CGGS Coursework 1 (Simulation): Rigid-Body Simulation

Amir Vaxman

Errata

- Currently none.

Introduction

This is the 1^{st} coursework for the "simulation" choice. The purpose is to write `C++` code for a simple simulation of rigid bodies moving (and rotating) through space and interacting with each other. Visualization will be done with PolyScope as in Practical 0 (Section 4). The concrete objectives are:

- Implement discrete-time integration of forces and torques into velocities, and then into positions and orientations (Lecture 6).
- Resolve collisions between objects (and the floor), correcting velocities and positions (Lecture 9)
- Enforce constraints by impulse-based projections of velocities and positions (Lecture 10).
- Allow the algorithm to scale by broad-phase collision detection and better constraint projection.

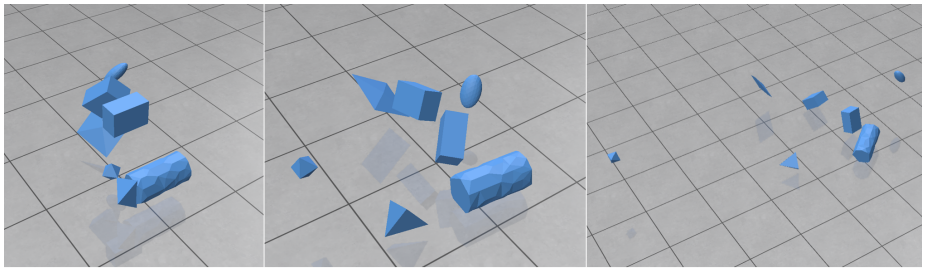

Figure 1: Results of the basic simulation in different time frames (left to right). The scene is `mixed_scene.txt` with no constraints.

General guidelines

The practical uses the same `Eigen` + `PolyScope` setup as in Practical 0. Your job is to complete several functions, given by signatures in the code, as described in the sections below. Such functions might have some default parameters that are supposed to induce the basic behaviour (of Section 1), and changing them "unlocks" the further features of the more advanced sections. The subproject `Section12` works both Section 1 and 2, and deals with the loading of scenes and constraints and visualizing them. Explore its code documentation and options to see how you can modify it to your settings—it does not get graded, but rather only the functions you have to complete in the respective header files

1 Basic Rigid-Body Simulation

Here you will implement a basic version of rigid bodies that are both in free fall and respond to collisions. You should do this by following the following steps:

1.1 Free Movement

Initialization The script loads and initializes (through `load_scene()` within `scene.h`) a set of tetrahedral meshes, and sets them as instances of the `Mesh` class (within `mesh.h`). The format of the scene file is as follows:

```
#num_meshes
#name0 density0 #is_fixed0 #xyz_position0, #quaternion_orientation0
#name1 density1 #is_fixed1 #xyz_position1, #quaternion_orientation1
...
```

In the `init_static_properties()` function, the original position of the mesh is normalized to have centre-of-mass (COM) at the origin, and its 3×3 *inverse* inertia tensor (as `Mesh.invIT`) is computed by integrating over the tets (this is not part of the coursework; you can find the derivation of this computation though in the Lecture notes so you can verify). Assuming uniform density, `Mesh.invMass` (note inverse again) is also computed by $\frac{1}{\rho \cdot V}$ where ρ is the density, and V is the volume of that mesh. the reason to only encode inverses is that we only need these quantities for collision; for fixed meshes (like the ground), they will be all zeroes (∞ resistance to movement).

Representing kinematics The `Mesh.origV` vertex positions are initialized to those of the loaded mesh, albeit translated to have their COM in the origin as detailed above. You *should never* update this variable. Your job in the entire CW is to update the COM (as `Mesh.COM`), the orientation (as `Mesh.orientation`), the linear velocity (as `Mesh.comVelocity`) and angular velocity (as `Mesh.angVelocity`). After each such update, the `Mesh.currV` will be updated (by uniform translation and rotation) to reflect the current position

of the vertices in any given time step; this is already coded in, so this coursework does not comprise any explicit update of vertex positions.

