Semester Project Report: Real-Time Cloth Simulation with CUDA and Vulkan

Mao Zhang ID: 218329

1 Project Overview

This project implements a real-time cloth simulation system that uses CUDA for physics computation and Vulkan for rendering and visualization. The main features are:

- Physics features of the simulated cloth:
  - Deformation under gravity and a wind field.
  - Collision effect with an infinite ground plane.
  - Collision effect with a movable box obstacle controlled via keyboard (W/A/S/D).
- Mouse interaction:
  - Clicking on the cloth applies a localized impulse, like "slapping" the fabric.
- Interactive ImGui-based UI:
  - Toggle pinning of the top edge of the cloth (hang vs. fall).
  - Enable/disable wind and adjust wind strength.
  - Adjust physical parameters (mass, spring stiffnesses etc.).
  - Switch between three material presets: cotton, elastane, and PVC.
- Rendering features:
  - Different visual appearances for different materials via textures and shading.
  - A fake soft shadow system for the cloth and box on the ground.

Overall, the result is an interactive real-time cloth simulation demo with configurable physics, materials, and rendering, controlled through a convenient GUI.

2 Physical and Computer Graphics Implementation

2.1 Cloth Model: Mass–Spring System

The cloth is modeled as a classic mass–spring system on a regular 2D grid: each grid point is a particle with position $\mathbf{x}$, velocity $\mathbf{v}$, and mass m. Particles are connected by three types of springs:

Structural springs. Connect immediate horizontal and vertical neighbors. They control stretch along the warp/weft directions (stiffness: `k_struct`).

Shear springs. Connect diagonal neighbors within each grid cell. They control shear deformation (stiffness: `k_shear`).

Bend springs. Connect particles that are two grid steps apart. They control out-of-plane bending (stiffness: `k_bend`).

Each spring approximately follows Hooke's law:

$$\mathbf{F}_{\text{spring}} = -k\left(\|\mathbf{d}\| - L_0\right)\frac{\mathbf{d}}{\|\mathbf{d}\|},$$

where k is the spring stiffness, $\mathbf{d}$ is the vector between two particles, and L_0 is the rest length.

Additional forces include:

- **Gravity**: $\mathbf{F}_g = m\mathbf{g}$.
- **Damping**: reduces oscillation and improves stability (applied to velocities or spring extension).
- **Wind**: a time-varying force that depends on wind direction and the cloth's surface orientation.

Pinning (the "pin top edge" option) is implemented by marking particles on the top row as fixed so their positions are not updated during integration.

2.2 Time Integration and Stability

The simulation uses a fixed time step (e.g. $\Delta t = 1/120$) with an explicit integration scheme (semi-implicit Euler style):

1. Compute forces on each particle:

$$\mathbf{F}_{\text{total}} = \mathbf{F}_{\text{springs}} + \mathbf{F}_g + \mathbf{F}_{\text{wind}} + \mathbf{F}_{\text{collision}}.$$

2. Compute acceleration:

$$\mathbf{a} = \frac{\mathbf{F}_{\text{total}}}{m}.$$

3. Update velocity and position:

$$\mathbf{v}_{t+\Delta t} = \mathbf{v}_t + \mathbf{a}\,\Delta t, \qquad \mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{v}_{t+\Delta t}\,\Delta t.$$

Explicit integration can become unstable for very stiff springs. To keep the simulation stable and visually plausible, the project:

- Uses a relatively small fixed time step.
- Restricts the range of stiffness parameters exposed in the UI.
- Adds damping and performs a short *warm-up phase* after certain operations (e.g. rebuilding the cloth), so the cloth settles into a natural hanging pose before normal simulation continues.

2.3 Collision Detection and Response

The collision environment consists of:

- A ground plane at $y = y_{\text{ground}}$.
- An axis-aligned box defined by its center $\mathbf{c}$ and half extents $\mathbf{h}$.

For each particle, after integration:

Ground collision. If a particle's y-coordinate is below the ground plane, the particle is projected back onto or slightly above the plane, and its velocity is adjusted to avoid sticking or tunneling.

Box collision. If a particle lies inside the axis-aligned box (in x, y, z), it is projected to the nearest point on the box surface and its velocity is corrected.

