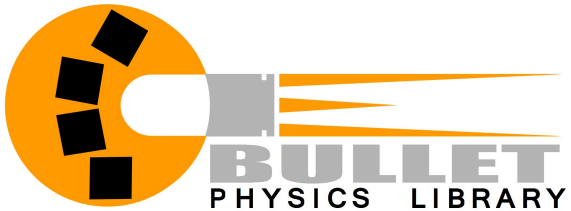

SIGGRAPH2015
Xroads of Discovery



Bullet Physics Simulation
Introduction to rigid body dynamics and collision detection

Erwin Coumans
Google Inc.

SIGGRAPH 2015
Xroads of Discovery
The 42nd International Conference and Exhibition on Computer Graphics and Interactive Techniques

Course Agenda

- Motivation
- Rigid Body Simulation Loop
- Collision Detection
- MLCP Constraint Solving
- Featherstone
- OpenCL Acceleration

Erwin Coumans

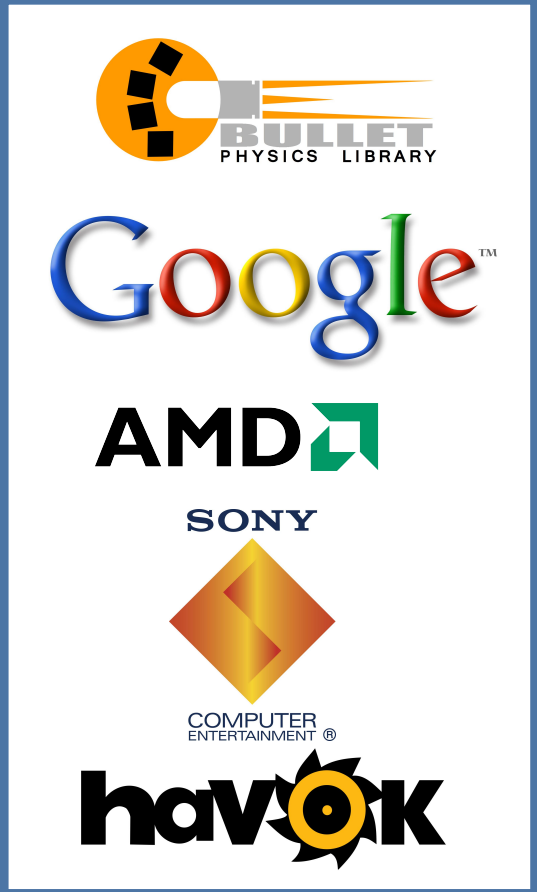
2003-

2014- Robotics

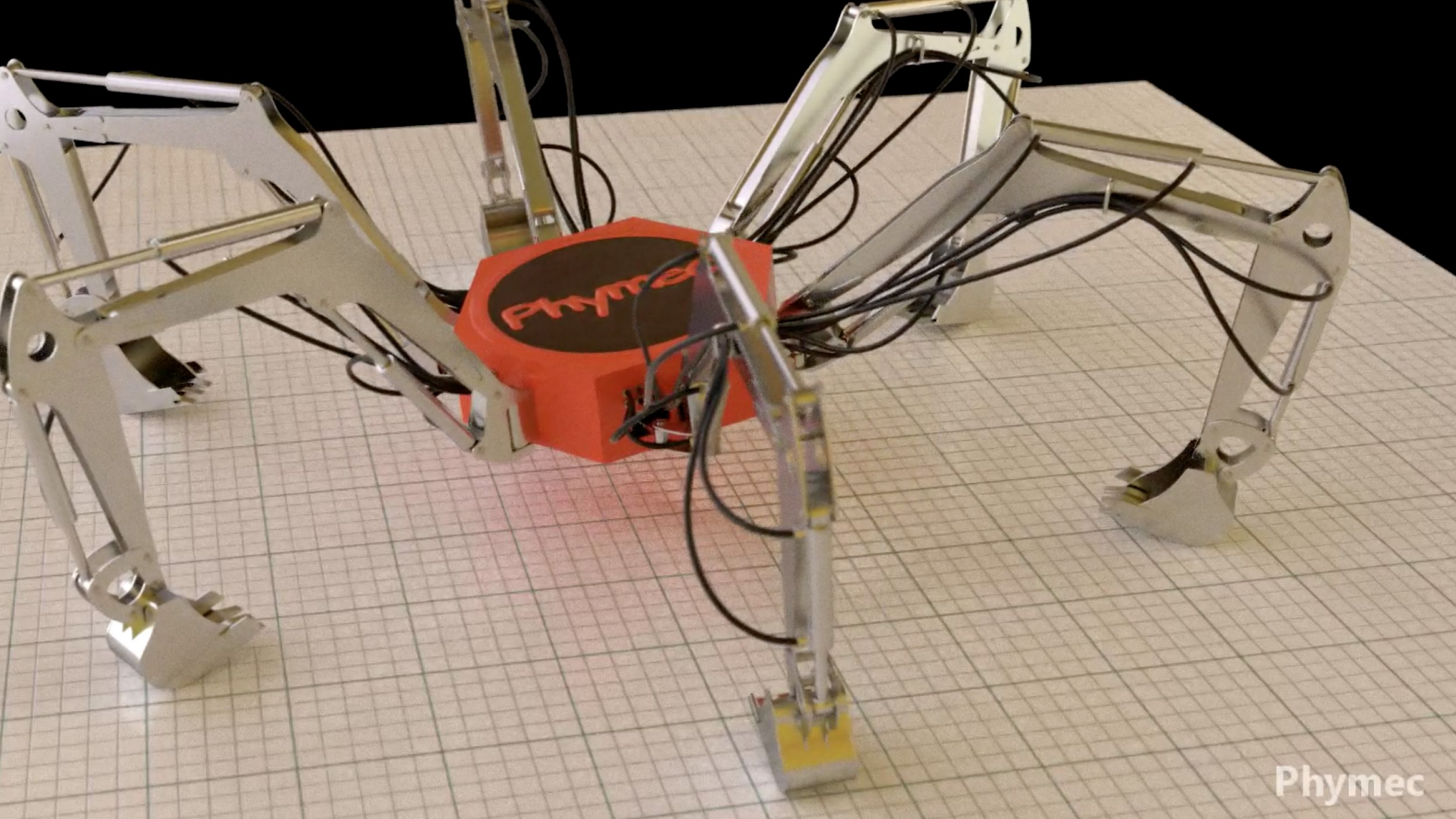
2010-2014 OpenCL,
Games, VFX

2003-2010 PS2, PS3,
PS4, Games

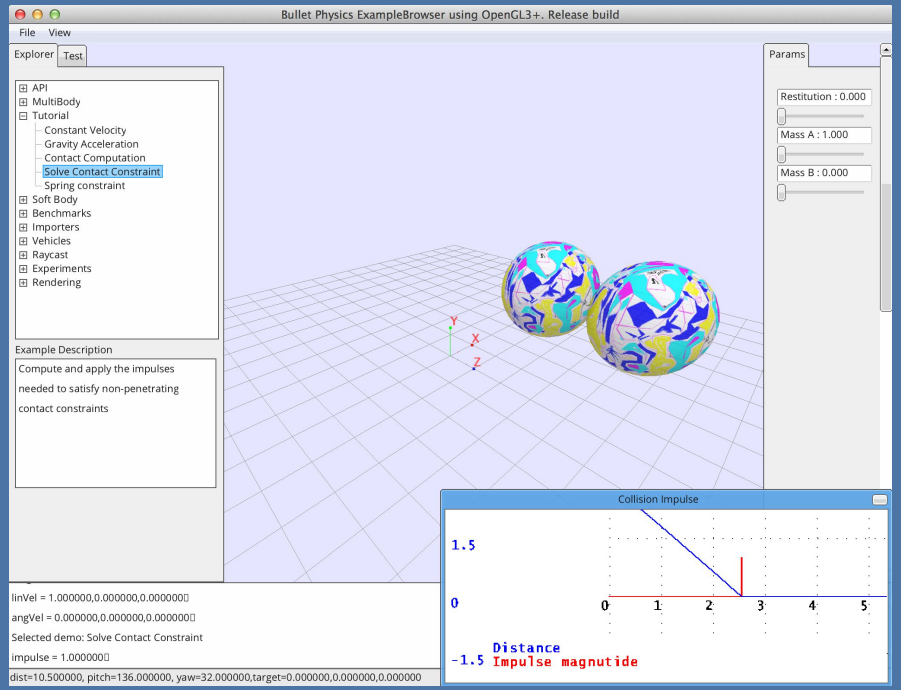
2001-2003, Havok 2.x

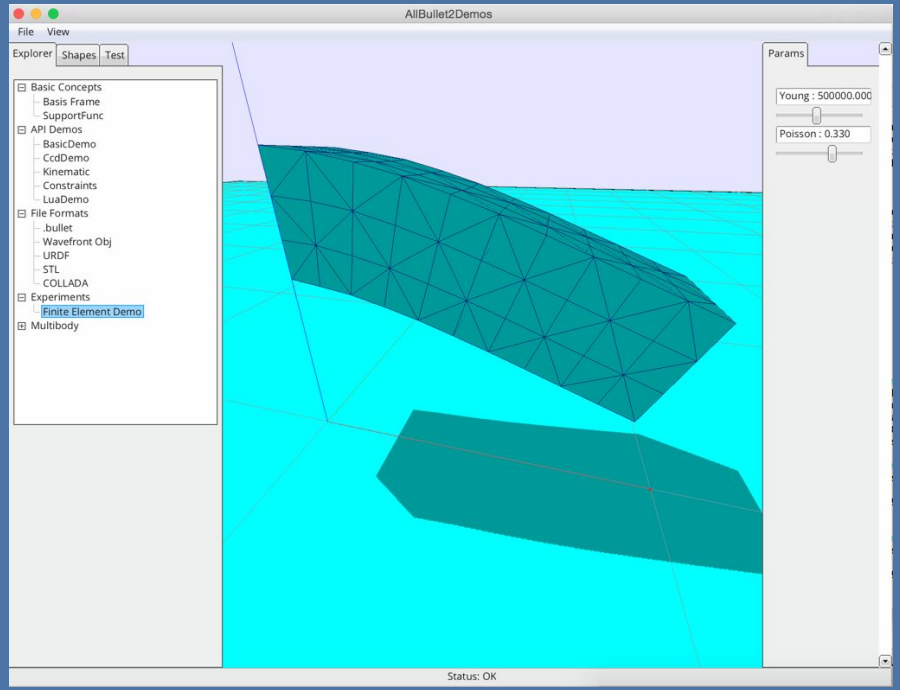


Some movies using Bullet rigid body sim


Phymec

Bullet Example Browser

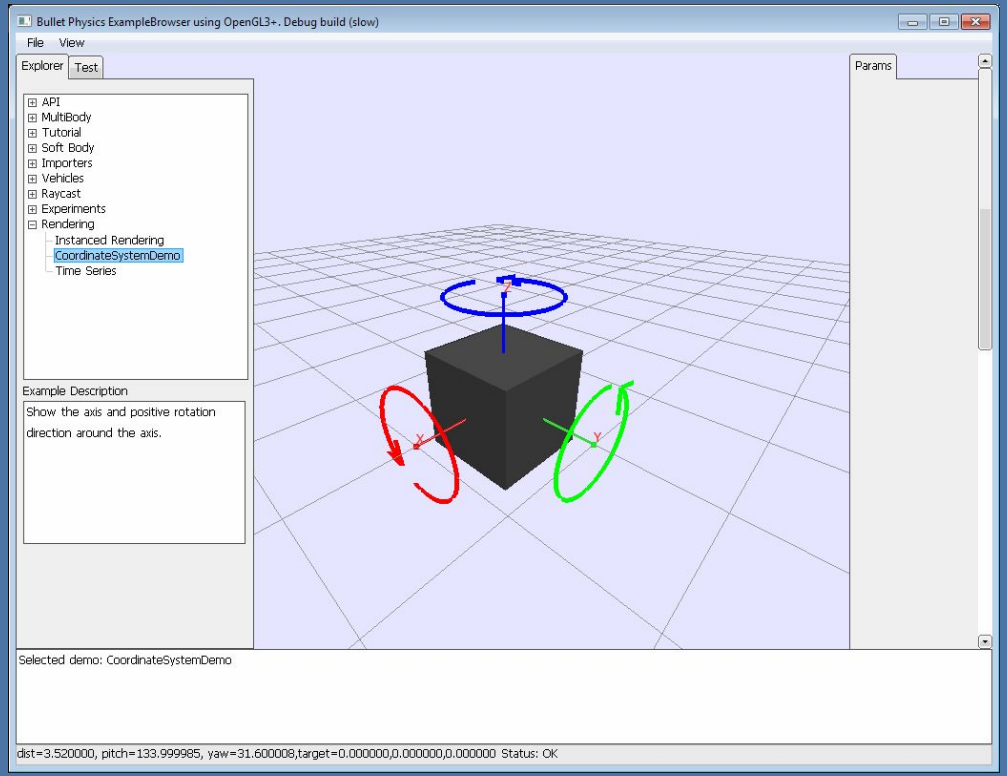




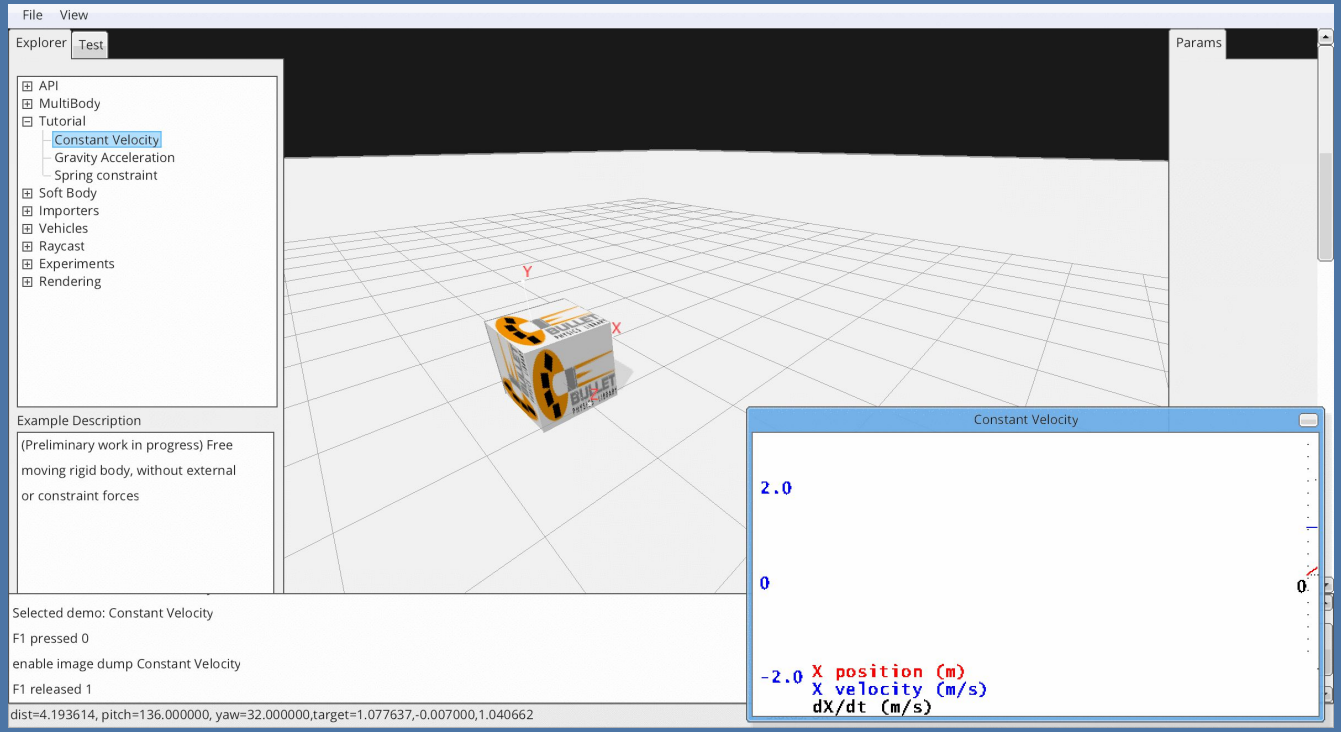
github.com/bulletphysics/bullet3

GNU Make

Reference Coordinate Frame



