Descripción General

Sistema Solar N-Body es una simulación física matemática del Sistema Solar que implementa gravedad newtoniana real, sin usar texturas precargadas. Toda la visualización es generada proceduralmente mediante shaders y matemáticas puras.

Características Principales

- ✅ **Física Gravitacional Real**: Implementación de la Ley de Gravitación Universal de Newton
- ✅ **Integrador Velocity Verlet**: Precisión numérica de segundo orden
- ✅ **Renderizado Procedural**: Planetas generados matemáticamente sin texturas
- ✅ **Datos Reales**: Masas, velocidades y distancias del Sistema Solar real
- ✅ **Tiempo Real**: Actualización vía WebSocket con arquitectura cliente-servidor
- ✅ **Visualización 3D**: Three.js con shaders GLSL personalizados

📐 Arquitectura del Sistema

Stack Tecnológico

Backend:

- Python 3.8+
- Flask (servidor web)
- Flask-SocketIO (comunicación en tiempo real)
- NumPy (cálculos numéricos)
- SciPy (integradores avanzados)

Frontend:

- Three.js (renderizado 3D WebGL)
- Socket.IO (cliente WebSocket)
- JavaScript ES6+

- HTML5 + CSS3

Flujo de Datos

text

```
1. Backend (Python)
   ├── Inicialización del sistema solar
   ├── Cálculo de fuerzas gravitacionales
   ├── Integración numérica (Verlet)
   └── Emisión de estados vía WebSocket
         ↓
2. WebSocket (Socket.IO)
   └── Transmisión JSON en tiempo real
         ↓
3. Frontend (JavaScript)
   ├── Recepción de estados
   ├── Creación de geometrías procedurales
   ├── Aplicación de shaders
   └── Renderizado WebGL (Three.js)
```

📁 **Estructura de Archivos**

text

```
solar-system-physics/
│
├── app.py                  # Servidor Flask principal
│
├── physics/                # Motor de física
│   ├── __init__.py
```

```
│   ├── constants.py          # Constantes físicas y datos del sistema solar
│   ├── nbody.py              # Simulador N-body con gravedad
│   └── integrator.py         # Métodos de integración numérica
│
├── visualization/            # Generación procedural
│   ├── __init__.py
│   ├── sphere_generator.py   # Generación de geometría esférica
│   └── shader_math.py        # Funciones matemáticas para shaders
│
├── static/
│   ├── js/
│   │   ├── main.js           # Cliente principal y WebSocket
│   │   ├── renderer.js       # Motor de renderizado Three.js
│   │   └── utils.js          # Funciones auxiliares
│   └── css/
│       └── style.css         # Estilos de la interfaz
│
├── templates/
│   └── index.html            # Interfaz web
│
├── requirements.txt          # Dependencias Python
└── README.md                 # Documentación
```

⚙️ Física Implementada

1. Ley de Gravitación Universal

Ecuación de Newton:

F=Gm1m2r2*F*=*Gr*2*m*1*m*2

Donde:

- F: Fuerza gravitacional (N)
- G: Constante gravitacional = 6.67430×10^{-11} m^3 kg^{-1} s^{-2}
- m_1, m_2: Masas de los cuerpos (kg)
- r: Distancia entre centros de masa (m)

Implementación en código:

python

```
def compute_gravitational_acceleration(self, body_index):
    body = self.bodies[body_index]
    acceleration = np.zeros(3, dtype=np.float64)

    for i, other_body in enumerate(self.bodies):
        if i == body_index:
            continue

        r_vec = other_body.position - body.position
        r_magnitude = np.linalg.norm(r_vec)
        r_hat = r_vec / r_magnitude

        # a = G * M / r^2
        a_magnitude = G * other_body.mass / (r_magnitude ** 2)
        acceleration += a_magnitude * r_hat

    return acceleration
```

2. Integrador Velocity Verlet

Ecuaciones del método:

r(t+Δt)=r(t)+v(t)Δt+12a(t)Δt2$r(t+\Delta t)=r(t)+v(t)\Delta t+21a(t)\Delta t2$

v(t+Δt)=v(t)+12[a(t)+a(t+Δt)]Δt$v(t+\Delta t)=v(t)+21[a(t)+a(t+\Delta t)]\Delta t$

