Congyao Zheng

congyao119@gmail.com

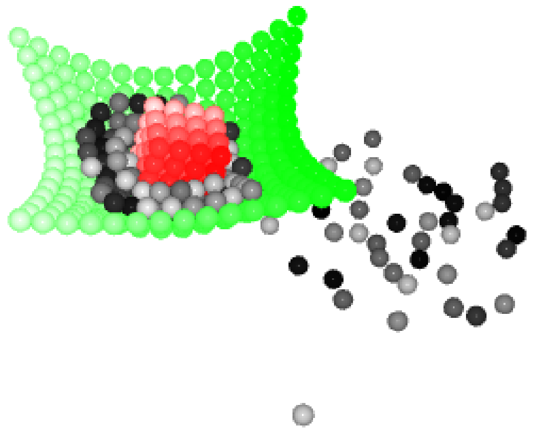
Final Project Report
INF585 Computer Animation, 22/23

2023-03-25

Implementation of Unified Particle Physics for Real-Time Applications

1 Introduction

In this project, we implemented the unified particle system [1]. Macklin et al. offered a comprehensive dynamics framework for in-the-moment visual effects. Through their demonstration of the system in the paper, we were able to create rigid bodies, textiles, and granular bodies with two-way interactions by using particles coupled with constraints as our fundamental building block. Using conventional particle-based techniques, we resolve a few common issues and outline a parallel constraint solver based on position-based dynamics that is effective enough for real-time applications.



2 Implementation

The structure of this project inherets the style of INF535 lab projects. CGP library and Eigen are the main dependencies used in this project.
Overall, to implement a unified particles system, we keep global variables of $all\_particles$ and obj_list to keep track of all the particles and their status. For each time step, we calculate each particle's $collision\ constraints$. And then, for each constraint, we project the collision constraints and all the other static constraints on the particles.
While in the original paper [1] stabilization iteration (1-2) was applied to collision constraints and solver iteration (1-5) was applied to all the constraints, we adopted the solver iteration only. Therefore, our simulation of collision would more likely to crash at a large time step.

2.1 Simulation

Algorithm 1 Simulation

```
procedure SIM($\delta_{time}$)
    for each particle $p$ in $all\_particles$ do
        $p.v \leftarrow p.v + \delta_{time} \cdot m\_gravity$    ▷ Apply external forces
        $p.v \leftarrow p.v * 0.98$    ▷ Dampen velocities
        $p.xg \leftarrow p.x + p.v \cdot \delta_{time}$    ▷ Initialise estimate positions
        $p.tinvmass \leftarrow 1.0/((1.0/p.invmass) \cdot exp(-p.x.z))$    ▷ Mass scaling
    end for
    $generate\_collision\_constraints()$    ▷ Update all collision constraints
    for $iter \leftarrow 0$ to $2$ do
        for each collision constraint $c$ in $g\_collision\_constraints$ do
            $c.project()$
        end for
        for each constraint $c$ in $g\_constraints$ do    ▷ distance & shape match
            $c.project()$
        end for
    end for
    for each object $obj$ in $dobj\_list$ do
        $obj.update\_mesh()$
    end for
    for each particle $p$ in $all\_particles$ do
        if $norm(p.xg - p.x) < \epsilon$ then    ▷ xg is the updated x by contraints
            $p.v \leftarrow \{0., 0., 0.\}$
        else
            $p.v \leftarrow (p.xg - p.x)/\delta_{time}$
            $p.x \leftarrow p.xg$
        end if
    end for
end procedure
```

The generate collision constraints function is responsible for generating collision constraints for all particles in the simulation. It first clears the global collision constraints list, then iterates through all particles, checking for collisions with the ground plane and other particles. If a collision is detected, the appropriate constraint is created and added to the global collision constraints list. This ensures that the simulation handles collisions between particles and the ground plane, as well as collisions between different particles.

2.2 Contraints

All constraint classes have a `cardinality` property that implies the number of particles connect to it and a `project()` function that applies the effects of the constraints on particles by updating the `xg` property.

2.3 Distance Constraint

`DistanceConstraint` class and its associated functions handle distance constraints between two particles in a particle-based simulation. The `buildDistanceConstraint` function creates a `DistanceConstraint` object for two given particles with a specified distance, initializing the constraint object, setting its `cardinality` to 2, and adding the particles to the constraint's particles list. The project function of the `DistanceConstraint` class calculates the displacement needed to maintain the specified distance between the particles and applies it to their positions, considering their inverse masses and whether they are fixed or not. This constraint is useful for maintaining the structure of objects like cloth, where particles need to maintain a certain distance from their neighbors.