6 Degrees of Freedom (DOF) Rigid Body



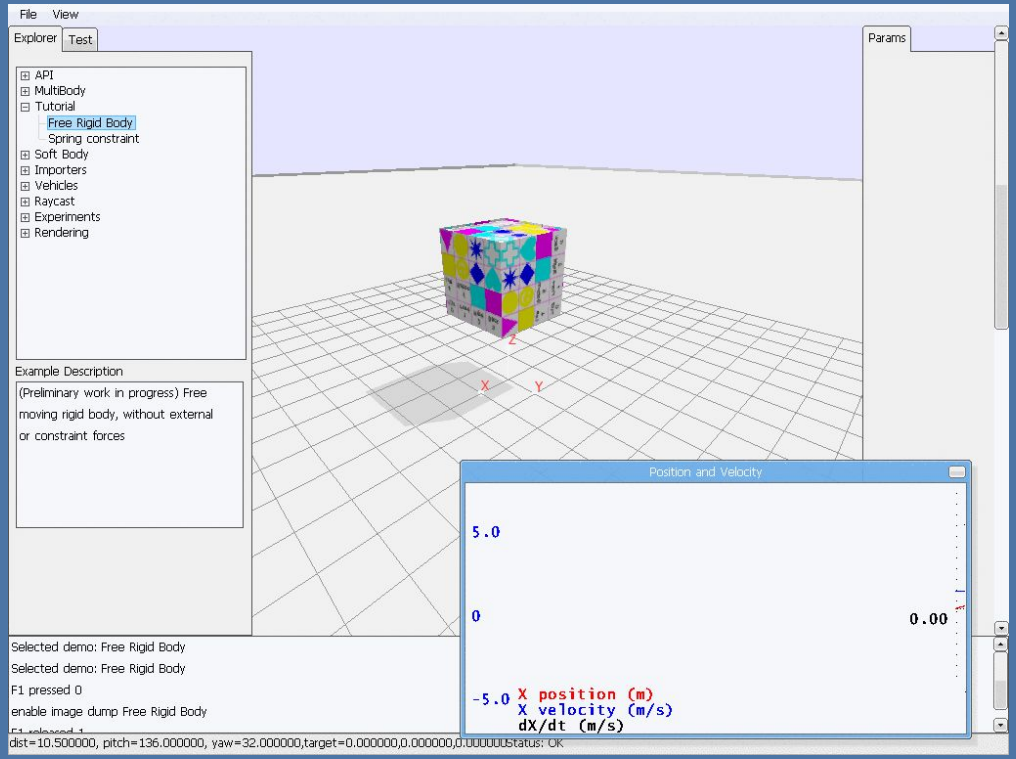
See Tutorial with simplified LWRigidBody

#DOF might not be #Position variables!

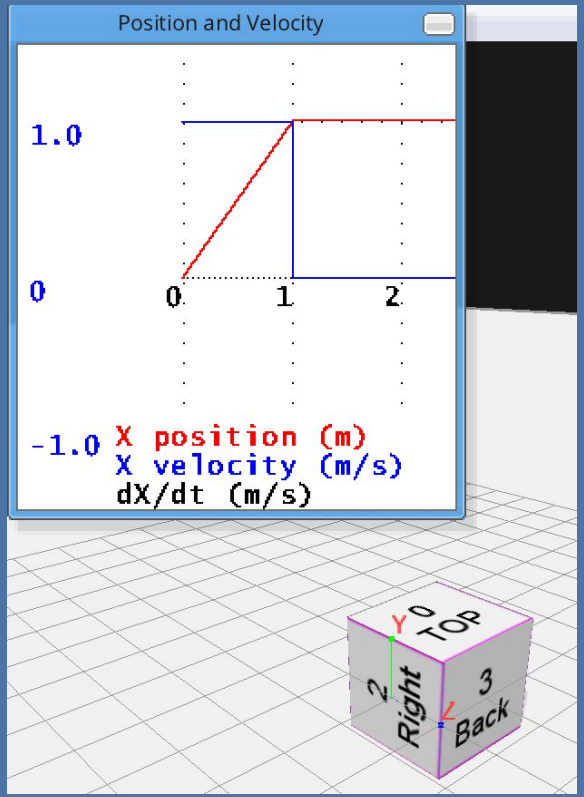
Orientation at RPY euler angles, 3x3 matrix, quaternion, exponential map, etc...

See also “Practical Parameterization of Rotations Using the Exponential Map”

Constant Linear Velocity and Position



Constant Linear Velocity and Position

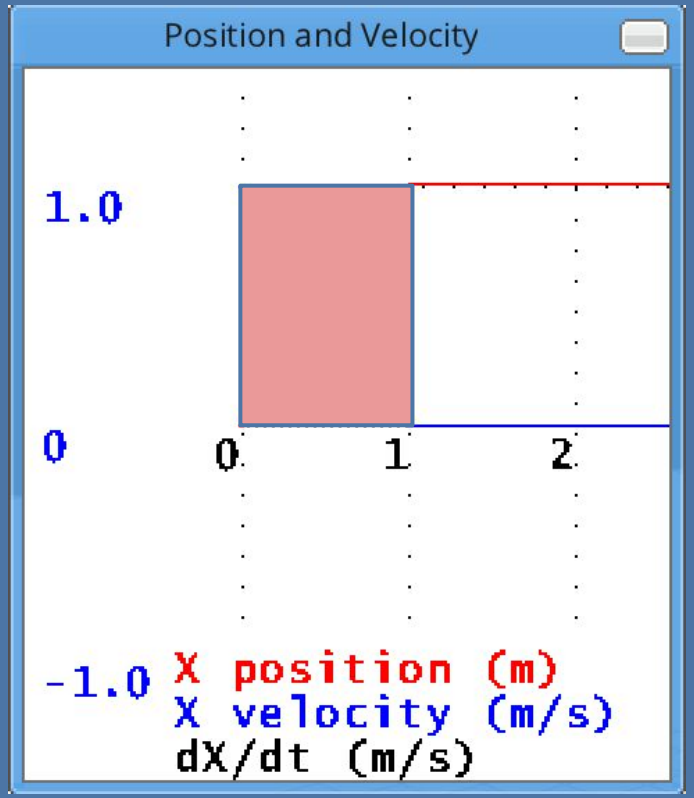


Numerical Integration

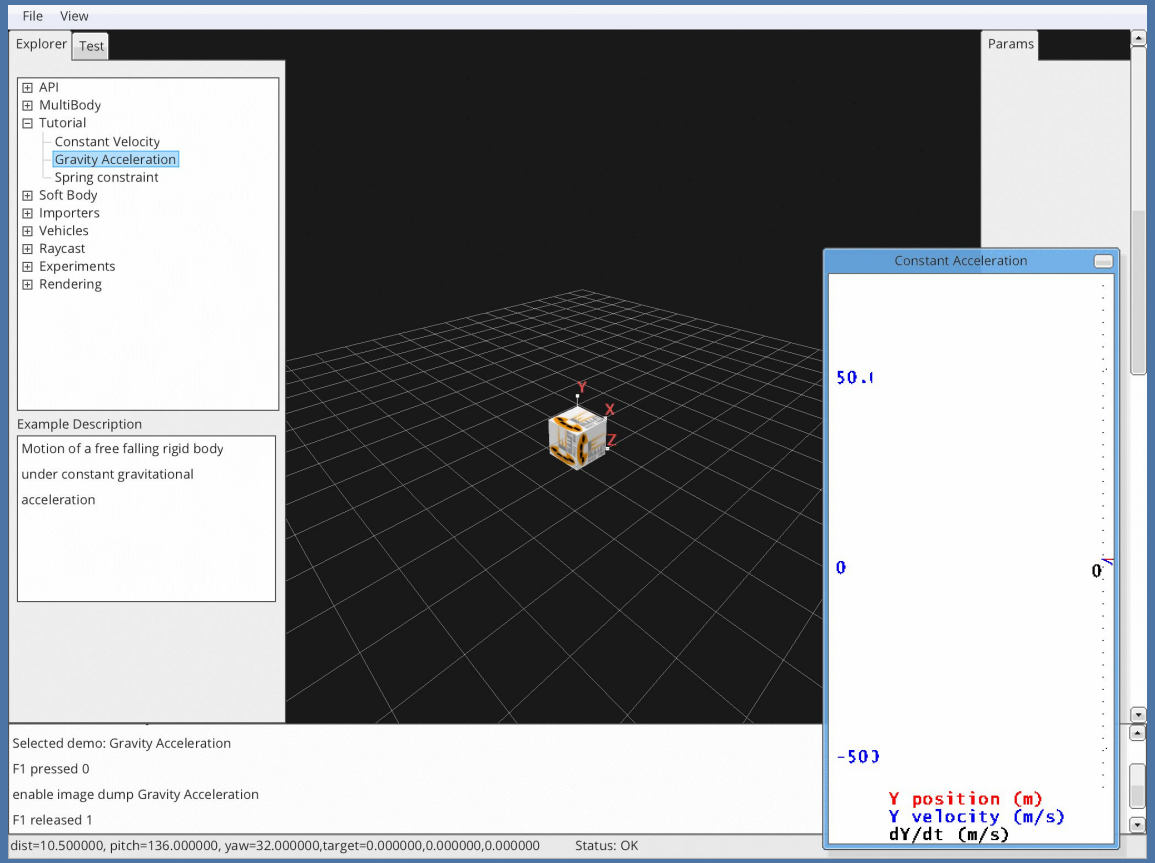
Change in position is
the area under the
velocity curve

v = dX/dt, we have constant velocity so

dX = v * dt



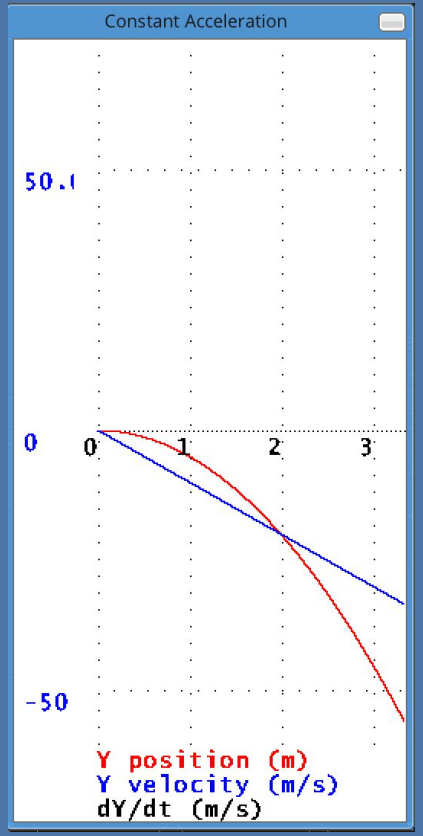
Constant Acceleration



Constant Acceleration

Change in position is the area under the velocity curve

v is changing so
$dX = \int v\, dt$



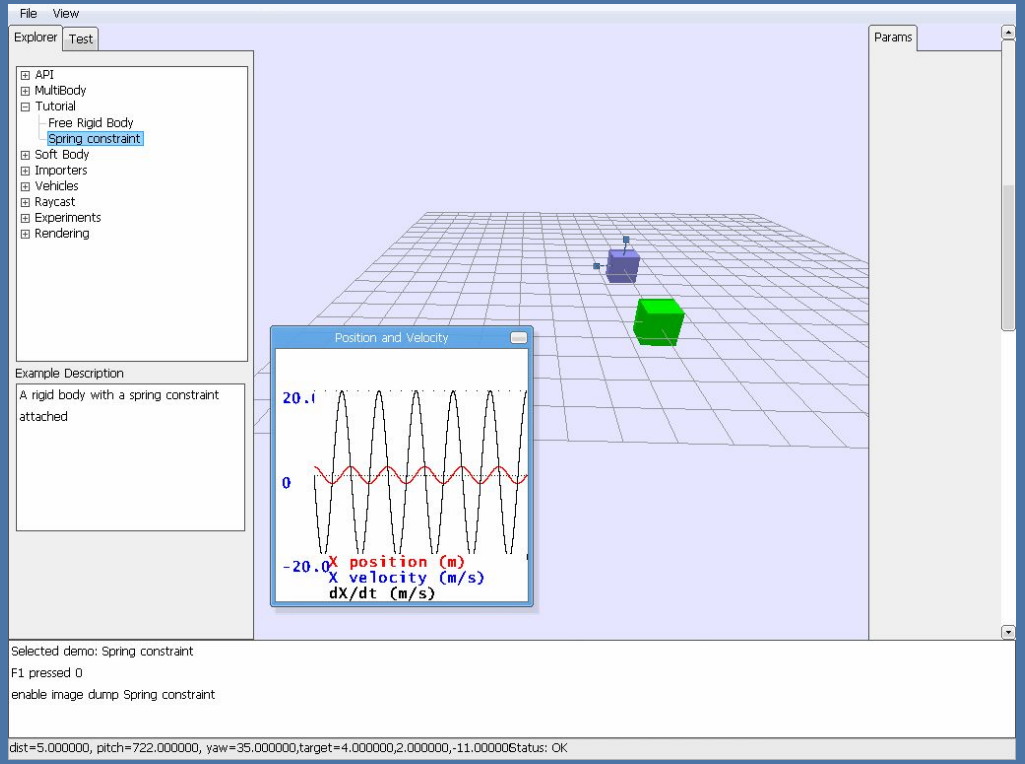
Force and acceleration as function of time

v is changing

over time…

and acceleration a

as well...

