Game Physics – Programming Exercise

Exercise 2 – *Rigid Bodies*

Task Overview

In this exercise, first take pen and paper and manually calculate one time step for a single moving rigid body (setup in Table 2.1) with the Euler method.

Then, you can turn to implementing it. Switch to the Ex2 branch of the template project (https://github.com/tum-pbs/gamephysicstemplate). **Make sure to pull again from the upstream repository because we updated branch Ex2 to work with VS 2022**. Based on the given header file, implement your simulator for 3D rigid body systems. Set up the test case in Table 2.1, and check the one-step results with your manually calculated results.

Afterward, implement **collision detection** and **collision response** methods. Set up a two-box scene to show a corner-to-face collision between them. Finally, show off your system with multiple rigid bodies and interactive external forces and collisions.

Details of the submission, i.e., the deliverables, are listed below.

Table 2.1, Setup of a scene with a single rigid box

Box center:$\mathbf{(0,0,0)^T}$, size:**(1, 0.6, 0.5)**, mass: **2**,

Initial orientation: **rotated around Z by 90 degrees**

The initial linear and angular velocity are both **zero**.

Simulate the rigid body for 1 step with an external force $\mathbf{f = (1,1,0)^T}$ applied at world space position $\mathbf{(0.3,\ 0.5,\ 0.25)^T}$. Compute the resulting linear and angular velocity and the world space velocity of point $\mathbf{(-0.3,\ -0.5,\ -0.25)^T}$. (Ignore that the force position is not on the body.)

Demo requirements:

Your submission should contain the following demos and should support **interactive switching between these demos**:

- **Demo 1: A simple one-step test**
  - Manually calculate the solution to the single rigid box setup from Table 2.1, with the parameters given there (no need to submit this), for an Euler time step of size h=2.

- According to the class definition in the “RigidBodySystemSimulator.h” file, implement your “RigidBodySystemSimulator” class to compute a solution for the same problem with your simulator.
- The following steps are necessary to add the rigid bodies to the Visual Studio project: In Visual Studio, set the “simulationsRunner” as the startup project. In main.cpp, replace “#define TEMPLATE_DEMO” with “#define RIGID_BODY_SYSTEM” at line 23. Add the “PublicRigidBodiesTests.cpp” file into the “SimulationsTester” project. Then you can run and test your RigidBodySystemSimulator class.
- Build and run the basic test case, and print your solution (i.e., linear and angular velocity of the body and the world space velocity of point $(0.3, 0.5, 0.25)^T$) after one time step to the command line. Verify your manual calculation.

- **Demo 2: Simple single-body simulation (sample video on Moodle page)**

  - Simulate the single rigid body with a smaller time step $h = 0.01$, for multiple steps.
  - Add extra forces on the body through the mouse or keyboard and display the simulation interactively for longer. Verify as much as possible that it gives the correct behavior.

- **Demo 3: Two-rigid-body collision scene (sample video on Moodle page)**

  - Set up your scene with two rigid body boxes. Their initial linear velocities (no angular velocities) should make them collide. Ensure that only one rigid body’s corner collides with the other on its face.
  - Manually calculate your collision test case by hand. (Choose a test case with simple values!) Calculate the collision point's world position, the collision face's normal, the velocity difference between the two boxes at the collision point in world space, and the impulse J.
    Update the velocity and momentum of the two boxes according to J as described in the slides.
  - Add support for collision in your simulator. Use the given collision detection function to check if there is a collision. Based on the returned data, implement your collision response method by calculating the impulse and updating the momentums.
  - Run your two-box collision scene and validate it with your calculation. You do not have to submit your manual calculation. Your demo should show the two bodies colliding and separating correctly after the collision.

- **Demo 4: Complex simulation**

  - Set up a simulation with at least four boxes.
  - Provide interaction methods, e.g. by including extra forces. You can also add a ground floor (or walls) and enable gravity so that all rigid bodies will collide with it. (Note that bodies will not fully come to rest with this basic simulation algorithm.)

Submission

The deadline for this exercise is on **10 December, 23:59**.

Please pack all your source files (.h and .cpp) and project files (.vcxproj) under the “Simulations” directory into a zip file, name it “Group??_Ex2_VS20??.zip”, and upload it in the corresponding Moodle submission activity. Ensure not to include the compiler temporary files (they will be under the Simulations/Win32/ directory). Your package should be smaller than 100kb.

