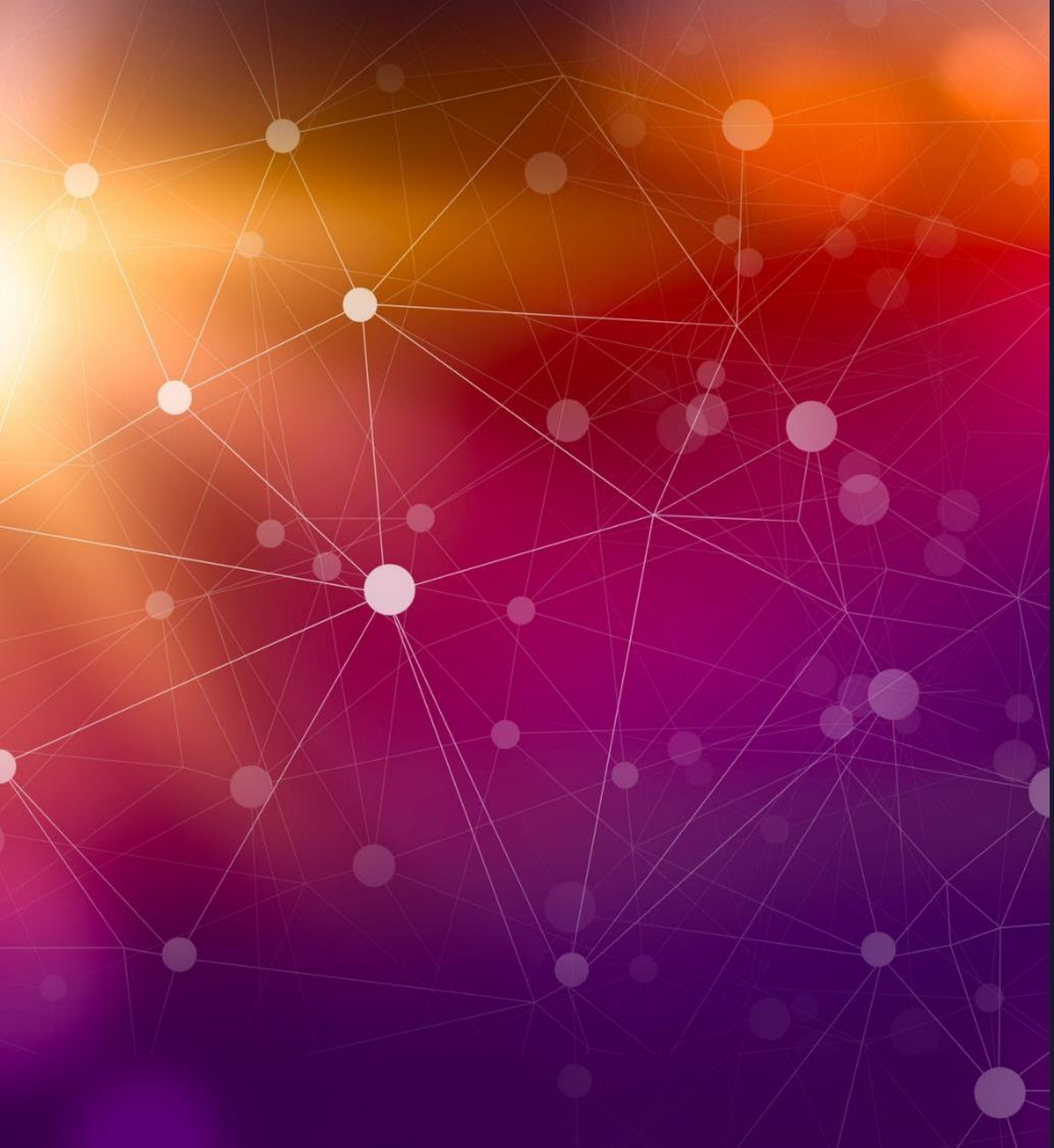
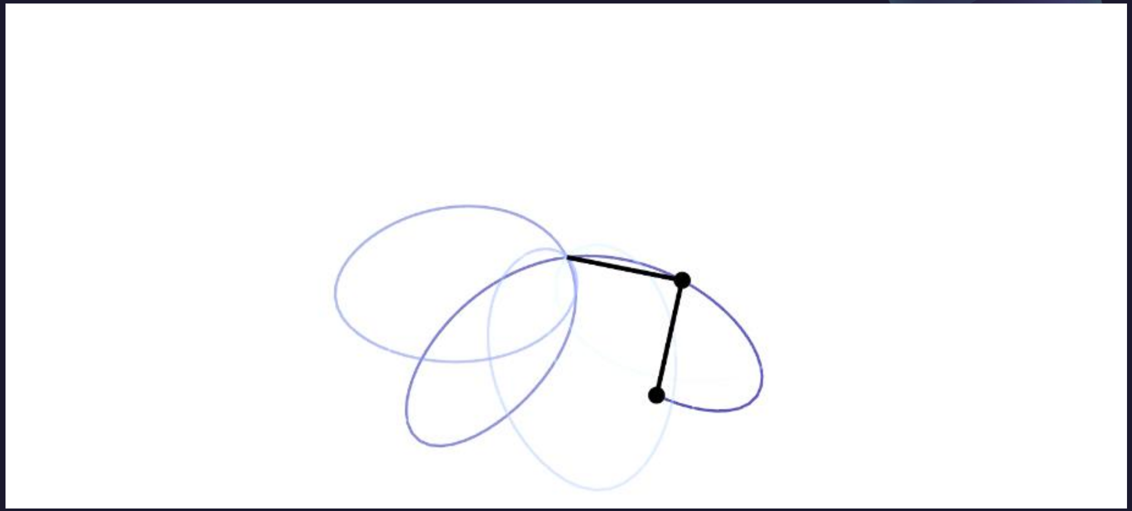


