Project Breakdown

November 3, 2024

In light of these two papers and their summaries, an overview of the project follows.

1 Idea

The project's main goal is to simulate a paper airplane with realistic bending and flexibility, using the discrete shell model. This model lets us represent the airplane as a flexible structure, capturing how paper would naturally bend and move.

We treat the paper airplane as a thin shell made up of a mesh with vertices and edges that can bend, similar to real paper. Bending energy captures the material's resistance to bending, making the airplane respond realistically to simple forces.

1.1 Blackbox Model

At a very high level, we can think of our simulation as a black box. The inputs, outputs, and core processes of the black box are as follows:

1.1.1 Inputs

- **Initial Mesh Configuration:** A 3D triangular mesh representing the paper airplane's shape. This mesh includes:
  - **Vertex positions:** The coordinates of each point on the paper airplane.
  - **Connectivity:** How vertices are connected to form edges and triangles.
- **Material Properties:** Properties that influence how the paper behaves
  - bending stiffness (resistance to bending)
  - mass distribution (mass per vertex or triangle)
  - damping coefficient (rate of energy dissipation).

- **Simulation Parameters:**
  - time step Δt (frequency of simulation updates)
  - duration (total simulation time)
  - optional external forces (such as gravity or an initial push)

1.1.2 Outputs

- **Deformed Mesh Configuration Over Time:**
  - Vertex positions at each time step, showing the paper airplane's shape as it bends, folds, or deforms dynamically
- **Velocity and Acceleration at Each Vertex:** Captures how the airplane's speed and direction change over time, affecting flight and stability.
- **Bending Energy:** Shows the internal energy due to bending, indicating how much deformation occurs in response to forces.

1.1.3 How the Model Works Internally

In simple terms, the model simulates the physics of a thin shell structure over time.

- **Initialization:**
  - Load the mesh (airplane shape)
  - Set initial vertex positions
  - Initialize parameters like stiffness and damping.
- **Time-Stepping:**
  - For each time step, calculate how forces affect each vertex based on bending energy, mass, and any external forces.
  - Use Newmark integration to update vertex positions and velocities, simulating how the airplane deforms or bends dynamically.
  - Solve for displacements at each time step using a solver, ensuring realistic deformation.
- **Energy Minimization:** The model minimizes bending energy to reach a new equilibrium shape at each step, helping achieve realistic deformation.

1.1.4 What the Model Simulates Physically

The simulation aims to capture realistic paper-like behavior, specifically:

- **Bending:** How the airplane's structure bends or folds due to forces and constraints.
- **Dynamics:** How the airplane would move or settle under its weight, stiffness, and damping.

For example, if you were to "throw" this virtual airplane by applying an initial force, the simulation would show how the paper structure deforms and moves over time.

1.1.5 Summary

As a black box:

- **Inputs:** Initial mesh, material properties, simulation parameters, optional forces
- **Outputs:**
  - Sequence of deformed shapes (vertex positions) over time
  - Velocity
  - Acceleration
  - Bending energy

1.2 Considerations

1.2.1 Underlying Engine

The paper suggests taking a working cloth simulator (like Baraff and Witkin) and replacing the bending energy as well as the integration scheme.

However, for our purposes, it's important to note that paper has next to zero bending energy since it does not stretch or shear. So we might be able to get away with not implementing membrane energies, and simply calculate flexural energies and use that as the total energy. In this case, we do not need to implement anything from Baraff and Witkin since both the integration scheme as well as the bending energy are replaced in the paper.

This will also speed up the simulation. And without this, the baseline project is suuuuper simple. Like, illegally simple.

1.2.2 Automatic Differentiation

We should probably ask if we can use the OG paper's AD classes for automatic differentiation to compute Jacobians and Hessians.

1.2.3 Stiffness

Currently, the stiffness parameter is a scalar that is equal for every vertex of the mesh (i.e. every spring in the system.) It's important to see that this fails to handle the case where paper is folded and then flattened again. In this case, one would simply reset the dihedral angle of the undeformed configuration and the paper sheet would still behave as it did initially, which does not accurately model the behavior of irl paper.

A suggestion for a simple modification is to have a stiffness coefficient for every edge, i.e. to use a vector instead of a scalar. That way, once a paper is folded, we can adjust the spring stiffness (for example double or triple it, since now the adjacent faces on the other side of the crease will resist this bending more) if there is a fold along an edge.

1.2.4 UI

Letting the user pick a fold is way too involved. We can work with predefined folds. UI details can be considered later (i.e. highlighting the edge to be folded, how to proceed with the simulation etc.)

1.2.5 External Forces

We need to decide on some external forces. One simple external force could be a simple unit vector parallel to the y axis on the corner of a page, or on every vertex along an edge, etc. Gravity is also fairly simple to implement as it is just a uniform downward force on every vertex. If we don't have external forces, our mesh will simply stay in place without acceleration.

It's important to note that the folding mechanic isn't modelled *physically.* We're simply modifying the dihedral angles of rest state of the mesh, so it is not sufficient to show the capabilities of the simulator.

1.2.6 Collision

I honestly think it would be highly worth it for us to implement a simple collision mechanic so that our plane can fall and collide with the ground, since we've pretty much minimized our simulation by a lot by choosing paper since we don't have to implement membrane energies... I'm not even sure we'll be allowed to do this, tbh.

I think the easiest thing to do is to make our super simple simulator look really nice, and so making a scene where the plane collides with the paper is highly highly worth it.

The "crumpling" should already be handled by our discrete shell model. We might have to tune parameters or refine the mesh, but the colliding force should already crumple it somewhat realistically.

