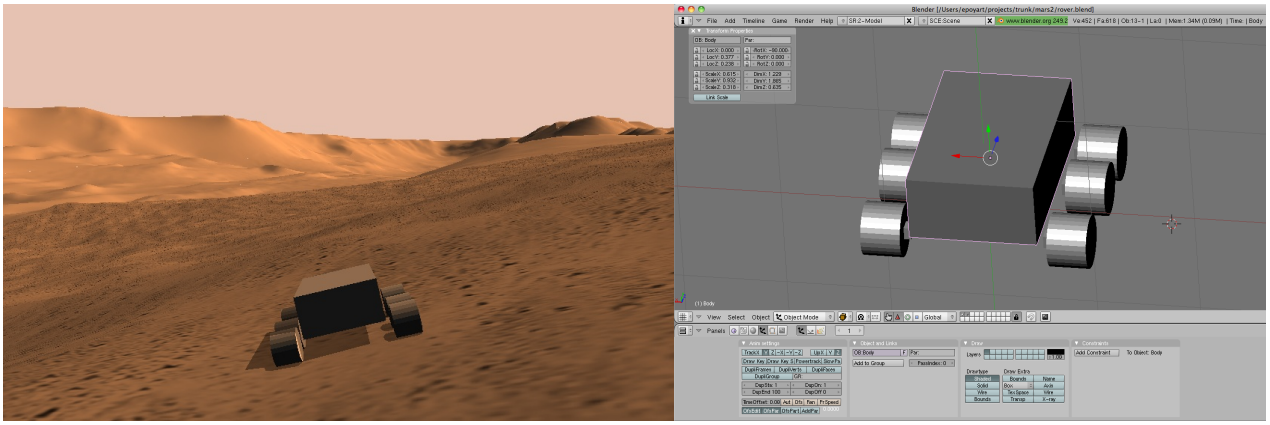
Interactive Articulated Rover on Mars

Eduardo Poyart – UCLA – March 2010

CS 274C – Computer Animation – Prof. Petros Faloutsos



Introduction

The purpose of this project was to implement an interactive virtual environment of the surface of Mars in which the user can drive and articulated rover. The Martian terrain is derived from the MOLA (Mars Orbiter Laser Altimeter) database available from NASA. The rover is subject to physics laws, and performs collision detection with the planetary surface. The ODE (Open Dynamics Engine) library provides the physics simulation and collision detection.

Mars

The MOLA database I used [NASA] was a heightmap of the surface of Mars with 128 samples per degree of longitude. This database is divided in 16 rectangular areas, in a 4x4 grid, that cover the surface of the planet from 88 degrees North to 88 degrees South. Only one of these segments was used.

Because the resolution of 128 samples per degree translates, close to the equator, to roughly one sample every 450 meters, the terrain had to be turned into something more detailed. A real-life-sized rover wouldn't interact in an interesting way with a terrain with that resolution, since it would be very small compared to the area of each triangle.

The method I chose to increase the resolution of the terrain was Fractional Brownian Motion. It consists of recursively subdividing the terrain and applying a random vertical displacement to all vertices at each step. Notice that the vertices created on earlier subdivisions receive more random displacements than those created on later subdivisions. This has a net effect of creating a distribution of frequencies in such a way that the magnitudes of higher frequencies are smaller. It results in a more realistically-looking terrain.

The technique of Simulated Erosion [Musgrave 1989] was also studied as a step to be applied after the fractal Brownian motion. It was not fully implemented due to time constraints and due to the scope of the project being on animation rather than terrain, but the system is prepared for integration with that technique.

The world is rendered using a quadtree technique, in which terrain closer to the viewer are rendered in higher resolution. This allows interactive frame rates. The highest resolution level is also texturized, to provide a ground rich in features at close inspection and to help with the

perception of motion. I used a texture taken from an actual photograph of Mars made by the Phoenix lander.

Since the total amount of terrain data is large, I used a streaming mechanism that I had previously implemented. This mechnism only loads in memory the sections of terrain that are to be rendered, at their appropriate resolution.

The Rover

In order to allow for flexibility in rover design and to help future projects, I created an entire modeling tool chain. It is based on the open-source software Blender [Blender], which provides 3D modeling capabilities. Figure 1 shows this tool chain.