Double Pendulum Simulation with WebGL

Antonio Serra
1893530

Introduction

- Overview of the Double Pendulum Simulation
- Importance of visualizing complex physical system
- Use of WebGL for rendering in real-time

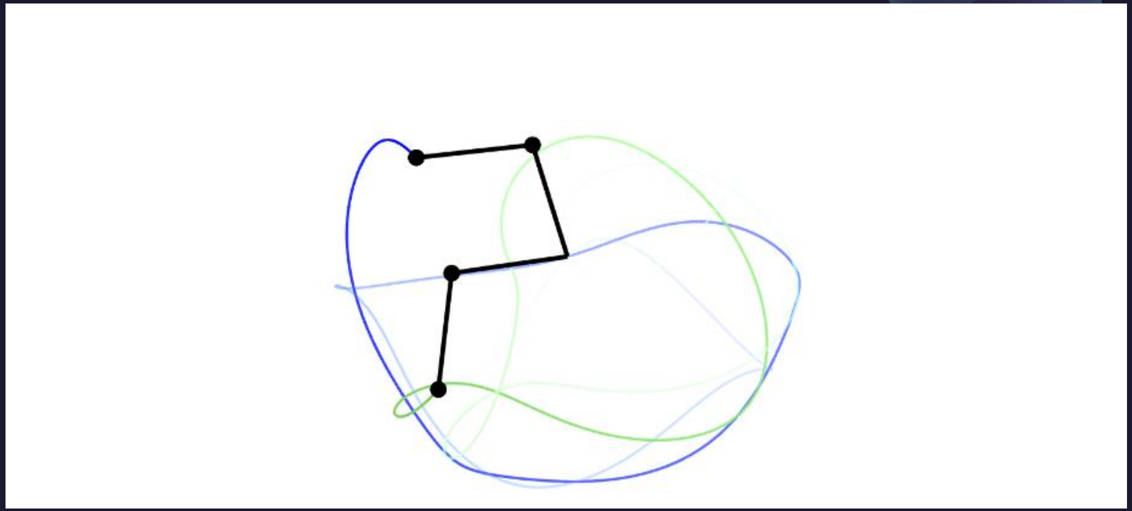


What is a Double Pendulum?