1.3 Visuals

Again: Look nice → better grade. Just slap some nice textures on the meshes, and maybe a cubemap.

`libigl` doesn't support cubemaps unfortunately, but it is built on top of OpenGL, and ChatGPT tells me that we can add a cubemap by injecting OpenGL commands into libigl's render loop as follows:

1. **Load Cubemap Textures**

   Load the six images for your cubemap (representing the six faces: right, left, top, bottom, front, and back) and create an OpenGL cubemap texture. You can use any image loading library like `stb_image` for this.

2. **Set Up a Cubemap Texture in OpenGL**

   Bind each image face to an OpenGL cubemap target. Here's how to set up the cubemap texture:

```
GLuint cubemapTexture;
glGenTextures(1, &cubemapTexture);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);

// Load each face of the cubemap
std::vector<std::string> faces = {
    "right.jpg", "left.jpg", "top.jpg", "bottom.jpg", "front.jpg", "
 back.jpg"
};

int width, height, nrChannels;
for (GLuint i = 0; i < faces.size(); i++) {
    unsigned char *data = stbi_load(faces[i].c_str(), &width, &height,
 &nrChannels, 0);
    if (data) {
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB,
 width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
        stbi_image_free(data);
    } else {
        std::cout << "Cubemap␣texture␣failed␣to␣load␣at␣path:␣" <<
 faces[i] << std::endl;
        stbi_image_free(data);
    }
}
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
 GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
 GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
 GL_CLAMP_TO_EDGE);
```

3. **Create a Skybox Shader**

Write a basic OpenGL shader for the cubemap. This shader will sample the cubemap texture based on the camera direction, making it appear as though the environment surrounds your scene.

Listing 1: Vertex Shader (skybox.vs)

```
// Vertex Shader (skybox.vs)
#version 330 core
layout(location = 0) in vec3 aPos;
out vec3 TexCoords;

uniform mat4 view;
uniform mat4 projection;

void main() {
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww; // keeps the skybox at a constant depth
}
```

Listing 2: Fragment Shader (skybox.fs)

```
// Fragment Shader (skybox.fs)
#version 330 core
in vec3 TexCoords;
out vec4 FragColor;

uniform samplerCube skybox;

void main() {
    FragColor = texture(skybox, TexCoords);
}
```

4. **Render the Skybox in libigl's Render Loop**

Within libigl's render loop, disable depth writing for the skybox and render a large cube with the skybox shader before rendering other objects.

```
// Load and compile the skybox shaders
GLuint skyboxShader = LoadShader("skybox.vs", "skybox.fs");

// Define a cube for the skybox
float skyboxVertices[] = {
    // positions for a 3D cube ...
};

// Set up skybox VAO
GLuint skyboxVAO, skyboxVBO;
glGenVertexArrays(1, &skyboxVAO);
glGenBuffers(1, &skyboxVBO);
glBindVertexArray(skyboxVAO);
glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices,
     GL_STATIC_DRAW);
```

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)
    0);
glBindVertexArray(0);

// In the render loop
while (!glfwWindowShouldClose(window)) {
    // Render skybox first
    glDepthFunc(GL_LEQUAL);  // Change depth function so depth test passes
     when values are equal to depth buffer's content
    glUseProgram(skyboxShader);
    glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix())); //
     remove translation from the view matrix
    setMat4(skyboxShader, "view", view);
    setMat4(skyboxShader, "projection", projection);
    // skybox cube
    glBindVertexArray(skyboxVAO);
    glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
    glDepthFunc(GL_LESS); // Set depth function back to default

    // Render the rest of your scene with libigl here...
}
```

Or, we can also pass information to `RayLib` using the `Mesh` class after computing everything in our simulator @Hugues :)

1.3.1 Flight

If we ever get to this, a very simple way to model it would be as follows: 1) "Collapse" the plane to its center of mass, 2) Compute trajectory for this point which applies to the whole model, 3) Use simple forces to simulate drag etc. to show how our folded shell behaves.

2 Implementation Steps

- **Modeling the Mesh:** The easiest way to achieve this is to model the mesh as the final plane and unfold it into a sheet, e.g., in Blender.
- **Basic Structure and Initialization:**
  - Initialize parameters
  - Load the mesh
  - Set initial states for the simulation.
- **Basic Time Stepping (Without Forces):** Implement `advanceOneStep` using the Newmark integration scheme. Use a placeholder `linearSolve` with a dummy value, such as setting `du = 0`.

- **Bending Energy and Forces:**
  - Implement `addShellBendingEnergy`: calculate bending energy based on the mesh deformation
  - Implement `addShellBendingForce`:mcompute internal forces that resist bending
  - Implement `addShellBendingHessian` to populate the stiffness matrix K with bending-related entries for accurate force calculations
- **Linear Solver:** Implement `linearSolve` to solve for `du`, updating the system based on the forces and bending constraints
- **Time Integration with Forces:** Integrate forces in `advanceOneStep` by using `du` to update positions and velocities
- **External Forces:** Add simple external forces like gravity to show that our paper plane behaves realistically with the shell engine
- **Energy and Residual Checks:** Implement functions to calculate total energy and residuals
- **Add a scene:**
  - Texture for our plane!!! pls
  - Achievable goal: Simple collision mechanic with the ground plane
  - Advanced goal: Simple simple simple flight mechanic

3 Comments

Guys I think we can get away with a really simple discrete shell implementation if we slap collision mechanics and some nice visuals on top of it.

It's probably super worth making something extremely simple that looks really polished for a decent grade rather than something hard that also looks shit.