Mesh Destruction
Final Report

Students Name *
Edinburgh Napier University
Physics-Based Animation (SET09119)

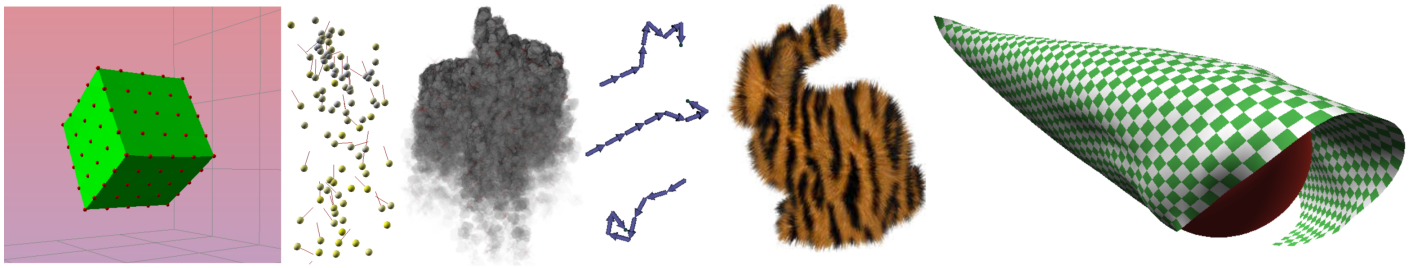

Figure 1: *Place a teaser image at the top of your report to show key examples of your work (e.g., multiple screenshots of the different test situations) - Every figure should have a caption and a description. For example, each figure is labelled and explained: (a) soft bodies, (b) particles, (c) inverse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

Abstract

In most physics based games, any destructible objects are normally loaded in with their fractured components on runtime. What 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 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.

3 Simulation

Overview

Detailed description

Implementation The implementation of this project can be broken into three major segments. These are: rigid bodies, collision detection and meshes.

Collision detection

There are three types of colliders that are used in this simulation. These are the plane collider, the sphere collider and the orientated bounding box.

Collider

This is the basic collider, which every other collider will inherit from. As it is the most general form it only contains a position.

```
Class Collider
{
        Vect3 position;

        // This is used to update the position of the c
        void Update();
}
```

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;
}
```

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

Sphere collider

This is a simple sphere which encompasses the entire model object. It only requires a position in the world, which is updated whenever the rigid body is updated, and a radius. The radius is calculated by finding the distance from the centre of the model to its furthest point.

```
Class SphereCollider : public Collider
{
        float radius;
}
```

Orientated bounding box

This is similar to an axis aligned box however it can handle rotations. Detecting collisions for an OBB however is more cost as there are more potential ways two OBB can collide. Therefore, in order to minimise calculations, the OBB checks only occur if both object pass the sphere collider check.

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

With the 6 points provided the corners of the box can be easily computed.

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.

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.

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.

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

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

DAY, R., AND GASTEL, B. 2012. *How to write and publish a scientific paper*. Cambridge University Press.

SAKO, Y., AND FUJIMURA, K. 2000. Shape similarity by homotropic deformation. *The Visual Computer 16*, 1, 47–61.