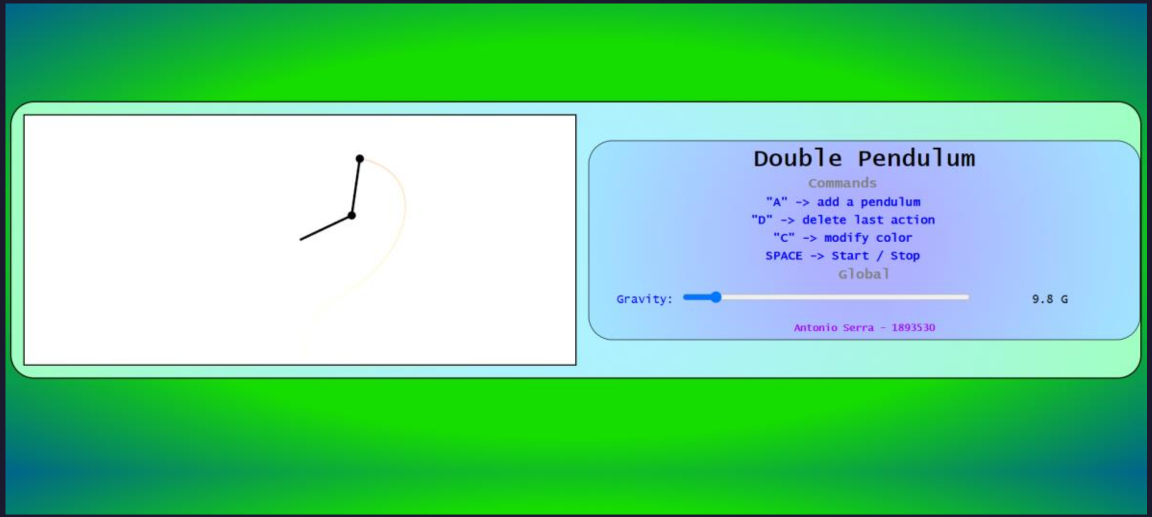
- A Double Pendulum consisit of two pendulums attached end-to-end

- Exhibits chaotic behavior and is a classic example of a complex system in physics, small changes in initial conditions give very different outcome

- Simulation shows the unpredictability of the double pendulum



Project Overview



Technologies Used:

- HTML for structure
- Javascript for simulation logic
- WebGL for rendering graphics

User Interactivity:

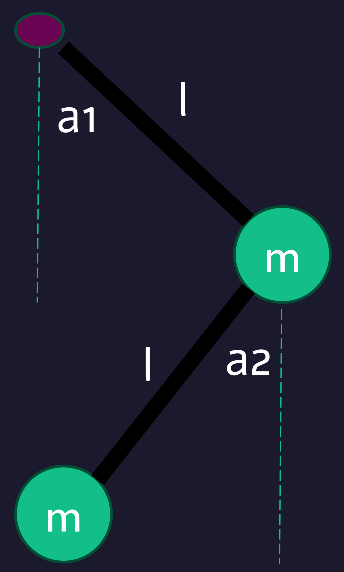
- Controls for adding/removing pendulums
- Slider for gravity adjustment
- Real-time visual feedback

WebGL Overview

- What is WebGL?
  - A JavaScript API for rendering 2D and 3D graphics within a web browser
  - Utilizes the GPU for rendering, allowing for efficient graphics processing
- Why WebGL for this project?
  - Provides high-performance rendering capabilities necessary for dynamic simulations

Pendulum Dynamics

- State Variables:
  - Angles (a1, a2) for the two pendulums
  - Momenta (p1, p2) representing the momentum of each pendulum
- Derivative Calculation:
  - ‘derivative()’ function computes changes in angles and momenta
  - Interaction between the two pendulums lead to complex motion



Simulation Logic

```
function deriviative(a1, a2, p1, p2) {
    let ml2 = M * L * L;
    let cos12 = Math.cos(a1 - a2);
    let sin12 = Math.sin(a1 - a2);
    //rate of change of angle a1
    let da1 = 6 / ml2 * (2 * p1 - 3 * cos12 * p2) / (16 - 9 * cos12 * cos12);
    //rate of change of angle a2
    let da2 = 6 / ml2 * (8 * p2 - 3 * cos12 * p1) / (16 - 9 * cos12 * cos12);
     // rate of change of momentum p1
    let dp1 = ml2 / -2 * (+da1 * da2 * sin12 + 3 * G / L * Math.sin(a1));
    // rate of change of momentum p2
    let dp2 = ml2 / -2 * (-da1 * da2 * sin12 + 3 * G / L * Math.sin(a2));
    return [da1, da2, dp1, dp2];
}
```