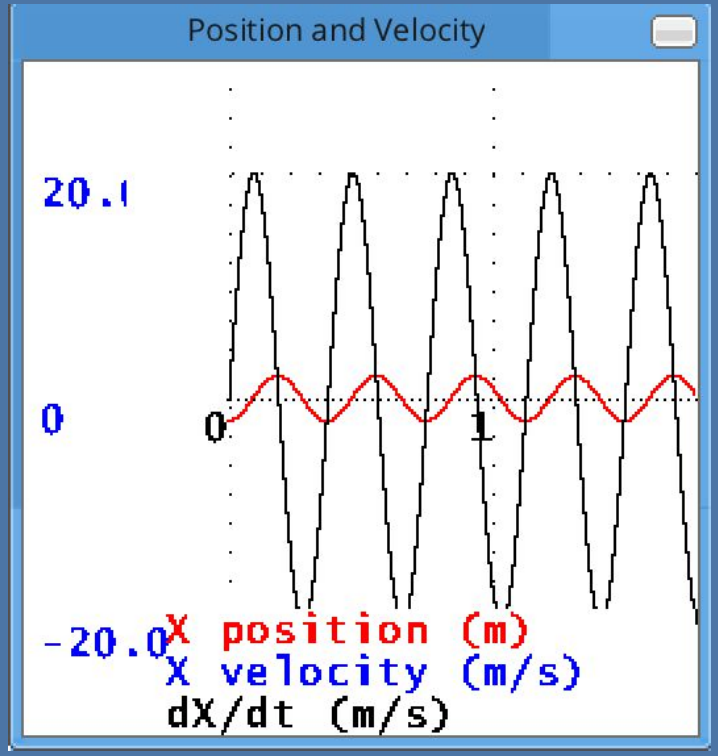


Variable Force and Acceleration

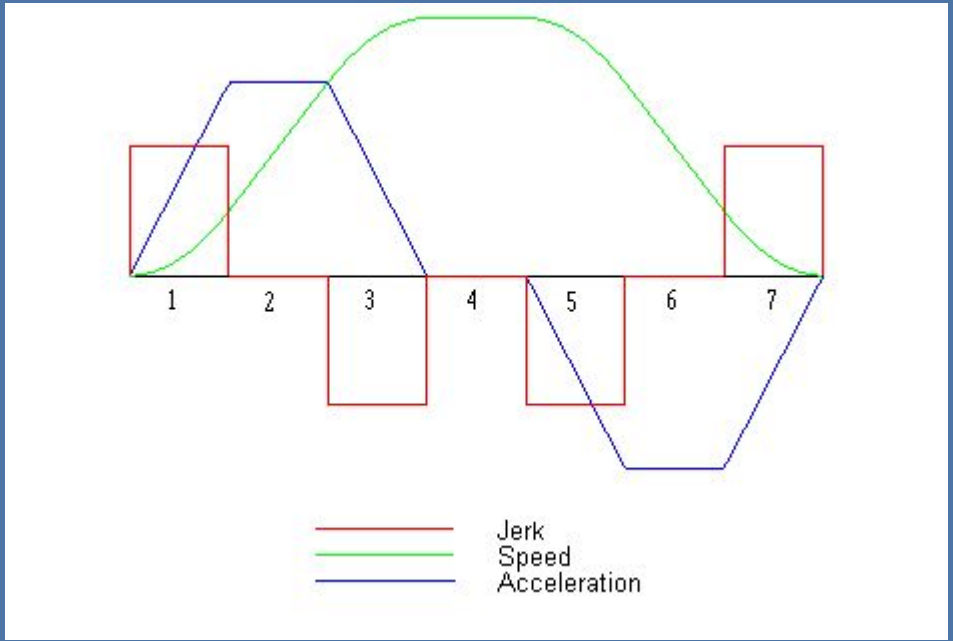
Change in position is the area under the velocity curve

$dX = \int v\, dt$

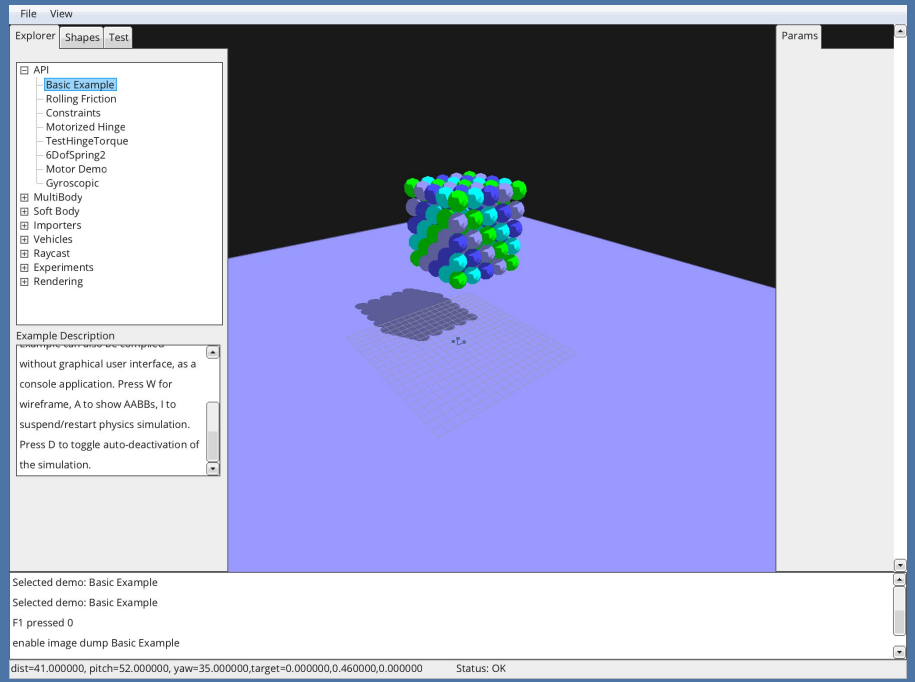
$dv = \int a\, dt$

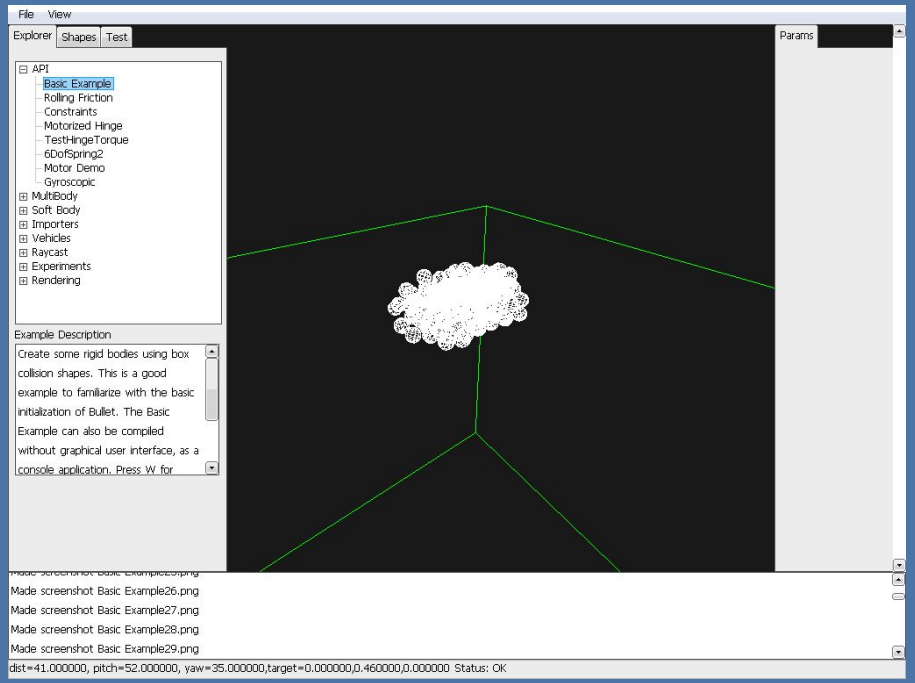


Constant Jerk

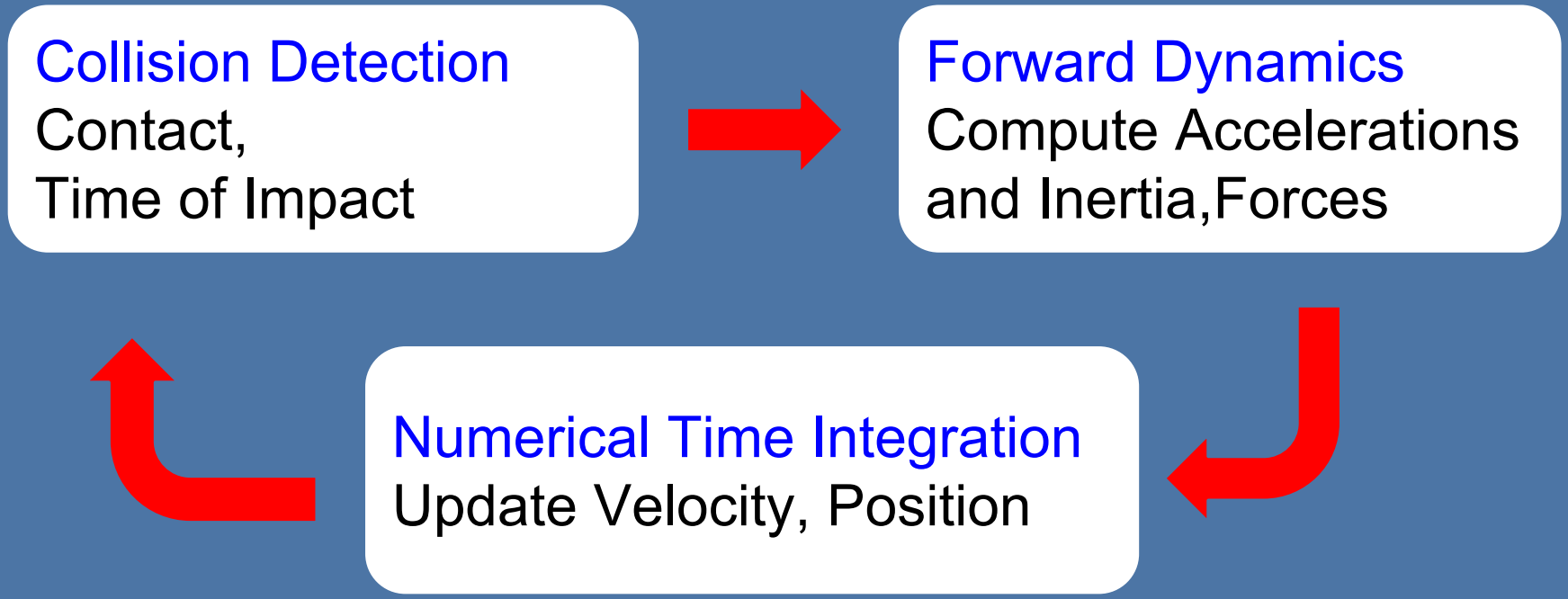


Collision and Friction Forces





Rigid Body Simulation Loop



Equations of Motion and Symplectic Euler

$$F = ma$$

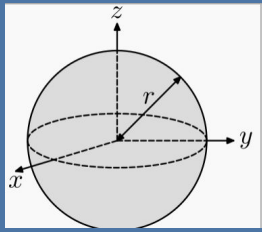
$$\tau = I\dot{\omega} + \omega \times I\omega$$

F_ext could be gravity, wind force field, user forces

$$v_{t+\Delta t} = v_t + a\Delta t = v_t + \frac{F_{ext} + F_c}{m}\Delta t = v_t + \frac{F_{ext}}{m}\Delta t + \frac{Impulse_c}{m}$$

F_c could be constraint forces such as contact, friction, joints

$$x_{t+\Delta t} = x_t + v_{t+\Delta t}\Delta t$$



$$I = \begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix}$$

C++ Numerical Integrator Symplectic Euler

```
struct LWPose {
	btVector3		m_worldPosition;
	btQuaternion		m_worldOrientation;
	LWPose():m_worldPosition(0,0,0), m_worldOrientation(0,0,0,1){}
};
struct LWRightBody {
	LWPose	m_worldPose;
	btVector3 m_linearVelocity;
	btVector3 m_angularVelocity;
	btVector3 m_gravityAcceleration;
	void		integrateAcceleration(double deltaTime) {
			m_linearVelocity += m_gravityAcceleration*deltaTime;
	}
	void		integrateVelocity(double deltaTime) {
			LWPose newPose;
			newPose.m_worldPosition = m_worldPose.m_worldPosition + m_linearVelocity*deltaTime;
			newPose.m_worldOrientation = m_worldPose.m_worldOrientation;
			newPose.m_worldOrientation += (m_angularVelocity * newPose.m_worldOrientation) * (deltaTime * btScalar(0.5));		//quaternion derivative
			newPose.m_worldOrientation.normalize();
			m_worldPose = newPose;
	}
}
int main(int argc, char* argv[]) {
	double deltaTime = 1./60.;//60 Hertz time step
	while (1) {
			m_body->integrateAcceleration(deltaTime);
			m_body->integrateVelocity(deltaTime);
	}
}
```

Symplectic, Explicit or Implicit Euler?

$$F = ma$$

$$\tau = I\dot{\omega} + \omega \times I\omega$$

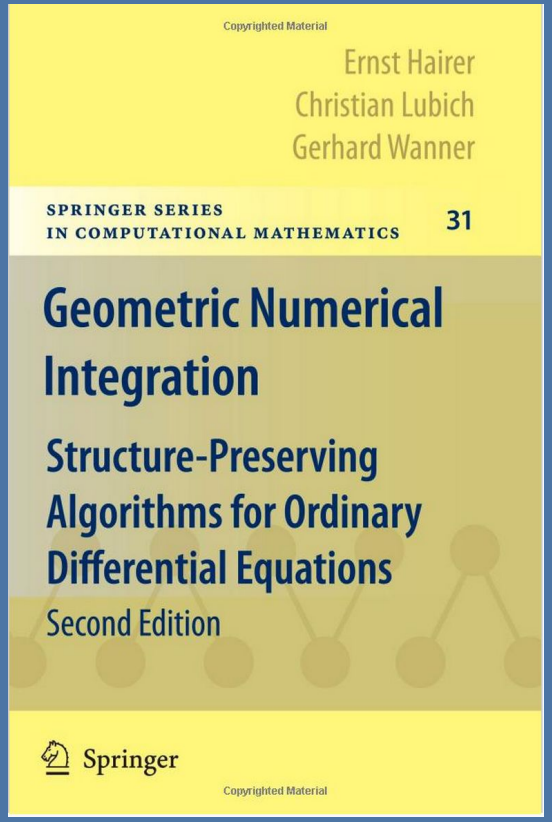
$$v_{t+\Delta t} = v_t + a\Delta t = v_t + \frac{F_{ext} + F_c}{m}\Delta t = v_t + \frac{F_{ext}}{m}\Delta t + \frac{Impulse_c}{m}$$

$$x_{t+\Delta t} = x_t + v_{t+\Delta t}\Delta t$$

Gyroscopic force is velocity dependent.