Ventajas:

- **Simpléctico**: Conserva energía mejor que Euler
- **Segundo orden**: Precisión $O(\Delta t^2)$
- **Reversible en el tiempo**: Estabilidad numérica

Implementación:

python

```
def step_verlet(self):
    dt = self.time_step

    # 1. Calcular aceleraciones actuales
    accelerations = [self.compute_gravitational_acceleration(i)
                     for i in range(len(self.bodies))]

    # 2. Actualizar posiciones
    for i, body in enumerate(self.bodies):
        body.position += body.velocity * dt + 0.5 * accelerations[i] * dt**2

    # 3. Calcular nuevas aceleraciones
    new_accelerations = [self.compute_gravitational_acceleration(i)
                         for i in range(len(self.bodies))]

    # 4. Actualizar velocidades
```

```
    for i, body in enumerate(self.bodies):
        body.velocity += 0.5 * (accelerations[i] + new_accelerations[i]) * dt
```

3. Conservación de Energía

Energía total del sistema:

Etotal=Ecine´tica+Epotencial*Etotal=Ecine´tica+Epotencial*

Ek=∑i12mivi2*Ek*=∑*i*21*mivi*2

Ep=−∑i<jGmimjrij*Ep*=−∑*i*<*jrijGmimj*

En un sistema ideal, E_total debe permanecer constante.

🎨 **Renderizado Procedural**

1. Geometría de Esferas

Generación mediante IcosahedronGeometry:

javascript

```
const geometry = new THREE.IcosahedronGeometry(radius, subdivisions);
```

- **Icosaedro**: Poliedro de 20 caras triangulares
- **Subdivisiones**: Refinamiento recursivo para suavidad

2. Shaders GLSL

Vertex Shader:

text

```
varying vec3 vNormal;
varying vec3 vPosition;
varying vec3 vWorldPosition;

void main() {
    vNormal = normalize(normalMatrix * normal);
    vPosition = position;
```

```
  vec4 worldPosition = modelMatrix * vec4(position, 1.0);
  vWorldPosition = worldPosition.xyz;
  gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Fragment Shader:

```
text
// Ruido fractal para patrones procedurales
float fractalNoise(vec3 p) {
  float value = 0.0;
  float amplitude = 1.0;
  float frequency = 1.0;

  for(int i = 0; i < 5; i++) {
    value += amplitude * noise(p * frequency);
    amplitude *= 0.5;
    frequency *= 2.0;
  }
  return value;
}

// Gradiente en 3 ejes
float gradientMix = (gradientX + gradientY + gradientZ) / 3.0;
vec3 baseColor = mix(colorGradient[0], colorGradient[1], gradientMix);

// Iluminación difusa
float diffuse = max(0.0, dot(vNormal, lightDir));
```

vec3 finalColor = baseColor * pattern * (0.35 + diffuse * 0.65);

3. Anillos de Saturno

Geometría:

javascript

```
const geometry = new THREE.RingGeometry(innerRadius, outerRadius, 128, 8);
```

Rotación al plano XZ:

javascript

```
ringMesh.rotation.x = Math.PI / 2; // 90 grados
```

🔌 **Comunicación WebSocket**

Eventos del Backend (Flask-SocketIO)

Emisión de estados:

python

```
socketio.emit('simulation_update', {
  'state': {
    'time': self.time,
    'bodies': [body.to_dict() for body in self.bodies]
  },
  'energy': {
    'kinetic': kinetic,
    'potential': potential,
    'total': kinetic + potential
  },
  'fps': fps
}, namespace='/')
```

Eventos recibidos:

- start_simulation: Inicia el loop de simulación
- stop_simulation: Pausa la simulación
- reset_simulation: Reinicia desde t=0
- set_time_scale: Ajusta velocidad del tiempo

Cliente JavaScript

Conexión:

javascript