The box position is updated per frame from keyboard inputs, and the collision scene is passed to the CUDA solver through the `Simulator`. This can be viewed as a geometric constraint step that enforces non-penetration.

2.4 Rendering and Shading

Rendering is done using Vulkan with a simple but effective lighting model:

- The cloth, box, and ground are rendered using vertex and index buffers.
- The fragment shader computes Lambertian diffuse lighting:

$$\text{diffuse} = \max(\mathbf{n} \cdot \mathbf{l}, \text{ambient}),$$

  where $\mathbf{n}$ is the normal, $\mathbf{l}$ is the light direction, and a small ambient term avoids fully black regions.
- Different materials are distinguished via:
  - Different base colors and textures.
  - Different shading parameters (e.g. specular strength, opacity for `pvc`).

A fake soft shadow is implemented in the fragment shader using additional geometric information passed through push constants; this is described in Section 4.2.

3 Program Architecture and Data Flow

3.1 Module Overview

The project is split into several main modules:

- **Physics**
  - `cuda_solver.cu` (in `physics/cuda/`): CUDA-based solver for cloth dynamics.
  - `simulator.hpp` (in `sim/`): scene initialization, force updates, stepping, and collision configuration.
- **Cloth Core**
  - `cloth_builder.cpp` (in `core/cloth/`): builds a regular grid cloth mesh (positions, indices, pinning).
- **Rendering**
  - `vk_renderer` (in `render/vulkan/`): handles Vulkan instance/device, swapchain, render pass, pipelines, vertex/index buffers, textures, and drawing.
- **UI**

  - `ui_state.hpp`, `ui_panel.hpp`, `ui_panel.cpp` (in `ui/`): hold the UI parameters and ImGui logic.

- **Application Entry**
  - `main.cpp`: ties everything together, manages the main loop, and coordinates physics, rendering, and UI.

3.2 Data Flow: CUDA → CPU → Vulkan

To keep the implementation simpler, the current design uses the CPU as a bridge between CUDA and Vulkan:

Initialization.

- `build_regular_grid(params)` generates the cloth mesh (positions, indices, pin mask) on the CPU.
- A `CudaSolver` instance is created and wrapped by `Simulator`.
- `Simulator::init_scene(scene)` uploads the cloth data to CUDA device memory.
- `VulkanRenderer` initializes Vulkan resources:
  - Swapchain, render pass, pipeline, command buffers.
  - Vertex/index buffers (initially with small or placeholder sizes).
  - Textures, samplers, descriptor set.

Per-frame simulation.

- GLFW events are processed: keyboard (box movement) and mouse input.
- The ImGui UI is updated:
  - When users change mass, stiffness, material, or pinning:
    - `apply_material_preset(ui)` updates physical parameters.
    - `rebuild_cloth_from_ui()` rebuilds the cloth model, reinitializes the simulation, and runs a warm-up phase.
- A fixed-step loop accrues time and performs several physics substeps per frame:
  - `sim.update_forces(forces)` (gravity, wind, click impulses).
  - `sim.step(FIXED_DT)` advances the CUDA simulation.

Reading back simulation data. After physics updates, the project calls:

`sim.download_positions_normals(host_pos, host_normals)`.

This copies current particle positions and normals from CUDA device memory to CPU arrays (`std::vector<Vec3>`).

On the CPU, the code builds a `std::vector<Vertex>`:

- `pos` from `host_pos`,
- `normal` from `host_normals`,
- `uv` derived from grid indices or world-space coordinates.

Uploading to Vulkan. `renderer.update_mesh(host_vertices, scene.cloth.indices)`:

- Checks if the existing vertex/index buffers are large enough; if not, recreates them.
- Uses `vkMapMemory` and `memcpy` to upload the vertex and index data to GPU memory.

  Similar update functions handle the box mesh and ground mesh.

Rendering. `renderer.draw_frame()`:

- Acquires the next swapchain image via `vkAcquireNextImageKHR`.
- Resets and records the command buffer for that image:
  - Begins the render pass.
  - Binds pipelines and descriptor sets.
  - Draws ground (material index 4), box (material index 3), and cloth (material index 0/1/2).
  - Invokes the ImGui Vulkan backend to draw the UI overlay.
- Submits the command buffer and presents the image.