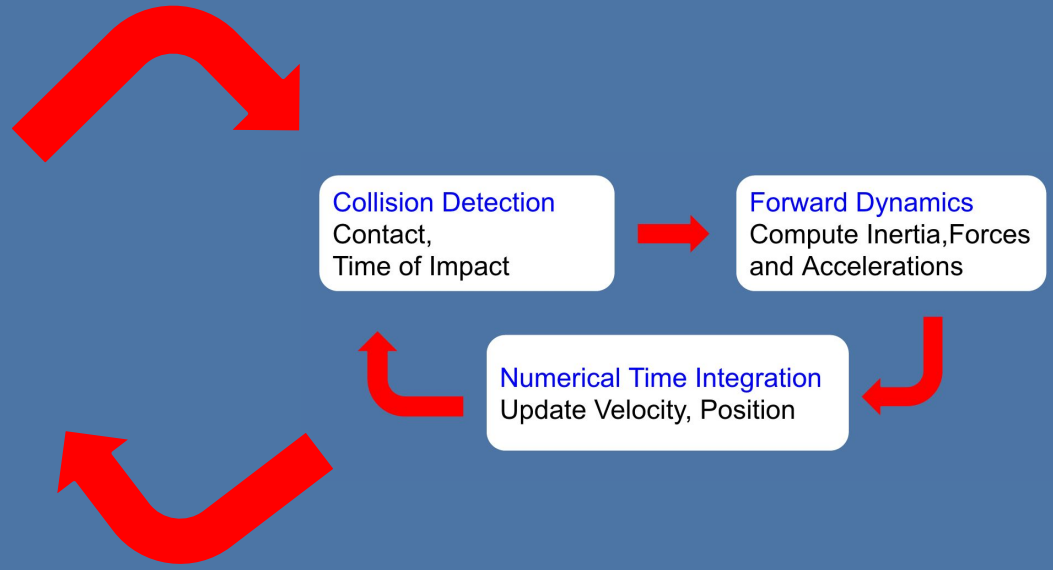
Implicit integration requires omega at t+Δt

Numerical Integrators



SimBody

Error Control
Adaptive Time Step
Higher Order Integrators, RK4,
Acceleration Level Constraints

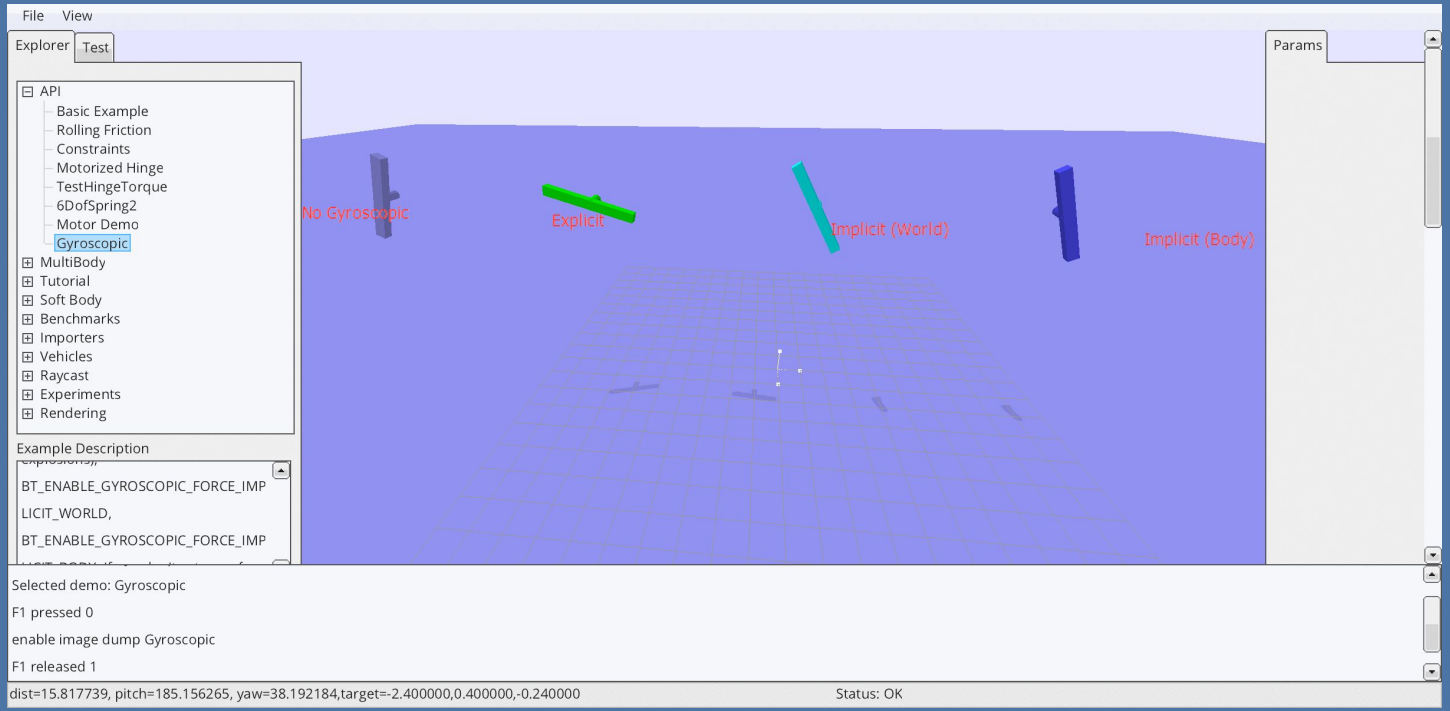


Michael “Sherm” Sherman

https://github.com/simbody/simbody

Dzhanibekov Effect

Implicit vs Explicit Gyroscopic Force

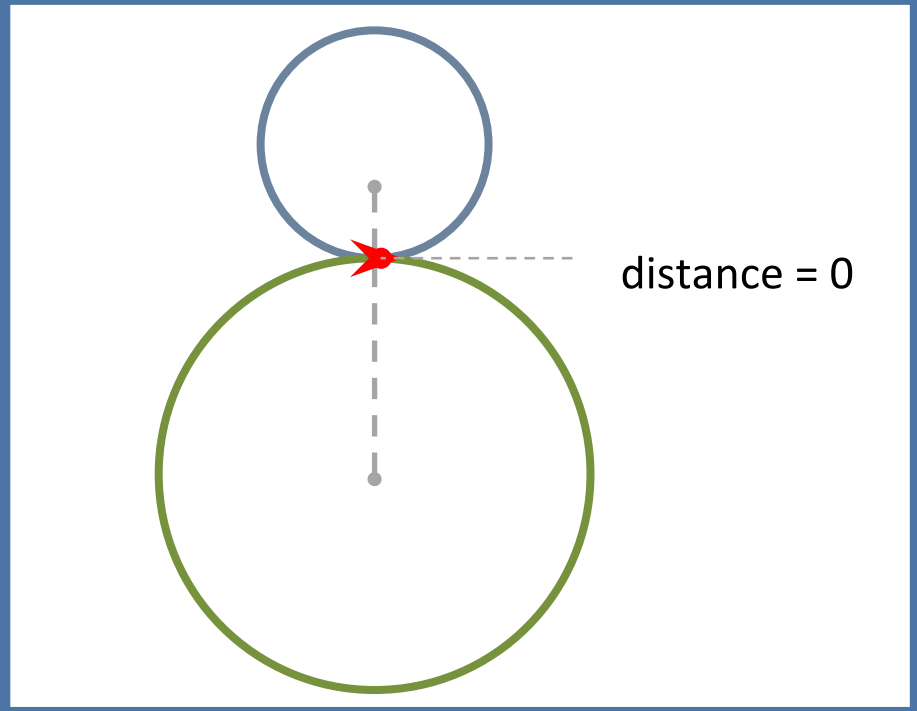


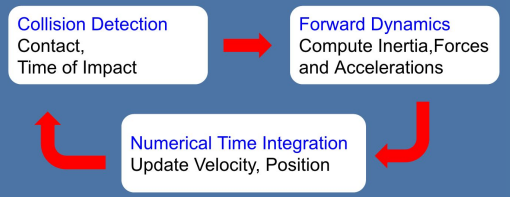
$$\tau = I\dot{\omega} + \omega \times I\omega$$

See “Stabilizing Gyroscopic Forces In Rigid Multibody Simulations”, Claude Lacoursiere,

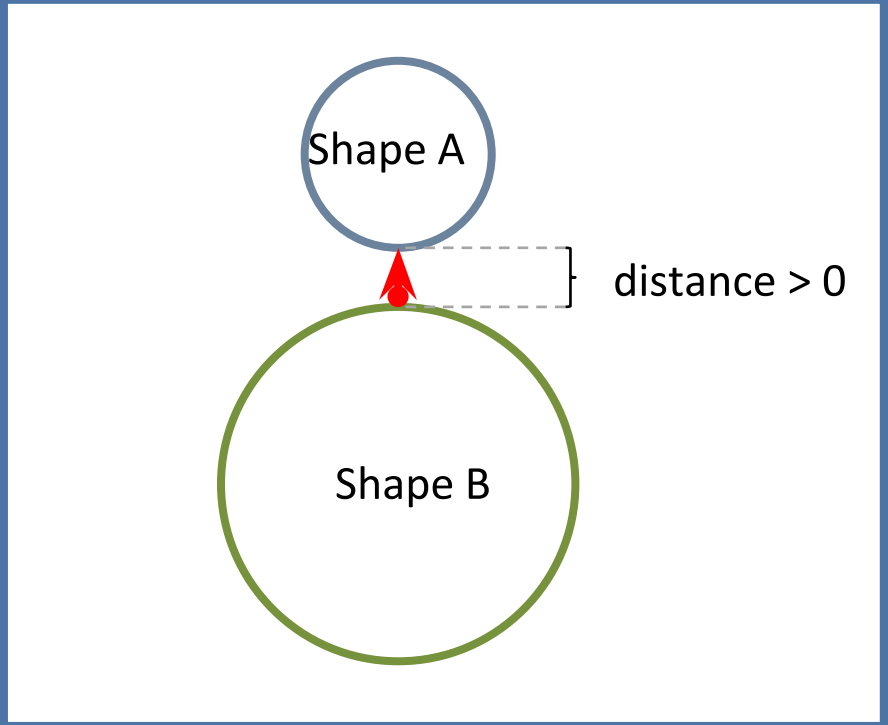
Also http://box2d.org/files/GDC2015/ErinCatto_NumericalMethods.pdf

Collision Detection: Contact Information

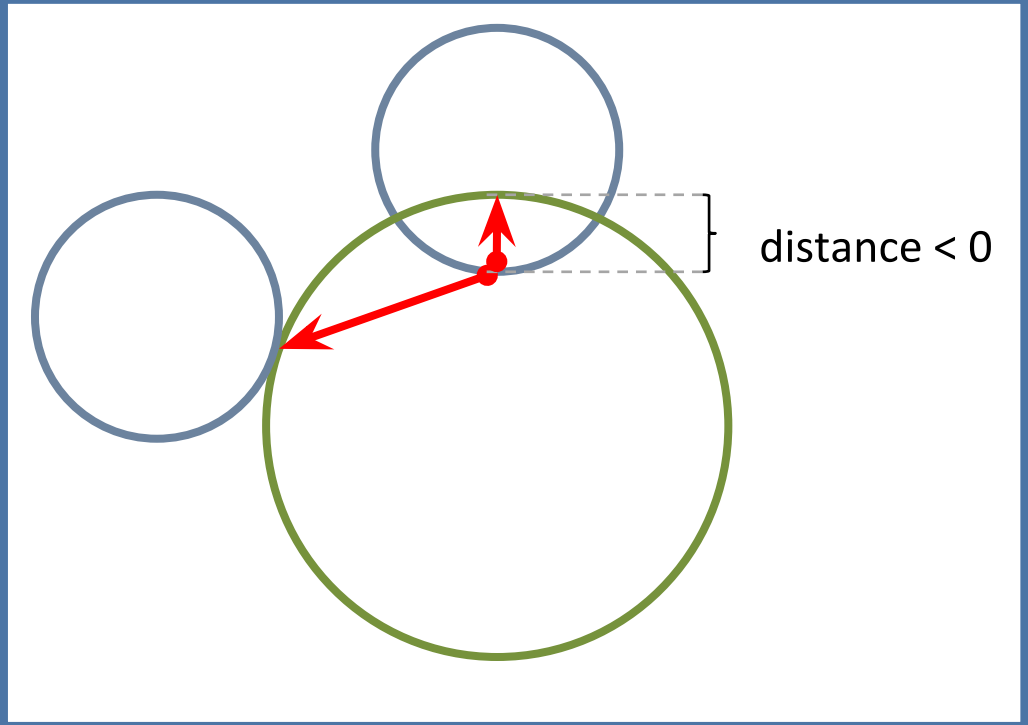




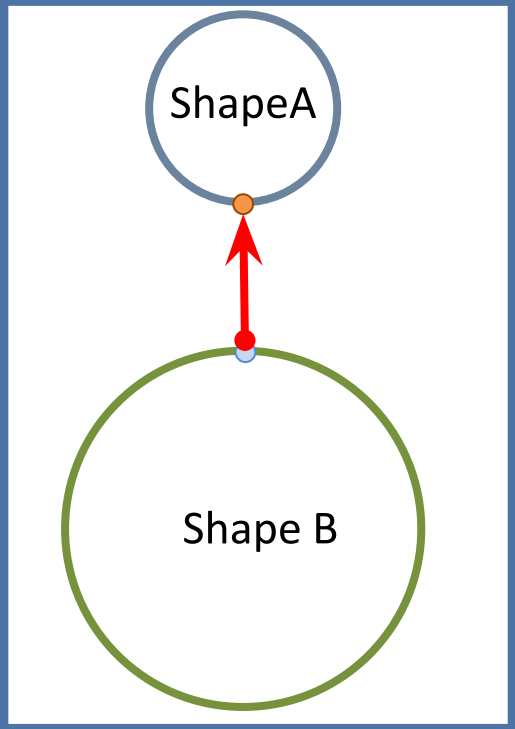
Shortest Distance



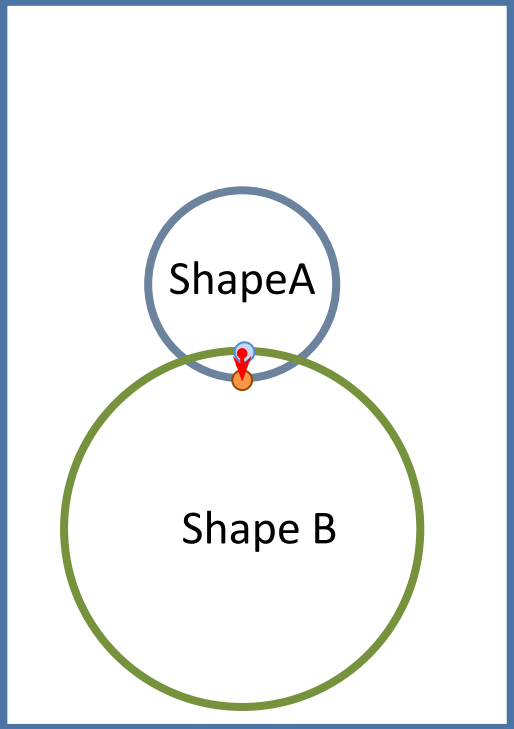
Separating Vectors



Contact Point

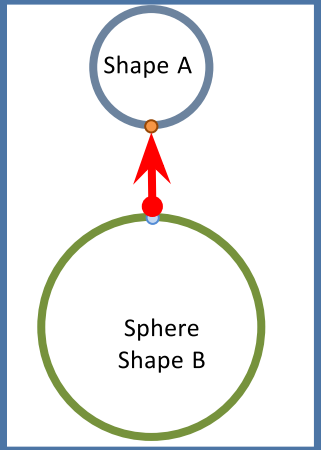


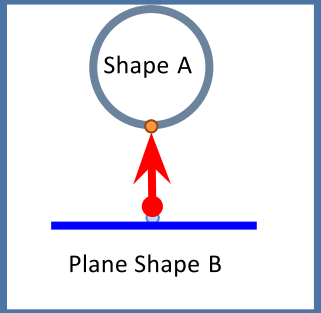
point on A
point on B
normal B->A
distance