- The simulation is based on physical laws governing pendulum motion
- Uses the Runge-Kutta method (RK4) for numerical integration and solve ordinary differential equation (other method can be Euler method, Heun's method, midpoint method, etc…)
- The derivative function contains normalized constant to simplify the expression
- Models forces acting on the pendulum and updates thier state over time

Shaders in WebGL

- What are Shaders?
  - Small programs that run on the GPU
  - Essential for customing the visual output of 3D graphics
- Two Main Types:
  - Vertex Shader: process vertex data
  - Fragment Shader: computes color and other attribute of pixels

Vertex Shader

- Responsable for transforming vertex position and passing data to fragment shader
- Key Components:
  - Takes vertex positions as input
  - Applies transformations (e.g. aspect ratio adjustments, translation, etx…)
  - Output processed vertex positions
- Example Code Snippet:

```
void main() {
    v_point = a_point;
    gl_Position = vec4(a_point * ${massRadius} / u_aspect + u_center, 0, 1);
}
```

Fragment Shader

- Handles the coloring and shading of each pixel on the rendered object
- Utilizes interpolated values from the vertex shader
- Key Features:
  - Calculates distance from the center for smooth gradient effects
  - Outputs color based on distance and provided uniform values
- Example code Snippet:

```
void main() {
    float dist = distance(vec2(0, 0), v_point);
    float v = smoothstep(1.0, 0.9, dist);
    gl_FragColor = vec4(u_color, v);
}
```

Rendering the Pendulum

- The rendering process involves using shaders to visualize the pendulum and its tail
- Rendering Steps:
    - Set up WebGL content and buffers
    - Compile shaders and create a program
    - Draw pendulum using vertex and fragment shader
    - Update the scene based on simulation state

Tail Visualization

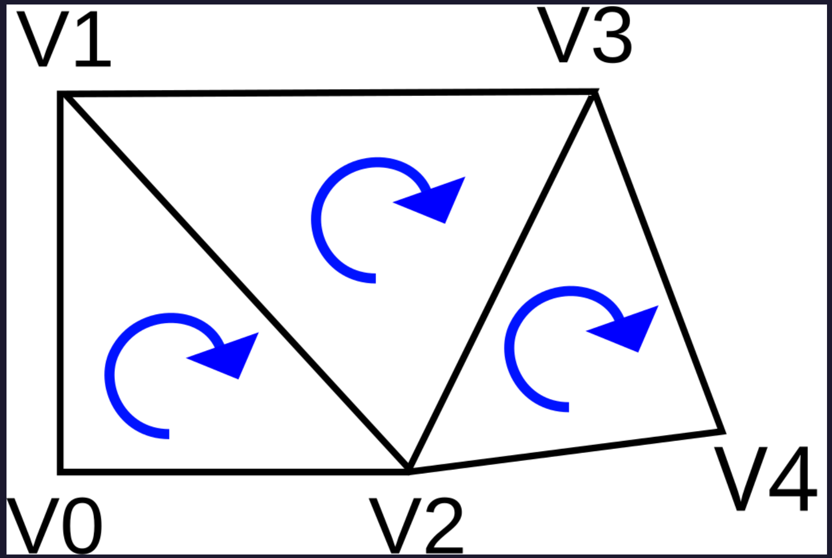
- Each pendulum has a trailing effect to visualize its path over time
- Implemented using a polyline constructed from historical positions
- Create a visually appealing representation of the pendulum’s motion
- ‘polyline()’ function is defined to draw the trajectory of the double pendulum

Polyline Function

- Is defined to draw a series of connected line segments based on a set of points
- Problem: the default drawing of line strip geometry thickness.
- To solve the problem represent each line as a set of triangles, this method ensures that the lines have a consistent thickness and smooth transition at the joints

