Research and Implement of Rigid Body Fracturing Simulation Based on Ogre and Newton Engine

Jiangfan Ning
School of Computer Science
National University of Defense Technology

Shi Lu
School of Computer Science
National University of Defense Technology

Bo Wu
School of Computer Science
National University of Defense Technology

Sikun Li
School of Computer Science
National University of Defense Technology

***Abstract*-Recently, the fracture simulation of rigid body has gained more and more attention in the area of computer graphics for its widely use in movies, computer games and military simulation. With the amalgamation of graphic rendering engine and physics engine, a fracture simulation framework of rigid body can be constructed fast and easily. The basic concepts and principles of OGRE and Newton game dynamics are introduced and the OgreNewt is adopted to bind the two engines. At last, the implementation of rigid body fracture simulation is given and the results demonstrate that our framework is robust and efficient.**

Keywords-OGRE, Newton Game Dynamics, Fracture simulation, Rigid body

I. INTRODUCTION

Fracture is a familiar phenomenon to us and appears in many popular movies and games, e.g. the splashing debris caused by explosion in the magnificence war scene in the movie Avatar and the various fracture special effects of various materials in the latest 3D game Star Wars: The Force Unleashed produced by Lucas Arts. With the rapid development of graphics hardware, people are no longer satisfied with the traditional simple, stiff and artificial effect of fracture.

In this paper, we designe a completely real-time rigid body fracturing simulation architecture based on open-source graphics rendering engine Ogre combined with a fast physical system Newton Game Dynamics. With this simulation architecture, we implement a prototype system of rigid body fracture simulation. The results can meet the requirement of reality and real-time interaction.

II. KEY TECHNOLOGY

A. Ogre engine

1) Introduction

OGRE stands for Object-oriented Graphics Rendering Engine. It is developed using C++. It's an object-oriented 3D rendering engine and is flexible to use. Its purpose is to allow developers to develop hardware-based 3D applications or games more easily and directly. The class library abstracts the details of more underlying libraries (Direct3D and OpenGL), and provides an interface based on real-world objects and other classes. The benefits of characteristics of object-oriented of Ogre are: abstraction, encapsulation, and polymorphism [1].

Ogre is a large and confused collection of objects and modules, because of its completely object-oriented design, most of the details have been hidden in the mature hierarchy of structure. Just a simple call can achieve very brilliant features. The object-oriented framework of Ogre provides all the object models for the rendering process. Rendering system abstracts the complex and different underlying API function into a unified operating interface; the scene graph also becomes another set of interfaces, and different scene management algorithm is allowed to adopt; all the renderable objects, whether dynamic or static, are abstracted as a set of interfaces, to be called by specific rendering operation; movable objects provides a set of common interfaces to accept a variety of operating methods.

2) Core object

Figure1 shows the core objects of OGRE and their interconnected relationships [2].

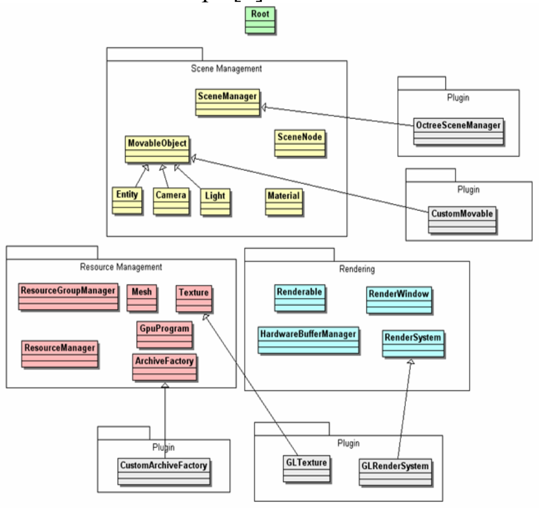

Figure 1. A simple UML of OGRE

Top of the chart is the root object. It is the portal of the OGRE system, and is used to create all the basic elements, such as: Scene Managers, Rendering Systems, Render

Window and Loading Plugins. Root object is the beginning of everything in Ogre, it provides user with the core object to achieve specific functions, so the root object is more like an organizer of Ogre. The rest classes of Ogre can be divided into the following three roles:

① Scene Management: Scene Management describes the content included in the scene and how they are organized together. It also provides the interface closest to the natural for user to call.