Integrating free movement Your first task is to implement a semi-implicit Euler integrator for velocity and then position, by filling the functions called from `Mesh::integrate()`. The only force you will use (in this basic version) will be gravity, inducing the acceleration of $g = (0, -9.8, 0)$. Having implemented this, first check that the scene was loaded correctly, and that all objects are in free-fall when animating.

1.2 Implementing collisions

The main function implementing the simulation loop is `Scene::update_scene()`, which is already implemented. It will do the following steps:

1. Integrate velocity and position for all meshes (calling `Mesh::integrate()` that you just implemented).
2. Try to detect collisions for all pairs of meshes. If a collision is detected, it will resolve the collision (see below).
3. Constraint handling (unlocked in Section 2).

Collision detection (within `Mesh::is_collide()`) is done in two steps: the medium phase checks the intersections of bounding boxes by calling the function `Mesh::is_box_collide()`. If positive, it runs the small-phase GJK algorithm. This algorithm is only guaranteed to work on convex objects! We limit our CW to such objects. This is already implemented entirely; your job is to implement `Scene::handle_collision()` according to what you learned in class.

Resolving interpenetration linearly The input to `Scene::handle_collision()` is the contact between two meshes m_1 and m_2 that are already interpenetrating. This is parameterized in three variables: d (as `depth`) is the depth of the interpenetration along the normal $\vec{n}_{21}$ (as `contactNormal`), which is the normal vector from m_2 to m_1. That is, you need to move m_2 in the positive direction of $\vec{n}_{21}$, and m_1 in the negative direction. Finally, p_2 (as `penPosition`) is a point on mesh m_2 which is closest to m_1. To resolve interpenetration linearly, you should compute the inverse mass weights:

$$w_1 = \frac{m_1^{-1}}{m_1^{-1} + m_2^{-1}},\ w_2 = \frac{m_2^{-1}}{m_1^{-1} + m_2^{-1}},$$

where m_1^{-1} is `m1.invMass` and respectively for m_2^{-1}, and then set

$$COM_2 := COM_2 + w_2 \cdot d \cdot \vec{n}_{21},\ COM_1 := COM_1 - w_1 \cdot d \cdot \vec{n}_{21}.$$

Note that fixed objects have $m^{-1} = 0$ which will cause the other object to move the entire way back. The new (single) contact point is $p_{12} = p_2 + w_2 \cdot d \cdot \vec{n}_{21}$ and we use it next.

Resolving velocities The second part of `Scene::handle_collision()` is to implement the impulse-based linear and angular velocity correction that is learned in Lecture 9. Essentially, you will compute an impulse $j\vec{n}_{21}$ that will be applied to both closing velocities to compute separating velocities. Both closing and separating velocities are total velocities at the meeting point p_{12} of both objects. They should be computed as in the lecture by combining linear COM velocity and angular velocity. For this, you will need to consider the arms $\vec{r}_1$ and $\vec{r}_2$ from the respective COMS to p_{12}, and the *rotated inverse* inertia tensors I_1 and I_2. Recall that the original tensors are computed in world coordinates around the respective COMs; thus you need to rotate them to fit the current orientation, as explained in Lecture 3. Finally, you should employ the coefficient of restitution (as `CRCoeff`), which models the elasticity of an impact, to compute the impulse $j\vec{n}_{21}$. *Important note:* You must implement the rotation of the inertia tensor through the function `Mesh::get_curr_inv_IT()`, which the grader will also use.

Remember, you are only to update COMs and velocities; the end of the function will broadcast the update to `Mesh.currV`.

Having completed `Scene::handle_collisions()`, you will have finished the basic version, and can run the simulation fully. Check all files that are marked as `"<name>-scene.txt"`.