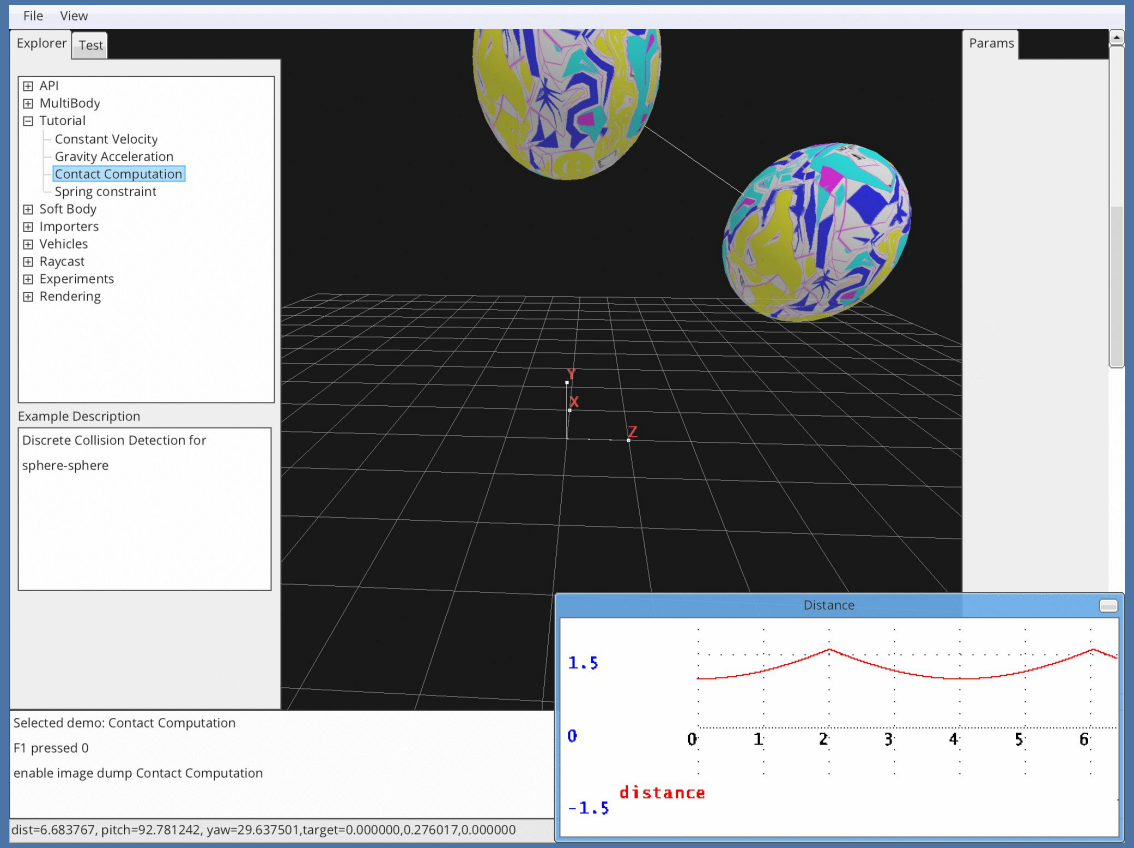
C++ Collision Contact Detector

```
struct LWContactPoint {
        b3Vector3 ptOnAWorld;
        b3Vector3 ptOnBWorld;
        b3Vector3 normalOnB;
        btScalar  distance;
};
struct LWPlane {
        b3Vector3   m_normal;
        btScalar    m_planeConstant;
};
struct LWSphere {
        btScalar    m_radius;
};
void ComputeClosestPointsSphereSphere(      const LWSphere& sphereA, const LWPose& sphereAPose,
                                            const LWSphere& sphereB, const LWPose& sphereBPose,
                                            LWContactPoint& pointOut) {
        b3Vector3 diff = sphereAPose.m_position-sphereBPose.m_position;
        btScalar len = diff.length();
        pointOut.distance = len - (sphereA.m_radius+sphereB.m_radius);
        pointOut.normalOnB = b3MakeVector3(1,0,0);
        if (len > B3_EPSILON) {
                pointOut.normalOnB = diff / len;
        }
        pointOut.ptOnAWorld = sphereAPose.m_position - sphereA.m_radius*pointOut.normalOnB;
        pointOut.ptOnBWorld = pointOut.ptOnAWorld-pointOut.normalOnB*pointOut.distance;
}
```

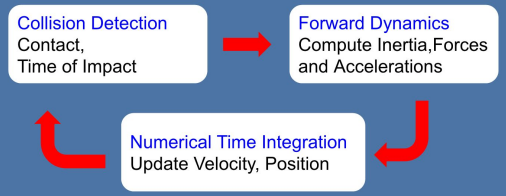




Collision Detection: Contact Point



Forward Dynamics



Forward Dynamics
Compute Inertia,Forces and Accelerations

In a nutshell f = ma becomes a = f/m

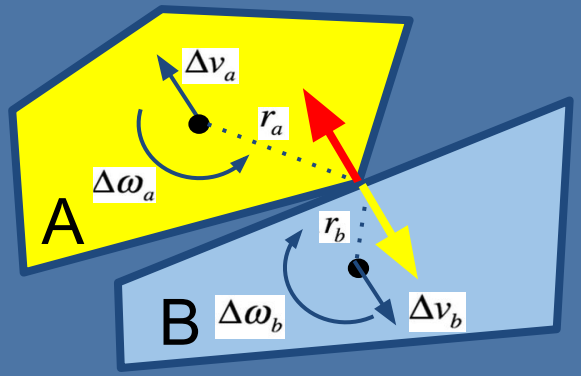
- Forces include gravity, friction, constraint, velocity dependent forces (gyroscopic), position dependent forces (spring)
- m (mass, Inertia) depends on system and position

“Solving” a Contact Constraint

The ultimate question we want to answer is - how big of an impulse do we need to apply to make the objects stop moving towards each other?

Let's assume we're applying an impulse of magnitude 1.0 (unit impulse) and see how big of a velocity change that would cause.

Collision Constraint



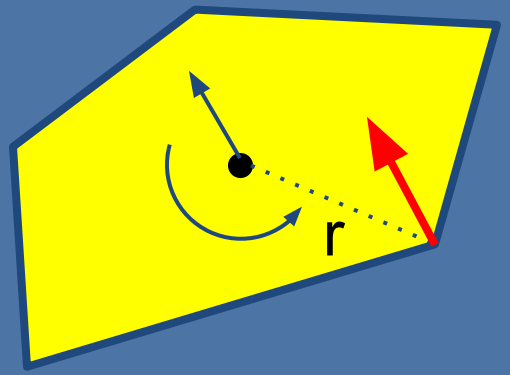
Effect of Impulse on Velocity

$$Impulse = F\Delta t = m\Delta v$$

$$Impulse_{torque} = \tau\Delta t = I\Delta\omega$$

$$\Delta v = \frac{Impulse}{m}$$

$$\Delta\omega = \frac{Impulse_{torque}}{I} = \frac{r \times Impulse}{I}$$



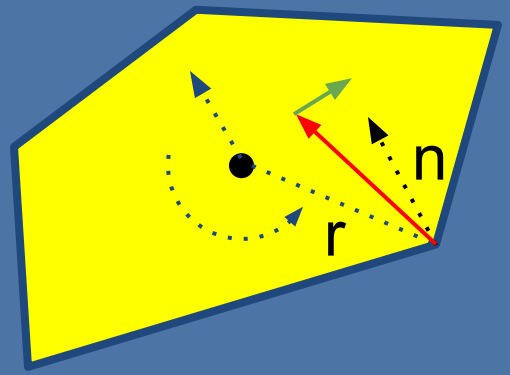
Single Body Effective Mass (inverse)

- Effective mass is change in velocity per impulse
  - projected on the normal

$$u_a = v_a + \omega_a \times r_a$$

$$\Delta u_a = \frac{Impulse}{m_a} + (\frac{r_a \times Impulse}{I_a}) \times r_a$$

$$effective\_mass_{inv} = \frac{dot(\Delta u_a, normal)}{|Impulse|}$$



Two Body Effective Mass (Inverse)

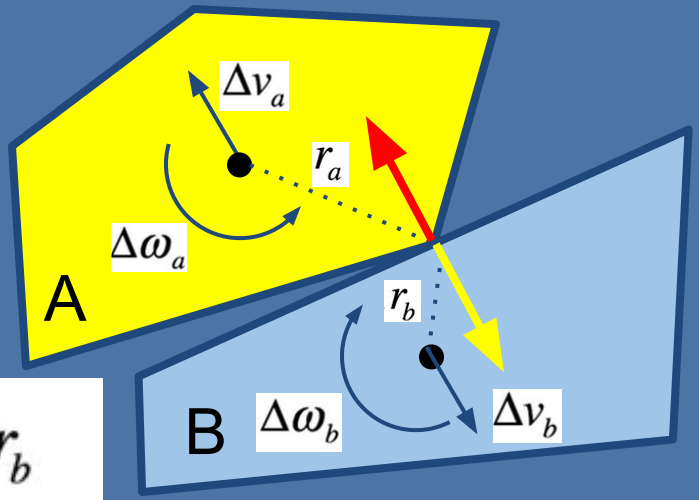
$$u_{ab} = u_a - u_b$$

$$= v_a - v_b + \omega_a \times r_a - \omega_b \times r_b$$

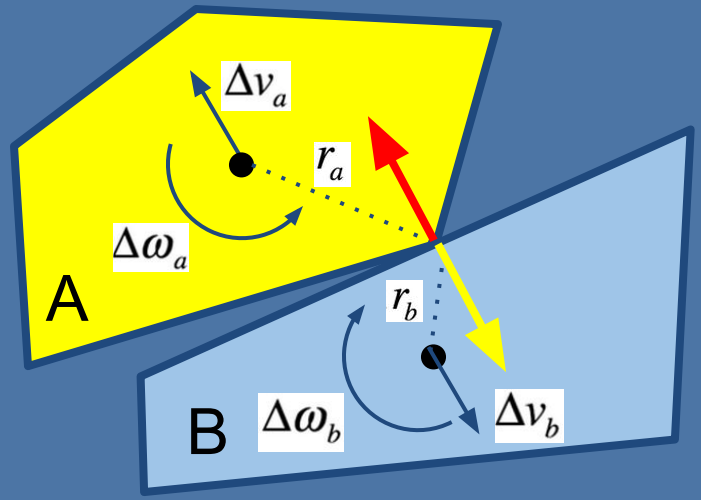
$$\Delta u_{ab} = \Delta v_a - \Delta v_b + \Delta \omega_a \times r_a - \Delta \omega_b \times r_b$$

$$\Delta u_{ab} = \frac{n}{m_a} - \frac{-n}{m_b} + (\frac{r_a \times n}{I_a}) \times r_a - (\frac{r_b \times -n}{I_b}) \times r_b$$

