sphere);
        });
    }
}
```

五、总结

本实验通过 Three.js 和 D3.js 库实现了生物演化史的 3D 可视化，展示了 DNA 双螺旋结构及其演化过程。通过创新的时间旅行模式和动态交互功能，用户能够更直观地理解生物演化的复杂性。未来可以进一步优化数据处理和渲染性能，以支持更大规模的数据集和更复杂的可视化效果。

4-screen-biological-evolution-force-2D.html.md

实验报告：生物演化史-力导向-2D 可视化

一、引言

本实验旨在通过力导向图的方式展示生物演化史，利用 D3.js 库实现数据的可视化。该项目不仅展示了生物演化的时间线，还通过交互功能增强了用户体验。本文将详细介绍该项目的创新功能、算法描述、代码实现及所用库函数的具体细节。

二、创新功能点

1. **交互式节点信息展示**：用户可以通过鼠标悬停在节点上查看详细信息，节点信息包括名称、时间和描述等。
2. **节点拖拽功能**：用户可以通过拖拽节点来重新排列图形，增强了可视化的灵活性。
3. **全屏显示功能**：用户可以选择全屏模式以更好地查看图形。
4. **动态缩放和平移**：用户可以通过鼠标滚轮进行缩放，按住鼠标左键进行平移，提升了图形的可操作性。
5. **实时更新的力导向模拟**：节点之间的关系通过力导向算法动态更新，提供了更真实的视觉效果。

三、算法描述

3.1 力导向算法

力导向算法是一种基于物理模型的图形布局算法。每个节点被视为一个带有电荷的粒子，节点之间的连接线被视为弹簧。算法通过计算节点之间的斥力和吸引力来调整节点的位置，最终达到一种平衡状态。

- **斥力**：节点之间的斥力使得节点尽量远离，避免重叠。
- **吸引力**：连接线的吸引力使得相连的节点靠近，形成合理的结构。

3.2 D3.js 库的使用

D3.js 是一个强大的数据可视化库，提供了丰富的功能来处理数据和生成图形。以下是本项目中使用的主要 D3.js 功能：

- **选择和操作 DOM 元素**：使用 `d3.select()`选择 SVG 元素并进行操作。
- **数据绑定**：使用 `data()`方法将数据绑定到 DOM 元素。
- **力导向模拟**：使用 `d3.forceSimulation()`创建力导向图，并通过 `force()`方法添加不同的力。

四、代码实现

4.1 HTML 结构

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>生物演化史-力导向-2D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <script src="biological-evolution-data.js"></script>
    <style>
        /* CSS 样式省略 */
    </style>
</head>
<body>
    <div id="background-container" class="background-image-container"><
/div>
    <div class="controls">
        <!-- 控制按钮 -->
    </div>
    <div id="container">
        <div id="graph"></div>
        <div id="magnifier"></div>
    </div>
    <script>
        // JavaScript 代码省略
    </script>
</body>
</html>
```

4.2 JavaScript 实现细节

4.2.1 初始化 SVG 容器

```
let width = window.innerWidth;
let height = window.innerHeight;

const svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height);
```

4.2.2 力导向模拟

```
const simulation = d3.forceSimulation()
    .force("link", d3.forceLink().id(d => d.id).distance(100)) // 弹簧
力
```

```
    .force("charge", d3.forceManyBody().strength(-300)) // 节点间的斥力
    .force("collide", d3.forceCollide().radius(50)) // 防止节点重叠的碰撞
    .force("center", d3.forceCenter(0, 0).strength(0.1)); // 添加中心力
```

4.2.3 更新函数

```
function update(source) {
    // 计算新的树布局
    tree(root);

    const nodes = root.descendants();
    const links = root.links();

    // 更新力导向模拟
    simulation.nodes(nodes);
    simulation.force("link").links(links);
    simulation.alpha(1).restart();

    // 更新节点和连接线
    // 省略具体实现
}
```

4.3 交互功能实现

4.3.1 鼠标悬停事件

```
nodeEnter.on("mouseover", showNodeInfo)
    .on("mouseout", hideNodeInfo);
```

4.3.2 拖拽功能

```
const dragHandler = d3.drag()
    .subject(function (event, d) {
        return { x: d.x, y: d.y };
    })
    .on("start", dragstarted)
    .on("drag", dragged)
    .on("end", dragended);

nodeUpdate.call(dragHandler);
```

五、总结

本实验通过 D3.js 实现了生物演化史的力导向可视化，展示了生物演化的复杂关系。通过交互功能的设计，用户可以更直观地理解生物演化的过程。未来可以考虑引入 Three.js 库，进一步增强 3D 可视化效果，提升用户体验。

六、参考文献

1. D3.js 官方文档: https://d3js.org/
2. 力导向图相关文献与研究

以上是本实验的详细报告，涵盖了创新功能、算法描述、代码实现及库函数的具体细节。希望对读者理解该项目有所帮助。

4-screen-biological-evolution-star-3D.html.md

实验报告：生物演化史-星空图-3D

一、引言

本实验旨在通过 3D 可视化技术展示生物演化史，利用 Three.js 库实现动态交互效果，帮助用户更直观地理解生物演化的过程。该项目结合了数据可视化和交互设计，提供了多种功能，如节点缩放、连接线显示/隐藏、全屏模式等，增强了用户体验。

二、创新功能点

1. **动态交互**：用户可以通过鼠标滚轮缩放视图，左键拖动视角，点击节点聚焦，提升了交互性。
2. **节点信息展示**：点击节点后，展示该节点的详细信息，包括时间、描述和相关图片，增强了信息的可获取性。
3. **连接线控制**：用户可以选择隐藏或显示节点之间的连接线，便于用户根据需要调整视图。
4. **星空背景**：通过动态星空背景，增加了视觉美感，使得整个可视化效果更加生动。
5. **呼吸效果**：节点具有呼吸效果，提升了视觉吸引力，使得用户在观察时更具沉浸感。

三、算法描述

3.1 数据结构

本项目使用树形结构来表示生物演化的各个节点。每个节点包含以下属性： - `name`：节点名称 - `time`：节点时间 - `description`：节点描述 - `children`：子节点数组

3.2 主要算法

1. **节点创建**：根据树形数据结构递归创建节点，使用 Three.js 的 `SphereGeometry` 生成节点的 3D 模型。
2. **连接线创建**：使用贝塞尔曲线连接节点，增强视觉效果。通过 `QuadraticBezierCurve3` 实现曲线连接。
3. **动态更新**：通过事件监听器实现动态更新节点信息和视图，确保用户交互的实时反馈。

四、代码实现

4.1 HTML 结构

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>生物演化史-星空图-3D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <style>
        /* 样式定义 */
    </style>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/t
hree.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/three@0.128.0/examples/js
/controls/OrbitControls.js"></script>
    <script src="biological-evolution-data.js"></script>
</head>
<body>
    <!-- 页面内容 -->
</body>
</html>
```

4.2 JavaScript 实现

4.2.1 初始化场景

```
function init() {
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x000000);

    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.i
nnerHeight, 0.1, 10000);
    camera.position.set(120, 270, 500);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;

    createStarField();
    networkGroup = createNetworkVisualization();

    animate();
}
```

4.2.2 创建星空背景

```
function createStarField() {
    const starsGeometry = new THREE.BufferGeometry();
    const starsMaterial = new THREE.PointsMaterial({
        color: 0xFFFFFF,
        size: 0.3,
        transparent: true,
        opacity: 1.0,
        sizeAttenuation: true
    });

    const starsVertices = [];
    for(let i = 0; i < 15000; i++) {
        const x = (Math.random() - 0.5) * 2000;
        const y = (Math.random() - 0.5) * 2000;
        const z = (Math.random() - 0.5) * 2000;
        starsVertices.push(x, y, z);
    }

    starsGeometry.setAttribute('position', new THREE.Float32BufferAttri
bute(starsVertices, 3));
    starField = new THREE.Points(starsGeometry, starsMaterial);
    scene.add(starField);
}
```

4.2.3 创建网络可视化

```
function createNetworkVisualization() {
    const networkGroup = new THREE.Group();

    function createNodes(data, position = new THREE.Vector3(), level =
0, angle = 0) {
        const nodeGeometry = new THREE.SphereGeometry(networkParameters.
nodeSize, 32, 32);
        const color = categoryColors[data.name] || new THREE.Color(0xff
ffff);

        const nodeMaterial = new THREE.MeshPhongMaterial({
            color: color,
            emissive: color,
            emissiveIntensity: 0.5,
            transparent: true,
            opacity: 0.8
        });

        const node = new THREE.Mesh(nodeGeometry, nodeMaterial);
        node.position.copy(position);
        node.userData = data;

        networkGroup.add(node);
```

```
        if (data.children) {
            const childCount = data.children.length;
            const radius = networkParameters.radius - level * 6;
            const angleStep = (Math.PI * 2) / childCount;
            const startAngle = 0;

            data.children.forEach((child, index) => {
                const childAngle = startAngle + angleStep * index;
                const childPosition = new THREE.Vector3(
                    position.x + Math.cos(childAngle) * radius,
                    position.y + (level * networkParameters.levelHeigh
t),
                    position.z + Math.sin(childAngle) * radius
                );

                createNodes(child, childPosition, level + 1, childAngl
e);
            });
        }
    }

    createNodes(treeData);
    scene.add(networkGroup);
    return networkGroup;
}
```

4.3 事件处理

4.3.1 鼠标点击事件

```
window.addEventListener('click', onMouseClick);

function onMouseClick(event) {
    mouse.x = (event.clientX / (window.innerWidth * 0.5)) * 2 - 1;
    mouse.y = -(event.clientY / (window.innerHeight * 0.5)) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(networkGroup.children.
filter(child => child instanceof THREE.Mesh));

    if (intersects.length > 0) {
        const selectedObject = intersects[0].object;
        const nodeData = selectedObject.userData;
        showEventDetails(nodeData);
    } else {
        hideEventDetails();
    }
}
```

五、库函数引用

5.1 Three.js

- `THREE.Scene`：创建 3D 场景。
- `THREE.PerspectiveCamera`：设置透视相机。
- `THREE.WebGLRenderer`：渲染 3D 图形。
- `THREE.SphereGeometry`：生成球体几何体，用于节点表示。
- `THREE.LineBasicMaterial`：创建线条材质，用于连接线。

5.2 D3.js

虽然本项目主要使用 Three.js 进行 3D 可视化，但 D3.js 可以用于处理和转换数据，特别是在需要将数据转换为树形结构时。

六、结论

本实验通过 Three.js 实现了生物演化史的 3D 可视化，提供了丰富的交互功能，增强了用户体验。未来可以考虑进一步优化性能，增加更多的交互功能，如节点搜索、时间轴控制等，以提升可视化效果和用户体验。

4-screen-biological-evolution-tree-2D.html.md

实验报告：生物演化树可视化

一、引言

本实验旨在通过可视化技术展示生物演化树，帮助用户更直观地理解生物的演化过程。我们使用了 D3.js 库来构建树形结构，并通过交互式功能增强用户体验。该项目的创新点在于结合了多种可视化方式，并实现了动态过滤和搜索功能。

二、创新功能点

1. **动态过滤**：用户可以通过下拉框选择不同的地质时代，实时更新树形结构，展示特定时期的生物演化情况。
2. **搜索功能**：用户可以输入生物名称进行搜索，系统会高亮显示匹配的节点，并自动滚动到该节点位置。
3. **工具提示**：鼠标悬停在节点上时，显示详细信息，包括生物名称、图片和描述，增强了信息的可读性。
4. **懒加载图片**：为了提高性能，使用懒加载技术，仅在节点进入视口时加载相关图片。
5. **全屏显示**：用户可以选择全屏模式，提供更好的视觉体验。

三、算法描述

1. 数据结构

我们使用 D3.js 提供的层次结构（hierarchy）来构建树形数据。每个节点包含以下属性： - `name`: 生物名称 - `src`: 图片链接 - `description`: 生物描述 - `children`: 子节点数组

2. 树形布局

使用 D3.js 的 `d3.tree()` 方法创建树形布局。根据节点数量动态计算 SVG 高度，以确保所有节点都能在视口内显示。

3. 事件处理

- **鼠标事件**：通过 `mouseover` 和 `mouseout` 事件处理工具提示的显示与隐藏。
- **搜索事件**：监听输入框的 `input` 事件，实时过滤并显示搜索结果。

四、代码实现

1. HTML 结构

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>生物演化史-Tree-2D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <div class="control-buttons">
        <!-- 控制按钮 -->
    </div>
    <div class="time-filter">
        <div class="filter-title">Tidy Tree</div>
        <select id="era-select">
            <option value="all">全部时期</option>
            <!-- 其他选项 -->
        </select>
    </div>
    <div class="search-container">
        <input type="text" id="search-input" placeholder="搜索节点...">
        <div id="search-results"></div>
    </div>
```

```
    <div id="tree"></div>
    <div class="tooltip"></div>
    <script src="biological-evolution-data.js"></script>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

2. JavaScript 代码

```
// 设置画布尺寸
const width = window.innerWidth - 50;
let height = 800; // 初始高度
const margin = {top: 20, right: 90, bottom: 20, left: 90};

// 创建 SVG 容器
const svg = d3.select("#tree")
    .append("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

// 创建树布局
const tree = d3.tree()
    .size([height - margin.top - margin.bottom, width - margin.left - m
argin.right]);

// 加载数据并渲染树
function renderTree(data, filter = 'all') {
    // 清除现有内容
    svg.selectAll("*").remove();

    // 过滤数据
    let filteredData = JSON.parse(JSON.stringify(data));
    if (filter !== 'all') {
        filteredData.children = filteredData.children.filter(d => d.nam
e === filter);
    }

    // 创建层次结构
    const root = d3.hierarchy(filteredData);

    // 计算需要的高度
    const nodeCount = root.descendants().length;
    const minHeightPerNode = 40; // 每个节点的最小高度
    const calculatedHeight = nodeCount * minHeightPerNode;
```

```
// 更新高度
let svgHeight = Math.max(height, calculatedHeight);

// 更新树形布局
const treeLayout = d3.tree()
    .size([svgHeight - margin.top - margin.bottom,
        filter === 'all' ?
            width - margin.left - margin.right :
            (width - margin.left - margin.right) * 0.6
    ]);

// 调整 SVG 容器高度
d3.select("#tree svg")
    .attr("height", svgHeight);

// 计算节点位置
treeLayout(root);

// 绘制连接线
const links = svg.selectAll(".link")
    .data(root.links())
    .enter()
    .append("path")
    .attr("class", "link")
    .attr("d", d3.linkHorizontal()
        .x(d => d.y)
        .y(d => d.x));

// 绘制节点
const nodes = svg.selectAll(".node")
    .data(root.descendants())
    .enter()
    .append("g")
    .attr("class", "node")
    .attr("transform", d => `translate(${d.y},${d.x})`);

// 添加节点圆圈
nodes.append("circle")
    .attr("r", 5);

// 添加图片
nodes.append("image")
    .attr("data-src", d => d.data.src)
    .attr("class", "node-image")
    .attr("x", -15)
    .attr("y", -15)
    .attr("width", 30)
    .attr("height", 30);
```

```
    // 添加文本标签
    nodes.append("text")
        .attr("dy", ".35em")
        .attr("x", d => d.children ? -40 : 40)
        .attr("text-anchor", d => d.children ? "end" : "start")
        .text(d => d.data.name);
}

// 加下拉框事件监听
d3.select("#era-select").on("change", function() {
    const selectedEra = this.value;
    renderTree(treeData, selectedEra);
});

// 初始渲染
renderTree(treeData);
```

3. D3.js 库函数引用

- `d3.select()`: 用于选择 DOM 元素。
- `d3.hierarchy()`: 将数据转换为层次结构。
- `d3.tree()`: 创建树形布局。
- `d3.linkHorizontal()`: 生成连接线的路径。

五、总结

本实验通过 D3.js 实现了生物演化树的可视化，提供了动态过滤、搜索和工具提示等功能，增强了用户的交互体验。通过使用懒加载技术，优化了性能，确保了在处理大量数据时的流畅性。未来可以考虑引入 Three.js 实现三维可视化，进一步提升展示效果。

4-screen-biological-evolution-tree-3D.html.md

实验报告：生物演化史-Tree-3D 可视化项目

一、项目概述

本项目旨在通过三维可视化技术展示生物演化史，利用 `Three.js` 和 `D3.js` 库实现动态交互式的生物演化树。用户可以通过鼠标操作进行视角旋转、缩放和节点聚焦，直观地了解不同地质年代的生物演化过程。

二、创新功能点

1. **动态交互**：用户可以通过键盘和鼠标进行自由移动和视角调整，增强了用户体验。
2. **节点高亮显示**：点击节点后，系统会高亮显示该节点，并展示相关信息，便于用户获取详细数据。
3. **历史记录功能**：用户的操作可以撤销和重置，方便用户进行多次尝试和调整。
4. **自适应布局**：根据不同的参数动态调整节点的布局和连接线的样式，使得可视化效果更加美观和易于理解。

三、算法描述

3.1 数据结构

项目使用树形结构来表示生物演化的各个节点。每个节点包含以下信息：`- name`：节点名称（如地质年代）。`- children`：子节点数组，表示该节点下的生物种类。`- time`：时间信息，表示该节点对应的地质年代。`- description`：节点的描述信息。

3.2 主要算法

1. **节点创建算法**：根据树形数据结构递归创建节点，并根据层级和类别设置不同的颜色和位置。
2. **曲线连接算法**：使用贝塞尔曲线连接父子节点，增强可视化效果。
3. **交互处理算法**：通过射线投射技术检测用户点击的节点，并更新视图和信息展示。

四、代码实现

4.1 主要库引用

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/three@0.128.0/examples/js/controls/OrbitControls.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/tween.js/18.6.4/tween.umd.js"></script>
```

4.2 关键代码实现

4.2.1 初始化场景

```
function init() {
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x000000);

    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight, 0.1, 10000);
```

```
    camera.position.set(-2170, 847, 792);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.maxDistance = 6000;
    controls.minDistance = 10;
    controls.target = new THREE.Vector3(0, 0, 0);

    createStarField();
    networkGroup = createNetworkVisualization();
    animate();
}
```

4.2.2 创建网络可视化

```
function createNetworkVisualization() {
    const networkGroup = new THREE.Group();

    // 定义基准角度和高度偏移
    const categoryBaseAngles = { '冥古宙': 0, '太古宙': Math.PI/3, ... };
    const categoryHeightOffset = { '冥古宙': networkParameters.heightSt
ep * 2, ... };

    function createNodes(data, position = new THREE.Vector3(0, -500, 0),
 level = 0, angle = 0) {
        const nodeGeometry = new THREE.SphereGeometry(10, 32, 32);
        const color = categoryColors[data.name] || new THREE.Color(0xff
ffff);

        const nodeMaterial = new THREE.MeshPhongMaterial({ color: color
 });
        const node = new THREE.Mesh(nodeGeometry, nodeMaterial);
        node.position.copy(position);
        networkGroup.add(node);

        if (data.children) {
            const childCount = data.children.length;
            const radius = networkParameters.radius - level * 6;
            const angleStep = (Math.PI * 2) / childCount;
            const startAngle = angle - (angleStep * (childCount - 1) /
2);

            data.children.forEach((child, index) => {
                const childAngle = startAngle + angleStep * index;
                const childPosition = new THREE.Vector3(
                    position.x + Math.cos(childAngle) * radius,
```

```
                    position.y + (level * networkParameters.levelHeigh
t),
                    position.z + Math.sin(childAngle) * radius
                );
                createNodes(child, childPosition, level + 1, childAngl
e);
            });
        }
    }

    createNodes(treeData);
    scene.add(networkGroup);
    return networkGroup;
}
```

4.2.3 交互处理

```
function onMouseClick(event) {
    mouse.x = (event.clientX / (window.innerWidth / 2)) * 2 - 1;
    mouse.y = -(event.clientY / (window.innerHeight / 2)) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(networkGroup.children.
filter(child => child instanceof THREE.Mesh));

    if (intersects.length > 0) {
        const clickedNode = intersects[0].object;
        // 高亮显示节点
        highlightNode(clickedNode);
        // 显示节点信息
        showEventDetails(clickedNode.userData);
    }
}
```

4.3 重要功能实现细节

- **节点高亮**：通过创建环形几何体并将其放置在节点位置，使用 MeshBasicMaterial 实现高亮效果。
- **撤销与重置功能**：使用数组保存历史状态，允许用户撤销最近的操作或重置到默认参数。
- **动态参数调整**：通过滑块控制节点的高度差、线长度和垂直间距，实时更新可视化效果。

五、总结

本项目通过 Three.js 和 D3.js 实现了一个生物演化史的三维可视化工具，提供了丰富的交互功能和动态效果。通过对数据结构的合理设计和算法的有效实现，用户能够直观地理解生物演化的过程和各个地质年代的特征。未来可以考虑增加更多的交互功能和数据展示方式，以进一步提升用户体验。

4-screen-biological-evolution-visualization-3D.html.md

实验报告：3D 生物演化可视化系统

引言

本实验旨在开发一个基于 Web 的 3D 生物演化可视化系统，利用 Three.js 和 D3.js 库实现生物演化数据的动态展示。该系统不仅能够展示生物演化的时间线，还能通过交互式界面让用户深入了解每个节点的详细信息。本文将详细描述系统的创新功能、算法实现、代码细节以及所使用的库函数。

创新功能点

1. **动态 3D 可视化**：利用 Three.js 实现生物演化数据的三维展示，用户可以通过鼠标拖拽和滚轮缩放来调整视角。
2. **交互式节点信息展示**：用户点击节点后，系统会显示该节点的详细信息，包括名称、时期和描述。
3. **多种布局选择**：用户可以选择不同的布局方式（如力导向布局、树形布局、球形布局等），以便更好地理解数据结构。
4. **背景和边的自定义**：用户可以自定义背景颜色、透明度以及边的类型（直线、曲线、箭头），增强可视化效果。
5. **全景和立方体天空盒**：支持多种背景天空盒的选择，提升视觉体验。

算法描述

数据处理

系统首先通过 `processData` 函数处理输入的生物演化数据。该函数递归地遍历数据结构，构建节点和连接关系，并为每个节点分配颜色。节点的颜色根据其所属的地质年代进行分类，具体如下：

- 冥古宙：红色
- 太古宙：橙色
- 元古宙：黄色
- 显生宙：绿色
- 中生代：青色
- 新生代：蓝色
- 第四纪：紫色

物理模拟

使用 D3.js 的力导向布局算法，系统通过 `d3.forceSimulation` 创建一个物理模拟环境。该模拟包括以下几种力：

- **链接力**：通过 `d3.forceLink` 实现节点之间的连接。
- **斥力**：通过 `d3.forceManyBody` 实现节点之间的排斥。
- **中心力**：通过自定义的 `createCenter3D` 函数实现节点向中心的吸引力。
- **碰撞力**：通过 `d3.forceCollide` 避免节点重叠。

3D 图形创建

在 Three.js 中，节点通过 `THREE.SphereGeometry` 创建为球体，连接通过 `THREE.Line` 或 `THREE.Curve` 实现。每个节点和连接都被添加到场景中，以便进行渲染。