Grading: Grading simulations is problematic because small numeric differences can make big changes locally and through time (for instance, a closely-missed collision). Thus, our automatic graders (subproject `Grading1`) will only unit-test both `Mesh::integrate()` and `Scene::handle_collisions()`. It is likely that if you implemented them correctly, the entire simulation of the basic step would be correct. The automatic grader for this section will constitute 20% of your grade; an extra 10% percent will be given by the marker for running the simulation and seeing that there are no further possible issues missed. Finally, 10% will be graded on a report of your (concise) insights detailing the issues you find with the simulation, in terms of the disadvantages of correct behaviour and why they happen. Include imagery and explanations of what you see. The grader will take all default options and therefore should not invoke any extensions (you should take care that any such extensions, in the next Sections, only work with non-default arguments). In total, this section is worth 40% of your grade.

Coding advice: We use the `Quaternion` class to do quaternion computations. Especially note `QExp()` for quaternion exponent (for the orientation integration from angular velocity), and `Q2RotMatrix()` to obtain the equivalent rotation matrix, which is needed to transform the IT.

2 Extension: working with distance constraints

Constraint representation We next augment our simulation with flexible distance constraints; that is, we consider a set of *holonomic* constraints $C(p_i, p_j)$

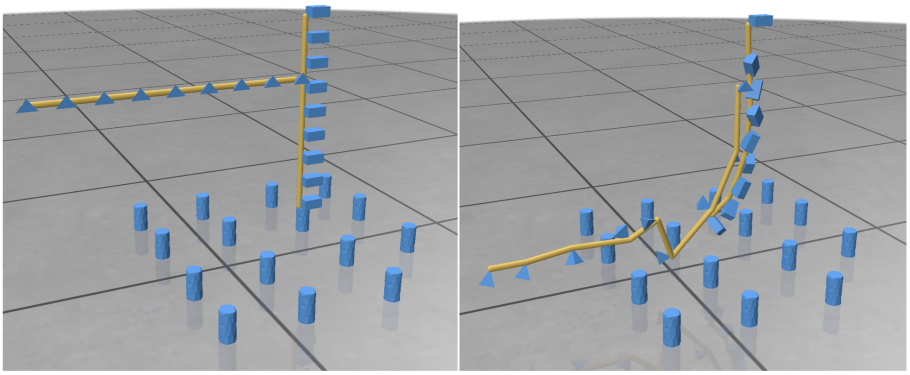

Figure 2: Simulation with distance constraints (left then right). This is `two_chain-scene.txt` with `two_chain-constraints.txt`.

such that each seeks to preserve the distance d_{ij} between vertex p_i in mesh m_i to that of p_j in mesh m_j, when measured in the initial translated and rotation configuration (that is, in `currV` at $t = 0$). Constraints are encoded in a file of the type `X-constraints.txt`, with the format:

```
#num_constraints
...
mi vi mj vj lower_bound_ij, upper_bound_ij
...
```

for each such constraint, the lower bound l_{ij} and the upper bound u_{ij} are numbers $[0, 1]$ that denote the flexibility of a constraint. Specifically, every such constraint is implemented as two inequality constraints of the following functions:

$$C_l(p_1, p_2) = |p_1 - p_2| - l_{12}d_{12} \geq -\tau$$
$$C_u(p_1, p_2) = |p_1 - p_2| - u_{12}d_{12} \leq \tau$$

for some small tolerance τ. in case $l_{12} = u_{12}$, the two constraints implement a rigid distance constraint.

The constraints in the simulation loop. Upon initializing the scene with a non-empty constraints file, it "unlocks" the last stage of the simulation loop, after having resolved collisions. The action takes place in `constraint.h` in which you have to implement this section. The algorithm (which is already coded in `Scene::update_scene()`) is as follows:

1. Check all constraints sequentially. If the entire sequence was valid without change, stop.
2. When a constraint is not valid, call `Constraint::resolve_position_constraint()` to correct its position and orientation, then `Constraint::resolve_velocity_constraint()` to resolve its velocities.

3. If we passed `max_iterations`, the algorithm stops anyhow (the constraints were not fully resolved). You'll see this is actually common, but that visually it might not make much difference.

In fact, for encapsulation, the resolving functions are the ones performing the validity test and return either `positionWasValid` or `velocityWasValid` if they didn't have to change anything. Your task is to complete both functions `resolve_position_constraint()` and `resolve_velocity_constraint()`. These resolution (from "resolve") functions *only* provide updates to COM and velocity, and *not* to individual vertices, which is done as post-process.