$$effective\_mass_{inv} = \frac{dot(\Delta u_{ab}, normal)}{|Impulse|}$$



Resolve a Single Collision

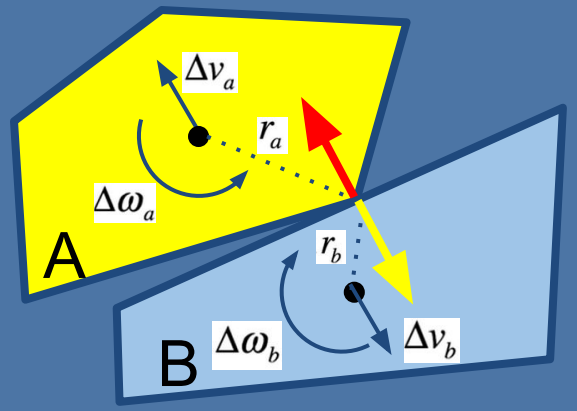


$$p_{correction} = M_{effective} \Delta v_{desired} \cdot n$$

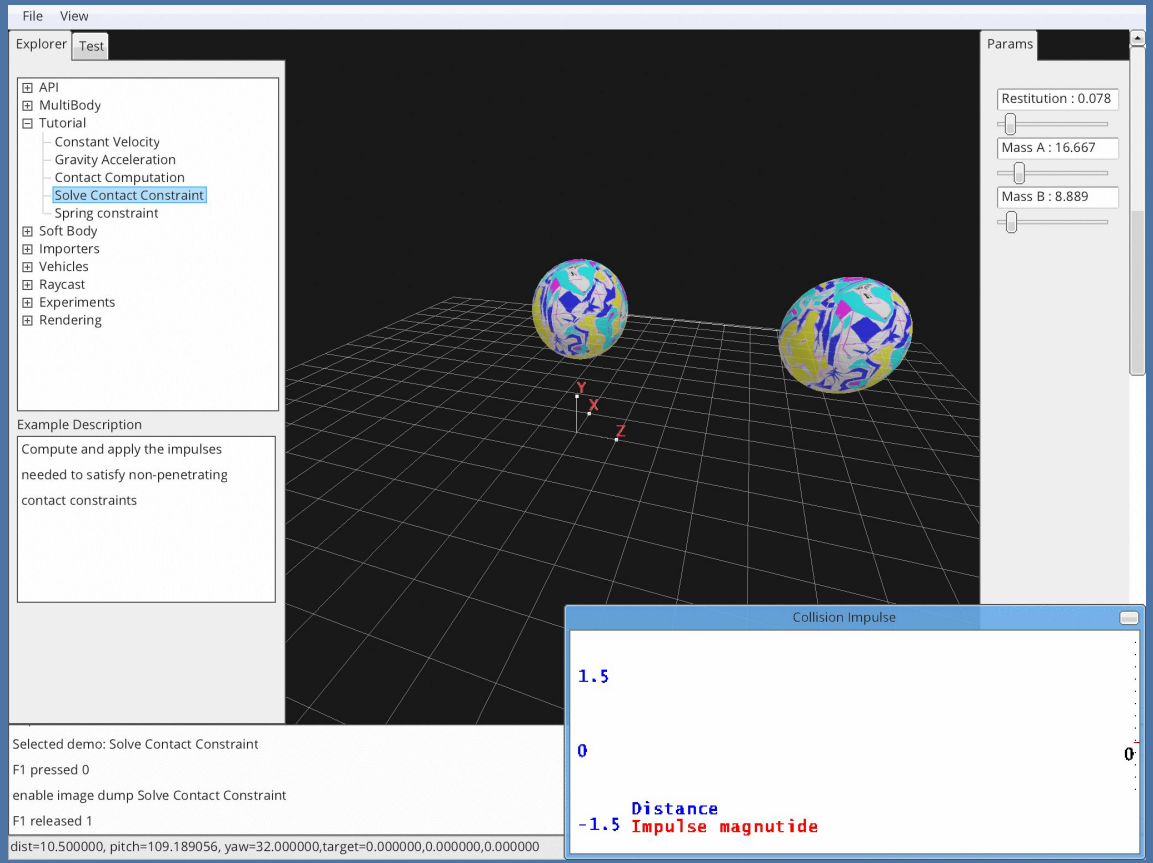
$$p_{correction} = \frac{-\Delta v_{ab} \cdot n}{n \cdot n\left(\frac{1}{M_a} + \frac{1}{M_a}\right) + \left(\frac{r_a \times n}{I_a}\right) \times r_a + \left(\frac{r_b \times n}{I_b}\right) \times r_b}$$

C++ Single Collision Constraint Solver

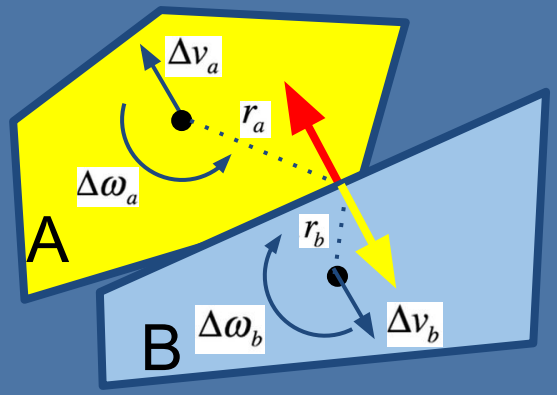
```
b3Scalar resolveCollision(LWRigidBody& bodyA,
                          LWRigidBody& bodyB,
                          LWContactPoint& contactPoint) {
        b3Assert(contactPoint.m_distance<=0);
        btScalar appliedImpulse = 0.f;
        b3Vector3 rel_pos1 = contactPoint.m_ptOnAWorld - bodyA.m_worldPose.m_position;
        b3Vector3 rel_pos2 = contactPoint.m_ptOnBWorld - bodyB.getPosition();
        btScalar rel_vel = contactPoint.m_normalOnB.dot(bodyA.getVelocity(rel_pos1) - bodyB.getVelocity(rel_pos2));
        if (rel_vel < -B3_EPSILON)     {
                b3Vector3 temp1 = bodyA.m_invInertiaTensorWorld * rel_pos1.cross(contactPoint.m_normalOnB);
                b3Vector3 temp2 = bodyB.m_invInertiaTensorWorld * rel_pos2.cross(contactPoint.m_normalOnB);
                btScalar impulse = -(1.0f + gRestitution) * rel_vel /
                (bodyA.m_invMass + bodyB.m_invMass + contactPoint.m_normalOnB.dot(temp1.cross(rel_pos1) + temp2.cross
(rel_pos2)));
                b3Vector3 impulse_vector = contactPoint.m_normalOnB * impulse;
                appliedImpulse = impulse;
                bodyA.applyImpulse(impulse_vector, rel_pos1);
                bodyB.applyImpulse(-impulse_vector, rel_pos2);
        }
        return appliedImpulse;
}

        if (contactPoint.m_distance<0) {
                m_bodies[0]->computeInvInertiaTensorWorld();
                m_bodies[1]->computeInvInertiaTensorWorld();
                b3Scalar appliedImpulse = resolveCollision(*m_bodies[0], *m_bodies[1], contactPoint);
        }
```



Resolve Contact Constraint



Position Error Correction (Penetration)



Baumgarte: Modify relative target velocity, so that after one or more integration steps, the error will reduce. dX = v*dt.

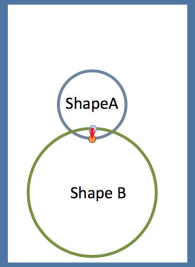
Drawback: can add a lot of kinetic energy, difficult to tune parameter.

Coordinate Projection, Split Impulse: Save the current velocity, set the relative target velocity for a constraint so that the error will reduce, perform an integration step with this temporary velocity, and revert to origin velocity.

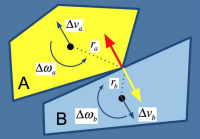
See “Stabilization of Constrained Mechanical Systems with DAEs and Invariant Manifolds”, Ascher, Chin, 1995.

Rigid Body Simulation Loop

point on A
point on B
normal B->A
distance



$$p_{correction} = M_{effective} \Delta v_{desired} \cdot n$$



$$p_{correction} = \frac{-\Delta v_{ab} \cdot n}{n \cdot n(\frac{1}{M_a} + \frac{1}{M_a}) + (\frac{r_a \times n}{I_a}) \times r_a + (\frac{r_b \times n}{I_b}) \times r_b}$$

Collision Detection
Contact,
Time of Impact

Forward Dynamics
Compute Inertia,Forces and Accelerations

Numerical Time Integration
Update Velocity, Position

$$v_{t+\Delta t} = v_t + a\Delta t = v_t + \frac{F_{ext} + F_c}{m}\Delta t = v_t + \frac{F_{ext}}{m}\Delta t + \frac{Impulse_c}{m}$$

$$x_{t+\Delta t} = x_t + v_{t+\Delta t}\Delta t$$

F_c could be constraint forces such as contact, friction, joints

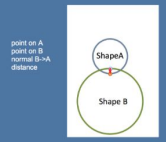
What Next?

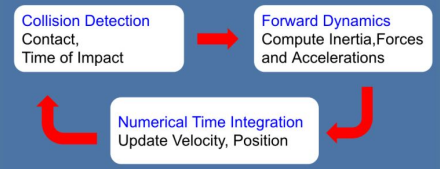
Discrete versus Continuous,

Time of Impact

Different Collision Shape Types

Collision Culling, Acceleration Structures

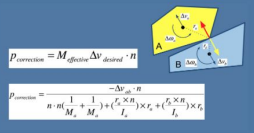




$$v_{t+\Delta t} = v_t + a\Delta t = v_t + \frac{F_{ext} + F_c}{m}\Delta t = v_t + \frac{F_{ext}}{m}\Delta t + \frac{Impulse_c}{m}$$
$$x_{t+\Delta t} = x_t + v_{t+\Delta t}\Delta t$$

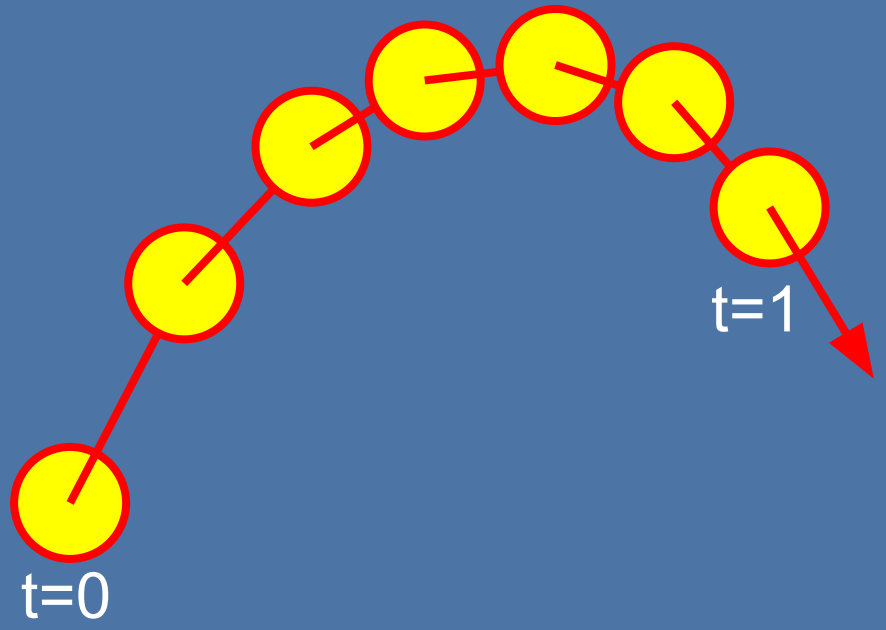
F_c could be constraint forces such as contact, friction, joints

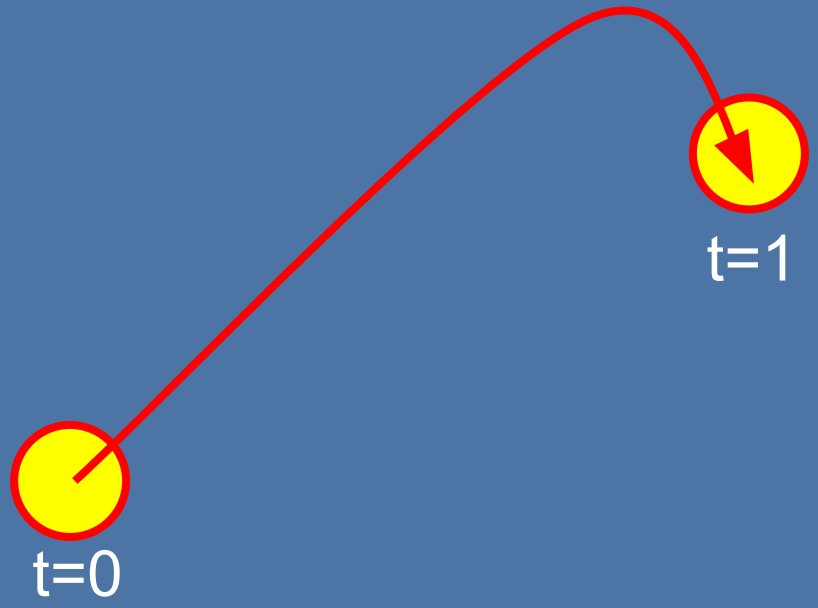
LCP Solvers and Featherstone



GPU OpenCL acceleration

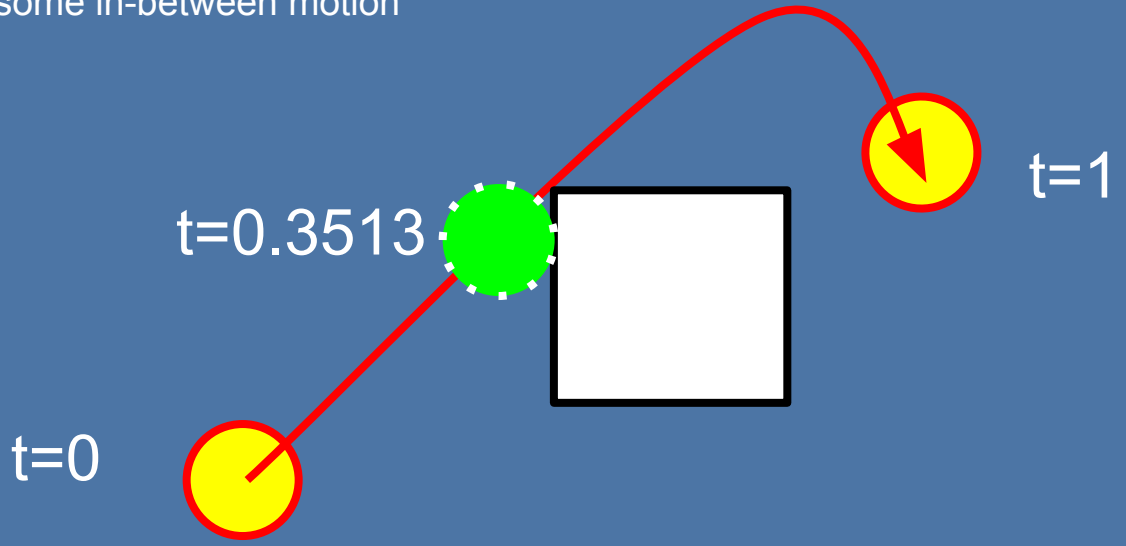
Discrete versus Continuous





Continuous Collision Check

- What is the first Time of Impact (TOI) ?
  - Given some in-between motion



See “Continuous Collision Detection of General Convex Objects under Translation”, Gino van den Bergen, http://dtecta.com/interesting

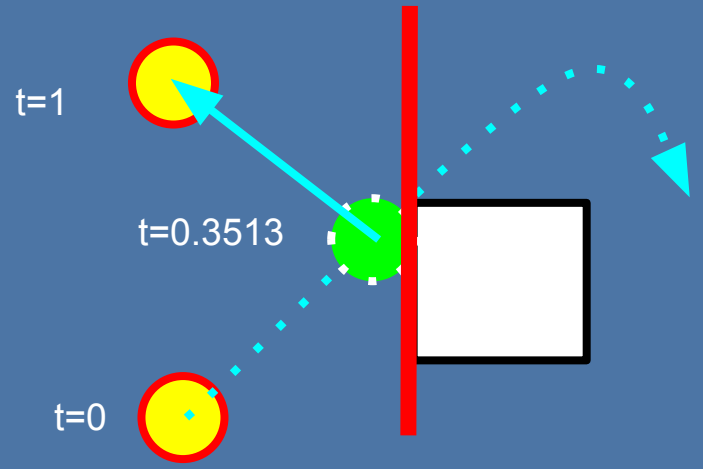
“Continuous Collision”, Erin Catto, http://box2d.org/downloads

“Controlled conservative advancement for continuous collision detection”, Y. Kim, D. Manocha

Continuous Physics Simulation

Apply collision forces at TOI would require subdivision of the time step.