代码实现

以下是关键代码片段的实现细节：

数据处理

```
function processData(data, parent = null) {
    const node = {
        id: data.name,
        name: data.name,
        time: data.time || "",
        description: data.description || "",
        src: data.src,
        size: 5
    };

    // 继承父节点的颜色
    if (parent && parent.color && parent.color !== '#FFFFFF') {
        node.color = parent.color;
    } else {
        // 根据节点名称确定颜色
        let currentNode = data;
        let color = '#FFFFFF';
        while (currentNode) {
            if (currentNode.name.includes('冥古宙')) {
                color = '#FF0000';  // 红色
                break;
            }
            // 其他颜色判断...
            currentNode = currentNode.parent;
        }
        node.color = color;
    }

    nodes.push(node);
    if (parent) {
        links.push({
```

```
            source: parent.id,
            target: node.id
        });
    }

    // 递归处理子节点
    if (data.children) {
        data.children.forEach(child => {
            child.parent = data;  // 设置父节点引用
            processData(child, node);
        });
    }
}
```

物理模拟

```
simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id).distance(100).stre
ngth(1))
    .force("charge", d3.forceManyBody().strength(-300))
    .force("center", createCenter3D(0, 0, 0).strength(0.1))
    .force("collision", d3.forceCollide().radius(20))
    .on("tick", updatePositions);
```

3D 图形创建

```
function createGraphObjects() {
    nodes.forEach(node => {
        const geometry = new THREE.SphereGeometry(node.size);
        const material = new THREE.MeshBasicMaterial({
            color: new THREE.Color(node.color || 0x00ff00) // 默认颜色
        });
        const sphere = new THREE.Mesh(geometry, material);
        node.object = sphere;
        scene.add(sphere);
    });

    links.forEach(link => {
        link.object = createEdge('line'); // 默认使用直线
        scene.add(link.object);
    });
}
```

交互功能

```
renderer.domElement.addEventListener('click', (event) => {
    event.preventDefault();
    // 计算鼠标坐标
    mouse.x = (event.clientX / (window.innerWidth / 2)) * 2 - 1;
    mouse.y = -(event.clientY / (window.innerHeight / 2)) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
```

```
    const intersects = raycaster.intersectObjects(nodes.map(node => nod
e.object).filter(obj => obj));

    if (intersects.length > 0) {
        const nodeData = nodes.find(n => n.object === intersects[0].obj
ect);
        if (nodeData) {
            selectedNode = nodeData;
            updateNodeInfo(nodeData);
            // 更新选择圈
            // ...
        }
    }
});
```

库函数引用

D3.js

- `d3.forceSimulation`: 创建一个物理模拟环境。
- `d3.forceLink`: 创建节点之间的链接力。
- `d3.forceManyBody`: 创建节点之间的斥力。
- `d3.forceCollide`: 创建节点之间的碰撞力。

Three.js

- `THREE.Scene`: 创建 3D 场景。
- `THREE.PerspectiveCamera`: 创建透视相机。
- `THREE.WebGLRenderer`: 创建 WebGL 渲染器。
- `THREE.SphereGeometry`: 创建球体几何体。
- `THREE.Line`: 创建线段。

结论

本实验成功实现了一个基于 Web 的 3D 生物演化可视化系统，利用 Three.js 和 D3.js 库实现了动态的生物演化数据展示。通过交互式界面，用户可以深入了解生物演化的各个节点及其关系。未来的工作可以集中在优化性能和增加更多的交互功能，以提升用户体验。

4-screen-display.html.md

实验报告：四象限展示网页

一、引言

本实验旨在开发一个四象限展示网页，用户可以通过 URL 参数选择四个不同的卡片内容，并在网页的四个象限中展示这些内容。该网页采用 HTML、CSS 和 JavaScript 技术实现，具有良好的用户体验和交互性。通过使用 iframe 技术，能够动态加载不同的子页面内容，提升了页面的灵活性和可扩展性。

二、创新功能点

1. **动态内容加载**：用户可以通过 URL 参数选择不同的卡片，网页会根据选择动态加载相应的内容，避免了页面的重复加载。
2. **响应式设计**：使用 CSS Flexbox 布局，使得网页在不同屏幕尺寸下都能保持良好的展示效果。
3. **消息传递机制**：通过 postMessage API 实现父页面与子页面之间的通信，能够实时更新象限内容。
4. **用户友好的错误提示**：当用户未选择四个卡片时，网页会显示明确的错误信息，提升用户体验。

三、代码实现

1. HTML 结构

网页的基本结构使用 HTML5 标准，包含了文档类型声明、头部信息和主体内容。以下是主要的 HTML 结构：

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <title>四象限展示</title>
    <style>
        /* CSS 样式 */
    </style>
</head>
<body>
    <div class="quadrant" id="top-left"></div>
    <div class="quadrant" id="top-right"></div>
```

```
    <div class="quadrant" id="bottom-left"></div>
    <div class="quadrant" id="bottom-right"></div>
    <script>
        // JavaScript 逻辑
    </script>
</body>
</html>
```

2. CSS 样式

使用 CSS Flexbox 布局来实现四个象限的排列，确保在不同设备上都能自适应。以下是主要的 CSS 样式：

```
body {
    display: flex;
    flex-wrap: wrap;
    height: 100vh;
    margin: 0;
    box-sizing: border-box; /* 包含边框在内的计算 */
}
.quadrant {
    width: calc(50% - 2px); /* 减去边框宽度 */
    height: calc(50% - 2px); /* 减去边框宽度 */
    position: relative;
    overflow: hidden;
    border: 1px solid #ccc; /* 边框样式 */
    box-sizing: border-box; /* 包含边框在内的计算 */
    margin: 1px; /* 添加间隔 */
}
```

3. JavaScript 逻辑

JavaScript 部分负责获取 URL 参数、加载内容、处理消息传递等功能。以下是主要的 JavaScript 逻辑：

```
// 获取 URL 中的参数
const urlParams = new URLSearchParams(window.location.search);
const cards = urlParams.get('cards').split(',');

// 确保有四个选中的卡片
if (cards.length === 4) {
    loadContent('top-left', cards[0]);
    loadContent('top-right', cards[1]);
    loadContent('bottom-left', cards[2]);
    loadContent('bottom-right', cards[3]);
} else {
    document.body.innerHTML = '<h1>错误: 请选择四个选项卡。</h1>';
}
```

```
// 加载子页面内容的函数
function loadContent(quadrantId, cardName) {
    const iframe = document.createElement('iframe');
    iframe.src = `./4-screen-${cardName}.html`; // 设置 iframe 的源
    document.getElementById(quadrantId).appendChild(iframe); // 将 ifra
me 添加到对应的象限
}

// 添加事件监听器以接收来自 iframe 的消息
window.addEventListener('message', (event) => {
    if (event.origin === window.location.origin) {
        const { quadrantId, data } = event.data;
        updateQuadrant(quadrantId, data);
    }
});

// 更新象限内容的函数
function updateQuadrant(quadrantId, data) {
    const quadrant = document.getElementById(quadrantId);
    quadrant.innerHTML = `<h2>选中数据: ${data}</h2>`; // 更新显示内容
}
```

4. 代码细节分析

- **URL 参数解析**：使用 `URLSearchParams` 对象解析 URL 中的参数，确保用户选择的卡片数量为四个。
- **动态加载 iframe**：通过创建 `iframe` 元素并设置其 `src` 属性，动态加载不同的子页面内容。
- **消息传递**：使用 `window.addEventListener` 监听来自子页面的消息，确保消息来源的安全性，并根据接收到的数据更新对应的象限内容。

四、库函数引用

在本实验中，虽然没有直接使用 D3.js 和 Three.js，但可以考虑在未来的扩展中引入这些库来增强数据可视化和三维效果。

- **D3.js**：可以用于创建动态数据可视化图表，增强用户交互体验。
- **Three.js**：可以用于在网页中创建三维场景，提供更丰富的视觉效果。

五、结论

本实验成功实现了一个四象限展示网页，具备动态内容加载、响应式设计和用户友好的错误提示等功能。通过使用现代前端技术，提升了用户体验和页面的灵活性。未来可以考虑引入更多的库和功能，以进一步增强网页的交互性和可视化效果。

4-screen-home-page.html.md

实验报告：旋转卡片展示页面

一、引言

本实验旨在创建一个动态的旋转卡片展示页面，用户可以通过点击卡片选择最多四张卡片，并在选择完成后跳转到新的页面以查看详细信息。该页面采用现代网页设计技术，结合了 CSS 动画、JavaScript 交互和响应式布局，提供了良好的用户体验。

二、创新功能点

1. **动态卡片选择**：用户可以通过点击卡片进行选择，最多选择四张卡片。选中的卡片会有明显的视觉反馈，提升用户的交互体验。
2. **3D 效果**：通过鼠标移动，卡片会呈现出 3D 旋转效果，增加了页面的趣味性和互动性。
3. **流动边框效果**：选中卡片时，卡片周围会出现动态流动的边框，增强了视觉吸引力。
4. **响应式设计**：页面布局采用 CSS Grid，使其在不同屏幕尺寸下都能良好展示。

三、代码实现细节

1. HTML 结构

页面的基本结构由 HTML 构成，包含了头部信息、样式定义和主要内容区域。以下是关键部分的代码：

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>旋转卡片展示</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <style>
        /* CSS 样式定义 */
    </style>
</head>
<body>
    <div class="container">
        <!-- 卡片内容将通过 JavaScript 动态生成 -->
```

```
    </div>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

2. CSS 样式

CSS 部分定义了页面的整体布局和卡片的样式。使用了 CSS Grid 布局来实现响应式设计，并通过 CSS 动画实现卡片的动态效果。

```
body {
    margin: 0;
    min-height: 100vh;
    display: flex;
    justify-content: center;
    align-items: center;
    background: linear-gradient(135deg, #1a1a1a, #2a2a2a);
    perspective: 1000px;
}

.container {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(400px, 1fr));
    gap: 40px;
    padding: 40px;
    max-width: 1800px;
    margin: 0 auto;
}

.card {
    transition: transform 0.3s ease;
    cursor: pointer;
    box-shadow: 0 10px 30px rgba(0,0,0,0.4);
}

.card.selected {
    border: 4px solid #4CAF50;
    background-color: rgba(76, 175, 80, 0.1);
}
```

3. JavaScript 交互

JavaScript 部分负责动态生成卡片、处理用户的点击事件以及实现 3D 效果。以下是关键代码片段：

```
const images = [
    'biological-classification-sunburst-2D',
    'biological-evolution-force-2D',
```

```
    // 其他图片
];

let selectedCards = [];

images.forEach(img => {
    document.write(`
        <div class="card-wrapper" data-img="${img}">
            <a href="javascript:void(0);" class="select-card">
                <div class="card">
                    <div class="card-face card-front">
                        <img src="./select-cards/${img}.png" alt="${img}
">
                    </div>
                    <div class="card-face card-back">
                        <span>点击查看详情</span>
                    </div>
                </div>
            </a>
        </div>
    `);
});

document.querySelectorAll('.select-card').forEach(card => {
    card.addEventListener('click', () => {
        // 处理卡片选择逻辑
    });
});
```

4. 3D 效果实现

通过监听鼠标移动事件，计算鼠标相对于卡片的位置，从而实现 3D 旋转效果。以下是实现代码：

```
document.querySelectorAll('.card-wrapper').forEach(card => {
    card.addEventListener('mousemove', (e) => {
        const rect = card.getBoundingClientRect();
        const x = e.clientX - rect.left;
        const y = e.clientY - rect.top;

        const centerX = rect.width / 2;
        const centerY = rect.height / 2;

        const rotateX = (y - centerY) / 10;
        const rotateY = -(x - centerX) / 10;

        card.style.transform = `perspective(1000px) rotateX(${rotateX}d
eg) rotateY(${rotateY}deg)`;
    });
```

```
    card.addEventListener('mouseleave', () => {
        card.style.transform = 'perspective(1000px) rotateX(0) rotateY
(0)';
    });
});
```

四、算法描述

在本实验中，主要的算法逻辑包括：

1. **卡片选择算法**：通过维护一个数组 `selectedCards` 来存储用户选择的卡片。当用户点击卡片时，检查该卡片是否已被选择。如果已选择，则从数组中移除；如果未选择且当前选择数量少于 4，则将其添加到数组中。
2. **页面跳转算法**：当用户选择的卡片数量达到 4 时，构建新的页面 URL 并进行跳转。

五、库函数引用

本实验未使用外部库如 D3.js 或 Three.js，但可以考虑在未来的扩展中引入这些库以实现更复杂的可视化效果。例如，D3.js 可以用于数据可视化，而 Three.js 可以用于更高级的 3D 效果。

六、结论

本实验成功实现了一个动态的旋转卡片展示页面，用户可以通过简单的交互选择卡片并查看详细信息。通过使用现代的网页技术，提升了用户体验和页面的视觉效果。未来可以考虑引入更多的功能和效果，以进一步增强页面的互动性和美观性。

4-screen-paleo-geography-3D.html.md

实验报告：地球古地理三维可视化

一、实验背景

随着科学技术的不断发展，地球科学的研究逐渐向可视化方向发展。通过三维可视化技术，用户可以更直观地理解地球的演变过程，尤其是古地理的变化。本实验旨在利用 Three.js 库实现一个地球古地理的三维可视化模型，展示不同地质时期的地球形态及其演变。

二、创新功能点

1. **动态时间选择**：用户可以通过下拉菜单选择不同的地质时期，模型会自动更新显示相应时期的地球纹理。

2. **自动旋转与手动控制**：用户可以选择自动旋转地球模型，或通过鼠标拖拽手动控制视角。
3. **云层显示**：用户可以选择显示或隐藏地球表面的云层，增强视觉效果。
4. **信息提示**：在不同时间点，用户可以查看该时期的地质信息，帮助理解地球的演变。

三、代码实现细节

1. 主要库函数引用

本实验主要使用了以下库：

- **Three.js**：用于创建和渲染 3D 图形。
- **OrbitControls.js**：用于实现相机的控制，使用户能够通过鼠标拖拽来旋转、缩放和移动视角。

2. 代码实现

以下是代码的主要实现部分，包含了初始化、纹理加载、时间切换等功能。

```
let scene, camera, renderer, earth, controls, clouds;

// 初始化场景
function init() {
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.
innerHeight, 0.1, 1000);
    renderer = new THREE.WebGLRenderer();
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    // 创建地球
    const geometry = new THREE.SphereGeometry(5, 32, 32);
    const textureLoader = new THREE.TextureLoader();
    const textures = {};
    const timePoints = [0, 20, 35, 50, 66, 90, 105, 120, 170, 200, 220,
 240, 260, 280, 300, 340, 370, 400, 430, 450, 470, 540, 600, 750];
    let currentTimeIndex = 0;

    // 加载纹理
    timePoints.forEach(time => {
        const fileName = time === 0 ? '0.jpg' : `${time}.jpg`;
        textureLoader.load(`./static-other/World_Texture/${fileName}`,
function (texture) {
            textures[time] = texture;
            if (time === 0) {
                const material = new THREE.MeshPhongMaterial({ map: tex
ture });
```

```
                earth = new THREE.Mesh(geometry, material);
                scene.add(earth);
            }
        });
    });

    // 添加控制器
    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.autoRotate = true;

    camera.position.z = 15;
    animate();
}

// 切换时间索引
function switchToTimeIndex(index) {
    if (index >= 0 && index < timePoints.length) {
        currentTimeIndex = index;
        const time = timePoints[index];
        if (earth && textures[time]) {
            earth.material.map = textures[time];
            earth.material.needsUpdate = true;
        }
    }
}

// 动画循环
function animate() {
    requestAnimationFrame(animate);
    controls.update();
    renderer.render(scene, camera);
}

init();
```

3. 关键算法描述

- **纹理加载**：使用 `THREE.TextureLoader` 异步加载不同时间点的地球纹理。通过 `forEach` 遍历时间点数组，加载对应的纹理文件，并在加载完成后更新地球模型的材质。
- **时间切换功能**：通过 `switchToTimeIndex` 函数实现时间的切换，更新地球的纹理和相关信息。
- **相机控制**：使用 `OrbitControls` 实现相机的平滑控制，允许用户通过鼠标操作来旋转和缩放视角。

4. 代码实现细节

- **地球模型创建**：使用 `THREE.SphereGeometry` 创建一个球体作为地球模型，并使用 `THREE.MeshPhongMaterial` 为其添加纹理。
- **光源设置**：添加环境光和方向光，以增强模型的立体感和真实感。
- **事件监听**：通过 `addEventListener` 监听用户的操作，如选择时间、鼠标点击等，动态更新模型状态。

四、总结

本实验通过 Three.js 实现了一个动态的地球古地理三维可视化模型，用户可以通过简单的交互操作，直观地了解地球的演变过程。该项目不仅展示了现代 Web 技术在科学教育中的应用潜力，也为未来的地理科学研究提供了新的思路和工具。通过不断优化和扩展功能，可以进一步提升用户体验和教育效果。

4-screen-radial-tidy-tree.html.md

实验报告：生物演化史放射状树形图

一、引言

本实验旨在通过使用 D3.js 库构建一个生物演化史的放射状树形图，以可视化生物的演化过程。该图形不仅展示了生物的演化关系，还提供了交互功能，如搜索、缩放和详细信息展示，增强了用户体验。

二、创新功能点

1. **动态搜索功能**：用户可以通过输入关键字快速搜索节点，系统会实时更新搜索结果，提升了信息获取的效率。
2. **时间过滤器**：用户可以通过滑块调整时间范围，动态显示不同时间段的生物演化节点，帮助用户更好地理解生物演化的时间线。
3. **详细信息面板**：点击节点后，系统会展示该节点的详细信息，包括名称、时间和描述，增强了信息的可读性。
4. **高亮路径功能**：用户可以高亮显示从根节点到目标节点的路径，帮助用户更直观地理解生物之间的演化关系。
5. **模态框展示**：点击节点或工具提示中的图片时，能够放大查看，提升了用户的交互体验。

三、算法描述

3.1 数据结构

使用 D3.js 的 `hierarchy` 方法构建树形结构，节点包含以下属性： - `id`: 节点唯一标识 - `data`: 包含生物的名称、时间、描述和图片链接等信息

3.2 布局算法

使用 D3.js 的 `tree` 布局算法，设置节点的角度和半径，以实现放射状的树形图。通过 `separation` 方法定义节点之间的间距。

3.3 事件处理

- **鼠标事件**：通过 `mouseover` 和 `mouseout` 事件展示和隐藏工具提示。
- **键盘事件**：监听 `Q` 和 `E` 键实现节点的旋转。
- **滑块事件**：通过滑块调整时间范围，动态更新可见节点。

四、代码实现

4.1 HTML 结构

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>生物演化史放射状树形图</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <div class="header">
        <h1>生物演化史-Tree-2D</h1>
        <a href="./4-screen-biological-evolution-tree-2D.html" class="n
av-link">Tidy Tree</a>
    </div>
    <div class="control-buttons">
        <!-- 控制按钮 -->
    </div>
    <div class="bottom-time-filter">
        <input type="range" id="timeSlider" min="0" max="500000" value=
"500000">
        <span id="timeDisplay">显示所有时期</span>
    </div>
    <div class="keyboard-hint">
```

```
        使用 'Q' 键逆时针旋转，'E' 键顺时针旋转
    </div>
    <div class="search-container">
        <input type="text" class="search-input" placeholder="搜索节点...
">
        <div class="search-result"></div>
    </div>
    <div class="detail-panel">
        <div class="content">
            <img src="" alt="">
            <h3></h3>
            <p class="time"></p>
            <p class="description"></p>
        </div>
    </div>
    <div id="imageModal" class="modal">
        <img class="modal-content" id="modalImage">
    </div>
    <script src="./biological-evolution-data.js"></script>
    <script>
        // JavaScript 代码实现
    </script>
</body>
</html>
```

4.2 JavaScript 代码实现

4.2.1 SVG 创建与布局

```
// 获取窗口尺寸
const width = window.innerWidth;
const height = window.innerHeight;

// 创建 SVG 容器
const svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height)
    .attr("viewBox", [-width / 2, -height / 2, width, height]);

// 添加一个专门用于缩放的组
const g = svg.append("g");

// 创建放射状树形图布局
const radius = Math.min(width, height) / 1;
const tree = d3.tree()
    .size([2 * Math.PI, radius])
    .separation((a, b) => {
        return (a.parent == b.parent ? 2 : 3) / a.depth;
    });
```

4.2.2 事件处理

```
// 添加键盘事件监听
document.addEventListener('keydown', (event) => {
    if (event.key === 'q') {
        // 逆时针旋转
        currentRotation -= rotationStep;
        g.attr("transform", `${d3.zoomTransform(svg.node())} rotate(${c
urrentRotation})`);
    } else if (event.key === 'e') {
        // 顺时针旋转
        currentRotation += rotationStep;
        g.attr("transform", `${d3.zoomTransform(svg.node())} rotate(${c
urrentRotation})`);
    }
});

// 修改滑块事件监听器
timeSlider.addEventListener('input', (event) => {
    const value = parseInt(event.target.value);
    updateView(value);
});
```

4.2.3 搜索功能实现

```
// 搜索函数
function searchNodes(query) {
    const results = [];
    root.descendants().forEach(node => {
        const name = node.data.name || '';
        const description = node.data.description || '';
        const time = node.data.time || '';

        if (!query || name.toLowerCase().includes(query.toLowerCase())
||
            description.toLowerCase().includes(query.toLowerCase()) ||
            time.toLowerCase().includes(query.toLowerCase())) {
            results.push(node);
        }
    });

    displaySearchResults(results);
}

// 显示搜索结果
function displaySearchResults(results) {
    searchResult.innerHTML = '';

    if (results.length === 0) {
        searchResult.innerHTML = '<div class="search-result-item">未找
```

```
到相关结果</div>';
        return;
    }

    results.forEach(node => {
        const div = document.createElement('div');
        div.className = 'search-result-item';
        div.textContent = `${node.data.name} ${node.data.time || ''}`;
        div.addEventListener('click', (event) => {
            event.stopPropagation();
            highlightPath(node);
        });
        searchResult.appendChild(div);
    });
}
```

4.4 D3.js 库函数引用

- **d3.select**: 用于选择 DOM 元素。
- **d3.hierarchy**: 用于构建层级数据结构。
- **d3.tree**: 用于创建树形布局。
- **d3.zoom**: 用于实现缩放功能。
- **d3.linkRadial**: 用于绘制放射状连接线。

五、结论

本实验通过 D3.js 库成功实现了生物演化史的放射状树形图，提供了丰富的交互功能，提升了用户体验。通过动态搜索、时间过滤和详细信息展示等功能，用户能够更直观地理解生物的演化过程。未来可以考虑引入 Three.js 库，进一步增强可视化效果，实现三维展示。

biological-classification-sunburst-2D.html.md

实验报告：生物分类可视化系统

一、引言

随着生物科学的不断发展，生物分类学作为一门重要的学科，越来越受到重视。为了帮助用户更好地理解生物分类的复杂性，我们开发了一个基于 Web 的生物分类可视化系统。该系统利用 D3.js 库实现了生物分类的交互式可视化，用户可以通过旋转、缩放等操作深入探索生物分类的层次结构。