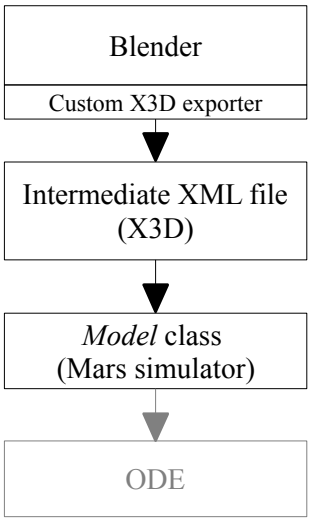


Figure 1 – the tool chain

Blender has an exporter plug-in framework in which each exporter is a Python program that accesses the Blender API. One of the exporters it comes with is the X3D exporter. The X3D format [Web 3D Consortium] is an open specification that is the successor of VRML. I found this format to be the best to use as a base, due to the fact that it represent boxes and cylinders with constant vertex coordinates in local space, and it stores the transformations needed to position, scale and rotate them. With X3D as a base, a few extra fields were introduced to represent physics constraints, and the bodies that these constraints relate to.

Blender has a built-in representation of constraints, but it was not used because it didn't appear practical. More specifically, from an user interface standpoint, it is difficult to position Blender's built in constraints and to define their axes. Instead, the following method was devised: for each constraint, the user creates a “box” objects and gives it a name that starts with the string “Const”. These objects do not represent geometry in the scene, but merely a constraint. They can be scaled to look very small and positioned between two geometry bodies in the scene. The position of the constraint object defines the pivot point of the physical constraint. The rotation of the constraint object defines the axes of the constraint: in case of a Hinge constraint, the X axis is the actuation axis; in case of a Hinge2 constraint (used in ODE for wheels), the X and Y axes are used.

By setting Blender's user-defined properties for these objects, the user can create key/value pairs that indicate the pair of objects this constraint is related with, as well as the constraint's limits. If the keys are the strings “Const1” or “Const2”, the value represent the name of the first or second body this constraint is linked to. If the keys are the strings “Limit1” or “Limit2”, the value is used to set the limits, in which case it should consist of two floats separated by a comma. Figure 2 exemplifies a pair of objects (the rover's main body and a wheel) and a constraint between them.

In the simulator's C++ code, the model is represented by a class called “Model”. This class is able to read the X3D file and instantiate all of the needed ODE bodies and constraints, as well as render the geometry of the model using OpenGL. The Model class is the run-time end of the modeling tool chain, and it talks directly with ODE.

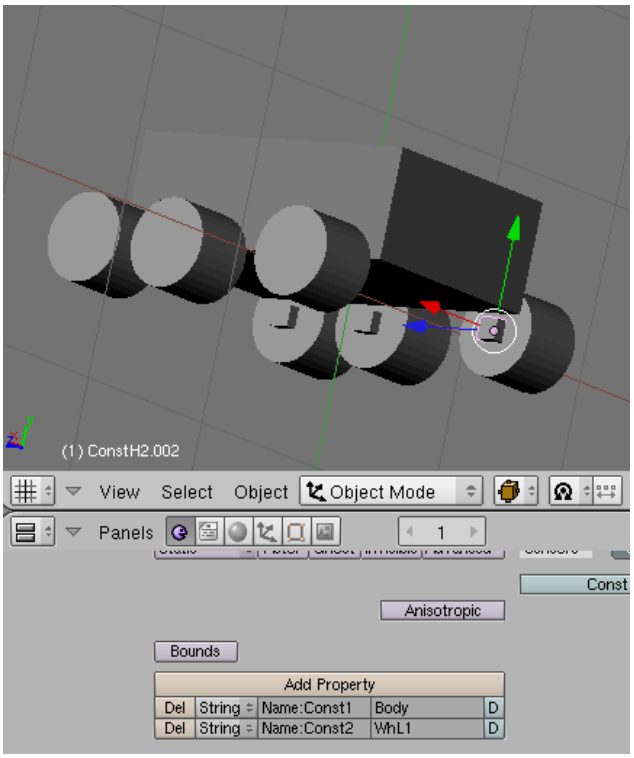


Figure 2 – A constrain between the rover's main body and a wheel. Notice the properties (box on the bottom) that define the parameters.