Triangle Strip

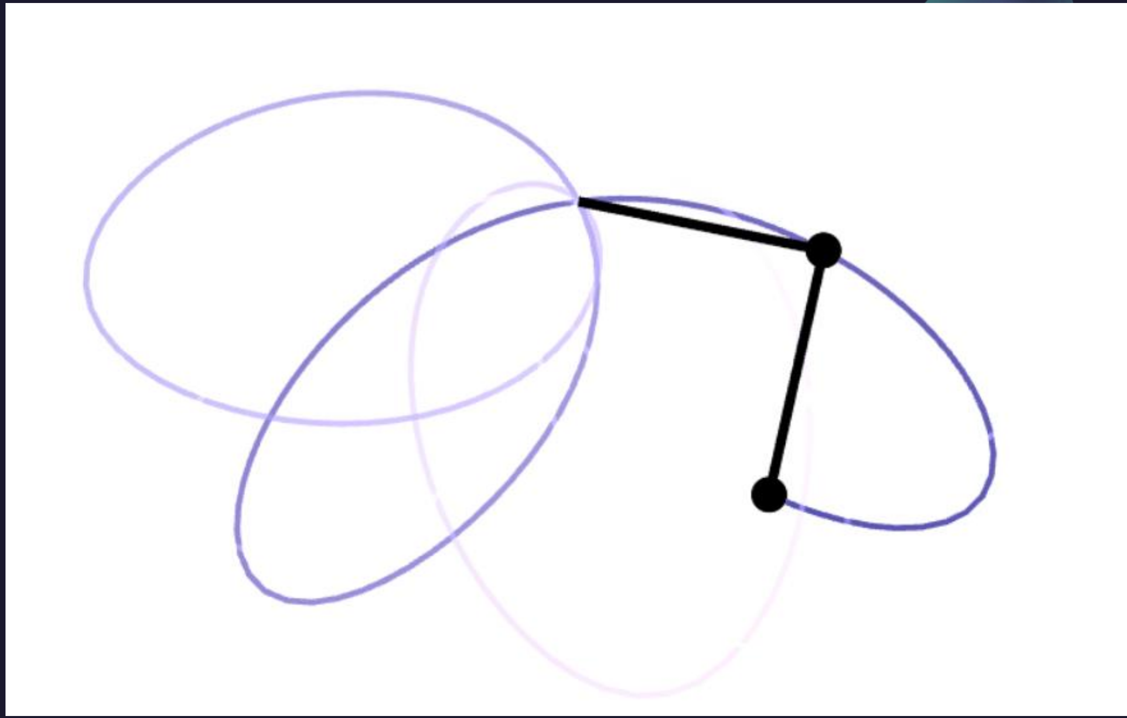
- Is a subset of triangles in a triangle mesh with shared vertices, and is a more memory-efficient method of storing information about the mesh
- Draws a series of triangles using vertices V0, V1, V2 then V2, V1, V3 (note the order)
- In this context, the polyline is converted into a strip of triangles to represent a thick line



Polyline Function Step

- For the first point (x1, y1), initialize the polyline using sub() function to compute difference between consecutive points and normalize() function to nomalize these vecetors
- For each subsequent segment defined by points (x1,y1) and (x2, y2) compute the direction and use the bisector to determine the vertices of the triangles that form the thick line segment
- Afert processing all segments, the function handles the final point to complete the polyline
- The color of the tail is managed on the Fragment Shader using the age to calculate alpha value:

```
gl_FragColor = vec4(u_color, max(0.0, v_alpha - u_cutoff) / icutoff);
```



History Function

- To manage and store a history of an angle sums in a circular buffer, for efficient storage and retrieval of the most recent ‘n’ entries, avoiding the need for shifting elements when adding new ones
- Is crucial for maintaining a history of points that can be visited or processed later
- Push(a1, a2) function computes the sum of sin and cos of the a1 and a2 angles and stores them in the current position of the circular buffer
- Visit(f) function iterates over the entries in the buffer and applies the function f to each pair of consecutive entries

User Interaction

- Interactive controls enhance user engagement:
  - Commands:
    - ‘A’ → Add a pendulum
    - ‘D’ → Delete the last action
    - ‘C’ → Modify color
    - SPACE → Start/Stop the simulation
  - Sliders:
    - Adjust gravity to see real-time effect on pendulum motion

Thanks

Antonio Serra

1893530