二、创新功能点

1. **交互式可视化**：用户可以通过鼠标拖拽、键盘操作等方式与可视化图形进行交互，增强了用户体验。
2. **放大镜功能**：用户可以在图形上方使用放大镜查看细节，提升了信息的可读性。
3. **搜索功能**：用户可以通过输入生物名称快速定位到相应的分类节点，方便快捷。
4. **信息面板**：当用户悬停在某个节点上时，右侧信息面板会显示该节点的详细信息，包括图片、描述等。

三、算法描述

3.1 数据结构

系统使用树形结构来表示生物分类数据。每个节点包含以下属性： - `name`：生物名称 - `src`：生物图片链接 - `description`：生物描述 - `children`：子节点数组

3.2 可视化算法

使用 D3.js 库的分区布局（partition layout）算法来生成生物分类的可视化图形。该算法将数据转换为层次结构，并根据节点的深度和权重计算每个节点的角度和半径。

3.3 交互功能

- **拖拽旋转**：通过监听鼠标事件，计算当前鼠标位置与中心点的角度差，更新当前旋转角度。
- **键盘控制**：使用 WASD 和方向键控制同层节点的移动，增强了用户的操作灵活性。
- **搜索功能**：通过遍历树形结构，查找匹配的节点并高亮显示。

四、代码实现

4.1 HTML 结构

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>生物分类可视化</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <script src="./biological-classification-data.js"></script>
    <script src="https://html2canvas.hertzen.com/dist/html2canvas.min.j
```

```
s"></script>
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <div class="control-tips">...</div>
    <div class="content-container">
        <svg width="1000" height="1000"></svg>
        <div class="info-panel">...</div>
    </div>
    <div class="control-buttons">...</div>
    <div class="search-container">...</div>
    <div id="magnifier">...</div>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

4.2 D3.js 实现细节

4.2.1 数据处理

使用 D3.js 的 `hierarchy` 函数将数据转换为层次结构，并使用 `partition` 函数生成分区布局。

```
const partition = data => {
    const root = d3.hierarchy(data)
        .sum(d => 1)
        .sort((a, b) => b.value - a.value);
    return d3.partition()
        .size([2 * Math.PI, radius])
        .padding(0.005)(root);
};
```

4.2.2 弧生成器

使用 D3.js 的 `arc` 函数生成每个节点的弧形路径。

```
const arc = d3.arc()
    .startAngle(d => d.x0)
    .endAngle(d => d.x1)
    .padAngle(0.005)
    .padRadius(radius / 2)
    .innerRadius(d => d.y0)
    .outerRadius(d => d.y1 - 1;
```

4.2.3 交互功能

通过监听鼠标和键盘事件，实现节点的拖拽和旋转。

```
d3.select("body")
    .style("cursor", "move")
    .on("mousedown", dragstarted)
    .on("mousemove", dragged)
    .on("mouseup", dragended);

function dragstarted(event) {
    isDragging = true;
    dragStartAngle = Math.atan2(event.pageY - height / 2, event.pageX -
 width / 2) * 180 / Math.PI;
    dragStartRotation = currentRotation;
}
```

4.3 放大镜功能实现

使用 Canvas 元素实现放大镜效果，通过鼠标移动事件更新放大镜内容。

```
function updateMagnifier(e) {
    const svgElement = document.querySelector('svg');
    if (!svgElement) return;

    // 获取 SVG 的位置和尺寸
    const svgRect = svgElement.getBoundingClientRect();
    const mouseX = e.clientX - svgRect.left;
    const mouseY = e.clientY - svgRect.top;

    // 清除离屏 canvas
    offscreenCtx.clearRect(0, 0, offscreenCanvas.width, offscreenCanvas.
height);

    // 创建圆形裁剪区域
    offscreenCtx.save();
    offscreenCtx.beginPath();
    offscreenCtx.arc(offscreenCanvas.width / 2, offscreenCanvas.height
/ 2, offscreenCanvas.width / 2, 0, Math.PI * 2);
    offscreenCtx.clip();

    // 将 SVG 转换为图片
    const svgString = new XMLSerializer().serializeToString(svgElement);
    const img = new Image();
    const blob = new Blob([svgString], {type: 'image/svg+xml'});
    const url = URL.createObjectURL(blob);

    img.onload = () => {
        // 计算缩放和位置
```

```
            const scale = ZOOM;
            const dx = offscreenCanvas.width / 2 - mouseX * scale;
            const dy = offscreenCanvas.height / 2 - mouseY * scale;

            // 在离屏 canvas 上绘制
            offscreenCtx.drawImage(img, dx, dy, svgRect.width * scale, svgR
ect.height * scale);
            offscreenCtx.restore();

            // 将离屏 canvas 的内容复制到显示 canvas
            ctx.clearRect(0, 0, canvas.width, canvas.height);
            ctx.drawImage(offscreenCanvas, 0, 0);

            URL.revokeObjectURL(url);
        };

        img.src = url;
}
```

五、总结

本实验通过 D3.js 库实现了一个生物分类可视化系统，提供了丰富的交互功能和用户体验。系统不仅能够有效地展示生物分类的层次结构，还通过放大镜和搜索功能提升了信息的可读性和可访问性。未来，我们计划进一步优化系统性能，并增加更多的生物分类数据，以便用户能够更全面地了解生物分类学的知识。

biological-evolution-dna-3D.html.md

实验报告：生物演化史-DNA 双螺旋-3D 可视化

一、引言

本实验旨在通过 3D 可视化技术展示生物演化史中的 DNA 双螺旋结构。利用 Three.js 库实现 3D 图形渲染，并结合 D3.js 库处理数据，创建一个交互式的时间旅行模式，用户可以通过界面控制播放速度、前进和后退，直观地了解生物演化的过程。

二、创新功能点

1. **时间旅行模式**：用户可以通过按钮控制时间的前进和后退，动态展示生物演化的不同阶段。
2. **交互式信息展示**：当用户将鼠标悬停在时间轴上的事件时，相关信息会以弹出框的形式展示，增强用户体验。
3. **全屏显示功能**：用户可以选择全屏模式，以更好地沉浸在 3D 可视化中。

4. **动态星空背景**：通过动态变化的星空背景，增强视觉效果，使得整个场景更加生动。

三、算法描述

3.1 数据处理

使用 D3.js 库处理生物演化的时间数据。数据结构为树形结构，每个节点包含事件名称、时间、描述和相关图片。通过递归函数将树形数据展平，并根据时间值进行排序。

3.2 3D 场景构建

使用 Three.js 库构建 3D 场景，包括相机、光源、星空背景和 DNA 双螺旋结构。相机使用透视投影，光源包括环境光和点光源，以确保场景的明亮度和层次感。

3.3 动画与交互

通过 TWEEN.js 库实现平滑动画效果，用户可以通过键盘控制相机的旋转，增强交互性。时间轴的更新通过定时器实现，确保在播放状态下，时间轴能够动态更新。

四、代码实现

4.1 HTML 结构

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>生物演化史-DNA 双螺旋-3D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <style>
        /* 样式定义 */
    </style>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/t
hree.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/three@0.128.0/examples/js
/controls/OrbitControls.js"></script>
    <script src="./biological-evolution-data.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/tween.js/18.9.5
/Tween.min.js"></script>
</head>
<body>
    <div id="navigation">
        <!-- 导航按钮 -->
    </div>
    <div id="info">生物演化史-DNA 双螺旋-3D</div>
    <div id="details"></div>
```

```
    <div id="controls-info">按 Q/E 键进行顺/逆时针旋转视角</div>
    <script>
        // JavaScript 代码实现
    </script>
</body>
</html>
```

4.2 JavaScript 实现细节

4.2.1 初始化场景

```
function init() {
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x000000);

    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.i
nnerHeight, 0.1, 10000);
    camera.position.set(100, 0, 100);
    camera.lookAt(0, 0, 0);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.maxDistance = 500;
    controls.minDistance = 50;

    // 添加光源
    const ambientLight = new THREE.AmbientLight(0xffffff, 0.5);
    scene.add(ambientLight);

    const pointLight = new THREE.PointLight(0xffffff, 1);
    pointLight.position.set(100, 100, 100);
    scene.add(pointLight);

    createStarField();
    createDNASpiral();
    createTimelineEvents();

    createTimeControls();

    animate();
}
```

4.2.2 创建 DNA 双螺旋

```
function createDNASpiral() {
    const curve1Points = [];
    const curve2Points = [];
```

```
    const radius = 20;
    const height = 500;
    const turns = 25;

    for(let i = 0; i <= 360 * turns; i++) {
        const angle = (i * Math.PI) / 180;
        const y = (i / (360 * turns)) * height - height/2;

        curve1Points.push(new THREE.Vector3(radius * Math.cos(angle), y,
 radius * Math.sin(angle)));
        curve2Points.push(new THREE.Vector3(radius * Math.cos(angle + M
ath.PI), y, radius * Math.sin(angle + Math.PI)));
    }

    const curve1Geometry = new THREE.BufferGeometry().setFromPoints(cur
ve1Points);
    const curve2Geometry = new THREE.BufferGeometry().setFromPoints(cur
ve2Points);

    const material = new THREE.LineBasicMaterial({ color: 0x00ff88, opa
city: 0.8, transparent: true });

    const dnaStrand1 = new THREE.Line(curve1Geometry, material);
    const dnaStrand2 = new THREE.Line(curve2Geometry, material);

    scene.add(dnaStrand1);
    scene.add(dnaStrand2);
}
```

4.3 交互功能实现

4.3.1 鼠标移动事件

```
window.addEventListener('mousemove', onMouseMove);

function onMouseMove(event) {
    const mouse = new THREE.Vector2();
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(timelineObjects.child
ren);

    if(intersects.length > 0) {
        const event = intersects[0].object.userData;
        showEventDetails(event);
    } else {
        hideEventDetails();
```

```
    }
}
```

五、库函数引用

- **Three.js**：用于 3D 图形的渲染和交互，提供了丰富的几何体、材质和光源选项。
- **D3.js**：用于数据处理和可视化，特别是在处理树形结构数据时，提供了强大的数据绑定和更新功能。
- **TWEEN.js**：用于实现平滑的动画效果，增强用户体验。

六、结论

本实验通过 Three.js 和 D3.js 的结合，实现了一个生动的生物演化史可视化工具。用户可以通过交互式界面探索 DNA 双螺旋的结构及其演化过程，增强了对生物学知识的理解。未来可以进一步扩展功能，例如增加更多生物演化事件的详细信息，或引入更多的交互方式，以提升用户体验。

biological-evolution-force-2D.html.md

实验报告：生物演化史-力导向-2D 可视化

一、引言

本实验旨在通过力导向图的方式展示生物演化史，利用 D3.js 库实现动态交互式可视化。该项目不仅展示了生物演化的时间线，还通过创新的功能点提升了用户体验，使得复杂的生物演化信息更加直观易懂。

二、创新功能点

1. **动态交互**：用户可以通过鼠标拖拽、缩放和旋转来探索生物演化图，增强了交互性。
2. **节点信息展示**：当用户悬停在节点上时，能够显示详细的节点信息，包括名称、时间和描述。
3. **全屏显示**：提供全屏模式，用户可以更好地查看复杂的图形。
4. **背景图像动态切换**：根据用户选择的节点，背景图像会动态更新，提供更丰富的视觉体验。
5. **图例和帮助提示**：在界面中添加图例和操作提示，帮助用户理解不同节点的含义和操作方式。

三、算法描述

3.1 力导向算法

力导向算法是一种基于物理模型的布局算法，主要通过模拟节点之间的力来确定节点的位置。该算法包括以下几个主要力的计算：

- **弹簧力**：通过 `d3.forceLink()`实现，模拟节点之间的连接，保持一定的距离。
- **斥力**：通过 `d3.forceManyBody()`实现，防止节点重叠。
- **碰撞力**：通过 `d3.forceCollide()`实现，确保节点之间有一定的间隔。
- **中心力**：通过 `d3.forceCenter()`实现，将节点吸引到中心位置。
- **径向力**：通过 `d3.forceRadial()`实现，模拟节点的层次结构。

3.2 更新函数

更新函数负责根据当前节点的状态重新计算节点的位置和连接线的路径。该函数的主要步骤包括：

1. 计算树的布局。
2. 更新节点和连接线的数据绑定。
3. 处理节点的进入、更新和退出动画。
4. 更新节点的颜色和文本位置。

四、代码实现

4.1 HTML 结构

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>生物演化史-力导向-2D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <script src="biological-evolution-data.js"></script>
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <div id="background-container" class="background-image-container"><
/div>
    <div class="controls">
```

```
        <button onclick="toggleFullscreen()">全屏显示</button>
        <!-- 其他控制按钮 -->
    </div>
    <div id="container">
        <div id="graph"></div>
        <div id="magnifier"></div>
    </div>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

4.2 D3.js 实现细节

4.2.1 创建 SVG 容器

```
const svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height);
```

4.2.2 力导向模拟

```
const simulation = d3.forceSimulation()
    .force("link", d3.forceLink().id(d => d.id).distance(100))
    .force("charge", d3.forceManyBody().strength(-300))
    .force("collide", d3.forceCollide().radius(50))
    .force("center", d3.forceCenter(0, 0).strength(0.1))
    .force("radial", d3.forceRadial(d => d.depth * 180, 0, 0).strength
(0.8));
```

4.2.3 更新函数

```
function update(source) {
    // 计算新的树布局
    tree(root);
    const nodes = root.descendants();
    const links = root.links();

    // 更新力导向模拟
    simulation.nodes(nodes);
    simulation.force("link").links(links);
    simulation.alpha(1).restart();

    // 更新节点和连接线
    const node = g.selectAll(".node")
        .data(nodes, d => d.id || (d.id = ++i));

    // 处理节点的进入、更新和退出
    const nodeEnter = node.enter()
        .append("g")
```

```
        .attr("class", "node")
        .attr("transform", d => `translate(${source.x0},${source.y0})`)
        .on("click", click)
        .on("mouseover", showNodeInfo)
        .on("mouseout", hideNodeInfo);

    nodeEnter.append("circle")
        .attr("r", 12)
        .style("fill", d => getNodeColor(d));

    nodeEnter.append("text")
        .attr("dy", ".31em")
        .attr("x", d => d._children ? -10 : 10)
        .attr("text-anchor", d => d._children ? "end" : "start")
        .text(d => d.data.name);
}
```

4.3 事件处理

4.3.1 鼠标事件

```
svg.on("mousedown", (event) => {
    if (event.target === svg.node()) {
        isDragging = true;
        startPoint = { x: event.clientX - currentTranslate.x, y: event.
clientY - currentTranslate.y };
    }
});
```

4.3.2 键盘事件

```
document.addEventListener('keydown', (event) => {
    const key = event.key.toLowerCase();
    if (key === 'w') currentTranslate.y += MOVE_SPEED;
    if (key === 's') currentTranslate.y -= MOVE_SPEED;
    if (key === 'a') currentTranslate.x += MOVE_SPEED;
    if (key === 'd') currentTranslate.x -= MOVE_SPEED;
});
```

五、总结

本实验通过 D3.js 实现了生物演化史的力导向可视化，结合动态交互和丰富的用户体验，成功地将复杂的生物演化信息以直观的方式呈现给用户。通过力导向算法的应用，节点之间的关系得以清晰展示，同时创新的功能点提升了用户的参与感和探索欲望。未来可以考虑引入更多的交互方式和数据源，以进一步丰富可视化效果。

biological-evolution-star-3D.html.md

实验报告：生物演化史-星空图-3D

一、引言

本实验旨在通过 3D 可视化技术展示生物演化史，利用 Three.js 库构建一个交互式的星空图，帮助用户更直观地理解生物演化的过程和各个地质年代的关系。该项目结合了数据可视化和交互设计，具有较高的教育意义和科学价值。

二、创新功能点

1. **交互式节点聚焦**：用户可以通过点击节点来聚焦特定的生物演化事件，系统会自动调整视角，提供详细信息。
2. **动态星空背景**：通过 Three.js 实现动态星空效果，增强视觉体验。
3. **参数调节功能**：用户可以实时调整节点大小、线条长度等参数，观察不同设置下的演化图变化。
4. **全屏模式**：支持全屏显示，提升用户体验。
5. **信息展示**：点击节点后，展示相关的时间、描述和图片信息，增强信息的可读性和趣味性。

三、算法描述

3.1 数据结构

使用树形结构存储生物演化数据，每个节点包含以下信息： - `name`: 节点名称（如地质年代） - `time`: 事件时间 - `description`: 事件描述 - `src`: 相关图片链接 - `children`: 子节点数组

3.2 3D 可视化算法

1. **节点创建**：根据数据生成 3D 球体，使用不同颜色表示不同地质年代。
2. **连接线绘制**：使用贝塞尔曲线连接父节点和子节点，增强视觉效果。
3. **动态效果**：实现节点的呼吸效果和星空的动态变化，提升用户体验。

四、代码实现

4.1 主要库引用

本项目主要使用了以下库： - **Three.js**：用于 3D 图形渲染。 - **D3.js**：用于数据处理和可视化（在本项目中未直接使用，但可用于数据的预处理）。

4.2 代码实现细节

以下是关键代码片段的实现细节：

4.2.1 初始化场景

```
function init() {
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x000000);

    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.i
nnerHeight, 0.1, 10000);
    camera.position.set(120, 270, 500);
    camera.lookAt(0, 0, 0);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.maxDistance = 3000;
    controls.minDistance = 50;

    createStarField();
    networkGroup = createNetworkVisualization();

    animate();
}
```

- **场景创建**：初始化 Three.js 场景，设置背景颜色。
- **相机设置**：使用透视相机，设置视角和位置。
- **渲染器**：创建 WebGL 渲染器并添加到 DOM 中。
- **控制器**：使用 OrbitControls 实现相机的平滑控制。

4.2.2 创建星空背景

```
function createStarField() {
    const starsGeometry = new THREE.BufferGeometry();
    const starsMaterial = new THREE.PointsMaterial({
        color: 0xFFFFFF,
        size: 0.3,
        transparent: true,
        opacity: 1.0,
        sizeAttenuation: true
    });

    const starsVertices = [];
    for(let i = 0; i < 15000; i++) {
        const x = (Math.random() - 0.5) * 2000;
        const y = (Math.random() - 0.5) * 2000;
        const z = (Math.random() - 0.5) * 2000;
        starsVertices.push(x, y, z);
    }
```

```
    starsGeometry.setAttribute('position', new THREE.Float32BufferAttri
bute(starsVertices, 3));
    const starField = new THREE.Points(starsGeometry, starsMaterial);
    scene.add(starField);
}
```

- **星空创建**：生成 15000 个随机位置的星星，使用点材质进行渲染。
- **动态效果**：通过调整星星的大小和透明度实现动态效果。

4.2.3 创建网络可视化

```
function createNetworkVisualization() {
    const networkGroup = new THREE.Group();

    function createNodes(data, position = new THREE.Vector3(), level =
0, angle = 0, parentColor) {
        const nodeGeometry = new THREE.SphereGeometry(networkParameters.
nodeSize, 32, 32);
        const color = level === 1 ? categoryColors[data.name] : parentC
olor;

        const nodeMaterial = new THREE.MeshPhongMaterial({
            color: color,
            emissive: color,
            emissiveIntensity: 0.5,
            transparent: true,
            opacity: 0.8
        });

        const node = new THREE.Mesh(nodeGeometry, nodeMaterial);
        node.position.copy(position);
        networkGroup.add(node);

        if (data.children) {
            data.children.forEach((child, index) => {
                const childPosition = new THREE.Vector3(position.x + Ma
th.cos(angle) * radius, position.y + level * networkParameters.levelHei
ght, position.z + Math.sin(angle) * radius);
                createNodes(child, childPosition, level + 1, angle + (M
ath.PI / 6), color);
            });
        }
    }

    createNodes(treeData);
    scene.add(networkGroup);
    return networkGroup;
}
```

- **节点创建**：根据数据生成 3D 球体，使用不同颜色表示不同地质年代。
- **递归创建子节点**：通过递归函数创建树形结构的节点。

4.3 事件处理

```
window.addEventListener('click', onMouseClick);

function onMouseClick(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(networkGroup.children.
filter(child => child instanceof THREE.Mesh));

    if (intersects.length > 0) {
        const selectedObject = intersects[0].object;
        showEventDetails(selectedObject.userData);
    } else {
        hideEventDetails();
    }
}
```

- **鼠标点击事件**：通过射线检测获取用户点击的节点，并显示相关信息。

五、结论

本实验通过 Three.js 实现了一个生物演化史的 3D 可视化项目，具有良好的交互性和可视化效果。通过动态星空背景和节点聚焦功能，用户能够更直观地理解生物演化的过程。未来可以进一步扩展功能，如增加更多的交互方式和数据展示形式，以提升用户体验和教育效果。

biological-evolution-tree-2D.html.md

实验报告：生物演化树可视化

引言

本实验旨在通过使用 D3.js 库构建一个生物演化树的可视化界面。该界面不仅展示了生物的演化关系，还提供了交互功能，如搜索、过滤和全屏显示等。通过这些功能，用户可以更直观地理解生物演化的复杂性。

创新功能点

1. **动态树布局**：根据用户选择的时间段动态调整树的高度和宽度，以便更好地展示不同时间段的生物演化情况。

2. **懒加载图片**：在节点中使用懒加载技术，只有当节点进入视口时才加载相关图片，从而提高页面性能。
3. **交互式工具提示**：当用户悬停在节点上时，显示详细信息，包括生物名称、时间和描述，增强用户体验。
4. **搜索功能**：用户可以通过输入生物名称快速定位到相应节点，并高亮显示路径。
5. **全屏显示**：提供全屏按钮，用户可以在全屏模式下更好地查看生物演化树。

算法描述

1. 数据结构

使用 D3.js 的层次结构（hierarchy）来表示生物演化树的数据。每个节点包含以下属性：`- name`：生物名称 `- src`：生物图片的 URL `- description`：生物的描述 `- time`：生物出现的时间

2. 树布局

使用 D3.js 的 `d3.tree()` 方法创建树布局。树的高度和宽度根据用户选择的时间段动态计算，以确保信息的清晰展示。

3. 懒加载实现

使用 `IntersectionObserver` API 监测节点图片的可见性，只有当图片进入视口时才加载，从而减少初始加载时间。

4. 交互功能

- **鼠标悬停事件**：当用户将鼠标悬停在节点上时，显示工具提示，包含生物的详细信息。
- **搜索功能**：通过监听输入框的变化，实时过滤并显示匹配的节点。

代码实现

以下是关键代码片段的实现细节：

1. 创建 SVG 容器

```
const svg = d3.select("#tree")
    .append("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);
```

2. 动态树布局

```
function getTreeLayout(filter) {
    let dynamicHeight;
    switch(filter) {
        case 'all':
            dynamicHeight = height * 8;
            break;
        case '显生宙':
            dynamicHeight = height * 3;
            break;
        case '新生代':
        case '中生代':
            dynamicHeight = height * 2;
            break;
        default:
            dynamicHeight = height;
    }

    const dynamicWidth = filter === 'all' ?
        width - margin.left - margin.right :
        (width - margin.left - margin.right) * 0.6;

    return d3.tree()
        .size([dynamicHeight - margin.top - margin.bottom, dynamicWidt
h]);
}
```

3. 懒加载图片