Recommendations & Tips:

1. **Orientation**. Use quaternions to represent the orientation of the rigid bodies.
2. **Integration.** Use explicit Euler integration in time. Refer to the “Simulation Algorithm 3D” from the rigid body lecture slides.
3. **Quaternions.** The update rule for the rigid body orientation does not conserve unit length. Don’t forget to re-normalize the quaternions manually after each update. Also, remember that you can always convert back and forth between rotation matrices and quaternions. E.g., it is easier to specify a given rotation as a matrix, and for rendering, you should also use the matrix representation. (An excellent way to debug quaternions is applying their corresponding rotation matrices to simple objects, such as a long box, and rendering them since quaternions are complex to interpret by inspecting the raw vector values. In this way, you can visualize the rotation they cause.)
4. **Inertia Tensors.** Analytic formulas for inertia tensors of ordinary objects like boxes or spheres can be found online, e.g., at http://en.wikipedia.org/wiki/List_of_moment_of_inertia_tensors.
5. **Interaction**. You can add extra forces, e.g., by dragging the mouse. Gravity forces may not be used for Demos 1 to 3 and are optional for Demo 4.
6. **Collision Detection**. You can find a simple function for collision detection in the Simulations folder (collisionDetect.h). You can find an explanation of that function on the last page of this exercise (p. 5). In this function, the separating axis theorem checks whether two boxes, A and B, are overlapped. If an overlap occurs, one collision point and one collision normal will be returned. It is a simple detection method and only returns one collision point. The limitations of this method are also explained on the last page.
7. **Render a Rigid box**. You can use the following lines to draw a rigid box.

```
DUC->setUpLighting(...); // set the lighting, like in the mass-spring exercise
DUC->drawRigidBody(GamePhysics::Mat4 transformation);
//input matrix is the transfer matrix from the object space of the box to the world space
```

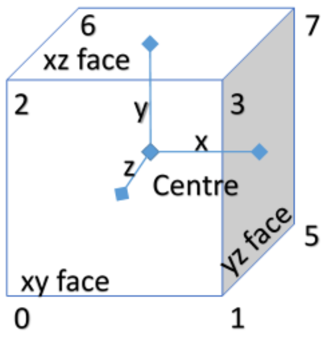
   Here is an example.

```
DUC->setUpLighting(Vec3(0,0,0),0.4*Vec3(1,1,1),2000.0, Vec3(0.5,0.5,0.5));
BodyA.Obj2WorldMatrix = BodyA.scaleMat * BodyA.rotMat * BodyA.translatMat;
DUC->drawRigidBody( BodyA.Obj2WorldMatrix );
```

8. **Left-handedness**. To work with DirectX, our matrix class works with left-handedness. That means we use row-major matrices, row vectors, and pre-multiplication. E.g., if you want to transform a vector, you should call v = Mat.transformVector(v). This will do the calculation of v = v * Mat . You will get the wrong result if you do v = Mat * v. For another example, the object to world matrix should be scaleMat * rotMat * translationMat instead of the other way around (translationMat * rotMat * scaleMat). Quaternions have no handedness, but they can be transferred into row-major (left-handed) matrices or column-major (right-handed) ones. Our Quat::getRotMat() will give you the left-handed matrix.

Collision Detection in 3D for boxes AB

- The method in collisionDetect.h:



```
CollisionInfo checkCollisionSAT (Mat4& obj2World_A,
Mat4& obj2World_B)
```

This method checks if A and B are overlapped using the Separating Axis Theorem.
obj2World_A, **obj2World_B**, are the transfer matrix from object space of A/B to the world space, including the rotation, the scaling and the translation.

- Data returned by the method:

```
struct CollisionInfo{
            bool      isValid;
            Vec3      collisionPointWorld;
            Vec3      normalWorld;
            float     depth;
}
```

- ○ **isValid:** whether a collision was found, true for yes
- ○ **collisionPointWorld :** the position of the collision point (one vertex of B) in world space

  - **normalWorld:** $\boldsymbol{n}$**,** the normalized direction of the impulse from B to A. For e.g., when one vertex of B is knocked into a face of A, the $\boldsymbol{n}$ will be negative of the face normal of that collision face.
  - **depth**: how far the collision point is inside A. It is a redundant value which should not be used in your impulse calculation
  - If **no collision** is detected, the method will return {false, Vec3(0.0), Vec3(0.0), 0.0}.

- Once you find a collision, your collision response function should calculate:
  - $\boldsymbol{v}_{rel}$, the relative velocity between A and B at the collision point (in world space).
  - If $\boldsymbol{v}_{rel} \cdot \boldsymbol{n} > \boldsymbol{0}$, this indicates that the bodies are separating.
  - Otherwise continue to calculate the impulse J, and apply it to both bodies.

- A simple example function, “testCheckCollision”, can be found at the end of collisionDetect.h

- Limitations:

  When checking collision on edges, edges are discretized into several points. Therefore, edge-to-edge collisions may be slow and not entirely accurate.