With this flexible system, many kinds of models and constraints can be easily created in Blender and exported to be used in C++ code under ODE. The primitives implemented were boxes and cylinders. Other primitives can be added in the future. The Model class also takes care of generating collision geometry for use with ODE's collision system. Currently each geometric primitive of the model corresponds to a rigid body and a collision primitive, but this can be changed in the future. In particular, separating rendering geometry from rigid bodies and collisions allow for rigid geometry of arbitrary shapes to be created without the use of an inflexible constraint between pairs of bodies, which is currently necessary but computationally expensive.

A six-wheeled rover was modeled. Its two front wheels are traction wheels. They also steer left and right. All of the rover's movement arise from its physical interaction with the terrain. The rover's position and orientation do not respond directly to user input, only indirectly. It achieves movement through the wheels' rotational speed, steering angle and friction with the terrain.

Steering was implemented by a PD controller on each of the traction wheels. This controller sets a desired angle for the wheel and applies rotational velocity relative to the steering axis so that it reaches that angle. Acceleration was implemented by directly applying angular velocity relative to the main rotation axis to the weel.

The Simulation

ODE, the Open Dynamics Library [ODE], is the library I used for physics simulation and collision detection. Internally, its constraint solver follows the techniques described in [Baraff 1995]. Differently from the ODE demos, in this project I implemented separation between step and draw functionalities, which is a better design. However, the time step was limited to a maximum of 1/30s, to help with the stability of the physics library, which can provide inaccurate and unstable results if large time steps are used.

The world's mesh is provided to ODE in form of a TriMesh object. Only the higher resolution meshes receive a TriMesh representation, since the rover will never be colliding with low-resolution terrain.

Many simulation parameters had to be fine-tuned. Friction between the wheels and the ground and the spring and damping constants for the wheel suspension were particularly tricky. The wheel suspension is parametrized in terms of ODE's Error Reduction Parameter and Constraint Force Mixing, which re related to k_s and k_d. My objective is for the user to drive the rover over almost any terrain in a pleasant way and without flipping over too much, provided that the speed isn't excessive. At the same time, the experience should not be too dull and slow.

A third-person camera was implemented. The positioning was done as follows: the camera is positioned at a constant negative translation along the forward axis of the rover's main body, and then it is translated up along the world's vertical axis. The camera orientation is such that it points to a fixed point above the rover (i.e. the center of the rover translated up on the world's vertical axis).

The user interaction with the simulator is very simple. The W/S/A/D keys are used for steering (these keys are commonly used in first-person shooter games). W accelerates, S decelerates, A turns left and D turns right. The 0 key decelerates the traction wheels to zero velocity instantly.

Results

The end result was, in my opinion, quite attractive and interesting. It does invite the user to explore the world and to learn the physical characteristics and limits of the rover. The system runs at 30 frames per second on a 2.5GHz Intel Core 2 Duo and a Nvidia GeForce 8600M GT GPU.

A few bits of emergent behavior were oberved. Due to the camera positioning, which somewhat follows the rover's actions, the whole scenery moves and shakes on screen as the rover moves, especially at high speeds. Also, when the rover is moving and the user “brakes” it (“0” key, which sets the traction wheel's angular velocities to 0), the resulting angular motion of the rover's main body becomes an angular motion of the camera, making the view “dip down” for a second, exactly as if you were inside a car in a sudden stop. The opposite happens if you suddenly accelerate the rover.

The fact that braking was done by setting the wheels' angular velocities to 0 had other interesting results. One can drive at high speeds and suddenly hit the brakes, and watch the rover slide. There is even a risk of flipping over when this is done.

Future work

There are many topics to explore, for which a planetary simulator with physically-based interaction is an important tool: how to efficiently render an entire planet in real time, how to generate detail, what would be the result of simulated erosion on the generated detailed surface, how to perform on-line simulated erosion in order to reduce the amount of data that has to be pre-processed, how to use such a system as a scenario for crowd simulations, or what other kinds of physically-based interaction users can have with it.

References

BARAFF, D. 1997. Physically Based Modeling: Principles and Practice. SIGGRAPH Online Course Notes.

MUSGRAVE, F. K., KOLB, C. E., MACE, R. S. 1989. The Synthesis and Rendering of Eroded Fractal Terrains. SIGGRAPH.

NASA PDS GEOSCIENCES NODE – Washington University in St. Louis. http://pds-geosciences.wustl.edu/missions/mgs/mola.html

ODE – OPEN DYNAMICS ENGINE. http://www.ode.org/

WEB 3D CONSORTIUM: Open Standards for Real-Time 3D Communication. http://www.web3d.org/x3d/