```
function lazyLoadImages() {
    const images = document.querySelectorAll('image.node-image');

    const observer = new IntersectionObserver((entries, observer) => {
        entries.forEach(entry => {
            if (entry.isIntersecting) {
                const img = entry.target;
                const src = img.getAttribute('data-src');
                if (src) {
                    img.setAttribute('href', src);
                    img.removeAttribute('data-src');
                }
                observer.unobserve(img);
            }
        });
    });

    images.forEach(image => {
        observer.observe(image);
    });
}
```

4. 交互式工具提示

```
nodes.on("mouseover", function(event, d) {
    tooltip.style("display", "block")
        .html(`
            <strong>${d.data.name}</strong>
            ${d.data.src ? `<img src="${d.data.src}" alt="${d.data.name}
">` : ''}
            <div class="tooltip-text">
                ${d.data.name ? '<br>时间: ' + d.data.name : ''}
                ${d.data.description ? '<br>描述: ' + d.data.descriptio
n : ''}
            </div>
        `)
        .style("left", tooltipX + "px")
        .style("top", tooltipY + "px");
})
.on("mouseout", function() {
    tooltip.style("display", "none");
});
```

5. 搜索功能实现

```
searchInput.addEventListener('input', function(e) {
    const searchTerm = e.target.value.trim().toLowerCase();
    if (!searchTerm) {
        searchResults.innerHTML = '';
        return;
    }

    const matches = allNodes.filter(node =>
        node.data.name && node.data.name.toLowerCase().includes(searchT
erm)
    );

    searchResults.innerHTML = matches
        .map(node => `
            <div class="search-result-item" data-node-id="${node.data.n
ame}">
                ${node.data.name}
            </div>
        `).join('');
});
```

库函数引用

D3.js

- **d3.tree()**：用于创建树布局，支持动态调整节点位置。
- **d3.hierarchy()**：将数据转换为层次结构，便于树形布局的计算。
- **d3.select()**：用于选择 DOM 元素并进行操作。

IntersectionObserver

- **IntersectionObserver**：用于实现懒加载功能，监测元素的可见性。

结论

本实验成功实现了一个生物演化树的可视化界面，结合 D3.js 的强大功能，提供了动态、交互式的用户体验。通过创新的功能点，如懒加载、搜索和全屏显示，用户能够更方便地探索生物演化的复杂关系。未来可以考虑进一步优化性能和增加更多的交互功能，以提升用户体验。

biological-evolution-tree-3D.html.md

实验报告：生物演化史-Tree-3D 可视化项目

一、项目概述

本项目旨在通过三维可视化技术展示生物演化的历史，利用 `Three.js` 库构建一个交互式的三维场景，用户可以通过鼠标和键盘进行操作，探索不同地质年代的生物演化信息。该项目不仅展示了生物演化的时间线，还通过动态效果增强了用户体验。

二、创新功能点

1. **交互式节点聚焦**：用户可以通过鼠标点击节点，聚焦到特定的生物演化事件，并显示详细信息。
2. **动态视角控制**：用户可以使用键盘的 W/S/A/D 键进行上下左右移动，使用鼠标滚轮进行缩放，增强了场景的可操作性。
3. **高亮显示功能**：当用户将鼠标悬停在节点上时，节点会高亮显示，提供更好的视觉反馈。
4. **全屏模式**：用户可以选择全屏模式，提升视觉体验。
5. **撤销与重置功能**：用户可以撤销最近的操作或重置到默认参数，方便用户进行实验和调整。

三、算法描述

3.1 数据结构

项目使用树形结构来表示生物演化的各个节点，每个节点包含以下信息：
- `name`：节点名称（如地质年代）。
- `time`：节点对应的时间信息。
- `description`：节点的详细描述。
- `children`：子节点数组，表示该节点下的演化分支。

3.2 主要算法

1. **节点创建算法**：根据树形数据结构递归创建节点，使用 `THREE.SphereGeometry` 创建节点的三维模型，并根据层级和角度计算节点的位置。
2. **曲线连接算法**：使用贝塞尔曲线连接父节点和子节点，增强视觉效果。通过 `THREE.QuadraticBezierCurve3` 创建曲线，并使用 `THREE.Line` 绘制连接线。
3. **动态效果算法**：实现节点的呼吸效果和星空背景的动态变化，提升场景的生动性。

四、代码实现

4.1 主要库引用

- **Three.js**：用于创建和渲染三维场景。
- **OrbitControls**：用于实现相机的旋转和缩放控制。
- **TWEEN.js**：用于实现平滑的动画效果。

4.2 代码实现细节

以下是项目中关键代码片段的实现细节：

4.2.1 初始化场景

```
function init() {
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x000000);

    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.i
nnerHeight, 0.1, 10000);
    camera.position.set(-2170, 847, 792);
    camera.lookAt(-47, -62, -43);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.maxDistance = 6000;
    controls.minDistance = 10;
    controls.screenSpacePanning = true;
    controls.target = new THREE.Vector3(0, 0, 0);
    controls.zoomToCursor = true;

    // 添加光源
    const ambientLight = new THREE.AmbientLight(0xffffff, 0.5);
    scene.add(ambientLight);
```

```
    const pointLight = new THREE.PointLight(0xffffff, 1);
    pointLight.position.set(100, 100, 100);
    scene.add(pointLight);

    createStarField();
    networkGroup = createNetworkVisualization();

    initializeSliders();
    animate();
}
```

4.2.2 创建节点和连接线

```
function createNodes(data, position = new THREE.Vector3(0, -500, 0), le
vel = 0, angle = 0, parentColor, baseAngle = null) {
    const nodeGeometry = new THREE.SphereGeometry(10, 32, 32);
    let color = categoryColors[data.name] || new THREE.Color(0xffffff);

    const nodeMaterial = new THREE.MeshPhongMaterial({
        color: color,
        emissive: color,
        emissiveIntensity: 0.5,
        transparent: true,
        opacity: 0.8
    });

    const node = new THREE.Mesh(nodeGeometry, nodeMaterial);
    node.position.copy(position);
    node.userData = data;

    networkGroup.add(node);

    if (data.children) {
        const childCount = data.children.length;
        const radius = networkParameters.radius - level * 6;

        let angleStep = (Math.PI * 2) / childCount;
        let startAngle = 0;

        data.children.forEach((child, index) => {
            const childAngle = startAngle + angleStep * index;
            const childPosition = new THREE.Vector3(
                position.x + Math.cos(childAngle) * radius,
                position.y + (level * networkParameters.levelHeight),
                position.z + Math.sin(childAngle) * radius
            );

            const connection = createCurvedLine(position, childPosition,
 color);
```

```
            networkGroup.add(connection);

            createNodes(child, childPosition, level + 1, childAngle, co
lor);
        });
    }
}
```

4.2.3 高亮显示功能

```
function onMouseClick(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(networkGroup.children.
filter(child => child instanceof THREE.Mesh));

    if (intersects.length > 0) {
        const clickedNode = intersects[0].object;
        // 创建高亮环
        const highlightRing = new THREE.Mesh(new THREE.RingGeometry(15,
 20, 32), new THREE.MeshBasicMaterial({ color: 0xffff00, side: THREE.Do
ubleSide, transparent: true, opacity: 0.8 }));
        highlightRing.position.copy(clickedNode.position);
        highlightRing.lookAt(camera.position);
        scene.add(highlightRing);

        // 显示节点信息
        showEventDetails(clickedNode.userData);
    }
}
```

4.3 交互功能实现

用户可以通过键盘和鼠标进行交互，以下是相关代码实现：

```
window.addEventListener('keydown', (event) => {
    switch(event.key.toLowerCase()) {
        case 'w': keyState.w = true; break;
        case 's': keyState.s = true; break;
        case 'a': keyState.a = true; break;
        case 'd': keyState.d = true; break;
    }
});

window.addEventListener('keyup', (event) => {
    switch(event.key.toLowerCase()) {
        case 'w': keyState.w = false; break;
        case 's': keyState.s = false; break;
        case 'a': keyState.a = false; break;
```

```
        case 'd': keyState.d = false; break;
    }
});
```

五、总结

本项目通过 Three.js 和相关库实现了一个生物演化史的三维可视化展示，用户可以通过交互操作深入了解生物演化的历史。项目的创新功能点如动态视角控制、高亮显示和撤销重置功能，极大地提升了用户体验。未来可以考虑进一步优化性能和增加更多的交互功能，以便更好地服务于用户的学习和探索需求。

biological-evolution-visualization-3D.html.md

实验报告：生物演化可视化 3D 系统

一、引言

随着科学技术的不断发展，数据可视化在生物学研究中扮演着越来越重要的角色。本实验旨在开发一个基于 Web 的生物演化可视化 3D 系统，利用 Three.js 和 D3.js 库实现生物演化树的动态展示。该系统不仅能够展示生物的演化关系，还提供了多种交互功能，增强了用户体验。

二、创新功能点

1. **3D 可视化**：通过 Three.js 实现生物演化树的三维展示，用户可以从不同角度观察演化关系。
2. **动态交互**：用户可以通过鼠标拖拽、滚轮缩放等方式与可视化图形进行交互，增强了可视化的灵活性。
3. **节点信息展示**：点击节点后，系统会显示该节点的详细信息，包括名称、时期和描述等。
4. **多种布局选择**：用户可以选择不同的布局方式（如力导向、树形、球形等），以适应不同的可视化需求。
5. **背景和边类型自定义**：用户可以自定义背景颜色、透明度以及边的类型（直线、曲线、箭头），提升了可视化的个性化。

三、算法描述

3.1 数据处理

系统首先通过 processData 函数处理输入数据，构建节点和连接关系。每个节点包含以下属性：- id：节点唯一标识 - name：节点名称 - time：节点对应的时间 - description：节点描述 - src：节点相关图像的 URL - size：节点大小 - color：节点颜色

3.2 力导向布局

使用 D3.js 的力导向算法，节点之间的关系通过力的作用进行动态调整。主要力的类型包括： - **链接力**：控制节点之间的距离。 - **斥力**：防止节点重叠。 - **中心力**：将节点吸引到中心。 - **碰撞力**：防止节点之间的重叠。

3.3 3D 渲染

使用 Three.js 进行 3D 渲染，创建场景、相机和渲染器。节点通过 `THREE.SphereGeometry` 创建，连接通过 `THREE.Line` 或 `THREE.Curve` 实现。

四、代码实现

4.1 主要库引用

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/three@0.128.0/examples/js/controls/OrbitControls.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/7.0.0/d3.min.js"></script>
```

4.2 数据处理函数

```
function processData(data, parent = null) {
    const node = {
        id: data.name,
        name: data.name,
        time: data.time || "",
        description: data.description || "",
        src: data.src,
        size: 5
    };

    // 继承父节点颜色
    if (parent && parent.color && parent.color !== '#FFFFFF') {
        node.color = parent.color;
    } else {
        // 根据节点名称确定颜色
        node.color = determineColor(data.name);
    }

    nodes.push(node);

    if (parent) {
        links.push({
            source: parent.id,
            target: node.id
        });
    }
```

```
    // 递归处理子节点
    if (data.children) {
        data.children.forEach(child => {
            child.parent = data;  // 设置父节点引用
            processData(child, node);
        });
    }
}

function determineColor(name) {
    // 根据节点名称确定颜色
    if (name.includes('冥古宙')) return '#FF0000';
    if (name.includes('太古宙')) return '#FFA500';
    if (name.includes('元古宙')) return '#FFFF00';
    if (name.includes('显生宙')) return '#00FF00';
    if (name.includes('中生代')) return '#00FFFF';
    if (name.includes('新生代')) return '#0000FF';
    if (name.includes('第四纪')) return '#800080';
    return '#FFFFFF'; // 默认颜色
}
```

4.3 力导向布局实现

```
simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id).distance(100).stre
ngth(1))
    .force("charge", d3.forceManyBody().strength(-300))
    .force("center", createCenter3D(0, 0, 0).strength(0.1))
    .force("collision", d3.forceCollide().radius(20))
    .on("tick", updatePositions);
```

4.4 3D 渲染实现

```
function createGraphObjects() {
    nodes.forEach(node => {
        const geometry = new THREE.SphereGeometry(node.size);
        const material = new THREE.MeshBasicMaterial({
            color: new THREE.Color(node.color || 0x00ff00) // 默认颜色
        });
        const sphere = new THREE.Mesh(geometry, material);
        node.object = sphere;
        scene.add(sphere);
    });

    links.forEach(link => {
        link.object = createEdge('line'); // 默认使用直线
        scene.add(link.object);
    });
}
```

4.5 更新节点位置

```
function updatePositions() {
    nodes.forEach(node => {
        if (node.object) {
            node.object.position.x += (node.x - node.object.position.x)
 * 0.1;
            node.object.position.y += (node.y - node.object.position.y)
 * 0.1;
            node.object.position.z += (node.z - node.object.position.z)
 * 0.1;
        }
    });

    links.forEach(link => {
        const source = link.source;
        const target = link.target;
        // 更新连接线位置
        updateLinkPosition(link, source, target);
    });
}
```

五、总结

本实验成功实现了一个基于 Web 的生物演化可视化 3D 系统，利用 Three.js 和 D3.js 库实现了生物演化树的动态展示。系统的创新功能点和交互设计提升了用户体验，为生物学研究提供了有效的可视化工具。未来可以进一步扩展功能，如增加更多的交互方式和数据分析功能，以满足更广泛的应用需求。

device-testing.html.md

实验报告：生物演化史导航页

一、引言

本实验报告旨在详细介绍生物演化史导航页的设计与实现。该页面旨在为用户提供一个友好的界面，以便于在移动端和 PC 端之间进行切换。通过自动检测设备类型，用户可以在 10 秒内自动跳转到适合其设备的页面。此外，页面还提供了手动选择按钮，以便用户在需要时进行手动切换。

二、创新功能点

1. **设备自动检测**：通过分析用户的 User-Agent 信息，自动识别用户的设备类型（移动端或 PC 端），并根据设备类型调整页面内容和样式。

2. **倒计时跳转**：在页面加载后，用户会看到一个倒计时提示，告知他们将在 10 秒后自动跳转到相应的页面。这种设计增强了用户体验，使用户在等待时不会感到无聊。
3. **手动选择功能**：提供了手动选择按钮，允许用户在自动跳转前进行选择，增加了灵活性。

三、代码实现细节

1. HTML 结构

页面的基本结构使用 HTML5 标准，包含头部和主体部分。头部定义了页面的元数据、样式和脚本，主体部分则包含了标题、提示信息和手动选择按钮。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>生物演化史导航页</title>
    ...
</head>
<body>
    <h1 class="title">生物演化史 - EvoViz</h1>
    ...
</body>
</html>
```

2. JavaScript 实现

2.1 设备检测与跳转逻辑

在页面加载时，JavaScript 会执行以下操作：

- 检测设备类型
- 显示手动选择按钮
- 启动倒计时并在倒计时结束后自动跳转

```
window.onload = function() {
    // 显示手动选择按钮
    document.getElementById('manual-selection').style.display = 'block';

    // 检测设备类型
    let target = '';
    if (/Mobi|Android/i.test(navigator.userAgent)) {
        target = '<span class="highlight">移动端</span>';
        document.querySelector('.title').style.fontSize = '30px';
    } else {
```

```
        target = '<span class="highlight">PC 端</span>';
    }

    // 倒计时
    let countdown = 10;
    const countdownElement = document.getElementById('countdown');
    countdownElement.innerHTML = `根据自动检测，将在 ${countdown} 秒后自动跳转到${target}页面...`;
    const interval = setInterval(() => {
        countdown--;
        countdownElement.innerHTML = `根据自动检测，将在 ${countdown} 秒后自动跳转到${target}页面...`;
        if (countdown === 0) {
            clearInterval(interval);
            // 自动跳转
            if (target.includes('移动端')) {
                window.location.href = './mobile-index.html';
            } else {
                window.location.href = './pc-index.html';
            }
        }
    }, 1000);
}
```

3. CSS 样式

页面的样式使用 CSS 进行定义，确保页面在不同设备上都能良好显示。主要样式包括标题、按钮和倒计时文本的样式设置。

```
.title {
    color: #FF0000;
    font-size: 48px;
    font-family: "Microsoft YaHei";
    text-align: center;
    margin-top: 20px;
    padding: 20px;
}
#manual-selection button {
    width: 300px;
    height: 80px;
    margin: 10px auto;
    padding: 0;
    font-size: 24px;
    display: block;
}
```

四、算法描述

本项目的核心算法是设备类型检测和倒计时逻辑。设备类型检测使用正则表达式匹配 User-Agent 字符串，以判断用户的设备类型。倒计时逻辑则使用 `setInterval` 函数，每秒更新一次倒计时，并在倒计时结束时执行页面跳转。

五、库函数引用

在本项目中，虽然没有直接使用 D3.js 和 Three.js 库，但可以考虑在未来的扩展中引入这些库来增强页面的交互性和可视化效果。例如，D3.js 可以用于数据可视化，而 Three.js 可以用于 3D 图形的展示。

5.1 D3.js 示例

```
// 使用D3.js 创建简单的柱状图
d3.select("body").append("svg")
    .attr("width", 500)
    .attr("height", 300)
    .selectAll("rect")
    .data([4, 8, 15, 16, 23, 42])
    .enter().append("rect")
    .attr("width", (d) => d * 10)
    .attr("height", 20)
    .attr("y", (d, i) => i * 25);
```

5.2 Three.js 示例

```
// 使用Three.js 创建简单的3D 立方体
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth / wind
ow.innerHeight, 0.1, 1000);
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

const geometry = new THREE.BoxGeometry();
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);

camera.position.z = 5;

function animate() {
    requestAnimationFrame(animate);
    cube.rotation.x += 0.01;
    cube.rotation.y += 0.01;
    renderer.render(scene, camera);
}
animate();
```

六、结论

本实验成功实现了生物演化史导航页的设计与开发，具备设备自动检测、倒计时跳转和手动选择功能。通过合理的代码结构和清晰的逻辑，确保了页面的用户体验。未来可以考虑引入更多的交互性和可视化效果，以进一步提升用户体验。

home-page.html.md

实验报告：旋转卡片展示网页

一、实验目的

本实验旨在创建一个动态的旋转卡片展示网页，用户可以通过点击卡片进入不同的页面。该网页结合了现代网页设计的美学和交互性，旨在提升用户体验。

二、创新功能点

1. **动态卡片效果**：通过鼠标移动实现 3D 旋转效果，增强了用户的交互体验。
2. **响应式设计**：使用 CSS Grid 布局，使得卡片在不同屏幕尺寸下自适应排列。
3. **渐变背景**：使用 CSS 渐变背景，提升视觉吸引力。
4. **卡片翻转效果**：卡片的正面和背面展示不同内容，增加了信息的展示方式。

三、代码实现细节

1. HTML 结构

HTML 部分定义了网页的基本结构，包括头部信息、样式链接和主体内容。主要的内容是一个包含多个卡片的容器。

```
<div class="container">
    <!-- 卡片内容通过 JavaScript 动态生成 -->
</div>
```

2. CSS 样式

CSS 部分负责网页的视觉效果，包括布局、颜色、阴影等。以下是关键样式的实现：

```
body {
    background: linear-gradient(135deg, #1a1a1a, #2a2a2a);
    display: flex;
    justify-content: center;
    align-items: center;
```

```
    min-height: 100vh;
}

.container {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(400px, 1fr));
    gap: 40px;
    padding: 40px;
    max-width: 1800px;
    margin: 0 auto;
}

.card-wrapper {
    position: relative;
    width: 400px;
    height: 400px;
    transition: transform 0.3s ease;
}

.card {
    box-shadow: 0 10px 30px rgba(0,0,0,0.4);
    cursor: pointer;
    transition: transform 0.3s ease;
}
```

3. JavaScript 动态生成卡片

JavaScript 部分负责动态生成卡片内容。通过 `document.write` 方法，将每个卡片的 HTML 结构插入到页面中。

```
const images = [
    'biological-classification-sunburst-2D',
    'biological-evolution-force-2D',
    // 其他图片...
];

images.forEach(img => {
    document.write(`
        <div class="card-wrapper">
            <a href="./${img}.html">
                <div class="card">
                    <div class="card-face card-front">
                        <img src="./select-cards/${img}.png" alt="${img}
">
                    </div>
                    <div class="card-face card-back">
                        <span>点击查看详情</span>
                    </div>
                </div>
```

```
            </a>
        </div>
    `);
});
```

4. 鼠标移动 3D 效果

通过监听鼠标移动事件，实现卡片的 3D 旋转效果。根据鼠标相对于卡片的位置计算旋转角度。

```
document.querySelectorAll('.card-wrapper').forEach(card => {
    card.addEventListener('mousemove', (e) => {
        const rect = card.getBoundingClientRect();
        const x = e.clientX - rect.left;
        const y = e.clientY - rect.top;

        const centerX = rect.width / 2;
        const centerY = rect.height / 2;

        const rotateX = (y - centerY) / 10;
        const rotateY = -(x - centerX) / 10;

        card.style.transform = `perspective(1000px) rotateX(${rotateX}d
eg) rotateY(${rotateY}deg)`;
    });

    card.addEventListener('mouseleave', () => {
        card.style.transform = 'perspective(1000px) rotateX(0) rotateY
(0)';
    });
});
```

四、算法描述

在实现 3D 效果时，使用了简单的几何计算。通过获取鼠标在卡片上的位置，计算出相对于卡片中心的偏移量，从而得出旋转角度。该算法的复杂度为 O(n)，其中 n 为卡片的数量。

五、库函数引用

在本实验中，虽然没有直接使用 D3.js 和 Three.js 库，但可以考虑在未来的扩展中引入这些库以增强功能：

- **D3.js**：用于数据可视化，可以将卡片内容与数据绑定，动态更新卡片信息。
- **Three.js**：用于更复杂的 3D 效果和动画，可以实现更高级的 3D 场景和交互。

六、总结

本实验成功实现了一个动态的旋转卡片展示网页，结合了现代网页设计的多种技术。通过 CSS 和 JavaScript 的结合，创造了良好的用户体验。未来可以考虑引入更多的库和功能，以进一步提升网页的交互性和视觉效果。

index-earth.html.md

实验报告：地球古地理可视化项目

一、项目概述

本项目旨在通过 Web 技术实现一个交互式的地球古地理可视化工具，用户可以通过该工具观察地球在不同历史时期的状态。项目使用了 Three.js 库进行 3D 图形渲染，结合 HTML 和 JavaScript 实现用户交互功能。用户可以通过按钮和下拉框选择不同的历史时期，查看地球的变化。

二、创新功能点

1. **动态纹理加载**：根据用户选择的历史时期动态加载对应的地球纹理，提升了用户体验。
2. **交互式控制**：用户可以通过鼠标和键盘控制地球的旋转和缩放，增强了可操作性。
3. **云层显示/隐藏功能**：用户可以选择是否显示地球上的云层，增加了视觉效果的多样性。
4. **全屏显示功能**：支持全屏模式，提供更沉浸的观看体验。
5. **触摸事件支持**：为移动设备用户提供良好的触摸交互体验。

三、代码实现细节

1. 主要库函数引用

本项目主要使用了以下库：

- **Three.js**：用于 3D 图形渲染。
- **D3.js**：虽然本项目未直接使用 D3.js，但可以考虑在未来版本中用于数据可视化。

2. 代码实现

以下是项目的主要代码实现细节：

2.1 HTML 结构

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0, maximum-scale=1.0, user-scalable=no">
    <title>地球古地理</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <div id="loadingText">地球模型加载中...</div>
    <div id="info">地球古地理-大陆漂移</div>
    <button id="rotateButton">停止自动旋转</button>
    <button id="cloudsButton">显示云层</button>
    <select id="timeSelect" class="timeSelect">
        <!-- 时间选项 -->
    </select>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/t
hree.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/three@0.128.0/examples/js
/controls/OrbitControls.js"></script>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

2.2 JavaScript 实现