Instead, we can add ‘speculative’ geometric constraints at the beginning of the time step. Instead of infinite half-plane, could use polytope geometry.



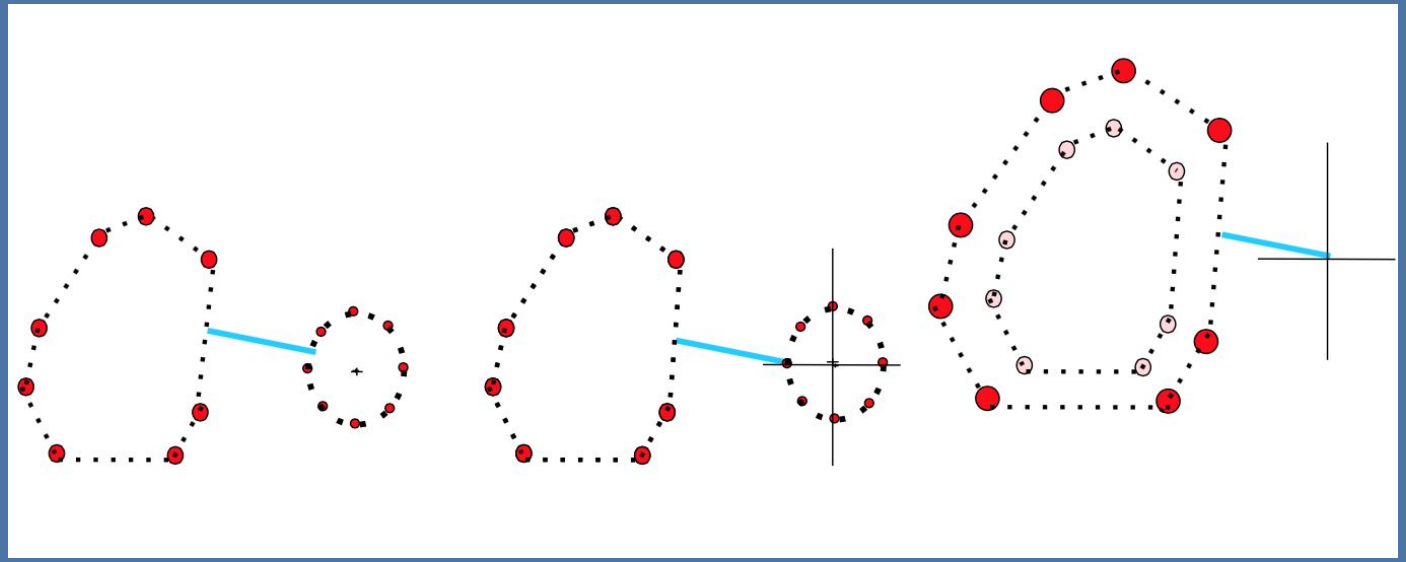
See (new!) “ A complementarity based contact model for geometrically accurate treatment of polytopes in simulation” by Jedediyah Williams, Ying Lu, Jeff Trinkle.

“A Different Approach for Continuous Physics”, Vincent Robert, Ubisoft, GDC 2012

Discrete Collision Detection for Convex Shapes

- GJK or SAT overlap check+GJK distance
- EPA, MPR or SAT for penetration info
- Contact Cache or Clipping for contact points

GJK



- Combine two convex objects into a single convex object
- Iteratively compute the distance of the combined object and the world origin
  Termination condition are sensitive to numerical issues. Cannot compute penetration information.

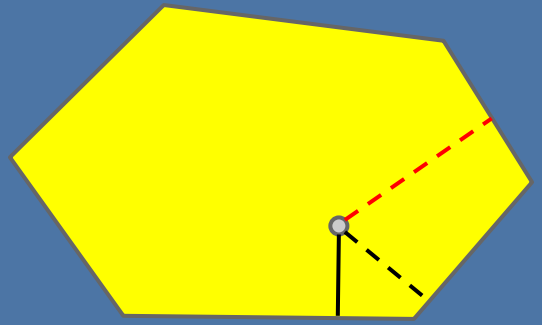
See “Collision Detection in Interactive 3D Environments” book by Gino van den Bergen

Expanding Polytope Algorithm, EPA

EPA gives global penetration depth for general convex shapes

EPA is build on top of GJK, an algorithm that computes if convex objects overlap, and if not, compute closest points.

Minkowski Portal Refinement, MPR

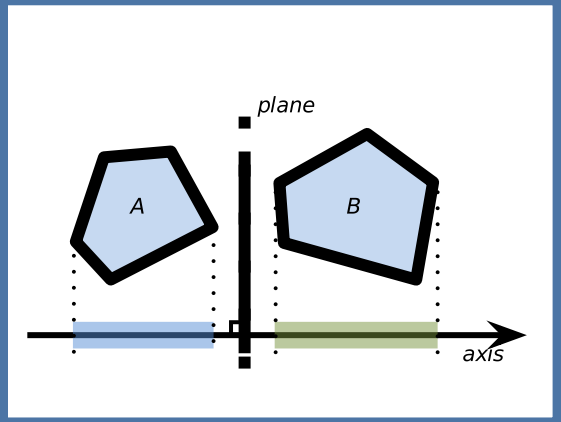


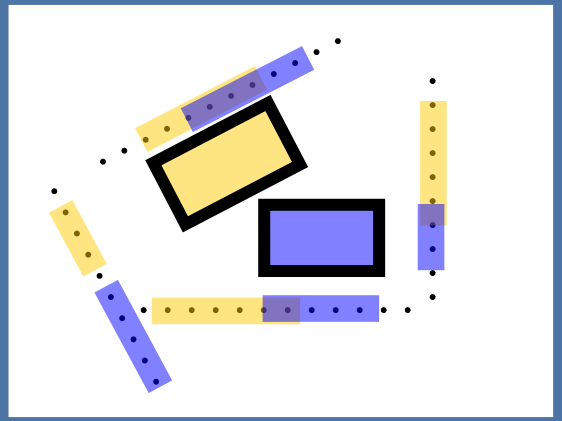
Local Penetration Depth, General Convex, Fast

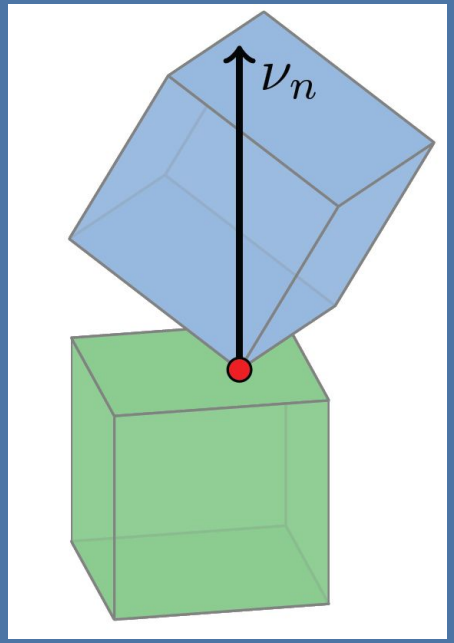
Gary Snethen (GPG7, http://github.com/erwincoumans/xenocollide)

Daniel Fiser, libccd, Bullet

Separating Axis Test



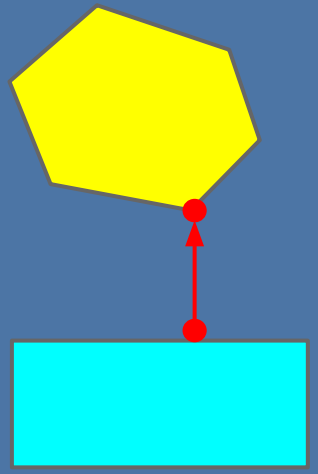




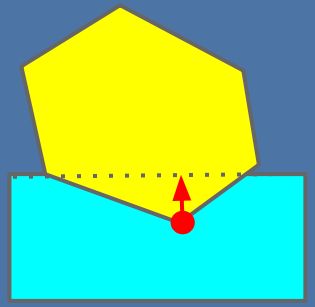
Number of SAT Tests

3D Objects	Face dirs (A)	Face dirs (B)	Edge dirs (AxB)	Total
Segment–Tri	0	1	1x3	4
Segment–OBB	0	3	1x3	6
AABB–AABB	3	0(3)	0(3x0)	3
OBB–OBB	3	3	3x3	15
Tri–Tri	1	1	3x3	11
Tri–OBB	1	3	3x3	13

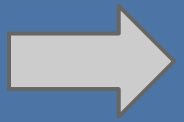
Multiple Contact Points

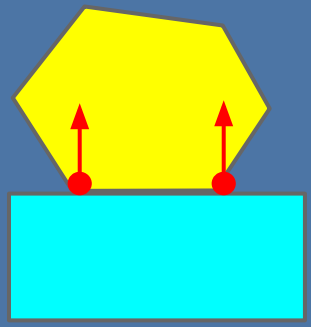


Closest Points,
Distance,
Normal (GJK)



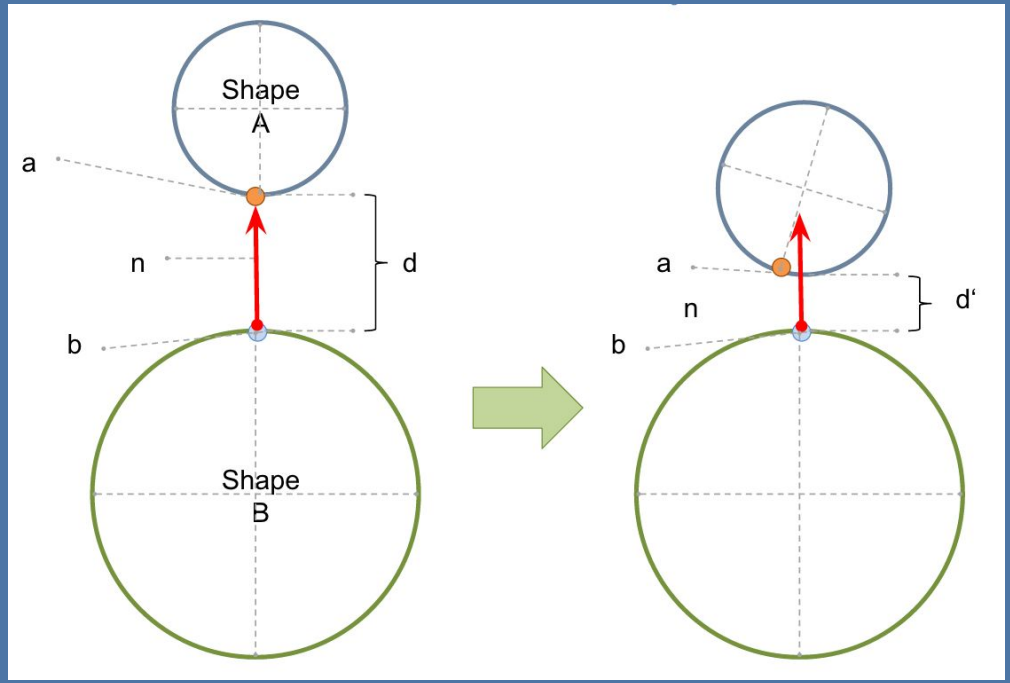
Closest Points,
Penetration Depth
(EPA, MPR)