② Resource Management: A dedicated group of classes for user to load, reuse or unload the resources for rendering the terrain, texture, front and all other objects.

③ Rendering: Rendering is also a subsystem of the Scene Management class. It organizes all the objects relate to rendering and provides simpler and high-level interfaces for user to call. It includes the low-level API objects, such as the rendering pipeline, all the cache transferred to the pipeline, and rendering state, etc.

3) Rendering framework

A typical Ogre rendering framework can be described with the following pseudo-code:

```
create AppExample
  call AppExample go method{
     create root example
     load resource;
     select scene manager;
     create camera;
     create viewport;
     create scene;
     create FrameListener;
     }
   call root example loop render method{
     While(!quit){
     ...
   call frameStarted method{
   response system input;
   calculate and set new states of dynamic nodes;
   calculate and set new state of camera;
   ...
   }
   ...
   }
   ...
  }
```

B. Newton Game Dynamics Engine

Newton Game Dynamics is an integrated solution for real-time simulation of physics environments. The API provides scene management, collision detection, dynamic behavior and yet it is small, fast, stable and easy to use.

In our framework, we use the Newton Game Dynamics to handle the collisions between objects and other dynamic behavior. Newton Game Dynamics is developed by Julio Jerez and Alain Suero. It can not only provide fast and accurate collision detection mechanism, but also can be easily integrated into other applications. The collision detection mechanism for the implementation is as follows: Define a collision primitive with the same size for the entities in the physical world before your creation to constrain its shape. The multiple entities of the same size attach to one collision geometry simultaneously. If there are multiple entities, you can create, impact, and destruct them by the callback mechanism built in Newton, thus greatly improve the efficiency of collision detection. When collision detection is completed, then delete the instance of the collision.

The collision geometries provided by Newton are: the Primitive Shapes, such as Boxes, Ellipsoids, Cylinders, Capsules, Chamfer Cylinders, etc; Convex Hulls, using a series of points in the space to create the smallest convex; Tree Collisions, the model built by the collision tree is endowed with infinite quality, mainly for background model [3]. In our system, we mainly use the Primitive Shapes and Convex Hulls as the collision geometries for fragments.

C. The combination of Ogre and Newton

Ogre is a rendering engine, it doesn't include physics module. Newton engine is a physical environment for real-time simulation and is not suitable for rendering. So we must integrate Newton with OGRE engine. In practice, we adopt an OOP class library named OgreNewt. OgreNewt is developed by Walaber, it integrates all the physical interface functions of Newton SDK to a group of object-oriented classes based on OGRE so as to binding the collision geometries in Newton to the entity mesh of OGRE [4][5]. Consequently we can combine the physical characteristic and rendering characteristic of object.

III. IMPLEMENTATION

A. OgreNewtonApplication

The OgreNewtonApplication class inherits the ExampleApplication class included in the Ogre example programme. It contains the following member functions as showed in Tab I.:

TABLE I. OGRENEWTONAPPLICATION CLASS

```
public:
          OgreNewtonApplication(void);
          ~OgreNewtonApplication(void);
protected:
          void createFrameListener();
          void createScene();
private:
          OgreNewt::World* m World;
          Ogre::SceneNode* msnCam;
          Ogre::FrameListener* mNewtonListener;
```

The function createFrameListener() is used to add a frame listener of the project to manage the events of the scene. The function createScene() is used to add entities, camera, lighting and other necessary information of the scene for rendering. OgreNewt::World* World is a class used to create the physical world. Ogre::SceneNode* msnCam is the carema and the position and angle can be set by it.

B. *OgreNewtonFrameListener*

The OgreNewtonFrameListener class inherits the ExampleFrameListener class included in the Ogre example programme and is used to handle the events of the scene. It contains the following members as showed in Table II.:

TABLE II. OGRENEWTONFRAMELISTENER CLASS

```
    protected:
        OgreNewt::World* m_World;
        SceneNode* msnCam;
        SceneManager* mSceneMgr;
        int count;
        float timer;
    public:
        OgreNewtonFrameListener(RenderWind*        win,
Camera* cam, SceneManager* mgr, OgreNewt::World* W,
SceneNode* ncam);
        ~OgreNewtonFrameListener(void);
        bool frameStarted(const FrameEvent &evt);
```

C. *Shadow and lighting*

To obtain more realistic results, we add shadow and lighting for our system. The code is given in Table III:

IV. RESULTS

We have implemented a real-time fracture simulation framework of brittle board hitting by a rigid ball on PC. The development enviroment is Visual C++.net 2008 and Microsoft Windows XP SP2. The hardware platform is a PC with Inter(R) Core(TM)2 6320 1.86 GHz, 2.0 GB RAM and NVIDIA GeForce 9800 GTX+. Figure2 is the results of our simulation.