```
let scene, camera, renderer, earth, controls, clouds;

function init() {
    // 初始化场景、相机和渲染器
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.
innerHeight, 0.1, 1000);
    renderer = new THREE.WebGLRenderer({ antialias: false, powerPrefere
nce: "high-performance" });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    // 创建地球
    const geometry = new THREE.SphereGeometry(5, 32, 32);
    const textureLoader = new THREE.TextureLoader();
```

```
    const textures = {};
    const timePoints = [0, 20, 35, 50, 66, 90, 105, 120, 170, 200, 220,
 240, 260, 280, 300, 340, 370, 400, 430, 450, 470, 540, 600, 750];
    let currentTimeIndex = 0;

    // 动态加载纹理
    timePoints.forEach(time => {
        const fileName = time === 0 ? '0.jpg' : `${time}.jpg`;
        textureLoader.load(
            `./static-other/World_Texture/${fileName}`,
            function (texture) {
                textures[time] = texture;
                if (time === 0) {
                    const material = new THREE.MeshPhongMaterial({ map:
 texture, specular: new THREE.Color('grey'), shininess: 10 });
                    earth = new THREE.Mesh(geometry, material);
                    scene.add(earth);
                }
            },
            undefined,
            function (error) {
                console.error('纹理加载失败:', error);
            }
        );
    });

    // 事件监听
    document.getElementById('timeSelect').addEventListener('change', fu
nction () {
        const time = parseInt(this.value);
        currentTimeIndex = timePoints.indexOf(time);
        if (earth && textures[time]) {
            earth.material.map = textures[time];
            earth.material.needsUpdate = true;
        }
    });

    // 初始化光源
    const ambientLight = new THREE.AmbientLight(0x404040, 0.5);
    scene.add(ambientLight);
    const directionalLight = new THREE.DirectionalLight(0xffffff, 0.7);
    camera.add(directionalLight);
    directionalLight.position.set(0, 0, 1);
    scene.add(camera);
    camera.position.z = 15;

    // 控制器设置
    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
```

```
    controls.autoRotate = true;

    // 动画循环
    function animate() {
        requestAnimationFrame(animate);
        controls.update();
        renderer.render(scene, camera);
    }
    animate();
}

init();
```

3. 算法描述

- **纹理加载算法**：使用 `THREE.TextureLoader` 异步加载地球的纹理，确保在用户选择不同历史时期时，能够快速切换显示。
- **交互控制算法**：通过事件监听器捕获用户的输入（如按钮点击、下拉框选择），并根据输入更新地球的显示状态。

4. 关键技术细节

- **WebGL 渲染**：使用 `THREE.WebGLRenderer` 进行高效的 3D 渲染，支持抗锯齿和高性能模式。
- **相机控制**：使用 `THREE.OrbitControls` 实现相机的平滑移动和缩放，增强用户交互体验。
- **光源设置**：通过环境光和方向光的组合，提升场景的光照效果，使地球模型更加真实。

四、总结

本项目通过使用 Three.js 库实现了一个交互式的地球古地理可视化工具，用户可以方便地查看不同历史时期的地球状态。项目的创新功能和良好的用户体验使其在教育和科研领域具有广泛的应用前景。未来可以考虑进一步优化性能，增加更多的交互功能和数据可视化效果。

index.html.md

实验报告：生物演化史导航页

一、引言

本实验旨在开发一个生物演化史的导航页面，提供用户友好的界面以便于访问不同设备上的内容。该页面通过检测用户的设备类型（移动端或 PC 端）来实现自动跳转，并提供手动选择的功能。页面设计简洁明了，旨在提升用户体验。

二、创新功能点

1. **设备自动检测与跳转**：通过 JavaScript 检测用户的设备类型，自动跳转到相应的页面（移动端或 PC 端），提高了用户的访问效率。
2. **手动选择功能**：在自动跳转的同时，提供手动选择按钮，确保用户在特殊情况下仍能访问所需内容。
3. **响应式设计**：页面设计考虑了不同设备的屏幕尺寸，确保在各种设备上都能良好显示。

三、代码实现细节

1. HTML 结构

页面的基本结构使用 HTML5 标准，包含头部和主体部分。头部定义了页面的元数据、样式和脚本，主体部分则包含了标题和手动选择的按钮。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>生物演化史导航页</title>
    ...
</head>
<body>
    <h1 class="title">生物演化史 - EvoViz</h1>
    <div id="manual-selection" style="display: none;">
        ...
    </div>
</body>
</html>
```

2. JavaScript 实现

设备检测与跳转

使用 `navigator.userAgent` 来检测用户的设备类型。根据检测结果，页面会自动跳转到相应的内容页面。

```
window.onload = function() {
    // 显示手动选择按钮
    document.getElementById('manual-selection').style.display = 'block';

    // 检测设备类型
    let target = '';
    if (/Mobi|Android/i.test(navigator.userAgent) && !/Windows NT/i.tes
t(navigator.userAgent)) {
```

```
            target = '<span class="highlight">移动端</span>';
            // 缩小标题字体
            document.querySelector('.title').style.fontSize = '30px';
            // 立即跳转
            window.location.href = './mobile-index.html';
        } else {
            target = '<span class="highlight">PC 端</span>';
            // 立即跳转
            window.location.href = './pc-index.html';
        }
}
```

3. CSS 样式

页面的样式使用 CSS 进行定义，确保页面在不同设备上的美观性和可读性。

```
.title {
    color: #FF0000;
    font-size: 48px;
    font-family: "Microsoft YaHei";
    text-align: center;
    margin-top: 20px;
    padding: 20px;
}
#manual-selection button {
    width: 300px;
    height: 80px;
    margin: 10px auto;
    padding: 0;
    font-size: 24px;
    display: block;
}
```

四、算法描述

本实验的核心算法是设备类型检测算法。该算法通过正则表达式匹配用户代理字符串，判断用户的设备类型。具体步骤如下：

1. 获取用户的 `userAgent` 字符串。
2. 使用正则表达式检查是否包含“Mobile”或“Android”。
3. 如果匹配成功，认为用户使用的是移动设备，进行相应的跳转；否则，跳转到 PC 端页面。

五、库函数引用

在本实验中，虽然没有直接使用 D3.js 和 Three.js 库，但可以考虑在后续版本中引入这些库以增强页面的交互性和可视化效果。

- **D3.js**：用于数据驱动的文档操作，可以在生物演化史的可视化展示中使用。
- **Three.js**：用于 3D 图形的渲染，可以在未来的版本中实现生物演化的三维模型展示。

六、总结

本实验成功实现了一个生物演化史的导航页面，具备设备自动检测、手动选择和响应式设计等功能。通过 JavaScript 和 CSS 的结合，页面在不同设备上均能良好展示。未来可以考虑引入更多的可视化库，以进一步提升用户体验和页面的交互性。

mobile-biological-classification-sunburst-2D.html.md

实验报告：生物分类可视化系统

一、引言

本实验旨在开发一个基于 Web 的生物分类可视化系统，利用 D3.js 库实现生物分类数据的动态展示。该系统通过交互式的方式，帮助用户更好地理解生物分类的层次结构和相关信息。系统的创新功能包括放大镜效果、搜索功能、全屏显示等，提升了用户体验和数据可视化效果。

二、创新功能点

1. **动态交互**：用户可以通过鼠标拖拽和键盘操作（WASD 和方向键）来旋转和移动生物分类图，增强了交互性。
2. **放大镜效果**：用户可以通过点击按钮显示放大镜，实时查看生物分类图的细节，提升了可视化效果。
3. **搜索功能**：用户可以通过输入生物名称进行搜索，系统会高亮显示匹配的节点，并更新信息面板。
4. **全屏显示**：用户可以选择进入全屏模式，提供更大的视图空间以便于观察复杂的生物分类结构。

三、算法描述

3.1 数据结构

系统使用树形结构来表示生物分类数据。每个节点包含以下信息： - `name`: 生物名称 - `src`: 生物图像链接 - `time`: 生物出现的时期 - `description`: 生物的简要描述 - `description_more`: 生物的详细描述 - `children`: 子节点数组

3.2 颜色生成算法

根据节点的深度和所属分支，使用线性插值生成不同的颜色，以便于区分不同的生物分类。例如，动物界使用橙色系，植物界使用绿色系等。

```
const color = d => {
    let node = d;
    while (node.parent) {
        if (node.data.name === "动物界") {
            return d3.scaleLinear()
                .domain([0, 5])
                .range(["#FFA500", "#FF8C00"])(d.depth);
        }
        // 其他分支的颜色处理...
        node = node.parent;
    }
    return d3.scaleLinear()
        .domain([0, 5])
        .range(["#FF8C00", "#FF4500"])(d.depth);
};
```

3.3 SVG 生成与交互

使用 D3.js 库生成 SVG 元素，并通过事件监听实现用户交互。用户可以通过鼠标拖拽来旋转图形，使用键盘控制移动。

```
const svg = d3.select("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${width/2},${height/2})`);

d3.select("body")
    .on("mousedown", dragstarted)
    .on("mousemove", dragged)
    .on("mouseup", dragended);
```

四、代码实现

4.1 HTML 结构

HTML 文件的基本结构包括头部信息、样式、脚本和主体内容。使用了 D3.js 和 html2canvas 等库。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
```

```
    <title>简简单的生物分类</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <script src="./biological-classification-data.js"></script>
    <script src="https://html2canvas.hertzen.com/dist/html2canvas.min.j
s"></script>
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <div class="control-tips">...</div>
    <div class="content-container">
        <svg width="1000" height="1000"></svg>
        <div class="info-panel">...</div>
    </div>
    <div class="control-buttons">...</div>
    <div class="search-container">...</div>
    <div id="magnifier">...</div>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

4.2 D3.js 实现细节

4.2.1 创建分区布局

使用 D3.js 的分区布局来处理数据并生成 SVG 路径。

```
const partition = data => {
    const root = d3.hierarchy(data)
        .sum(d => 1)
        .sort((a, b) => b.value - a.value);
    return d3.partition()
        .size([2 * Math.PI, radius])
        .padding(0.005)(root);
};
```

4.2.2 创建弧生成器

使用 D3.js 的弧生成器来绘制每个节点的弧形。

```
const arc = d3.arc()
    .startAngle(d => d.x0)
    .endAngle(d => d.x1)
    .padAngle(0.005)
    .padRadius(radius / 2)
```

```
    .innerRadius(d => d.y0)
    .outerRadius(d => d.y1 - 1;
```

4.3 事件处理

实现鼠标悬停、点击等事件处理，更新信息面板。

```
path.on("mouseover", function(event, d) {
    // 更新信息面板
})
.on("mouseout", function() {
    // 恢复初始状态
});
```

五、总结

本实验成功实现了一个基于 D3.js 的生物分类可视化系统，具备动态交互、放大镜效果、搜索功能和全屏显示等创新功能。通过合理的数据结构和算法设计，用户能够直观地理解生物分类的层次关系。未来可以进一步优化性能，增加更多的生物分类数据，以提升系统的实用性和可扩展性。

mobile-biological-evolution-dna-3D.html.md

实验报告：生物演化史-DNA 双螺旋-3D 可视化

一、引言

本实验旨在通过 3D 可视化技术展示生物演化史中的 DNA 双螺旋结构，利用 Three.js 库实现动态交互效果，增强用户的学习体验。该项目结合了生物学、计算机图形学和用户交互设计，展示了如何通过现代 Web 技术将复杂的科学概念以直观的方式呈现。

二、创新功能点

1. **动态时间旅行模式**：用户可以通过控制面板播放、暂停和调整播放速度，模拟生物演化的时间线，直观感受生物演化的过程。
2. **交互式信息展示**：用户可以通过鼠标悬停在时间线上的节点，查看详细的事件信息，包括时间、描述和相关图片。
3. **全景视图**：用户可以通过鼠标控制相机视角，360 度查看 DNA 双螺旋结构，增强沉浸感。
4. **自适应界面**：界面元素根据屏幕大小自适应调整，确保在不同设备上均能良好展示。

三、代码实现细节

3.1 技术栈

- **HTML/CSS**：用于构建页面结构和样式。
- **JavaScript**：实现动态交互和逻辑控制。
- **Three.js**：用于 3D 图形渲染。
- **D3.js**：用于数据可视化（在本项目中未直接使用，但可扩展用于数据处理）。

3.2 主要库函数引用

Three.js

Three.js 是一个强大的 JavaScript 库，用于创建和显示 3D 图形。以下是一些关键的实现细节：

- **场景创建：**

```
scene = new THREE.Scene();
scene.background = new THREE.Color(0x000000);
```

- **相机设置：**

```
camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.
innerHeight, 0.1, 10000);
camera.position.set(100, 0, 100);
camera.lookAt(0, 0, 0);
```

- **渲染器初始化：**

```
renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

- **光源添加：**

```
const ambientLight = new THREE.AmbientLight(0xffffff, 0.5);
scene.add(ambientLight);
```

- **创建 DNA 双螺旋：**

```
function createDNASpiral() {
    const curve1Points = [];
    const curve2Points = [];
    const radius = 20;
    const height = 500;
    const turns = 25;
```

```
    for(let i = 0; i <= 360 * turns; i++) {
        const angle = (i * Math.PI) / 180;
        const y = (i / (360 * turns)) * height - height/2;
        curve1Points.push(new THREE.Vector3(radius * Math.cos(ang
le), y, radius * Math.sin(angle)));
        curve2Points.push(new THREE.Vector3(radius * Math.cos(ang
le + Math.PI), y, radius * Math.sin(angle + Math.PI)));
    }

    const curve1Geometry = new THREE.BufferGeometry().setFromPoin
ts(curve1Points);
    const curve2Geometry = new THREE.BufferGeometry().setFromPoin
ts(curve2Points);
    const material = new THREE.LineBasicMaterial({ color: 0x00ff8
8, opacity: 0.8, transparent: true });
    const dnaStrand1 = new THREE.Line(curve1Geometry, material);
    const dnaStrand2 = new THREE.Line(curve2Geometry, material);
    scene.add(dnaStrand1);
    scene.add(dnaStrand2);
}
```

3.3 交互功能实现

- **时间控制面板：**

```
function createTimeControls() {
    const controlsContainer = document.createElement('div');
    controlsContainer.style.position = 'absolute';
    controlsContainer.style.bottom = '20px';
    controlsContainer.style.right = '20px';
    controlsContainer.style.zIndex = '1000';
    controlsContainer.style.display = 'flex';
    controlsContainer.style.flexDirection = 'column';
    controlsContainer.style.gap = '15px';
    controlsContainer.style.padding = '20px';
    controlsContainer.style.background = 'rgba(0,0,0,0.7)';
    controlsContainer.style.borderRadius = '10px';
    controlsContainer.style.backdropFilter = 'blur(5px)';
    controlsContainer.style.boxShadow = '0 0 20px rgba(0,255,0,0.
2)';

    // 添加播放、暂停、前进、倒退按钮
    const playButton = document.createElement('button');
    playButton.innerText = '播放';
    playButton.onclick = () => {
        isPlaying = !isPlaying;
        playButton.innerText = isPlaying ? '暂停' : '播放';
    };
    controlsContainer.appendChild(playButton);
```

```
    document.body.appendChild(controlsContainer);
}
```

- **鼠标事件处理：**

```
window.addEventListener('mousemove', onMouseMove);
function onMouseMove(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;
    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(timelineObjects.
children);
    if(intersects.length > 0) {
        const event = intersects[0].object.userData;
        showEventDetails(event);
    } else {
        hideEventDetails();
    }
}
```

3.4 数据处理与可视化

在本项目中，数据以 JSON 格式存储，包含生物演化的时间节点、事件描述和相关图片。通过解析这些数据，动态生成时间线上的节点，并为每个节点添加交互功能。

四、算法描述

1. **时间线更新算法：**
   - 每当播放状态为真时，定时更新当前时间索引，并根据索引更新可见的事件。
   - 通过 `requestAnimationFrame` 实现平滑的动画效果。
2. **事件处理算法：**
   - 鼠标移动时，使用射线投射技术检测鼠标与时间线节点的交互。
   - 根据交互结果显示或隐藏事件详情。

五、结论

本实验通过 Three.js 实现了生物演化史的 3D 可视化，展示了 DNA 双螺旋的结构及其演化过程。通过动态交互和信息展示，用户能够更直观地理解生物演化的复杂性。未来可以进一步扩展功能，例如增加更多的生物演化事件、优化用户界面和增强数据可视化效果。

六、参考文献

- Three.js 官方文档：https://threejs.org/docs/
- D3.js 官方文档：https://d3js.org/

- 相关生物学文献与资料。

mobile-biological-evolution-force-2D.html.md

实验报告：生物演化史-力导向-2D 可视化

一、引言

本实验旨在通过可视化技术展示生物演化的历史，利用 D3.js 库实现力导向图的交互式展示。该项目不仅展示了生物演化的时间线，还通过创新的功能点提升了用户体验，使得复杂的生物演化信息更加直观易懂。

二、创新功能点

1. **力导向布局**：使用 D3.js 的力导向图算法，动态展示节点之间的关系，用户可以通过拖拽和缩放操作自由探索图形。
2. **节点信息展示**：当用户悬停在节点上时，显示该节点的详细信息，包括名称、时间和描述，增强了信息的可获取性。
3. **背景图像切换**：根据用户选择的节点，动态更换背景图像，提供更丰富的视觉体验。
4. **交互式控制**：提供展开/收起所有节点的功能，用户可以快速查看或隐藏信息，提升了交互性。
5. **移动设备适配**：针对移动设备优化了触摸事件处理，确保在不同设备上均能流畅使用。

三、算法描述

3.1 力导向图算法

力导向图算法通过模拟物理力的作用来布局节点。每个节点之间的关系通过“弹簧力”和“斥力”来实现：

- **弹簧力**：通过 `d3.forceLink()`实现，控制节点之间的距离。
- **斥力**：通过 `d3.forceManyBody()`实现，防止节点重叠。
- **中心力**：通过 `d3.forceCenter()`实现，确保节点向中心聚集。

3.2 D3.js 库的使用

D3.js 是一个强大的数据可视化库，提供了丰富的功能来处理数据和生成图形。以下是本项目中使用的主要 D3.js 功能：

- **SVG 元素创建**：使用 `d3.select()`和 `append()`方法创建 SVG 容器和图形元素。

- **数据绑定**：通过.data()方法将数据绑定到 DOM 元素，实现动态更新。
- **力导向模拟**：使用 d3.forceSimulation()创建力导向模拟，并通过.on("tick", ...)方法更新节点和链接的位置。

四、代码实现

以下是项目的关键代码实现细节：

4.1 HTML 结构

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>生物演化史-力导向-2D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <script src="biological-evolution-data.js"></script>
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

4.2 SVG 容器创建

```
const svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height);
```

4.3 力导向模拟设置

```
const simulation = d3.forceSimulation()
    .force("link", d3.forceLink().id(d => d.id).distance(100)) // 弹簧
力
    .force("charge", d3.forceManyBody().strength(-300)) // 节点间的斥力
    .force("center", d3.forceCenter(width / 2, height / 2)); // 中心力
```

4.4 节点和链接的更新

```
function update(source) {
    const nodes = root.descendants();
    const links = root.links();
```

```
    // 更新力导向模拟
    simulation.nodes(nodes);
    simulation.force("link").links(links);
    simulation.alpha(1).restart();

    // 更新节点位置
    g.selectAll(".node")
        .attr("transform", d => `translate(${d.x},${d.y})`);

    // 更新链接路径
    g.selectAll(".link")
        .attr("d", d => `M${d.source.x},${d.source.y} C${(d.source.x +
d.target.x) / 2},${d.source.y} ${(d.source.x + d.target.x) / 2},${d.tar
get.y} ${d.target.x},${d.target.y}`);
}
```

4.5 事件处理

```
svg.on("mousedown", (event) => {
    isDragging = true;
    startPoint = { x: event.clientX - currentTranslate.x, y: event.clie
ntY - currentTranslate.y };
});
```

五、库函数引用

5.1 D3.js

- **选择器**：`d3.select()`用于选择 DOM 元素。
- **数据绑定**：`data()`方法用于将数据绑定到元素。
- **力导向模拟**：`forceSimulation()`用于创建力导向图。

5.2 Three.js（如有使用）

在本项目中未使用 Three.js，但可以通过类似的方式实现 3D 可视化。

六、结论

本实验通过 D3.js 实现了生物演化史的力导向可视化，展示了生物演化的复杂关系。通过创新的功能点和良好的用户体验，提升了信息的可获取性和可视化效果。未来可以考虑引入更多的交互功能和数据分析，以进一步丰富用户体验。

七、参考文献

- D3.js 官方文档：D3.js
- 力导向图算法相关文献
- 生物演化相关资料

以上是本实验的详细报告，涵盖了创新功能、算法描述、代码实现及库函数引用等方面。希望对您有所帮助！

mobile-biological-evolution-star-3D.html.md

实验报告：生物演化史-星空图-3D

一、引言

本实验旨在通过 3D 可视化技术展示生物演化史，利用 Three.js 库实现动态交互效果，增强用户体验。该项目结合了生物学、计算机图形学和用户交互设计，展示了生物演化的时间线和不同地质年代的特征。

二、创新功能点

1. **动态交互**：用户可以通过鼠标点击和移动与 3D 模型进行交互，查看不同节点的详细信息。
2. **自适应布局**：根据不同的生物分类，自动调整节点的高度和位置，形成清晰的层次结构。
3. **可视化效果**：使用 Three.js 的粒子系统创建星空背景，增强视觉效果。
4. **信息展示**：点击节点后，动态显示该节点的详细信息，包括时间、描述和相关图片。
5. **全屏模式**：支持全屏切换，提升用户的沉浸感。

三、算法描述

3.1 数据结构

使用树形结构表示生物演化的各个节点，每个节点包含以下信息：`- name`：节点名称（如地质年代）`- time`：时间信息 `- description`：节点描述 `- src`：相关图片链接 `- children`：子节点数组

3.2 主要算法

1. **节点创建**：根据树形数据结构递归创建节点，使用 Three.js 的 `SphereGeometry` 生成球体表示节点。
2. **连接线绘制**：使用贝塞尔曲线连接父节点和子节点，增强视觉效果。
3. **动态更新**：通过调整参数（如节点大小、线长度等）实时更新可视化效果。

四、代码实现

4.1 主要库引用

本项目主要使用以下库： - **Three.js**：用于 3D 图形渲染。 - **D3.js**：用于数据处理和可视化（在本项目中未直接使用，但可用于数据的预处理）。

4.2 代码实现细节

以下是关键代码片段的实现细节：

4.2.1 初始化场景

