Physics Cumulative Submission

Patrick O' Halloran

March 6, 2016

Student Number:	14301692
Module ID:	CS7057

1 Particle Systems

In my implementation there are three forces - gravity, drag and wind. There are three different integration methods implemented - Euler, RK2 (Midpoint method) and RK4.

```
return 0.5f * env.fluid.density * surfaceArea * dragCoefficient
    * (vel - (env.wind)).length() * -(vel - (env.wind));
```

Listing 1: The function calculating wind and drag

The particles can collide with a plane using the post-processing / projection method. The bounciness of the particle is governed by the coefficient of restitution. It is also possible to rotate the plane at run-time. The particles can be recycled when they collide with the plane, or when they reach a certain age.

To create a snow effect I changed the particle emitter to be an area instead of a single point (See Figure 1). The particles are given random initial velocities facing downwards.

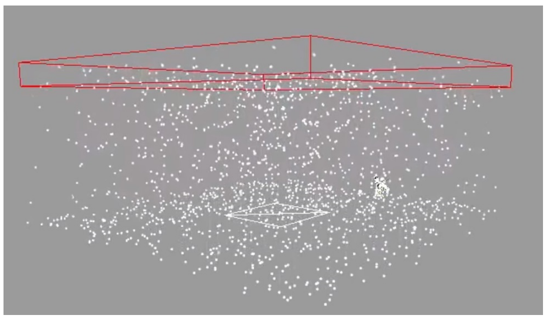


Figure 1: Snow effect

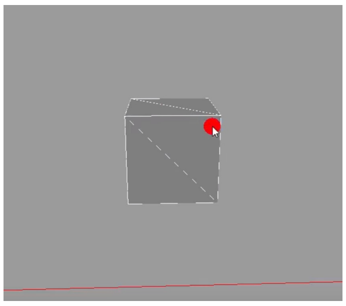


Figure 2: Impulse with mouse

2 Rigid Body Unconstrained Motion

Rigid bodies extend on particles by having an orientation. Other new variables of note are mass, centre of mass, angular velocity and momentum, and torques.

When the rigid body is created the inertial tensor is generated using cube approximation. The inertial tensor is analogous to mass when considering rotations, as are torques to forces. The inertial tensor should not change during the program. Assuming the mass of the object stays constant, the tensor just needs to be rotated.

```
glm::mat3 getIntertialTensor()
{
        return glm::transpose(glm::toMat3(model->worldProperties
            .orientation)) * glm::inverse(inertialTensor) * glm
            ::toMat3(model->worldProperties.orientation);
}
```

Listing 2: Getting the inertial tensor for the physics update

The user can make an impulse using the mouse as seen in Figure 2. The position of the cursor is found in 3d space (the red ball in the image) and the impulse is applied to that point by calling *ApplyForce* and passing it the point and a force vector. The resulting force and torque are added to *forceNet* and *torqueNet*, which are later used for integration.

The orientation is kept orthonormalised to avoid numerical drift.

3 Broadphase Collision Detection

The broadphase culls trivially non-colliding pairs to combat the all-pairs weakness of a brute force approach. I chose to implement the sweep and prune algorithm, which uses Axis Aligned Bounding Boxes (AABB).

When a rigid body is created an AABB is generated that fits the model of the rigid body. AABBs vary with rotation so if there is a change in the rotation of the object I recalculate the AABB. The vertices of the bounding box

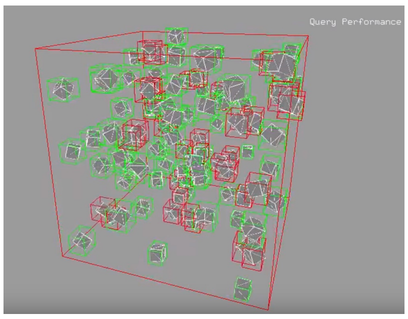


Figure 3: Sweep and prune in action

are rotated and used to calculate a new bounding box, instead of using the full model. This speeds up the calculation.

The sweep and prune algorithm maintains a sorted list of endpoints along each axis. I implemented an insertion sort to keep them sorted, as it is very fast on nearly sorted lists, which exploits temporal coherency in the scene.

I also implemented bounding spheres and a brute force check for colliding pairs. This is considerably slower than sweep and prune, especially when there are many objects, like in Figure 3.

4 Narrowphase Collision Detection

The narrowphase receives a list of potentially colliding pairs from the broadphase stage. I implemented the GJK algorithm to detect if the pairs were actually colliding.

The goal of the GJK algorithm is to find a 3-simplex (Tetrahedron) from the points of the Minkowski Difference that can enclose the origin .

I flag the colliding objects with a blue wireframe (Figure 4). I also drew the simplex output by the GJK algorithm for debugging purposes. In my program I use boxes but the implementation can support any combination of convex polyhedra.

5 Collision Response

For collision response I implemented the Expanding Polytope Algorithm (EPA). This algorithm naturally extends from GJK, taking the simplex that GJK terminated with as its input.

EPA iteratively expands the simplex, adding new faces. It is searching for the closest face (or within some threshold distance) on the convex hull of the

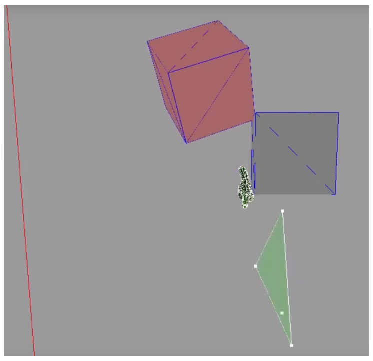
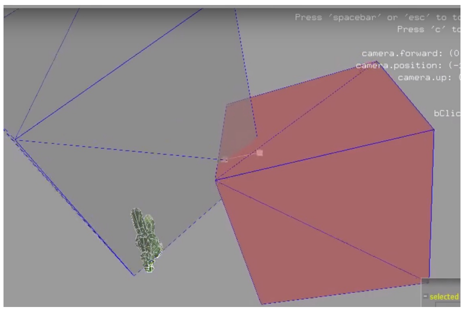


Figure 4: GJK detecting a collision



```
Press 'spacebar' or 'esc' to to
                    Press 'c' to

      camera.forward: (0
    camera.position: (-
           camera.up: (

                       bClic
```

Figure 5: I allow the objects to intersect to show the contact points and penetration vector

Minkowski Difference.

The penetration vector is given by the normal of the closest face on the convex hull. The contact points are found by projecting the origin on to the closest face. Both are shown in Figure 5.

Finally an impulse magnitude is calculated and applied in the contact normal direction.