Model shattering using real-time collisions
Final Report

Conner Weatherston *
Edinburgh Napier University
Physics-Based Animation (SET09119)

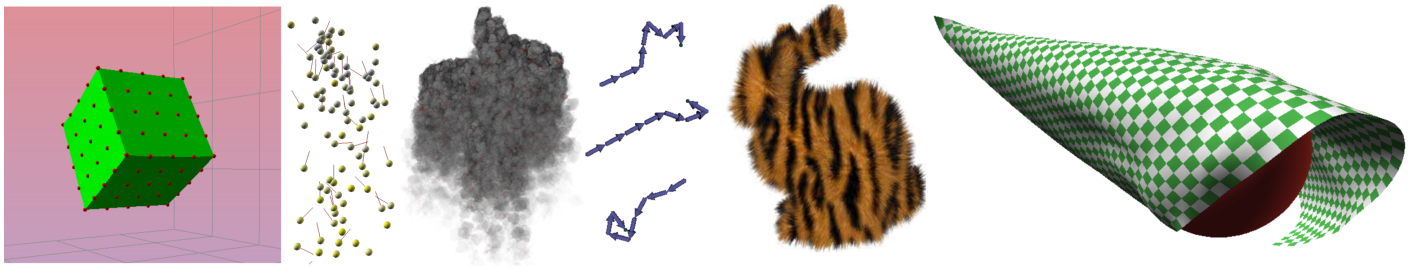

Figure 1: *Place a teaser image at the top of your report to show key examples of your work (e.g., multiple screenshots of the different test situations) - Every figure should have a caption and a description. For example, each figure is labelled and explained: (a) soft bodies, (b) particles, (c) inverse kinematics, (e) fur shells, and (f) position-based dynamics for cloth effects.*

Abstract

In most physics based games, destructible objects are typically loaded in with their shattered fragments on run-time. This means that sometimes the fracture look unnatural. The aim of this physics simulation is hoping to look at is ways to slice meshes in real time to generate new fragments.

The approach looked at the in the design report suggested a technique called voronoi shattering in order to break the objects up into random fragments. However this was not implemented in the final simulation due to difficulty of producing 'Delaney Triangulation' and time constraints. Therefore a simpler approach was used where the object is split is recursively using a series of random planes which intersect through the model mesh.

Keywords: Rigid body, collision detection, bounding volumes, mesh destruction, OpenGL, real-time, physics, computer graphics

1 Introduction

In most video games, destructible objects are pre-computed models which are the equivalent of the model fragments. The purpose of this simulation is to create proper mesh fragments in real-time just using the original meshes information. A key part of this simulation is optimisation as creating new objects is an expensive process.

2 Related Work

Currently there are several plugins that use either the same idea or has a similar solution. However these are mostly implemented into modelling software where optimisation is not an issue as it is not required in a real-time environment. The most similar idea is the plugin by [Esteve 2011]. The tutorial provided gave a insight on how to split the meshes by using planes.

*e-mail:40167111@live.napier.ac.uk

3 Simulation

Overview

The simulation is revolved around having objects in a real-time environment interact with each other. In this simulation, the objects have been kept to cube or cuboid shapes for simplicity. There is also a plane which acts as the 'floor' of the scene. When objects collide with each other they respond in a realistic manner. In order for a collision to cause an object to break a certain threshold must be reached. If this limit is reached for the object then it will split apart depending on its splitting planes, which determine how the model will shatter.

Implementation The implementation of this project can be identified by three major components. These are: rigid bodies, collision detection and meshes.

Collision detection

It is incredibly inefficient to test meshes directly against each other as this would require testing each vertex of object A against every vertex in object B. Therefore bounding volumes are required which are there to provide, simpler tests that are not costly in performance. The downside to this is the response may not be accurate when a collision is detected depending on what collider approach is used. In this simulation three types of colliders have been utilised. These are the plane collider, the sphere collider and the orientated bounding box (shortened to OBB).

As this simulation is focused on cubes the most suitable collider is the OBB as it fits well with the objects. However a sphere collider is attached in order to improve performance. Instead of testing for an OBB to OBB collision each tick, it is cheaper and faster to check two sphere colliders, thus if two sphere colliders are in contact then an OBB OBB intersection check shall be performed.

Collider

This is an abstract class which all colliders will inherit from. As every collider requires a position and an update method it has been initialised here.

```
abstract Class Collider
{
        Vec3 position;

        // This is used to update the position of the colliders.
        virtual void Update(double deltaTime);
}
```

Plane collider

The plane collider is used to simulate the floor. However as the floor is not meant to have true rigid body properties. i.e it should not 'fall' as it will never move, therefore there is a plane collider and a graphic drawn for it rather than it being a true 'Model' object in the scene.

```
Class PlaneCollider : public Collider
{
        Vec3 normal;
}
```

Sphere collider

The sphere collider encompasses the model in a sphere shape. In order to do this a new variables needs to be held. This is the radius of the sphere.

```
Class SphereCollider : public Collider
{
        float radius;
        void Update(deltaTime){}
}
```

Orientated bounding box

This is a stage further in complexity to an axis aligned box, however it can handle rotations. As there are more checks that a required for an OBB to OBB intersection it is important to only check when it is necessary to, and not every tick. In order to create an OBB the corners are required. To reduce the memory required to store an OBB, only two opposite corners are stored, as the other corners can be produced using a combination of these two corners.

```
Class BoundingBox : public Collider
{
        float topX, topY, topZ;
        float botX, botY, botZ;
}
```

The rotation of the box is done in the intersection check. The rotation is taking from the orientation of the rigid body. This will ensure that the object stays inside the bounding volume.