2.4 Static Collision Constraint

This constraint handles static collision constraints in a particle-based simulation. It defines two main functions: `buildStaticCollisionConstraint` and project (inside the `StaticCollisionConstraint` class). `buildStaticCollisionConstraint` function creates a `StaticCollisionConstraint` object for a given particle, normal, and position. It initializes the constraint object, sets its cardinality to 1 (since it involves one particle), adds the particle to the constraint's particles list, and sets the constraint's normal and position. The function returns a `shared_ptr` to the created `StaticCollisionConstraint` object.
The project function (inside the `StaticCollisionConstraint` class) is responsible for projecting the static collision constraint for a particle. It checks if the particle is colliding with a static object (when the collided displacement is greater than epsilon), and if so, it adjusts the particle's position (add the displacement back) to resolve the collision. Additionally, it applies friction to the particle if enabled.

2.5 Rigid Shape Match Constraint

To form rigid bodies with particles, for each given object, we build a `RigidShapeMatchingConstraint` which initializes the constraint object, sets the constraint's obj to the given obj, sets its cardinality to the number of particles in the obj, and assigns the obj's particles to the constraint's particles list. This constraint controls the displacement between particles and maintains the overall shape formed by the particles of the object.
The project function of this constraint updates the obj's rotation matrix, and for each particle in the object, according to the paper(page 5, Eq. 15). For each particle, we calculate the displacement needed to match the object's original shape. The displacement is then applied to the particle's position.

2.6 Rigid Collision Constraint

This constraint resolves collisions for all particles in the global scope and applies friction if necessary. The only mattered initialization is the project function:

1. Get the two particles $mp1$ and $mp2$ using their indices $p1$ and $p2$.
2. Get the signed distance fields sdf_1 and sdf_2 for the two particles.
3. Check if the distance between the two particles is greater than the sum of their radii plus ϵ.
   (a) If true, handle the normal collision:
      - Calculate the distance d and the normal vector n between the two particles.

   (b) If false, handle the interlocking (tunneling) case:
      - For inner particles, set d and n based on the signed distance fields.
      - For surface particles, update d and n based on the particle radius and the dot product of $p12$ and n.
4. Calculate the displacement needed to resolve the collision for both particles δ_1 and δ_2.

5. Apply half of the calculated displacements to each particle's position.
6. If friction is enabled, apply static or kinetic friction based on the tangential displacement.

2.7 Scenes

As suggested by Macklin et al.[1], we generated a signed distance field(SDF) for inner particles based on their id, which is according to the order of the particles (from outside to inside). We can resolve deeper overlaps between shapes by reusing the mechanism of our particle-particle collision detection by storing the SDF on particles. We keep track of the SDF's magnitude and gradient at each particle's location, which can be considered a sparse, first-order approximation of the underlying field. The SDF is only applied on rigid body collision to distinguish the external/inner particles, but for the structure of the whole simulation, we keep a copy of the SDF placeholder for cloth and granular objects.

2.7.1 cube

The `create_cube()` function is responsible for creating a cube object with a given center, edge length, and rotation angles around the x and y axes. We generate a mesh for the cube, rotates and translates it, and then builds particles for the cube. The cube particles are generated by iterating through the x, y, and z dimensions with a step size of `particle_dim`. For each point in the 3D grid, a particle is created and added to the `all_particles` list. The particle's position is calculated by applying rotation and translation transformations. The SDF for each particle is also created, with the gradient direction and distance being calculated based on the particle's position within the cube. The gradient direction is transformed using the same rotation matrices as the particle position. The SDFs are added to the `sdf_list`. The cube is added to the list of objects.

2.7.2 cloth

The cloth is represented as a grid of particles with fixed distances between them. The function takes the center of the cloth, its height and width, and a list of fixed indices as input parameters. It first calculates the positions of the four corners of the cloth and the number of particles in each dimension. Then, it creates a mesh for the cloth using the `mesh_primitive_grid` function. The cloth object is initialized with the generated mesh, and its type is set to `CLOTH`. The function then iterates through the mesh positions to create particles and their corresponding SDF objects. The particles are added to the cloth object and the global list of particles. The function also updates the distance constraints for the cloth particles, ensuring that they maintain their fixed distances from each other. Finally, the function initializes the cloth's visual representation on the GPU and adds the cloth object to the global list of objects.

2.7.3 Granular

The `cube_stack()` function creates particles that are initialized in cube shapes without the rigid shape matching or distance constraints. As the granular object is only a set of unconstrained particles, instead of an object as a whole, no mesh will display for the stack of particles when `mesh` is ticked in GUI.

3 Limitation & Future Improvements

Our implementation suffers from two major limitations: 1) only cubes are created while the shape match constraint is supposed to support any shapes, and 2) no applicable shaders for the granular material. We would like to address these two limitations in future improvements.

Literatur

[1] Miles Macklin u. a. „Unified Particle Physics for Real-Time Applications“. In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), S. 104.