1 Basic Bounc3 Documentation

1.1 Introduction

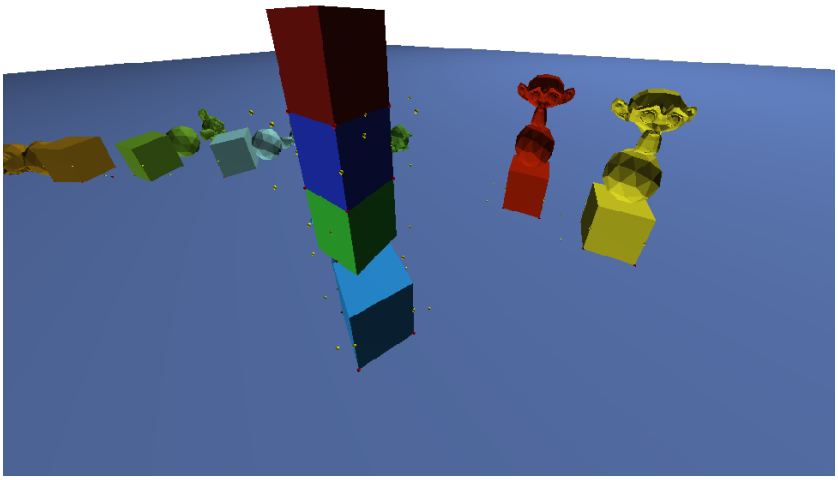
Bounc3 is a lightweight constraint-based rigid body physics engine developed by Irlan Robson (greatly inspired by Erin Catto's Box2D) to be used in mid-scale interactive applications. It was particularly designed to create games with the necessity of the physically plausable dynamics simulation of solid objects in a scene. The engine can simulate (rigid) bodies and the two most common physically interaction types between them, contact, and friction. For the time being, joints aren't currently available in *Bounc3*.

1.2 Features

- Ellastic and inellastic rigid body simulation with contact and friction;
- Convex polyhedras;
- Sensor shapes;
- Fast ray-casting and AABB queries;
- Box stacking;
- Broad-phase collision detection is performed using the (BVH) Dynamic AABB Tree;
- Half-edge centric polyhedras;
- (Narrow-phase) Collision detection and distance queries between shapes using the Separating Axis Test (SAT) with the Gauss-map Optimization (BERGEN e GREGORIUS, 2010);
- Contact graph;
- Sequential impulses technique to satisfy (contact and friction) velocity constraints (CATTO, 2005). Basically, the technique is a specialization of the Projected-Gauss-Seidel (PGS) to solve the LCP defined by the contact constraint and the (not-so-technically) MLCP defined by the friction constraint;
- Semi-implicit euler integration method for solving the simulation system of ordinary differential equations;
- Simulation islands, and body sleeping;
- Contact, ray-casting, and AABB query listeners for query notifications;
- Built-in math and memory management;

1.3 Practical Use Case

1.3.1 Creating a scene



A scene rendered using Bounc3 as the physics engine.
Source: Research field (2015).

```
// At simulation initialization:

// Create a b3Scene object instance.

b3Scene m_scene;

// Create a b3TimeStep object that defines the configuration of a single
simulation step.

b3TimeStep m_step;

m_step.dt = 1.0 / 60.0f (60Hz); // The simulation frequency.

m_step.velocityIterations = 15; // Recomendable to set > 5.

m_step.allowSleeping = true; // Allow bodies to sleep under unconsiderable motion.

// At any frame (or step):

m_scene.Step(&m_step);
```

You may want to setup a fixed-time step framework for deterministic purposes and precisability (as in the above example). Contrarly, just set `m_step.dt` with the (not-so-used) variable time.

1.3.2 Creating a body and attaching a (convex) shape to it

A rigid body can hold as many shapes the user wants. Shapes are defined in the local space of the body. Agnostically, shapes should be rendered in world space.