Resolving positions Each call to `resolve_position_constraints()` resolves the COM position of a single constraint. It receives the two COMs of meshes m_i and m_j (as argument `currCOMPositions`), the current location of the constrained p_i and p_j (as `currConstPositions`), and returns their corrected versions. The class `Constraint` is initialized with the reference threshold distance (as `refValue`), either $l_{ij}d_{ij}$ for $C_l(p_i, p_j)$ or $u_{ij}d_{ij}$ for $C_u(p_i, p_j)$, both inverse masses m_1^{-1}, m_2^{-1} (as `invMass1|2`), a flag `isUpper` which is `true` when this is a C_u type constraint, and a `tolerance` value for τ. Your function should first check if the constraint is already valid, and if so return `positionWasValid` and just copy back the inputs `currCOMPosition` to `correctedCOMPositions`. Otherwise, you need to find a correction term $\pm\Delta p$ to the COMs of each mesh to return the constraint to threshold validity with the projection method shown in Lecture 10. You should return `correctedCOMPosition` with the corrected COMs for both meshes.

Note that linear collision resolution is just a special case of this. Technically, we could have included collision as a temporary constraint, but this would make the separation to basic and enhanced components of the CW complicated, so we leave it separate. A side effect is that the constraints can sometimes generate visible new collisions that will only be handled in the next time step.

Resolving velocities This is the more involved of the two, since you need to resolve both linear and angular velocity. The formulation for doing so is entirely in Lecture 9. Essentially, you need to create a gradient vector $J_{12\times1}$ for the constraint, and a velocity vector $v_{1\times12}$. If $J\cdot v = 0$, the constraint is initially valid, and you should return `velocityWasValid=true`, with the original velocities. Otherwise, you should find a correction vector Δv such that $J(v + \Delta v) = 0$, and use it to correct velocities, returned in `correctedCOMVelocities` and `correctedAngVelocities`. For this, you should also be using the (already rotated in the input) inertia tensors given by `invInertiaTensor1|2`.

Important note: this function is only called if the position constraint was violated. Note further that this is also a more general case of velocity correction for collisions, for the same arguments as for position correction.

Grading: The automatic grader (subproject `Grading2`) will unit-test both `resolve_position_constraint()` and `resolve_velocity_constraint()`. An

almost-zero result on the automatic grader is worth 20% in grade, with 5% graded on a plausible running of the simulation visually, and 5% for a correct summary of the results in the report; discuss the artifacts that you see, and the performance of the algorithm. This section is then worth 30% of your grade.

3 Scalable and efficient Rigid-body simulation

This section is a more advanced extension of the practical, which is worth the final 30% of your total grade. It will not be checked automatically, but rather entirely depend on your demonstration and explanations in the report.

You will have noticed that the constraint and collision system doesn't scale very well with many rigid bodies in the scene; our scenes are rather toy, but what will happen if you throw 1000 objects in? For a more refined result, you should implement two measures that will significantly reduce computation time:

1. Broad-phase space subdivision that will allow us to only check for the intersection of two meshes that are already close enough in space. There are several ways to do this, and you should research what's available and easy to integrate in your environment.

2. A more efficient constraint scheduling; rather that go through the entire streak over and over, have a system where the resolution of a constraint only flag up other constraints that are affected (meaning using the same mesh), and iterate sparsely.

4 Submission

You should submit the following:

- A report that must be at most 3 pages long including all figures, according to the instructions above, in PDF format.
- For Sections 1 and 2, *only* submit `mesh.h`, `scene.h` and `constraint.h`.
- For Section 3, submit any supporting files in case you made extensions that you would like to demonstrate.
- Any extra data files of scenes or constraints if you need them.
- You are encouraged to submit videos of your simulation with specific close-ups to demonstrate specific effects, and refer to them in the report.

The submission will be in the official place on Learn, where you should submit a single ZIP file of everything.