V. SUMMARIES

In this paper, the basic concepts and principles of professional graphics rendering engine OGRE and Newton Game Dynamics is introduced. Based on OGRE and Newton, we implement a rigid body fracture simulation framework through the OgreNewt class library. Profitting from the powerful capability of physical simualtion of Newton, we have gotten actual results with high performance, the simulation results can meet the requirement of reality and real-time.

Although the OGRE and Newton is completely open source, there are still some limitations for constructing specific application. For instance the OGRE can only use the pre-defined model file format .mesh, so if we use models with other file formats, we must try to convert them to .mesh. This is a troublesome work and some mistakes may occur during the conversion process. Our future work may mainly concern on constructing independent physical and rendering engines based on the existing technologies, and integrate them into the specific application framework for given purpose.

REFERENCES

[1] JUKER Gregory. Pro OGRE 3D Programming. Apress, 2006.9

[2] STREETING Steve. OGRE Manual v1.7. The OGRE Team, 2009.12

[3] JEREZ Julio; SUERO Alain. Unoficial Tutorials for Newton Game Dynamics WIKI [EB/OL]. http://newtondynamics.com/wiki/index.php5?title=Tutorials.

[4] WALABER. Ogre and Newton with OgreNewt Beginners Guide WIKI [EB/OL]. http://newtondynamics.com/wiki/index.php5?title=Tutorial_-Ogre_and_Newton_with_OgreNewt_beginners_ guide

[5] OUYANG Hui-qin; CHEN Fu-min. Research and Implementation of Binding Physics Engine and Graphics Rendering Engine. Computer Engineering and Design [J], 2008.11, PP:5580-5582

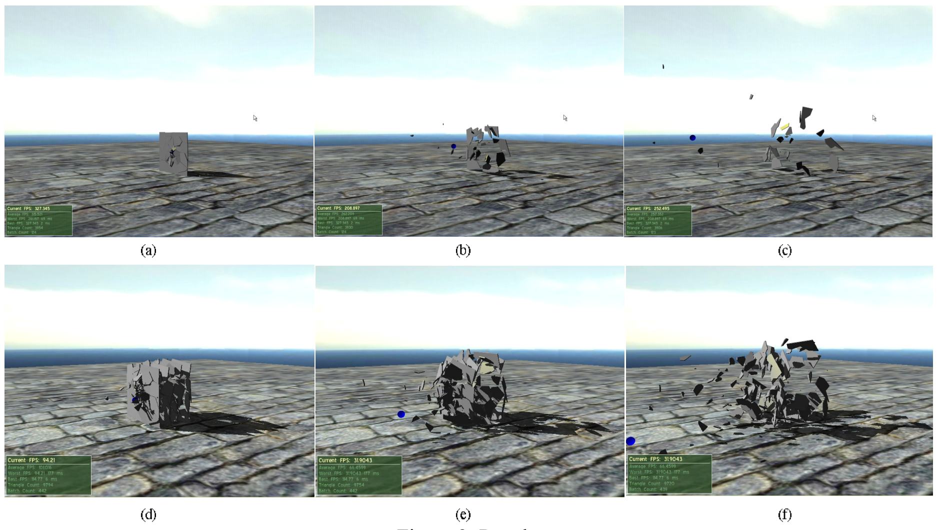

Figure 2. Results

TABLE III. SET THE SHADOW AND LIGHTING

```
//Set the type of shadow
mSceneMgr->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_MODULATIVE);
//Set the light source
    Ogre::Light* light;
    light = mSceneMgr->createLight( "Light1" );
//Set the type of light to point light source
    light->setType( Ogre::Light::LT_POINT);
    light->setDiffuseColour(ColourValue(0.5, 0.5, 0.5));//set the diffuse reflection
    light->setSpecularColour(ColourValue(0.5, 0.5, 0.5));//set the specular reflection
    light->setPosition( Ogre::Vector3(-1500,1500,-1500) );
```