```
const socket = io({
  reconnection: true,
  reconnectionDelay: 1000,
  reconnectionAttempts: 5
});
```

Recepción de datos:

javascript

```
socket.on('simulation_update', (data) => {
  if (data.state && data.state.bodies) {
    renderer.updateBodies(data.state.bodies);
  }
  updateStats(data);
});
```

🎯 Características Clave

1. Estelas con Desvanecimiento

Algoritmo:

javascript

```
history.push({
```

```
    position: new THREE.Vector3(x, y, z),
    timestamp: currentTime
});

// Calcular alpha basado en edad
const age = currentTime - point.timestamp;
const ageAlpha = Math.max(0, 1 - (age / maxAge));
const positionAlpha = index / history.length;
const finalAlpha = ageAlpha * positionAlpha * 0.9;
```

Resultado: Las estelas se desvanecen gradualmente después de 30 segundos.

2. Control de Velocidad

Multiplicadores preestablecidos:

- 0.1x (muy lento)
- 0.5x (lento)
- 1x (tiempo real)
- 5x, 10x (acelerado)
- 50x, 100x (muy rápido)

Implementación:

```
python
self.time_step = 3600 * scale  # Base: 1 hora
```

3. Sistema de Cámara

Características:

- Damping factor: 0.08 (movimiento suave)
- Límites polares: evita inversión
- Focus en planetas: click para centrar
- Reset: doble click

Animación de enfoque:

javascript

const easeProgress = 1 - Math.pow(1 - progress, 3); *// Ease-out cubic*

camera.position.lerpVectors(startPos, targetPos, easeProgress);

📊 Datos del Sistema Solar

Planetas Implementados

Planeta	Masa (kg)	Radio (m)	Distancia (AU)	Período (días)
Sol	1.989×10^{30}	6.96×10^{8}	0	-
Mercurio	3.301×10^{23}	2.44×10^{6}	0.387	87.97
Venus	4.867×10^{24}	6.05×10^{6}	0.723	224.7
Tierra	5.972×10^{24}	6.371×10^{6}	1.0	365.25
Marte	6.417×10^{23}	3.39×10^{6}	1.524	686.98
Júpiter	1.898×10^{27}	6.99×10^{7}	5.203	4332.59
Saturno	5.683×10^{26}	5.82×10^{7}	9.537	10759.22
Urano	8.681×10^{25}	2.54×10^{7}	19.191	30688.5
Neptuno	1.024×10^{26}	2.46×10^{7}	30.069	60182

Fuente de datos: NASA JPL Planetary Data

🚀 Instalación y Ejecución

Requisitos Previos

- Python 3.8 o superior
- pip (gestor de paquetes)
- Navegador moderno con WebGL

Paso 1: Clonar Repositorio

bash

```
git clone https://github.com/tu-usuario/solar-system-nbody.git
cd solar-system-nbody
```

Paso 2: Crear Entorno Virtual

bash

```
python -m venv venv
source venv/bin/activate  # En Windows: venv\Scripts\activate
```

Paso 3: Instalar Dependencias

bash

```
pip install -r requirements.txt
```

Contenido de requirements.txt:

text

```
Flask==3.0.0
Flask-SocketIO==5.3.5
Flask-CORS==4.0.0
python-socketio==5.10.0
python-engineio==4.8.0
werkzeug==3.0.1
eventlet==0.33.3
numpy==1.26.2
```

scipy==1.11.4

Paso 4: Ejecutar Servidor

bash

python app.py

Paso 5: Abrir en Navegador

text

http://localhost:5000

🎮 Guía de Uso

Controles de Cámara

- **Rotar**: Click izquierdo + arrastrar
- **Zoom**: Scroll del mouse
- **Pan**: Click derecho + arrastrar
- **Enfocar planeta**: Click en planeta
- **Reset vista**: Doble click en canvas

Controles de Simulación

- ▶️ **Iniciar**: Comienza la simulación
- ⏸️ **Pausar**: Detiene el tiempo
- 🔄 **Reiniciar**: Vuelve a t=0

Control de Velocidad

- **Botones**: 0.1x a 100x (instantáneo)
- **Slider**: Ajuste continuo
- **Tiempo simulado**: Mostrado en días y años

Visualización

- **Trayectorias**: Toggle ON/OFF
- **Nombres**: Toggle ON/OFF

- **Estelas**: Se desvanecen después de 30s

🔧 Personalización

Agregar Nuevos Cuerpos

En physics/constants.py:

```
python
'pluton': {
  'name': 'Plutón',
  'mass': 1.303e22,
  'radius': 1.188e6,
  'position': np.array([39.48 * AU, 0.0, 0.0]),
  'velocity': np.array([0.0, 0.0, 4.67e3]),
  'color': [0.8, 0.7, 0.6],
  'gradient': [0.7, 0.6, 0.5, 0.9, 0.8, 0.7],
  'orbital_elements': {
    'semi_major_axis': 39.48 * AU,
    'eccentricity': 0.248,
    'inclination': 17.2,
    'period': 90560 * DAY
  }
}
```

Cambiar Escalas

Ajustar factores de visualización:

```
python
SCALE_FACTORS = {
  'distance': 2e-9,  # Mayor = planetas más separados
```

```
    'radius': 3e-7,   # Mayor = planetas más grandes
    'time': 43200     # Mayor = simulación más rápida
}
```

Modificar Shader

Cambiar colores procedurales en renderer.js:

javascript

```
// En fragmentShader
float pattern = fractalNoise(vPosition * 3.0 + time * 0.03);
pattern = pattern * 0.25 + 0.75;  // Ajustar contraste

vec3 baseColor = mix(colorGradient[0], colorGradient[1], gradientMix);
```

📈 Optimizaciones

Performance Backend

1. **Threading**: Simulación en thread separado
2. **Locks**: Prevención de race conditions
3. **Actualización selectiva**: Envío cada 5 frames

Performance Frontend

1. **BufferGeometry**: Geometría optimizada
2. **Frustum Culling**: Solo renderiza lo visible
3. **Límite de estelas**: Máximo 500 puntos por planeta
4. **PixelRatio limitado**: Max 2x para retina

🐛 Debugging

Consola del Navegador (F12)

Mensajes de log:

text

✨ Renderizador inicializado

🪐 Mercurio creado

💍 Creando anillos para Saturno

🌟 Estela creada para Venus

Verificar Conexión

javascript

```
socket.on('connect', () => {
  console.log('✅ Conectado');
});
```

Monitorear Energía

python

```
energy = simulator.compute_energy()
print(f"E_total: {energy['total']:.2e} J")
```

📚 Referencias

Papers y Documentación

1. **Numerical Recipes in C** - Press et al.
2. **Classical Mechanics** - Goldstein
3. **Three.js Documentation** - threejs.org
4. **NASA JPL Horizons System** - ssd.jpl.nasa.gov

APIs Utilizadas

- **Three.js r128**: Renderizado WebGL
- **Socket.IO 4.5.4**: WebSocket real-time
- **NumPy 1.26**: Cálculos numéricos
- **SciPy 1.11**: Integradores ODEs

📄 Licencia

👨‍💻 Autor

Franklin - Ingeniero de Sistemas
Especializado en Machine Learning y Computer Vision

📧 franklinsecper@hotmail.com
📍 Montería, Córdoba, Colombia

🎓 Notas Técnicas

Precisión de la Simulación

- **Error de energía**: < 0.001% por órbita terrestre
- **Paso de tiempo**: 3600 segundos (1 hora)
- **Método**: Velocity Verlet (2º orden)

Limitaciones

- No incluye relatividad general
- Planetas considerados como puntos (sin forma)
- No incluye efectos de marea
- Luna de la Tierra no implementada aún

Futuras Mejoras

1. Agregar Luna y lunas galileanas
2. Cinturón de asteroides
3. Cometas con órbitas elípticas
4. Modo VR (WebXR)
5. Exportación de datos a CSV

¡Disfruta explorando el Sistema Solar matemático! 🌌 ✨