This architecture is not the most bandwidth-efficient (due to GPU→CPU→GPU traffic every frame), but it is conceptually straightforward and adequate for the grid sizes used in this project.

4 Additional Technical Details

4.1 Material System: cotton, elastane, PVC

The material system distinguishes cloth types along two axes: physics and rendering.

4.1.1 Physical Behavior

Each material preset corresponds to a different set of parameters in `UIState`:

- `cloth_mass`
- `k_struct` (structural stiffness)
- `k_shear`
- `k_bend`

  Conceptually:

- **cotton**: higher structural and bending stiffness; less curvature and more "blocky" folds.
- **elastane**: lower stiffness; more stretch and smoother, larger deformations.
- **PVC**: intermediate stiffness; stronger resistance to bending than elastane, combined with specific shading and transparency.

  When the material is changed:

1. `apply_material_preset(ui)` updates the physical parameters.
2. The cloth is rebuilt and the simulator is re-initialized.
3. A short warm-up simulation is run to let the cloth settle into a natural equilibrium for the new parameters.

4.1.2 Visual Appearance

On the rendering side:

- A material index (0/1/2 for cloth types, 3 for box, 4 for ground) is passed via push constants.
- The fragment shader uses this ID to:
  - Choose which texture to sample (for textured materials).
  - Adjust base color and tint.
  - Control specular intensity, roughness, and transparency.

  For **PVC** material:
- The shader emphasizes highlights and reduces diffuse contribution to simulate plastic.
- Alpha is set to a value less than 1 to make the cloth partially transparent.
- The Vulkan pipeline enables alpha blending, so the pvc cloth visually mixes with what is behind it.

Combining these physical and visual differences gives each material a distinct look and motion.

4.2 Fake Soft Shadow

Instead of a full shadow map, the project uses a geometric approximation to create soft-looking shadows for the box and cloth on the ground.

4.2.1 Shadow Parameters

Each frame, the CPU computes and uploads via push constants:

- `lightDir` – direction of the light.
- `boxCenter` – world-space center of the box.
- `boxRadius` – approximate radius in the xz-plane for the box shadow.
- `clothCenter` – center of the cloth's bounding box.
- `clothSize` – approximate size of the cloth in the x and z directions.

These parameters approximate where the box and cloth would cast their shadows on the ground.

4.2.2 Fragment Shader Approximation

For the ground, in the fragment shader:

1. For each ground fragment, compute its world position.
2. Use a simple analytic relationship (depending on `lightDir`) to determine whether this point is within the "shadow footprint" of:
   - the box (using a circular or elliptical region centered at `boxCenter` with radius `boxRadius`),
   - the cloth (using a rectangular/elliptical region defined by `clothCenter` and `clothSize`).

3. Compute smooth falloff using a function such as `smoothstep` on the distance from the center: inside the core region is darkest, near the boundary the shadow fades smoothly.

4. Modulate the lit color:

$$\text{finalColor} = \text{litColor} \times (1 - w_{\text{shadow}} \cdot \text{shadowStrength}),$$

where w_{shadow} is the local shadow weight.

Thus, the ground under the cloth and box is darkened in a soft-edged shape that visually reads as a shadow. A simplified logic can also be applied to slightly darken regions of the cloth near the box or ground, enhancing contact and depth perception.

This technique is very fast and easy to tune, at the cost of physical accuracy.

4.3 UI Implementation (ImGui)

The UI is built using Dear ImGui with GLFW + Vulkan backends.

- `UIState` stores all parameters that the user can manipulate:
  - `wind_enabled`, `wind_base_strength`
  - `cloth_mass`, `k_struct`, `k_shear`, `k_bend`
  - `pin_top_edge`
  - `current_material_index` (cotton / elastane / PVC)
- `ui_panel.cpp` defines:
  - `ui::new_frame()` – starts a new ImGui frame.
  - `ui::draw_panel(UIState& ui)` – creates collapsible sections ("Wind", "Cloth Parameters", "Pinning", "Material") and binds widgets to `UIState` fields.
  - `ui::end_frame(VulkanRenderer*)` – finalizes ImGui and passes draw data to Vulkan.

In `main.cpp`, within the render loop, the Vulkan backend appends ImGui's draw commands to the same command buffer used for the 3D scene, so the UI is rendered as an overlay on top of the cloth simulation.