```
function init() {
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x000000);

    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.i
nnerHeight, 0.1, 10000);
    camera.position.set(120, 270, 500);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.maxDistance = 3000;
    controls.minDistance = 50;

    createStarField();
    networkGroup = createNetworkVisualization();

    animate();
}
```

- **场景设置**：创建一个黑色背景的场景，并设置相机位置。
- **渲染器**：使用 WebGL 渲染器进行 3D 图形渲染。
- **控制器**：使用 `OrbitControls` 实现相机的平滑移动和缩放。

4.2.2 创建星空背景

```
function createStarField() {
    const starsGeometry = new THREE.BufferGeometry();
    const starsMaterial = new THREE.PointsMaterial({
        color: 0xFFFFFF,
        size: 0.3,
        transparent: true,
        opacity: 1.0,
        sizeAttenuation: true
```

```
    });

    const starsVertices = [];
    for(let i = 0; i < 15000; i++) {
        const x = (Math.random() - 0.5) * 2000;
        const y = (Math.random() - 0.5) * 2000;
        const z = (Math.random() - 0.5) * 2000;
        starsVertices.push(x, y, z);
    }

    starsGeometry.setAttribute('position', new THREE.Float32BufferAttri
bute(starsVertices, 3));
    const starField = new THREE.Points(starsGeometry, starsMaterial);
    scene.add(starField);
}
```

- **粒子系统**：使用 BufferGeometry 和 PointsMaterial 创建星空效果，生成 15000 个随机位置的星星。

4.2.3 创建网络可视化

```
function createNetworkVisualization() {
    const networkGroup = new THREE.Group();

    function createNodes(data, position = new THREE.Vector3(), level =
0, angle = 0, parentColor, baseAngle = null) {
        const nodeGeometry = new THREE.SphereGeometry(networkParameters.
nodeSize, 32, 32);
        let color = (level === 1) ? categoryColors[data.name] : parentC
olor || new THREE.Color(0xffffff);

        const nodeMaterial = new THREE.MeshPhongMaterial({
            color: color,
            emissive: color,
            emissiveIntensity: 0.5,
            transparent: true,
            opacity: 0.8
        });

        const node = new THREE.Mesh(nodeGeometry, nodeMaterial);
        node.position.copy(position);
        node.userData = data;

        networkGroup.add(node);

        if (data.children) {
            data.children.forEach((child, index) => {
                const childPosition = calculateChildPosition(position,
level, index);
                const connection = createCurvedLine(position, childPosi
```

```
tion, color);
                networkGroup.add(connection);
                createNodes(child, childPosition, level + 1, childAngle,
 color);
            });
        }
    }

    createNodes(treeData);
    scene.add(networkGroup);
    return networkGroup;
}
```

- **节点创建**：根据传入的数据递归创建节点，并为每个节点设置颜色和材质。
- **连接线**：使用 `createCurvedLine` 函数绘制父子节点之间的连接线。

4.3 事件处理

```
window.addEventListener('click', onMouseClick);

function onMouseClick(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(networkGroup.children.
filter(child => child instanceof THREE.Mesh));

    if (intersects.length > 0) {
        const selectedObject = intersects[0].object;
        showEventDetails(selectedObject.userData);
    } else {
        hideEventDetails();
    }
}
```

- **鼠标点击事件**：通过 `Raycaster` 检测用户点击的节点，并显示相应的详细信息。

五、总结

本实验通过 Three.js 实现了生物演化史的 3D 可视化，展示了不同地质年代的特征和演化过程。通过动态交互和视觉效果，提升了用户的体验。未来可以进一步优化数据处理和可视化效果，增加更多的交互功能，以便更好地展示生物演化的复杂性和美丽。

mobile-biological-evolution-tree-2D.html.md

实验报告：生物演化史-Tree-2D 可视化项目

一、项目概述

本项目旨在通过可视化技术展示生物演化的历史，采用 D3.js 库构建一个交互式的树形图。用户可以通过选择不同的时间段来过滤展示的生物演化信息，并通过搜索功能快速定位特定节点。该项目不仅展示了生物演化的复杂性，还提供了用户友好的交互体验。

二、创新功能点

1. **动态树形布局**：根据用户选择的时间段动态调整树形图的高度和宽度，确保信息的清晰展示。
2. **交互式工具提示**：当用户悬停或点击节点时，显示详细信息，包括生物名称、时间和描述，增强用户体验。
3. **搜索功能**：用户可以通过输入关键字快速搜索节点，系统会实时更新搜索结果并高亮显示路径。
4. **设备检测**：在移动设备上，提供特定的提示信息，确保用户体验的一致性。
5. **高亮路径**：用户点击搜索结果后，系统会高亮显示从目标节点到根节点的路径，帮助用户理解生物演化的关系。

三、算法描述

1. 数据结构

使用 D3.js 的层次结构（hierarchy）来表示生物演化数据。每个节点包含以下属性： - `name`: 生物名称 - `src`: 图片链接 - `description`: 生物描述 - `time`: 生物出现的时间

2. 树形布局算法

使用 D3.js 的树形布局算法，计算节点的位置和连接线的路径。根据节点数量动态调整 SVG 的高度，确保所有节点都能在视口内显示。

3. 事件处理

- **鼠标事件**：通过 `mouseover` 和 `mouseout` 事件处理工具提示的显示与隐藏。
- **搜索事件**：监听输入框的变化，实时过滤和显示匹配的节点。

四、代码实现

1. HTML 结构

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>生物演化史-Tree-2D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <style>
        /* CSS 样式省略 */
    </style>
</head>
<body>
    <div id="alertMessage" class="alert-message"></div>
    <div class="control-buttons">
        <!-- 控制按钮省略 -->
    </div>
    <div class="time-filter">
        <!-- 时间过滤器省略 -->
    </div>
    <div class="search-container">
        <input type="text" id="search-input" placeholder="搜索节点...">
        <div id="search-results"></div>
    </div>
    <div id="tree"></div>
    <div class="tooltip"></div>
    <script src="./biological-evolution-data.js"></script>
    <script>
        // JavaScript 代码省略
    </script>
</body>
</html>
```

2. D3.js 树形布局实现

```
// 创建 SVG 容器
const svg = d3.select("#tree")
    .append("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

// 创建树布局
const tree = d3.tree()
    .size([height - margin.top - margin.bottom, width - margin.left - m
```

```
argin.right]);

// 渲染树形图
function renderTree(data, filter = 'all') {
    // 清除现有内容
    svg.selectAll("*").remove();

    // 过滤数据
    let filteredData = JSON.parse(JSON.stringify(data));
    if (filter !== 'all') {
        filteredData.children = filteredData.children.filter(d => d.nam
e === filter);
    }

    // 创建层次结构
    const root = d3.hierarchy(filteredData);
    tree(root);

    // 绘制连接线
    svg.selectAll(".link")
        .data(root.links())
        .enter()
        .append("path")
        .attr("class", "link")
        .attr("d", d3.linkHorizontal()
            .x(d => d.y)
            .y(d => d.x));

    // 绘制节点
    const nodes = svg.selectAll(".node")
        .data(root.descendants())
        .enter()
        .append("g")
        .attr("class", "node")
        .attr("transform", d => `translate(${d.y},${d.x})`);

    // 添加节点圆圈
    nodes.append("circle")
        .attr("r", 5);

    // 添加文本标签
    nodes.append("text")
        .attr("dy", ".35em")
        .attr("x", d => d.children ? -40 : 40)
        .attr("text-anchor", d => d.children ? "end" : "start")
        .text(d => d.data.name);
}
```

3. 交互功能实现

```
// 添加鼠标事件
nodes.on("mouseover", function(event, d) {
    tooltip.style("display", "block")
        .html(`
            <strong>${d.data.name}</strong>
            ${d.data.src ? `<img src="${d.data.src}" alt="${d.data.name}
">` : ''}
            <div class="tooltip-text">
                ${d.data.name ? '<br>时间: ' + d.data.name : ''}
                ${d.data.description ? '<br>描述: ' + d.data.descriptio
n : ''}
            </div>
        `)
        .style("left", "50%")
        .style("top", "50%");
})
.on("mouseout", function() {
    tooltip.style("display", "none");
});
```

4. 搜索功能实现

```
searchInput.addEventListener('input', function(e) {
    const searchTerm = e.target.value.trim().toLowerCase();
    if (!searchTerm) {
        searchResults.innerHTML = '';
        return;
    }

    const matches = allNodes.filter(node =>
        node.data.name && node.data.name.toLowerCase().includes(searchT
erm)
    );

    searchResults.innerHTML = matches
        .map(node => `
            <div class="search-result-item" data-node-id="${node.data.n
ame}">
                ${node.data.name}
            </div>
        `).join('');
});
```

五、库函数引用

D3.js

- **d3.hierarchy**: 用于创建层次结构，便于后续的树形布局计算。
- **d3.tree**: 创建树形布局，计算节点的位置。

- **d3.linkHorizontal**: 生成连接线的路径。

Three.js

本项目未使用 Three.js，但可以考虑在未来版本中结合 Three.js 实现 3D 可视化效果。

六、总结

本项目通过 D3.js 实现了生物演化史的可视化，提供了动态交互功能，增强了用户体验。未来可以考虑引入更多的可视化库，如 Three.js，进一步提升展示效果。通过不断优化和扩展功能，项目将更好地服务于生物学研究和教育。

mobile-biological-evolution-tree-3D.html.md

实验报告：生物演化史-Tree-3D 可视化项目

1. 引言

本实验报告旨在详细介绍“生物演化史-Tree-3D”项目的设计与实现。该项目利用 Web 技术和 3D 可视化库（如 Three.js）展示生物演化的历史，提供用户交互功能，增强学习体验。报告将涵盖创新功能点、代码实现细节、算法描述以及所使用的库函数。

2. 创新功能点

2.1 交互式可视化

用户可以通过鼠标点击节点来聚焦特定生物种类，系统会平滑移动相机到该节点位置，并展示相关信息。这种交互方式使得用户能够深入了解生物演化的细节。

2.2 动态参数调整

用户可以通过滑块动态调整可视化参数（如高度差、线长度和垂直间距），实时更新 3D 图形。这种功能使得用户能够根据个人需求自定义可视化效果。

2.3 全屏模式

提供全屏切换功能，用户可以在更大的视图中体验生物演化的 3D 效果，增强沉浸感。

2.4 详细信息展示

点击节点后，系统会在侧边展示该节点的详细信息，包括时间、描述和相关图片，帮助用户更好地理解生物演化的背景。

3. 代码实现细节

3.1 HTML 结构

项目的 HTML 结构简单明了，主要包括头部信息、样式、脚本引用和主体内容。以下是关键部分的代码示例：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>生物演化史-Tree-3D</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <style>
        /* 样式定义 */
    </style>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/t
hree.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/three@0.128.0/examples/js
/controls/OrbitControls.js"></script>
    <script src="./biological-evolution-data.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/tween.js/18.6.4
/tween.umd.js"></script>
</head>
<body>
    <div id="info">生物演化史-Tree-3D</div>
    <div id="details"></div>
    <div id="controls-panel">
        <!-- 控制面板 -->
    </div>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

3.2 JavaScript 实现

3.2.1 初始化 Three.js 场景

在 `init` 函数中，创建了 Three.js 的场景、相机和渲染器，并设置了光源和控制器。

```
function init() {
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.i
nnerHeight, 0.1, 10000);
    renderer = new THREE.WebGLRenderer({ antialias: true });
```

```
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    const ambientLight = new THREE.AmbientLight(0xffffff, 0.5);
    scene.add(ambientLight);

    const pointLight = new THREE.PointLight(0xffffff, 1);
    pointLight.position.set(100, 100, 100);
    scene.add(pointLight);

    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.maxDistance = 6000;
    controls.minDistance = 10;
    controls.target = new THREE.Vector3(0, 0, 0);

    animate();
}
```

3.2.2 创建网络可视化

createNetworkVisualization 函数负责根据生物演化数据生成 3D 网络图。该函数使用了 Three.js 的几何体和材质来创建节点和连接线。

```
function createNetworkVisualization() {
    const networkGroup = new THREE.Group();
    // 创建节点和连接线的逻辑
    createNodes(treeData, new THREE.Vector3(0, -700, 0));
    scene.add(networkGroup);
    return networkGroup;
}
```

3.2.3 节点创建与连接

createNodes 函数递归地创建节点，并使用 createCurvedLine 函数生成连接线。节点的颜色和位置根据其在生物演化树中的层级和类别进行设置。

```
function createNodes(data, position = new THREE.Vector3(0, -500, 0), le
vel = 0, angle = 0, parentColor, baseAngle = null) {
    const nodeGeometry = new THREE.SphereGeometry(10, 32, 32);
    const color = categoryColors[data.name] || new THREE.Color(0xfffff
f);
    const nodeMaterial = new THREE.MeshPhongMaterial({ color: color });
    const node = new THREE.Mesh(nodeGeometry, nodeMaterial);
    node.position.copy(position);
    networkGroup.add(node);

    if (data.children) {
        data.children.forEach((child, index) => {
            const childPosition = calculateChildPosition(position, inde
```

```
x, data.children.length);
            const connection = createCurvedLine(position, childPosition,
 color);
            networkGroup.add(connection);
            createNodes(child, childPosition, level + 1, angle, color);
        });
    }
}
```

3.3 交互功能实现

3.3.1 鼠标点击事件

通过 onMouseClick 函数处理鼠标点击事件，使用 Raycaster 检测用户点击的节点，并展示详细信息。

```
function onMouseClick(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;

    raycaster.setFromCamera(mouse, camera);
    const intersects = raycaster.intersectObjects(networkGroup.childre
n);

    if (intersects.length > 0) {
        const clickedNode = intersects[0].object;
        showEventDetails(clickedNode.userData);
    }
}
```

3.3.2 滑块控制

通过滑块调整参数，实时更新可视化效果。每当滑块值变化时，都会更新网络的可视化。

```
function initializeSliders() {
    const sliders = ['heightStep', 'radius', 'levelHeight'];
    sliders.forEach(param => {
        const slider = document.getElementById(param);
        slider.addEventListener('input', function() {
            networkParameters[param] = parseFloat(this.value);
            updateVisualization();
        });
    });
}
```

4. 算法描述

4.1 数据结构

项目使用树形结构表示生物演化数据，每个节点包含名称、时间、描述和子节点信息。数据结构示例如下：

```
const treeData = {
    name: "生物演化",
    children: [
        {
            name: "冥古宙",
            time: "45.4 亿年前",
            description: "地球形成初期的生物。",
            children: []
        },
        // 其他节点...
    ]
};
```

4.2 渲染算法

渲染算法主要包括以下步骤：

1. 初始化 Three.js 场景。
2. 根据数据结构递归创建节点和连接线。
3. 处理用户交互，更新视图。

5. 库函数引用

5.1 Three.js

Three.js 是一个强大的 3D 图形库，提供了丰富的几何体、材质和光源选项。项目中使用了以下 Three.js 功能：

- `THREE.Scene`：创建 3D 场景。
- `THREE.PerspectiveCamera`：设置透视相机。
- `THREE.WebGLRenderer`：渲染 3D 图形。
- `THREE.Mesh`：创建 3D 网格对象。
- `THREE.Raycaster`：用于检测鼠标与 3D 对象的交互。

5.2 Tween.js

Tween.js 用于实现平滑动画效果，特别是在相机移动时。通过设置目标位置和动画持续时间，创建流畅的过渡效果。

```
new TWEEN.Tween(camera.position)
    .to(newCameraPosition, 1000)
    .easing(TWEEN.Easing.Quadratic.InOut)
    .start();
```

6. 结论

本项目通过结合 Three.js 和 Tween.js，实现了一个生动的生物演化可视化工具。用户可以通过交互式界面探索生物演化的历史，动态调整可视化参数，增强了学习的趣味性和有效性。未来可以考虑增加更多的生物数据和交互功能，以进一步提升用户体验。

mobile-biological-evolution-visualization-3D.html.md

实验报告：生物演化史可视化大屏-3D

引言

本实验旨在开发一个生物演化史的可视化大屏，利用 3D 图形技术展示生物演化的过程和节点。该项目结合了 D3.js 和 Three.js 两个强大的 JavaScript 库，前者用于数据处理和力导向布局，后者用于 3D 图形渲染。通过该可视化工具，用户可以直观地了解生物演化的历史和各个节点之间的关系。

创新功能点

1. **3D 可视化**：通过 Three.js 实现生物演化节点的 3D 展示，用户可以通过鼠标拖拽和滚轮缩放来查看不同角度的演化图。
2. **动态交互**：用户可以点击节点查看详细信息，节点的颜色和大小根据其重要性和属性动态变化。
3. **多种布局方式**：支持力导向布局、树形布局、球形布局和圆形布局，用户可以根据需求选择不同的布局方式。
4. **背景和天空盒选择**：用户可以选择不同的背景颜色和天空盒，增强视觉效果。
5. **实时数据更新**：通过 D3.js 的力导向算法，节点和连接的布局会根据用户的交互实时更新。

算法描述

数据处理

使用 D3.js 处理生物演化的数据，构建节点和连接的关系。每个节点包含以下属性： - `id`: 节点的唯一标识符 - `name`: 节点名称 - `time`: 节点对应的地质年代 -

description: 节点的描述 - src: 节点的图像源 - size: 节点的大小 - color: 节点的颜色

力导向布局

使用 D3.js 的力导向算法来处理节点之间的关系。通过设置不同的力（如引力、斥力和中心力），实现节点的动态布局。具体实现如下：

```
simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id).distance(100).stre
ngth(1))
    .force("charge", d3.forceManyBody().strength(-300))
    .force("center", createCenter3D(0, 0, 0).strength(0.1))
    .force("collision", d3.forceCollide().radius(20))
    .force("3d", create3DForce(1))
    .force("axis", createAxisForce())
    .on("tick", updatePositions);
```

3D 渲染

使用 Three.js 进行 3D 渲染，创建场景、相机和渲染器。节点通过球体表示，连接通过线段表示。具体实现如下：

```
function createGraphObjects() {
    nodes.forEach(node => {
        const geometry = new THREE.SphereGeometry(node.size);
        const material = new THREE.MeshBasicMaterial({
            color: new THREE.Color(node.color || 0x00ff00) // 默认颜色
        });
        const sphere = new THREE.Mesh(geometry, material);
        node.object = sphere;
        scene.add(sphere);
    });

    links.forEach(link => {
        link.object = createEdge('line'); // 默认使用直线
        scene.add(link.object);
    });
}
```

代码实现细节

主要库函数引用

1. **D3.js**：
    - d3.forceSimulation(): 创建一个力导向模拟。
    - d3.forceLink(): 创建连接力。
    - d3.forceManyBody(): 创建斥力。
    - d3.forceCollide(): 创建碰撞力。

2. **Three.js**：
   - `THREE.Scene()`: 创建一个场景。
   - `THREE.PerspectiveCamera()`: 创建一个透视相机。
   - `THREE.WebGLRenderer()`: 创建一个 WebGL 渲染器。
   - `THREE.SphereGeometry()`: 创建球体几何体。
   - `THREE.MeshBasicMaterial()`: 创建基本材质。

代码实现示例

以下是部分关键代码的实现示例：

```
function init() {
    // 初始化 Three.js 场景
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.
innerHeight, 0.1, 10000);
    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.getElementById('container').appendChild(renderer.domElemen
t);

    // 添加节点和连接
    processData(treeData);
    createGraphObjects();
    animate();
}

function animate() {
    requestAnimationFrame(animate);
    controls.update();
    renderer.render(scene, camera);
}
```

结论

本实验成功实现了一个生物演化史的 3D 可视化大屏，利用 D3.js 和 Three.js 的强大功能，提供了动态交互和多种布局方式。该项目不仅展示了生物演化的历史，还为用户提供了丰富的交互体验。未来可以进一步扩展功能，如增加更多的节点属性、支持更多的交互方式等，以提升用户体验和数据展示的丰富性。

mobile-comment.html.md

实验报告：生物演化史可视化网页 EvoViz

一、引言

生物演化史可视化网页 EvoViz 旨在为用户提供一个直观的界面，以便更好地理解生物演化的过程。该网页结合了现代前端技术，利用 D3.js 和 Three.js 库实现数据的动态可视化，增强用户体验。

二、创新功能点

1. **交互式可视化**：用户可以通过鼠标或触摸屏与可视化图形进行交互，查看不同生物的演化关系。
2. **多平台支持**：该网页在移动端和 PC 端均可流畅运行，确保用户在不同设备上的使用体验。
3. **实时数据更新**：通过 AJAX 请求，网页可以实时获取最新的生物演化数据，确保信息的时效性。
4. **友好的用户界面**：简洁明了的设计，使得用户能够快速上手，轻松获取所需信息。

三、代码实现细节

3.1 HTML 结构

网页的基本结构使用 HTML5 标准，包含头部信息和主体内容。以下是主要的 HTML 代码片段：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>评论区</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
</head>
<body>
    <!-- 主要内容 -->
</body>
</html>
```

3.2 CSS 样式

使用内联样式来控制网页的布局和外观，确保在不同设备上的适配性。样式包括字体大小、颜色、边距等。

3.3 JavaScript 功能实现

3.3.1 禁用移动端左右滑动

为了提升用户体验，禁用移动端浏览器的左右滑动翻页行为：

```
document.body.addEventListener('touchmove', function(event) {
    event.preventDefault();
}, { passive: false });
```

3.3.2 D3.js 与 Three.js 的使用

- **D3.js**：用于数据驱动的文档操作，能够将数据绑定到 DOM 元素上，并实现动态更新。

  ```
  d3.json('data/evolution.json').then(function(data) {
      // 处理数据并生成可视化
  });
  ```

- **Three.js**：用于创建和显示 3D 图形，能够实现复杂的三维可视化效果。

  ```
  const scene = new THREE.Scene();
  const camera = new THREE.PerspectiveCamera(75, window.innerWidth
  / window.innerHeight, 0.1, 1000);
  const renderer = new THREE.WebGLRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // 创建 3D 对象
  const geometry = new THREE.BoxGeometry();
  const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  camera.position.z = 5;

  function animate() {
      requestAnimationFrame(animate);
      cube.rotation.x += 0.01;
      cube.rotation.y += 0.01;
      renderer.render(scene, camera);
  }
  animate();
  ```

3.4 数据处理与可视化

数据通过 AJAX 请求获取，使用 D3.js 进行解析和处理。以下是数据处理的示例代码：

```
d3.json('data/evolution.json').then(function(data) {
    // 解析数据
    const processedData = data.map(d => ({
        name: d.name,
        evolution: d.evolution
    }));