Given a CW vertex-ordered convex mesh and the connectivety information about its faces at application initialization, we must create a face topology in order to create its half-edge centric convex polyhedra and set it to a rigid body. Thus, a face topology describes a convex polyhedra. To the topology, a face is simply a index to a face topology. Example:

```
// Number of faces = 2.
// Topology list (face size followed by the face vertex indices).
topology[0] = 4; // Start of new face.
topology [1] = v1; // Vertex index.
topology [2] = v2; // Vertex index.
topology [3] = v3; // Vertex index.
topology [4] = v4; // Vertex index.
topology [5] = 3; // Start of new face.
topology [6] = vx; // Vertex index.
topology [7] = vy; // Vertex index.
topology [8] = vz; // Vertex index.
// Face list (indices to the face topology).
faces[0] = 0;
faces[1] = 5;
```

The half-edge centric convex hull expects pointers to the vertices, faces, topology, and face planes of the polyhedra. With these data available here’s how we setup a polyhedra hull:

```
b3HullDef hullDef;
hullDef.vertexCount = size of vertices (not the size in bytes!);
hullDef.vertices = pointer to the first adress of the vertex array;
hullDef.planes = pointer to the first adress of the plane array;
hullDef.faceCount = size of faces;
hullDef.facesIndices = pointer to the first adress of the face array;
hullDef.faceTopology = pointer to the first adress of the topology array;
```

Note that we don't need to pass the actual polyhedron faces. Momentarly, most of the modeling tools generally gives you the face connectivity information. Then, this is a flexible way of creating convex shapes. Once the shape is created we can create a body, or use a new one, and attach the new shape to it. Example:

```
B3HullDef hullDef;
(...) Initialize the topology (see above).
b3Hull hull;
hull.Set(&hullDef); // This will compute its the half-edge structure.

// Create the body.
B3BodyDef bodyDef;
// (...) Set the rest of the body definition attributes.
B3Body* m_body = m_scene.CreateBody(&bodyDef);

// Create a shape and attach to the body.
B3ShapeDef shapeDef;
shapeDef.shape = &hull;
// (...) Set the rest of the shape definition attributes.
B3Shape* shape = m_body->CreateShape(&shapeDef);
```

To destroy the shape call:

```
m_body->DestroyShape(shape);
```

For a more detailed overview please see the code comments on b3Polyhedron.cpp and b3Hull.cpp, as well b3Scene.h and b3Scene.cpp.

1.4 Future work

- Convex hull creation;
- Continuous collision detection;
- Broad and narrow phase collision detection optimization;
- Joints;

The convex hull creation is flexible but not perfect. Therefore, a good idea is invest some (good) time in mesh cooking. Another thing, is that continuous collision detection is needed for (real) physical and visual fidelity. The simulation lacks of deterministic results with the current features. Gigantically, the determinism is a function of the bodies size and the time step.

1.5 References

BERGEN, G. V. D.; GREGORIUS, D. (Eds.). **Game Physics Pearls**. [S.l.]: A K Peters, Ltd., 2010.

CATTO, E. Iterative Dynamics with Temporal Coherence. **Box2D**, 2005. Available at: <https://box2d.googlecode.com/files/GDC2005_ErinCatto.zip>. Accessed in: 22 Feb. 2015.

CATTO, E. Modeling and Solving Constraints. **Box2D**, 2009. Available at: <https://box2d.googlecode.com/files/GDC2009_ErinCatto.zip>. Accessed in: 21 Feb. 2015.

CATTO, E. Soft Constraints. **Box2D**, 2011. Available at: <https://box2d.googlecode.com/files/GDC2011_Catto_Erin_Soft_Constraints.pdf>. Accessed in 23 Feb. 2015.

CATTO, E. Understanding Constraints. **Box2D:** A 2D Physics Engine for Games, 2014. Available at: <http://box2d.org/files/GDC2014/GDC2014_ErinCatto.zip>. Accessed in: 1 Jan. 2015.

CATTO, E. Numerical Methods. **Box2D**, 2015. Available at: <http://box2d.org/files/GDC2015/ErinCatto_NumericalMethods.pdf>. Acessed in. 27 May 2015.

1.6 Thanks

Special thanks to Erin Catto and Dirk Gregorius for making their incredible contributions to the physics simulation community; for sharing their knowledge with us. Thanks to Dirk for helping me (at gamedev.net) implementing (efficiently) the SAT algorithm. Thanks to Erin for creating and sharing Box2D for use and reference, and for encourage us to use his library as reference.