Intersections

Rigid bodies

In order to have a simulation which simulates proper physics, rigid bodies need to be attached to the objects. An rigid body consists of the following: position, previous position, mass, inverse mass, orientation, forces being applied to the object.

```
Struct RigidBody
{
        double mass;
        double inverseMass;
        dvec3 position;
        dvec3 previousPosition;
        dvec3 forces;
        dvec3 torques;
        dquat orientation;
}
```

Meshes

The mesh information needs to be stored and cannot be put straight onto buffers. The reasons for this is the vertex positions of each triangle are essential when splitting the geometry. This information is stored in a 'ModelInfo' struct, which holds the geometries vertex position, texture coordinates, normals and colours.

In order to split a mesh a splitting hyperplane is required. This plane separates the geometry into two segments, One which is the 'new' fragment and the other is the new updated original model which has been fractured.

In the case of the plane splitting the cube, or cuboid on an individual axis (x,y or z), then the following procedure is followed:

A Triagle is a class that has been created that will store 3 vertices, which is the same as one of the triangles rendered on the model.

Data: Model to be split, plane to split by
Result: New Model
Vector3 fragment vertices.
for *each Triangle in ModelInfo positions* **do**
  Integer pointsInfront = 0;
  for *each point in Triangle* **do**
    if *dot(TrianglePoint, planeNormal) - dot(planePoint,planePoint) > 0* **then**
      pointsInfront++;
    end
  end
  if *pointsInfront == 3* **then**
    Add triangles points to fragment vertices;
  end
  if *points == 0* **then**
    Add closest points to be plane to fragment vertices;
  end
  if *points == 1* **then**
    God help us all;
  end
  if *points == 2* **then**
    God help us all a little more;
  end
end

Algorithm 1: New fragment creation

The pseudo-code for finding the vertices for updating the original model except, the comparison between the points and the plane is flipped, thus the points should be behind the plane instead of infront of the plane.

Update

To create an realistic simulation a concept of a 'physics tick' is required. This is in place in order to decouple the physics from the frame-rate. Thus the physics will run the same if the frame rate is at 10 FPS or 60 FPS.

There are three parts to the physics update. There are summerised below:

Colliding object The first stage is detecting if any objects are currently colliding with each other. If there is a collision between two objects then a struct called 'CollisionInfo' is created which contains the following data:

- Pointer to object A.
- Pointer to object B.
- Depth of overlap in objects.
- Direction to be pushed towards.
- Position of collision.

This struct is then pushed back onto a vector; which contains all the collisions that have happened between the models in this current update.

Resolve collisions

For each of the collisions that have occurred a resolve stage needs to occur. This calculates what should happen to each object in terms of direction it is now going; the velocity of the objects and if the proper conditions have been met, one of the objects shatter.

Integrate rigid bodies

After collisions have been sorted, each model need to update their current position, previous position, orientation.

4 Testing and evaluation

In order to test the splitting a meshes in this simulation a series of hard coded planes were tested against. The results were already known thus is was just a simple comparison between the expected results and the actual results. If there was a discrepancy then there was an issue with mesh cutting algorithm. The first error was noticed when the split was done in the proper shape however some of the triangles were missing from the polygon. It turned out using rays to try and calculate where the new points should lie if there was a plane intersection would lead to incorrectly sized triangles. This would have been rectified by triangulating the model. However An easier solution was the one presented earlier; where finding the closest point to the plane proved not only simpler, but more accurate as well.

The collisions were done in two stages. The first stage was testing that the colliders recognised that they were in contact with each other. This was tested by having a message print to console when two colliders contacted and the message specified which colliders were in contact.

The following stage, which was resolving collisions, was more difficult. The only way the test if the algorithm put in place worked properly was to run it several times and used different conditions to ensure that the colliders responded in a sensible manner. What was noticed from this was an OBB collider had issues sitting on other objects. If it was ontop of a plane collider it would keep rotating, If it was on top of another OBB collider it would sink into the object before coming back out and starting jumping on top of it. This was resolved by...

5 Conclusion and Future work

To conclude, I am done.

Future work on this project can be undertaken. The next step would be creating convex shapes instead of just cubes and cuboid, once this has successfully undertaken then it would be possible to undertake the voronoi shattering technique that was originally suggested as this would provide the planes that the model would be split by. In order to create the convex shapes the current way of splitting the mesh should be sufficient, the difficulty would be triangulating the mesh in order for it to be displayed properly.

References