    // 生成可视化
    createVisualization(processedData);
});
```

四、算法描述

本项目的核心算法是基于树形结构的生物演化关系图。通过递归算法遍历生物的演化关系，生成可视化图形。具体步骤如下：

1. **数据解析**：将获取的 JSON 数据解析为可操作的对象。
2. **树形结构构建**：根据生物的演化关系构建树形结构。
3. **可视化生成**：使用 D3.js 和 Three.js 将树形结构转化为可视化图形。

五、结论

生物演化史可视化网页 EvoViz 通过结合 D3.js 和 Three.js，实现了一个功能丰富、用户友好的可视化平台。该项目不仅提升了用户对生物演化的理解，也为未来的研究提供了一个良好的基础。未来可以考虑增加更多的交互功能和数据源，以进一步丰富用户体验。

mobile-home-page.html.md

实验报告：旋转卡片展示网页

一、引言

本实验旨在创建一个具有交互性和视觉吸引力的旋转卡片展示网页。该网页使用 HTML、CSS 和 JavaScript 构建，旨在展示生物分类相关的图像。用户可以通过点击卡片进入不同的页面，增强了用户体验。本文将详细介绍该网页的创新功能、代码实现细节、算法描述以及所使用的库函数。

二、创新功能点

1. **旋转卡片效果**：通过 CSS 和 JavaScript 实现卡片的 3D 旋转效果，提升了视觉体验。
2. **移动端适配**：使用媒体查询确保在不同屏幕尺寸下的良好展示，增强了移动端用户体验。
3. **消息弹窗**：当用户点击特定卡片时，弹出提示信息，告知用户该页面仅在 PC 端可用。
4. **禁用滑动翻页**：在移动端禁用左右滑动翻页行为，避免用户误操作。

三、代码实现细节

1. HTML 结构

HTML 部分定义了网页的基本结构，包括头部信息、样式和主体内容。主要的内容是一个包含多个卡片的容器。

```
<div class="container">
    <!-- 卡片内容通过 JavaScript 动态生成 -->
</div>
```

2. CSS 样式

CSS 部分负责网页的布局和样式设计。使用了 Flexbox 和 Grid 布局来实现响应式设计。

```
body {
    margin: 0;
    min-height: 100vh;
    display: flex;
    justify-content: center;
    align-items: center;
    background: linear-gradient(135deg, #1a1a1a, #2a2a2a);
    perspective: 1000px; /* 3D 效果 */
}

.card-wrapper {
    position: relative;
    width: 160px;
    height: 160px;
    transition: transform 0.3s ease; /* 动画效果 */
}
```

3. JavaScript 逻辑

JavaScript 部分负责动态生成卡片和实现交互效果。

3.1 动态生成卡片

使用 document.write 方法动态生成卡片，第一张卡片有特殊处理，其他卡片则链接到不同的页面。

```
const images = [
    'biological-classification-sunburst-2D',
    'biological-evolution-force-2D',
    // 其他图像
];

images.forEach((img, index) => {
    if (index === 0) {
        // 第一张卡片的特殊处理
        document.write(`
            <div class="card-wrapper">
                <div class="card" onclick="showPCOnlyMessage()">
                    <div class="card-face card-front">
                        <img src="./select-cards/${img}.png" alt="${img}
">
                    </div>
                    <div class="card-face card-back">
                        <span>点击查看详情</span>
                    </div>
                </div>
            </div>
        `);
    } else {
        // 其他卡片保持不变
        document.write(`
            <div class="card-wrapper">
                <a href="./mobile-${img}.html">
                    <div class="card">
                        <div class="card-face card-front">
                            <img src="./select-cards/${img}.png" alt="
${img}">
                        </div>
                        <div class="card-face card-back">
                            <span>点击查看详情</span>
                        </div>
                    </div>
                </a>
            </div>
        `);
    }
});
```

3.2 3D 效果实现

通过监听鼠标移动事件，计算鼠标相对于卡片中心的坐标，动态调整卡片的旋转角度。

```
document.querySelectorAll('.card-wrapper').forEach(card => {
    card.addEventListener('mousemove', (e) => {
        const rect = card.getBoundingClientRect();
        const x = e.clientX - rect.left;
        const y = e.clientY - rect.top;

        const centerX = rect.width / 2;
        const centerY = rect.height / 2;

        const rotateX = (y - centerY) / 10;
        const rotateY = -(x - centerX) / 10;

        card.style.transform = `perspective(1000px) rotateX(${rotateX}d
eg) rotateY(${rotateY}deg)`;
    });

    card.addEventListener('mouseleave', () => {
        card.style.transform = 'perspective(1000px) rotateX(0) rotateY
(0)';
    });
});
```

4. 消息弹窗实现

当用户点击特定卡片时，创建一个消息弹窗，提示用户该页面仅在 PC 端可用。

```
function showPCOnlyMessage() {
    const messageDiv = document.createElement('div');
    messageDiv.className = 'message-popup';
    messageDiv.innerHTML = '在 pc 端才能访问<br>生物分类-太阳爆发图-2D 页面<
br>请尝试其他页面~';
    document.body.appendChild(messageDiv);

    messageDiv.style.display = 'block';

    setTimeout(() => {
        messageDiv.style.display = 'none';
        document.body.removeChild(messageDiv);
    }, 2500);
}
```

四、算法描述

本实验的核心算法主要集中在卡片的动态生成和 3D 效果的实现。通过监听鼠标事件，实时计算卡片的旋转角度，提供了良好的用户交互体验。

五、库函数引用

在本实验中，虽然没有直接使用 D3.js 和 Three.js 库，但可以考虑在未来的扩展中引入这些库来增强数据可视化和 3D 效果。例如，D3.js 可以用于处理复杂的数据集，而 Three.js 可以用于创建更复杂的 3D 场景。

六、结论

本实验成功实现了一个具有交互性和视觉吸引力的旋转卡片展示网页。通过使用现代 Web 技术，提升了用户体验。未来可以考虑引入更多的功能和库，以进一步增强网页的功能性和美观性。

mobile-index-earth.html.md

实验报告：地球古地理可视化项目

一、项目概述

本项目旨在通过 Web 技术实现一个交互式的地球古地理可视化工具，用户可以通过该工具查看不同历史时期的地球状态。项目使用了 Three.js 库进行 3D 图形渲染，并结合 HTML 和 JavaScript 实现用户交互功能。该工具不仅展示了地球的历史演变，还提供了丰富的用户交互体验。

二、创新功能点

1. **多时间点切换**：用户可以通过下拉菜单选择不同的历史时期，系统会自动加载对应的地球纹理。
2. **自动旋转与手动控制**：用户可以选择让地球模型自动旋转，或通过鼠标和键盘手动控制视角。
3. **云层显示/隐藏功能**：用户可以通过按钮控制云层的显示与隐藏，增强视觉效果。
4. **全屏显示功能**：用户可以选择全屏模式，提供更沉浸的体验。
5. **触摸控制**：针对移动端用户，提供触摸控制功能，支持单指旋转和双指缩放。

三、代码实现细节

1. HTML 结构

HTML 部分主要负责页面的基本结构和用户界面元素的布局。以下是关键部分的代码示例：

```
<div id="info">
    地球古地理-大陆漂移
    <div style="color: red; font-size: 14px; margin-top: 5px;">若移动端加载缓慢/失败,请转到 pc 端进行操作</div>
</div>
<button id="rotateButton">停止自动旋转</button>
<button id="cloudsButton">显示云层</button>
<select id="timeSelect" class="timeSelect">
    <option value="0">现代地球(第四纪-全新世)</option>
    <option value="20">2 千万年前(渐新世结束-新近纪开始-中新世开始)</option>
    <!-- 省略其他选项 -->
</select>
```

2. JavaScript 实现

JavaScript 部分是项目的核心，负责 3D 场景的创建、用户交互的处理以及纹理的加载。

2.1 初始化场景

使用 Three.js 库创建 3D 场景、相机和渲染器：

```
let scene, camera, renderer;

function init() {
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);
    renderer = new THREE.WebGLRenderer({ antialias: false, powerPreference: "high-performance" });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);
}
```

2.2 纹理加载

通过 THREE.TextureLoader 加载不同历史时期的地球纹理：

```
const textureLoader = new THREE.TextureLoader();
const textures = {};
const timePoints = [0, 20, 35, 50, 66, 90, 105, 120, 170, 200, 220, 240,
```

```
 260, 280, 300, 340, 370, 400, 430, 450, 470, 540, 600, 750];

timePoints.forEach(time => {
    const fileName = time === 0 ? '0.jpg' : `${time}.jpg`;
    textureLoader.load(`./static-other/World_Texture/${fileName}`, func
tion (texture) {
        textures[time] = texture;
        // 处理纹理加载完成后的逻辑
    });
});
```

2.3 用户交互

通过事件监听实现用户交互功能，例如切换时间点、控制旋转等：

```
document.getElementById('timeSelect').addEventListener('change', functi
on () {
    const time = parseInt(this.value);
    if (earth && textures[time]) {
        earth.material.map = textures[time];
        earth.material.needsUpdate = true;
    }
});

const rotateButton = document.getElementById('rotateButton');
rotateButton.addEventListener('click', function () {
    controls.autoRotate = !controls.autoRotate;
    this.textContent = controls.autoRotate ? '停止自动旋转' : '开始自动旋
转';
});
```

3. 使用的库函数

- **Three.js**：用于 3D 图形渲染，提供了丰富的几何体、材质和光源等功能。
- **D3.js**：虽然本项目主要使用 Three.js，但 D3.js 可以用于数据可视化和动态数据绑定，未来可以考虑将其集成以增强数据展示能力。

四、算法描述

本项目的核心算法主要包括：

1. **纹理加载算法**：使用异步加载的方式，确保在所有纹理加载完成后再进行地球模型的渲染。
2. **时间切换算法**：通过索引管理当前时间点，用户选择时间后，更新地球的纹理。
3. **用户输入处理算法**：通过键盘和鼠标事件监听，实时响应用户的输入，更新视角或切换时间。

五、总结

本项目通过结合 Three.js 和 HTML/JavaScript 实现了一个交互式的地球古地理可视化工具，用户可以方便地查看不同历史时期的地球状态。项目的创新功能和良好的用户体验使其在教育和科研领域具有广泛的应用前景。未来可以考虑进一步优化性能，增加更多的交互功能和数据展示能力。

mobile-index.html.md

实验报告：生物演化史导航页-移动端

一、引言

本实验旨在开发一个生物演化史的移动端导航页面，利用现代 Web 技术（HTML、CSS、JavaScript）和 3D 图形库（Three.js）实现一个交互式的用户体验。该页面不仅展示了生物演化的相关信息，还通过 3D 场景和动态效果增强了用户的沉浸感。

二、创新功能点

1. **3D 场景展示**：使用 Three.js 库创建一个动态的 3D 场景，用户可以通过触摸或鼠标操作来旋转视角，增强了交互性。
2. **移动端优化**：针对移动设备进行了特别优化，提供了适合触摸操作的按钮和提示信息。
3. **动态加载天空盒**：用户可以选择不同的场景背景，使用 equirectangular 和 cubemap 类型的纹理，提升了视觉效果。
4. **实时反馈**：在加载资源时提供加载提示，确保用户体验流畅。
5. **设备检测与提示**：在移动端访问时，提供了设备检测功能，提示用户使用 PC 端以获得最佳体验。

三、代码实现细节

3.1 HTML 结构

HTML 部分主要负责页面的基本结构和元素的布局。以下是关键部分的代码示例：

```
<div id="message" class="message">
    请使用 PC 端(电脑)访问以获得最佳体验
</div>
```

该部分用于提示用户在移动设备上访问时的体验限制。

3.2 CSS 样式

CSS 部分负责页面的样式设计，包括按钮、提示信息和 3D 场景的样式。以下是部分样式的实现：

```
.message {
    position: fixed;
    top: 5%;
    left: 50%;
    transform: translate(-50%, -50%);
    color: red;
    font-size: 1em;
    text-align: center;
    padding: 20px;
    z-index: 1000;
    white-space: nowrap;
    letter-spacing: 0.2em;
}
```

3.3 JavaScript 实现

JavaScript 部分是页面的核心，负责逻辑处理和 3D 场景的创建。以下是关键功能的实现细节：

3.3.1 3D 场景的创建

使用 Three.js 库创建 3D 场景的代码如下：

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth / wind
ow.innerHeight, 0.1, 1000);
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

3.3.2 加载天空盒

天空盒的加载使用了不同类型的纹理，以下是加载函数的实现：

```
function loadSkyboxTexture(textureConfig) {
    loadingDiv.style.display = 'block';
    const loadCallback = function() {
        loadingDiv.style.display = 'none';
        // 移除任何现有的提示
        const existingHint = document.querySelector('.vr-hint');
        if (existingHint) {
            existingHint.remove();
        }
        // 根据设备类型显示不同的提示
        const vrHint = document.createElement('div');
```

```
        vrHint.className = 'vr-hint';
        vrHint.textContent = isMobileDevice() ? '左右滑动体验 VR 全景' :
'点击并拖动体验 VR 全景';
        document.body.appendChild(vrHint);
    };

    if (textureConfig.type === 'cubemap') {
        const cubeTextureLoader = new THREE.CubeTextureLoader();
        cubeTextureLoader.load(textureConfig.urls, function(cubeTexture)
 {
            scene.background = cubeTexture;
            loadCallback();
        }, undefined, function(err) {
            loadingDiv.innerHTML = '天空盒资源加载失败，请刷新页面重试';
            console.error('天空加载错误:', err);
        });
    } else {
        const textureLoader = new THREE.TextureLoader();
        textureLoader.load(textureConfig.url, function(texture) {
            texture.mapping = THREE.EquirectangularReflectionMapping;
            scene.background = texture;
            loadCallback();
        }, undefined, function(err) {
            loadingDiv.innerHTML = '天空盒资源加载失败，请刷新页面重试';
            console.error('天空盒加载错误:', err);
        });
    }
}
```

3.3.3 交互功能

通过触摸和鼠标事件实现用户交互，以下是触摸事件的处理代码：

```
document.addEventListener('touchstart', onTouchStart, false);
document.addEventListener('touchmove', onTouchMove, false);

function onTouchStart(event) {
    touchStartX = event.touches[0].clientX;
    touchStartY = event.touches[0].clientY;
}

function onTouchMove(event) {
    event.preventDefault();
    const touchX = event.touches[0].clientX;
    const touchY = event.touches[0].clientY;
    const deltaX = (touchX - touchStartX) * 0.01;
    const deltaY = -(touchY - touchStartY) * 0.01;
    currentRotationY += deltaX;
    currentRotationX += deltaY;
```

```
    // 限制旋转角度
    currentRotationX = Math.max(-Math.PI / 2, Math.min(Math.PI / 2, cur
rentRotationX));
    camera.position.x = 5 * Math.sin(currentRotationY) * Math.cos(curre
ntRotationX);
    camera.position.y = 5 * Math.sin(currentRotationX);
    camera.position.z = 5 * Math.cos(currentRotationY) * Math.cos(curre
ntRotationX);
    camera.lookAt(scene.position);
    touchStartX = touchX;
    touchStartY = touchY;
}
```

3.4 库函数引用

- **Three.js**：用于创建和渲染 3D 场景，提供了丰富的 3D 图形处理功能。
- **D3.js**：虽然在本项目中未直接使用，但可以用于数据可视化和动态交互效果的实现。

四、总结

本实验成功实现了一个生物演化史的移动端导航页面，通过 3D 场景和动态效果提升了用户体验。未来可以考虑进一步优化性能，增加更多的交互功能和场景选择，以丰富用户的探索体验。

mobile-paleo-geography-3D.html.md

实验报告：地球古地理三维可视化系统

一、引言

本实验旨在开发一个基于 Web 的三维地球古地理可视化系统，利用 Three.js 库实现地球的三维模型展示，并通过 D3.js 库实现数据的动态交互。该系统不仅展示了地球在不同历史时期的地理变化，还提供了用户友好的交互功能，使用户能够直观地了解地球的演变过程。

二、创新功能点

1. **动态时间选择**：用户可以通过下拉菜单选择不同的地质时期，系统会自动更新地球的纹理和相关信息。
2. **自动旋转与手动控制**：用户可以选择自动旋转地球，或通过鼠标拖拽手动控制视角，增强了交互性。
3. **云层显示**：用户可以选择显示或隐藏地球表面的云层，增加了视觉效果的丰富性。

4. **加载提示**：在地球模型加载过程中，系统会显示加载提示，提升用户体验。
5. **全屏显示功能**：用户可以选择全屏模式，提供更沉浸式的体验。

三、算法描述

3.1 数据结构

- **时间点数组**：定义了不同地质时期的时间点，便于在用户选择时进行切换。
- **纹理映射对象**：使用对象存储不同时间点对应的地球纹理，便于快速访问和更新。

3.2 事件处理

- **键盘事件**：通过监听键盘的方向键，用户可以快速切换地质时期。
- **鼠标事件**：支持鼠标拖拽和双击操作，增强用户的交互体验。

四、代码实现

4.1 HTML 结构

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>地球古地理</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <style>
        /* 样式定义 */
    </style>
</head>
<body>
    <script>
        // JavaScript 代码
    </script>
</body>
</html>
```

4.2 CSS 样式

```
body {
    margin: 0;
}

canvas {
    display: block;
}
```

```css
#info {
    position: absolute;
    top: 10px;
    left: 10px;
    color: white;
    font-family: Arial;
    background: rgba(0, 0, 0, 0.7);
    padding: 10px;
    border-radius: 5px;
}

/* 其他样式定义 */
```

4.3 JavaScript 实现

4.3.1 初始化 Three.js 场景

```javascript
let scene, camera, renderer, earth, controls;

function init() {
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.
innerHeight, 0.1, 1000);
    renderer = new THREE.WebGLRenderer();
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    // 创建地球
    const geometry = new THREE.SphereGeometry(5, 32, 32);
    const textureLoader = new THREE.TextureLoader();
    const textures = {};

    // 加载纹理
    loadTextures(textureLoader, textures);

    // 添加光源
    const ambientLight = new THREE.AmbientLight(0x404040, 0.5);
    scene.add(ambientLight);

    const directionalLight = new THREE.DirectionalLight(0xffffff, 0.7);
    camera.add(directionalLight);
    scene.add(camera);

    camera.position.z = 15;

    // 添加控制器
    controls = new THREE.OrbitControls(camera, renderer.domElement);
    controls.enableDamping = true;
    controls.dampingFactor = 0.05;
    controls.autoRotate = true;
```

```
    animate();
}
```

4.3.2 纹理加载

```
function loadTextures(loader, textures) {
    const timePoints = [0, 20, 35, 50, 66, 90, 105, 120, 170, 200, 220,
 240, 260, 280, 300, 340, 370, 400, 430, 450, 470, 540, 600, 750];
    timePoints.forEach(time => {
        const fileName = time === 0 ? '0.jpg' : `${time}.jpg`;
        loader.load(`./static-other/World_Texture/${fileName}`, functio
n (texture) {
            textures[time] = texture;
            if (time === 0) {
                const material = new THREE.MeshPhongMaterial({ map: tex
ture });
                earth = new THREE.Mesh(geometry, material);
                scene.add(earth);
            }
        });
    });
}
```

4.3.3 事件监听

```
document.getElementById('timeSelect').addEventListener('change', functi
on () {
    const time = parseInt(this.value);
    if (earth && textures[time]) {
        earth.material.map = textures[time];
        earth.material.needsUpdate = true;
    }
});
```

4.4 D3.js 的使用

在本项目中，D3.js 主要用于处理数据的动态交互和可视化。通过 D3.js，我们可以轻松地绑定数据到 DOM 元素，并实现数据驱动的更新。

```
d3.select("#timeSelect")
    .selectAll("option")
    .data(timePoints)
    .enter()
    .append("option")
    .attr("value", d => d)
    .text(d => `${d}年`);
```

五、总结

本实验成功实现了一个基于 Three.js 和 D3.js 的地球古地理三维可视化系统。通过动态交互和丰富的视觉效果，用户能够直观地了解地球的演变过程。未来的工作可以集中在进一步优化性能和增加更多的交互功能，以提升用户体验。

mobile-radial-tidy-tree.html.md

实验报告：生物演化史放射状树形图可视化

1. 引言

本实验旨在创建一个交互式的生物演化史放射状树形图可视化系统。该系统利用 D3.js 库实现了复杂的数据可视化，展示了生物演化的历史进程，并提供了多种交互功能，使用户能够深入探索生物演化的细节。

2. 创新功能点

2.1 放射状树形图布局

采用 D3.js 的树形图布局算法，将生物演化数据以放射状的形式展现，直观地显示了物种间的演化关系。

2.2 时间轴过滤

实现了基于时间的数据过滤功能，用户可以通过滑动条来选择特定的时间段，动态更新显示的演化节点。

2.3 搜索和高亮功能

提供了搜索功能，允许用户快速定位特定的物种或时期，并高亮显示从根节点到目标节点的路径。

2.4 详细信息面板

点击节点后，会显示包含图片、名称、时间和描述的详细信息面板。

2.5 图片放大功能

支持点击图片进行放大查看，提供更清晰的视觉体验。

2.6 旋转控制

通过按钮或键盘操作，用户可以旋转整个树形图，以不同角度查看数据。

2.7 响应式设计

适配不同屏幕尺寸，确保在移动设备上也能良好展示。

3. 技术实现

3.1 数据结构

使用层次化的 JSON 结构存储生物演化数据，每个节点包含名称、时间、描述和图片 URL 等信息。

3.2 D3.js 布局算法

使用 D3.js 的树形图布局算法来构建放射状树形图：

```
const tree = d3.tree()
    .size([2 * Math.PI, radius])
    .separation((a, b) => {
        return (a.parent == b.parent ? 2 : 3) / a.depth;
    });
```

此代码创建了一个 360 度的放射状布局，节点间的分离度根据深度动态调整。

3.3 SVG 绘制

使用 D3.js 操作 SVG 元素来绘制树形图：

```
const svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height)
    .attr("viewBox", [-width / 2, -height / 2, width, height]);
```

3.4 缩放和平移

实现了缩放和平移功能，使用 D3.js 的 zoom 行为：

```
const zoom = d3.zoom()
    .scaleExtent([0.2, 5])
    .on("zoom", (event) => {
        g.attr("transform", `${event.transform} rotate(${currentRotatio
n})`);
    });
```

3.5 时间过滤算法

实现了基于时间的节点过滤算法：

```
function updateView(threshold) {
    const visibleNodes = new Set();
    root.descendants().forEach(d => {
```

```
            const time = parseTime(d.data.time);
            if (time === 0 || time <= threshold) {
                visibleNodes.add(d.id);
                let ancestor = d.parent;
                while (ancestor) {
                    visibleNodes.add(ancestor.id);
                    ancestor = ancestor.parent;
                }
            }
        });
        // 更新节点和连接线的显示
        // ...
}
```

3.6 搜索和路径高亮

实现了搜索功能和路径高亮显示：

```
function highlightPath(node) {
    g.selectAll('.link').classed('highlight-path', false);
    const path = [];
    let current = node;
    while (current.parent) {
        path.push([current.parent, current]);
        current = current.parent;
    }
    g.selectAll('.link')
        .classed('highlight-path', d => {
            return path.some(p =>
                (p[0].id === d.source.id && p[1].id === d.target.id)
            );
        });
    // 显示详细信息面板
    // ...
}
```

3.7 旋转控制

实现了通过按钮和键盘控制的旋转功能：

```
function startRotation(direction) {
    clearInterval(rotationInterval);
    rotationInterval = setInterval(() => {
        currentRotation += direction * rotationStep;
        g.attr("transform", `${d3.zoomTransform(svg.node())} rotate(${c
urrentRotation})`);
    }, rotationDelay);
}
```

4. 用户界面设计

4.1 控制面板

设计了包含各种功能按钮的控制面板，如切换不同可视化模式、旋转控制等。

4.2 时间滑动条

实现了底部的时间滑动条，允许用户选择特定的时间范围。

4.3 搜索框

顶部添加了搜索框，支持实时搜索和结果显示。

4.4 详细信息面板

设计了右侧弹出的详细信息面板，展示节点的详细信息和图片。

5. 性能优化

5.1 节点过滤

通过时间过滤算法，减少了需要渲染的节点数量，提高了大数据量下的渲染性能。

5.2 事件委托

使用事件委托处理节点的点击事件，减少了事件监听器的数量。

5.3 防抖和节流

在搜索和滑动条操作中应用了防抖技术，避免过于频繁的更新。

6. 跨平台兼容性

6.1 响应式设计

使用相对单位和媒体查询，确保在不同尺寸的设备上都能正常显示。

6.2 触摸事件支持

添加了对触摸事件的支持，使移动设备用户也能进行旋转操作。

7. 未来改进方向

7.1 数据加载优化

考虑实现懒加载或分片加载，以支持更大规模的数据集。

7.2 3D 渲染

考虑使用 WebGL 或 Three.js 实现 3D 版本的演化树，提供更丰富的视觉体验。

7.3 协作功能

添加用户注释和分享功能，促进科研协作。

8. 结论

本实验成功实现了一个功能丰富、交互性强的生物演化史可视化系统。通过 D3.js 的强大功能，我们创建了一个直观、易用的放射状树形图，使用户能够深入了解生物演化的复杂过程。该系统不仅在桌面环境下表现优异，在移动设备上也能提供良好的用户体验。未来，我们将继续优化性能，扩展功能，以满足更广泛的科研和教育需求。

paleo-geography-3D.html.md

实验报告：地球古地理动态可视化系统

1. 引言

本实验旨在创建一个互动式地球古地理动态可视化系统，利用 Three.js 库实现了对地球在历史不同时期的地理变化进行模拟。系统通过 3D 可视化的方式，展示了地球大陆的漂移过程，并在用户交互上提供了多项便于操作的功能。

2. 创新功能点

2.1 3D 地球展示

使用 Three.js 创建了一个 3D 地球模型，展示了地球在不同地质时期的地理变化。用户可以通过交互来观察地球的不同角度。

2.2 时光倒流功能

通过选择不同的时间节点，用户可以观察地球在从现代到 7.5 亿年前不同时期的地理变化。

2.3 自动旋转与手动控制

地球模型支持自动旋转，并允许用户通过鼠标和键盘进行手动控制，提供灵活的交互方式。

2.4 云层显示

地球表面可以显示和隐藏云层，增加了可视化的真实感。

2.5 星空背景

模拟了星空背景，使整个场景更加生动。

2.6 全屏显示

支持全屏模式以便用户获得更好的查看体验。

3. 技术实现

3.1 数据结构

地球地理变化的数据以图片的形式存储，每张图片对应一个特定的时间点。例如，`0.jpg` 表示现代地球，`750.jpg` 表示 7.5 亿年前的地球。

3.2 Three.js 库介绍

Three.js 是一个跨平台的 JavaScript 库，用于在 Web 浏览器中创建和显示动画 3D 计算机图形。它使用 WebGL 构建，因此可以在不使用插件的情况下在现代浏览器上运行。

3.2.1 地球模型创建

```
const geometry = new THREE.SphereGeometry(5, 32, 32);
const material = new THREE.MeshPhongMaterial({
    map: texture,
    specular: new THREE.Color('grey'),
    shininess: 10
});
earth = new THREE.Mesh(geometry, material);
scene.add(earth);
```

- `SphereGeometry`: 创建球体几何。
- `MeshPhongMaterial`: 用于创建具有光泽和反射的材质。
- `Mesh`: 将几何和材质组合成网格。

3.2.2 光照设置

```
const ambientLight = new THREE.AmbientLight(0x404040, 0.5);
scene.add(ambientLight);

const directionalLight = new THREE.DirectionalLight(0xffffff, 0.7);
directionalLight.position.set(0, 0, 1);
scene.add(camera);
```

- `AmbientLight`: 环境光，用于整体照亮场景。

- `DirectionalLight`: 方向光，用于模拟太阳光。

3.2.3 控制器

使用 `OrbitControls` 实现对地球的旋转和缩放控制：

```
controls = new THREE.OrbitControls(camera, renderer.domElement);
controls.enableDamping = true;
controls.dampingFactor = 0.05;
controls.autoRotate = true;
controls.autoRotateSpeed = 0.5;
```

`OrbitControls` 允许用户通过鼠标交互来旋转、缩放和移动 3D 模型。

3.3 纹理映射

不同时期的地球纹理使用 `TextureLoader` 加载，并根据用户选择的时间节点进行切换。

3.4 云层实现

```
const cloudsGeometry = new THREE.SphereGeometry(5.05, 32, 32);
const cloudsMaterial = new THREE.MeshPhongMaterial({
    map: texture,
    transparent: true,
    opacity: 1,
    side: THREE.DoubleSide
});
clouds = new THREE.Mesh(cloudsGeometry, cloudsMaterial);
clouds.visible = false;
scene.add(clouds);
```

云层通过半透明的双面材质实现，可以通过按钮控制其显示和隐藏。

3.5 星空背景

通过 `THREE.Points` 创建星空背景，模拟宇宙环境：

```
const starsGeometry = new THREE.BufferGeometry();
const starsMaterial = new THREE.PointsMaterial({
    color: 0xFFFFFF,
    size: 0.1
});
```

`BufferGeometry` 和 `PointsMaterial` 结合，创建出细小的星星效果。

3.6 交互功能

系统提供了键盘和鼠标的交互功能，通过事件监听器实现用户对地球的操控。

3.6.1 键盘交互

可以使用键盘的左右方向键或 A/D 键来切换时间节点。

3.6.2 鼠标交互

鼠标左键和右键可用于快速切换时间节点。

3.7 全屏功能

通过调用浏览器的全屏 API 实现全屏查看功能：

```
document.documentElement.requestFullscreen();
```

4. 用户界面设计

4.1 控制面板

包括全屏按钮、时间选择下拉菜单、云层控制按钮等，用户可以通过这些控件操作地球模型。

4.2 信息展示

在界面上显示当前时间节点的信息，并提供相关的地质时期描述。

5. 性能优化

5.1 懒加载纹理

只有在需要时才加载特定时间节点的纹理，以减少初始加载时间。

5.2 控制器阻尼

使用控制器的阻尼功能，使交互更加自然流畅。

6. 跨平台兼容性

6.1 响应式设计

使用 CSS 确保界面在不同设备和屏幕尺寸上都能正常显示。

6.2 事件支持

支持鼠标和触摸事件，以便在桌面和移动设备上都可以正常操作。

7. 未来改进方向

7.1 数据扩展

增加更多的时间节点和更详细的地质数据，以覆盖更广泛的地质历史。

7.2 3D 渲染优化

利用 WebGL 的高级特性，提升渲染质量和性能。

7.3 用户自定义功能

允许用户上传自定义的地质数据和纹理，以便进行个性化分析和展示。

8. 结论

本实验成功创建了一个集成多项先进功能的地球古地理动态可视化系统，利用 Three.js 实现了 3D 可视化和丰富的用户交互体验。未来，将继续在数据丰富性和系统性能上进行改进，以满足更广泛的科学研究和教育需求。

pc-index.html.md

实验报告：生物演化史导航页-PC 端

一、引言

本实验旨在开发一个生物演化史的导航页面，利用 Web 技术和 3D 图形库 Three.js，提供用户友好的交互体验。该页面不仅展示了生物演化的历史，还通过 3D 场景和动态效果增强了用户的沉浸感。

二、创新功能点

1. **3D 场景展示**：使用 Three.js 创建动态的 3D 场景，用户可以通过鼠标或触摸操作自由旋转视角，增强了交互性。
2. **天空盒选择**：用户可以选择不同的天空盒背景，提供多样化的视觉体验。
3. **设备检测**：自动检测用户设备类型，提供适配的操作提示，确保用户在不同设备上都能获得良好的体验。
4. **动态加载提示**：在加载 3D 资源时，显示加载动画，提升用户体验。
5. **按钮交互**：提供多个功能按钮，如跳转到首页、评论区等，方便用户操作。

三、代码实现细节

3.1 HTML 结构

HTML 文件的基本结构如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.
0">
    <title>生物演化史导航页-PC 端</title>
    <link rel="icon" href="./static-other/icon/favicon.ico" type="image
/x-icon">
    <style>
        /* CSS 样式定义 */
    </style>
</head>
<body>
    <div id="message" class="message no-select">生物演化史 - EvoViz</di
v>
    <div id="device-check" style="position: fixed; bottom: 10px; left:
10px; color: white; font-size: 1em; z-index: 1000;">
        页面格式有问题？点击重新进行设备检测→
        <button onclick="window.location.href='./device-testing.html'"
style="margin-left: 5px; padding: 5px 10px; background-color: rgba(0, 1
23, 255, 0.9); color: white; border: none; border-radius: 5px; cursor:
pointer;">设备检测</button>
    </div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/t
hree.min.js"></script>
    <script>
        // JavaScript 代码实现
    </script>
</body>
</html>
```

3.2 CSS 样式

CSS 样式用于美化页面，主要包括按钮、加载提示、消息框等的样式定义。以下是部分样式示例：

```
.message {
    position: fixed;
    top: 5%;
    left: 50%;
    transform: translate(-50%, -50%);
    color: red;
    font-size: 3em;
    text-align: center;
    padding: 20px;
    z-index: 1000;
    white-space: nowrap;
    letter-spacing: 0.2em;
}

.loading {
    position: fixed;
```

```
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    color: white;
    font-size: 1.2em;
    text-align: center;
    z-index: 1000;
}
```

3.3 JavaScript 实现

JavaScript 部分是实现页面交互和 3D 场景的核心。以下是主要功能的实现细节：

3.3.1 设备检测

通过 navigator.userAgent 检测用户设备类型：

```
function isMobileDevice() {
    return /Android|webOS|iPhone|iPad|iPod|BlackBerry|IEMobile|Opera Mi
ni/i.test(navigator.userAgent);
}
```

3.3.2 创建 3D 场景

使用 Three.js 创建 3D 场景、相机和渲染器：

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth / wind
ow.innerHeight, 0.1, 1000);
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

3.3.3 加载天空盒

天空盒的加载通过 THREE.CubeTextureLoader 和 THREE.TextureLoader 实现，支持多种类型的纹理：

```
function loadSkyboxTexture(textureConfig) {
    loadingDiv.style.display = 'block';
    const loadCallback = function() {
        loadingDiv.style.display = 'none';
        // 显示提示信息
    };

    if (textureConfig.type === 'cubemap') {
        const cubeTextureLoader = new THREE.CubeTextureLoader();
        cubeTextureLoader.load(textureConfig.urls, function(cubeTexture)
 {
            scene.background = cubeTexture;
            loadCallback();
```

```
        });
    } else {
        const textureLoader = new THREE.TextureLoader();
        textureLoader.load(textureConfig.url, function(texture) {
            texture.mapping = THREE.EquirectangularReflectionMapping;
            scene.background = texture;
            loadCallback();
        });
    }
}
```

3.3.4 交互功能

通过事件监听器实现用户交互，如鼠标和触摸控制：

```
document.addEventListener('mousedown', onMouseDown, false);
document.addEventListener('mousemove', onMouseMove, false);
document.addEventListener('mouseup', onMouseUp, false);

function onMouseDown(event) {
    isMouseDown = true;
    mouseStartX = event.clientX;
    mouseStartY = event.clientY;
}

function onMouseMove(event) {
    if (!isMouseDown) return;
    // 计算旋转角度
}
```

3.4 库函数引用

- **Three.js**：用于创建和渲染 3D 场景，提供了丰富的 3D 图形功能。
- **D3.js**：虽然在本项目中未直接使用，但可以用于数据可视化，未来可以考虑将生物演化数据以图表形式展示。

四、总结

本实验成功实现了一个生物演化史的导航页面，利用 Three.js 提供的 3D 图形能力，结合用户交互设计，提升了用户体验。未来可以进一步扩展功能，如增加更多的生物演化数据展示、优化加载性能等。通过不断迭代和优化，期望为用户提供更丰富的学习体验。

radial-tidy-tree.html.md

生物演化史放射状树形图实验报告

1. 引言

本实验旨在创建一个交互式的生物演化史放射状树形图，利用 D3.js 库实现数据可视化。该项目不仅展示了生物演化的历程，还提供了丰富的交互功能，使用户能够深入探索生物演化的细节。

2. 创新功能点

2.1 放射状树形图布局

采用 D3.js 的树形布局算法，创建了一个放射状的树形图，直观地展示了生物演化的层次结构。

2.2 时间轴过滤

实现了一个时间轴滑块，允许用户根据时间筛选显示的生物节点，动态展示不同时期的生物演化状况。

2.3 搜索功能

集成了实时搜索功能，用户可以快速定位特定的生物节点。

2.4 路径高亮

在搜索结果中点击节点时，会高亮显示从根节点到该节点的完整路径。

2.5 详细信息面板

点击节点后，会显示包含图片、名称、时间和描述的详细信息面板。

2.6 图片放大功能

支持点击图片查看大图，增强了用户体验。

2.7 旋转控制

通过键盘控制（'Q'和'E'键），用户可以旋转整个树形图，以不同角度查看数据。

2.8 全屏显示

提供全屏查看选项，让用户获得更沉浸式的体验。

3. 技术实现

3.1 D3.js 库的使用

3.1.1 树形布局创建

```
const tree = d3.tree()
    .size([2 * Math.PI, radius])
    .separation((a, b) => {
        return (a.parent == b.parent ? 2 : 3) / a.depth;
    });
```

这段代码创建了一个放射状的树形布局。size([2 * Math.PI, radius])设置了布局的大小，使其形成一个完整的圆。separation 函数定义了节点之间的间隔。

3.1.2 数据绑定与元素创建

```
const node = g.selectAll(".node")
    .data(treeData2.descendants())
    .join("g")
    .attr("class", "node")
    .attr("transform", d => `
        translate(${d3.pointRadial(d.x, d.y)})
    `);
```

这段代码将数据绑定到 DOM 元素，创建节点组，并设置它们的位置。
d3.pointRadial 函数用于计算极坐标系中的点位置。

3.1.3 缩放与平移

```
const zoom = d3.zoom()
    .scaleExtent([0.2, 5])
    .on("zoom", (event) => {
        g.attr("transform", `${event.transform} rotate(${currentRotatio
n})`);
    });
```

这段代码实现了缩放和平移功能。scaleExtent 设置了缩放的范围，on("zoom")定义了缩放时的行为。

3.2 时间轴过滤实现

```
function updateView(threshold) {
    const visibleNodes = new Set();

    root.descendants().forEach(d => {
        const time = parseTime(d.data.time);
        if (time === 0 || time <= threshold) {
            visibleNodes.add(d.id);
            let ancestor = d.parent;
            while (ancestor) {
                visibleNodes.add(ancestor.id);
                ancestor = ancestor.parent;
```

```
            }
        }
    });

    g.selectAll(".link")
        .style("display", d => {
            return (visibleNodes.has(d.source.id) && visibleNodes.has(d.
target.id)) ? "block" : "none";
        });

    g.selectAll(".node")
        .style("display", d => {
            return visibleNodes.has(d.id) ? "block" : "none";
        });
}
```

这个函数实现了基于时间阈值的节点过滤。它首先遍历所有节点，找出时间小于阈值的节点及其所有祖先节点，然后更新连接线和节点的显示状态。

3.3 搜索功能实现

```
function searchNodes(query) {
    const results = [];
    root.descendants().forEach(node => {
        const name = node.data.name || '';
        const description = node.data.description || '';
        const time = node.data.time || '';

        if (!query || name.toLowerCase().includes(query.toLowerCase())
||
            description.toLowerCase().includes(query.toLowerCase()) ||
            time.toLowerCase().includes(query.toLowerCase())) {
            results.push(node);
        }
    });

    displaySearchResults(results);
    searchResult.style.display = 'block';
}
```

这个函数实现了节点搜索功能。它遍历所有节点，检查节点的名称、描述和时间是否包含查询字符串，并返回匹配的结果。

3.4 路径高亮

```
function highlightPath(node) {
    g.selectAll('.link').classed('highlight-path', false);

    const path = [];
    let current = node;
    while (current.parent) {
```

```
        path.push([current.parent, current]);
        current = current.parent;
    }

    g.selectAll('.link')
        .classed('highlight-path', d => {
            return path.some(p =>
                (p[0].id === d.source.id && p[1].id === d.target.id)
            );
        });
}
```

这个函数实现了路径高亮功能。它首先清除之前的高亮，然后找出从根节点到目标节点的路径，最后将路径上的连接线添加高亮类。

3.5 旋转控制

```
document.addEventListener('keydown', (event) => {
    if (event.key === 'q') {
        currentRotation -= rotationStep;
        g.attr("transform", `${d3.zoomTransform(svg.node())} rotate(${c
urrentRotation})`);
    } else if (event.key === 'e') {
        currentRotation += rotationStep;
        g.attr("transform", `${d3.zoomTransform(svg.node())} rotate(${c
urrentRotation})`);
    }
});
```

这段代码实现了键盘控制旋转功能。当按下'Q'或'E'键时，会更新旋转角度并应用到 SVG 元素上。

4. 性能优化

4.1 虚拟 DOM

使用 D3.js 的 `join` 方法进行高效的 DOM 更新，减少不必要的 DOM 操作。

4.2 事件委托

利用事件冒泡，将事件监听器添加到父元素上，而不是每个子元素，减少了事件监听器的数量。

4.3 防抖

在搜索功能中可以考虑添加防抖，减少频繁的搜索操作，提高性能。

5. 用户界面设计

5.1 响应式布局

使用相对单位和弹性布局，确保在不同屏幕尺寸下都能正常显示。

5.2 交互式元素

添加悬停效果、点击反馈等，提升用户体验。

5.3 信息展示

使用工具提示和详情面板，在不影响整体布局的情况下展示更多信息。

6. 未来改进方向

6.1 数据加载优化

考虑实现数据的异步加载和分片加载，以支持更大规模的数据集。

6.2 交互增强

可以添加更多的交互方式，如拖拽重组、节点折叠等。

6.3 移动端适配

优化移动设备上的用户体验，如触摸控制、手势操作等。

6.4 数据分析功能

集成数据分析工具，如统计图表、时间线等，深入挖掘数据价值。

7. 结论

本实验成功创建了一个功能丰富、交互性强的生物演化史放射状树形图。通过 D3.js 库的强大功能，实现了复杂的数据可视化和交互效果。该项目不仅直观地展示了生物演化的过程，还提供了多种方式让用户探索和理解数据。未来，我们将继续优化性能，增强功能，以提供更好的用户体验和更深入的数据洞察。

z-new.html.md

生物演化史墙实验报告

1. 引言

本实验旨在通过 D3.js 库创建一个生物演化史墙，展示生物演化的结构及其细节。该项目采用了创新的懒加载图片技术，以提高页面的性能和用户体验。

2. 创新功能点

2.1 数据树形结构展示

利用递归的方式将树形数据结构展示成一个易于浏览的页面，每个节点代表一个生物演化的阶段或元素。

2.2 懒加载图片

使用 IntersectionObserver API 实现图片的懒加载，只有当图片进入视窗时才加载，以提升初始加载速度和页面性能。

2.3 简单而有效的布局

采用简单的 CSS 布局，每个节点以卡片形式展示，确保信息的清晰呈现。

3. 技术实现

3.1 D3.js 库的使用

3.1.1 数据绑定与 DOM 生成

```
function createWall(data, container) {
    const node = container.append("div")
        .attr("class", "node");
    node.append("h3").text(data.name);
    node.append("p").text(data.time);
    node.append("p").text(data.description);
    node.append("img")
        .attr("data-src", data.src)
        .attr("alt", data.name);

    if (data.children) {
        const childrenContainer = node.append("div").attr("class", "chi
ldren");
        data.children.forEach(child => {
            createWall(child, childrenContainer);
        });
    }
```

```
}

const wallContainer = d3.select("#evolution-wall");
createWall(treeData, wallContainer);
```

使用 D3.js 的选择器方法，递归地遍历树形数据，生成每个节点的 DOM 元素。每个节点包含名称、时间、描述和图片。

3.2 懒加载实现

3.2.1 IntersectionObserver API

```
const lazyLoadImages = document.querySelectorAll('img[data-src]');
const imageObserver = new IntersectionObserver((entries, observer) => {
    entries.forEach(entry => {
        if (entry.isIntersecting) {
            const img = entry.target;
            img.src = img.getAttribute('data-src');
            img.onload = () => img.style.opacity = 1;
            observer.unobserve(img);
        }
    });
});

lazyLoadImages.forEach(img => {
    imageObserver.observe(img);
});
```

使用 IntersectionObserver API 监听图片元素是否进入视窗。当图片进入视窗时，即加载图片，并通过设置 CSS 样式实现淡入效果。

3.3 CSS 布局

```
.node {
    border: 1px solid #ccc;
    padding: 10px;
    margin: 5px;
    display: inline-block;
    vertical-align: top;
}
.node img {
    max-width: 100px;
    display: block;
    margin: 0 auto;
    opacity: 0;
    transition: opacity 0.3s;
}
```

每个节点通过 CSS 样式定义外观，包括边框、内边距和图片的过渡效果。图片初始不透明度为 0，通过懒加载后渐变显示。

4. 性能优化

4.1 懒加载技术

通过 IntersectionObserver 实现图片懒加载，减少初始加载时的带宽消耗，并加快页面的初次渲染速度。

4.2 DOM 操作优化

使用 D3.js 的绑定和更新机制，确保 DOM 节点只创建一次，减少不必要的 DOM 操作。

5. 用户界面设计

5.1 简洁的卡片布局

使用卡片式设计展示每个节点的信息，确保在有限的空间中信息易于阅读。

5.2 渐进式图片加载

使用懒加载和淡入效果，使图片加载更具视觉上的平滑性，提高用户体验。

6. 未来改进方向

6.1 响应式设计

进一步优化页面布局，使其在不同屏幕尺寸下表现更佳。

6.2 动态数据加载

未来可以考虑从服务器端动态加载数据，以支持更大的数据集和更实时的应用场景。

6.3 可视化增强

集成更多的可视化效果，如动画和交互，以提高数据的可视性和用户的参与感。

7. 结论

本实验成功实现了一个简单而有效的生物演化史墙，通过 D3.js 库展示了复杂的树形结构数据，并通过懒加载技术优化了页面性能。该项目不仅提供了一种清晰直观的方式来展示生物演化的历程，还为未来的扩展和优化奠定了基础。未来，我们将继续探索更多的功能和优化方案，以提供更好的用户体验和数据价值展示。