Contact Point Cache

Contact Cache Update

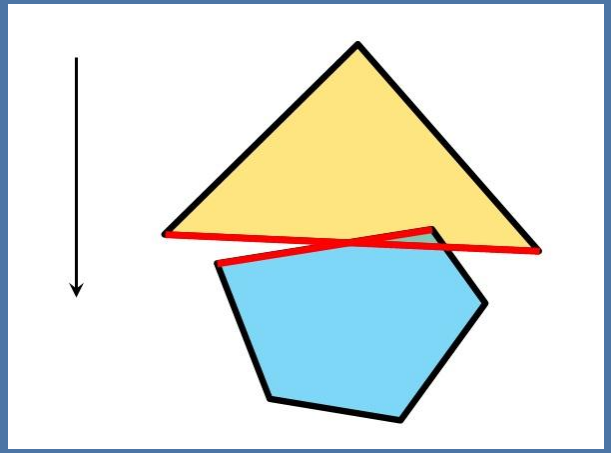


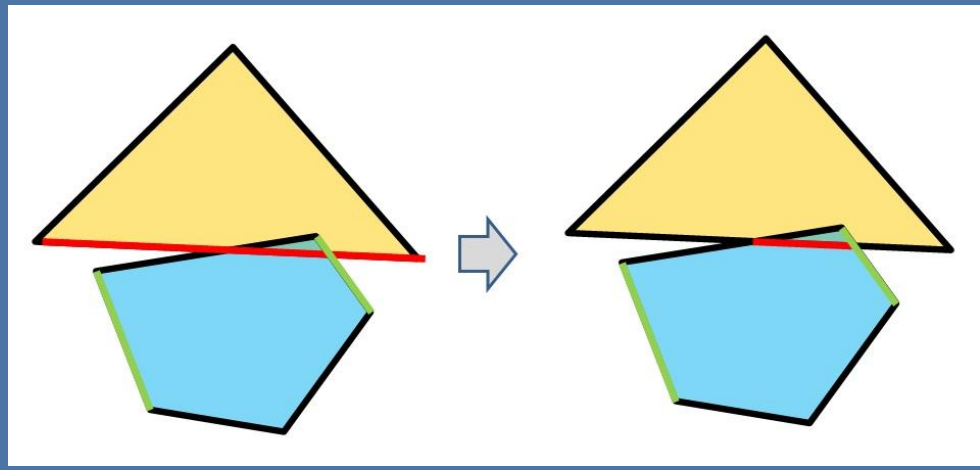
Maintain contact points in local space

Recompute distance based on origin contact normal in world space

Remove points based on tolerances

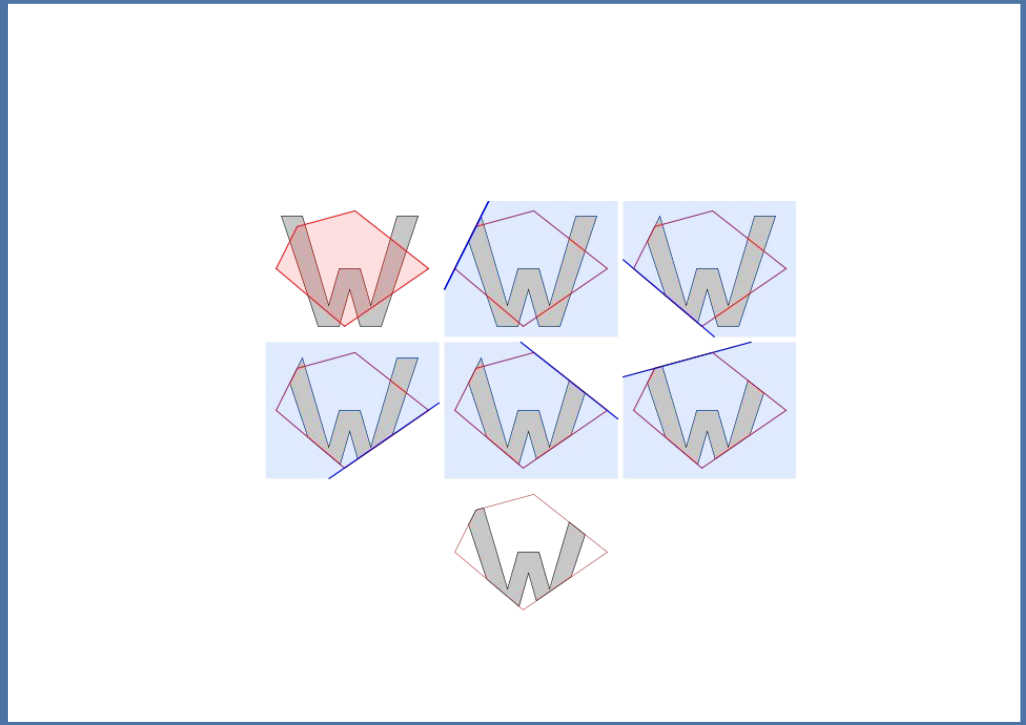
Polyhedral Clipping





Pick a plane in each convex polyhedraon, (based on separating axis) and clip

Sutherland Hodgeman Clipping

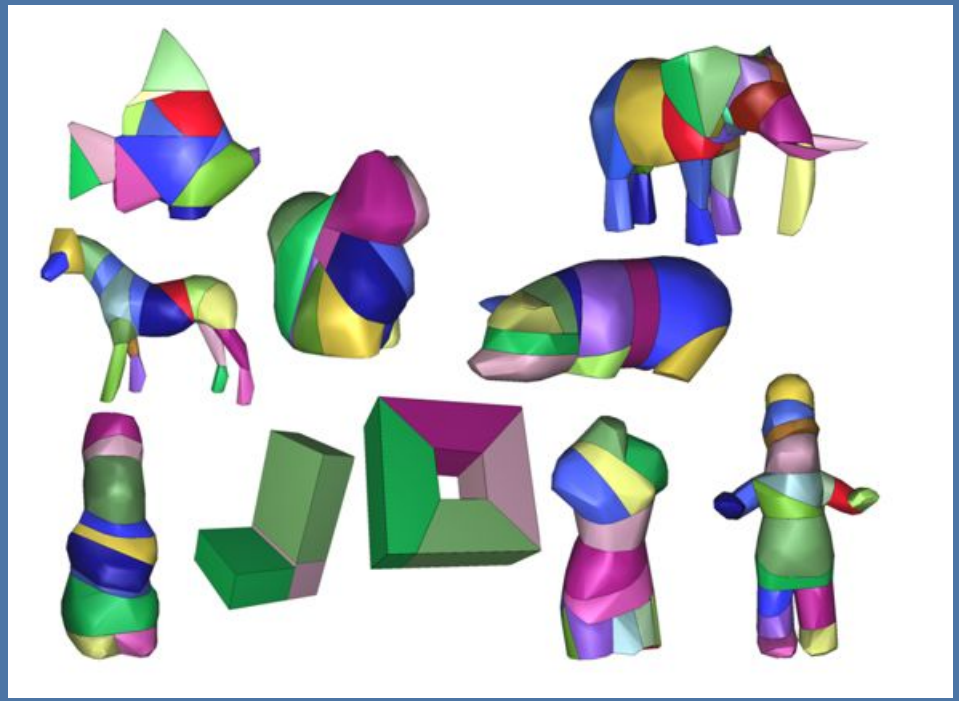


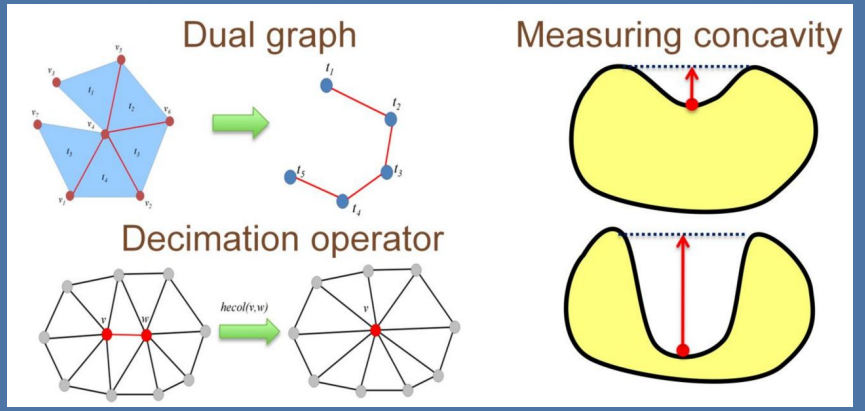
Clipping Single Planes not always sufficient.

See “Game Physics Pearls” book for more information about convex hull collision.

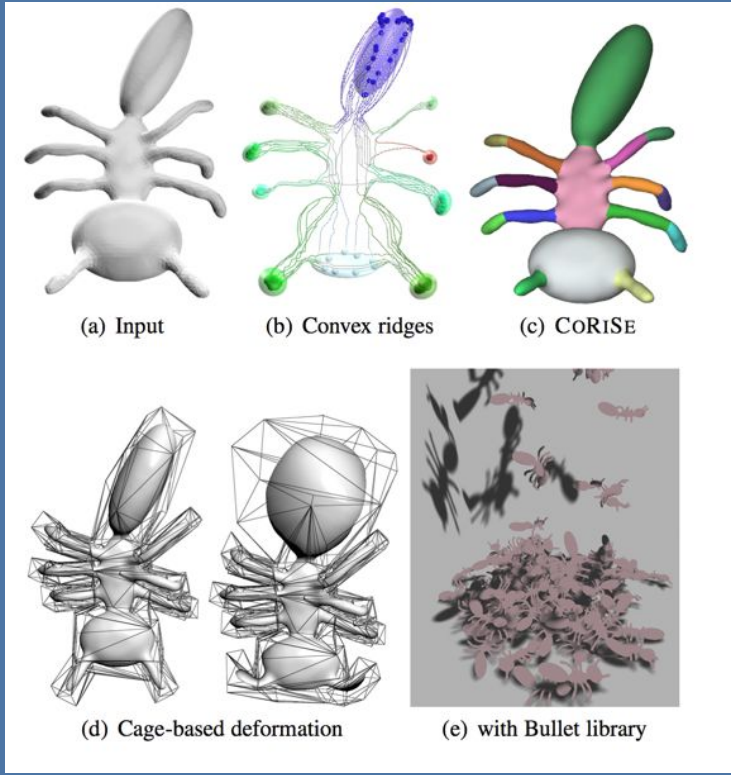
GDC Physics Diner

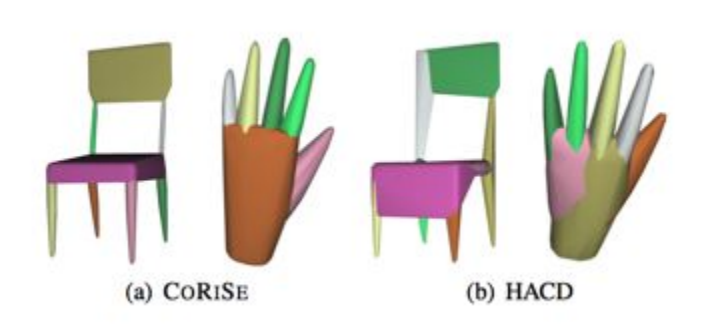
Convex Decomposition, HACD



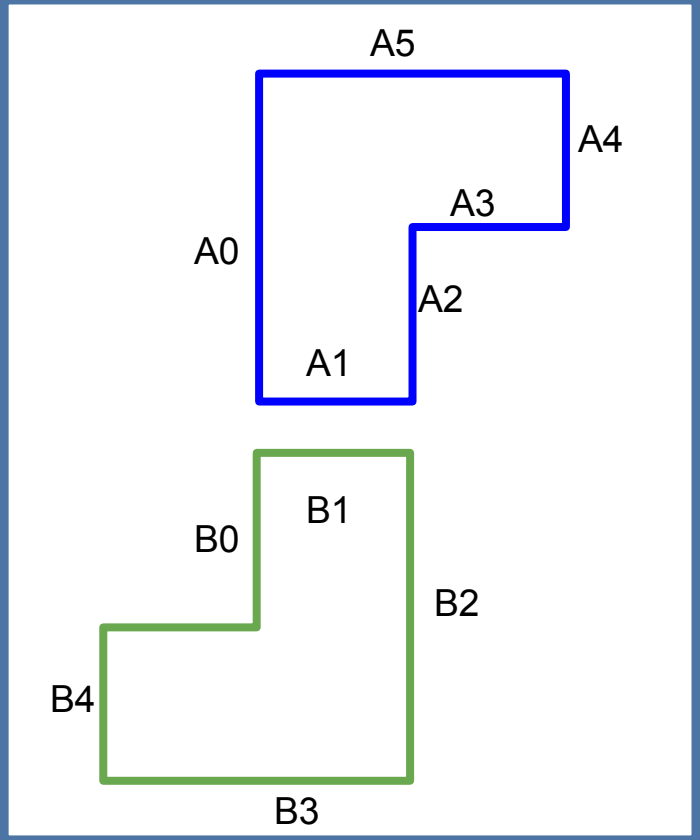


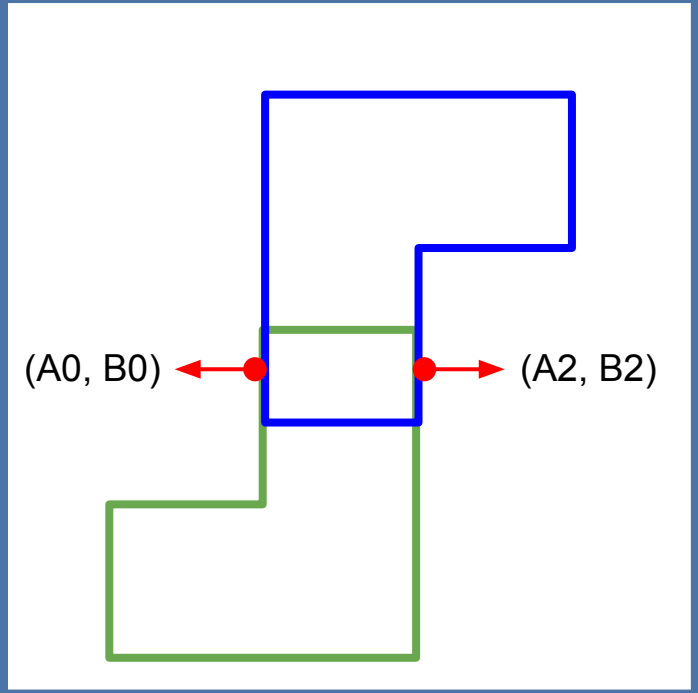
CoRise, Guilin Liu





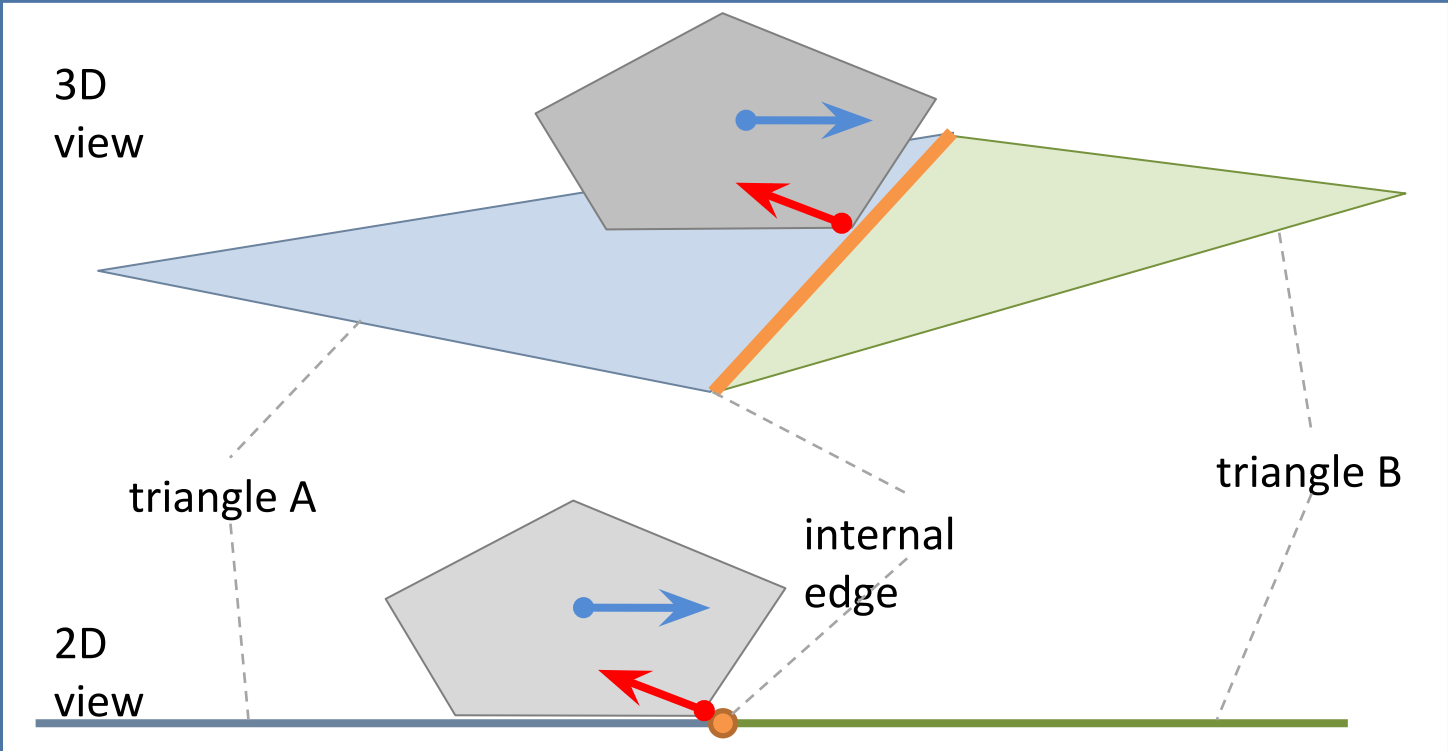
Concave Global Penetration





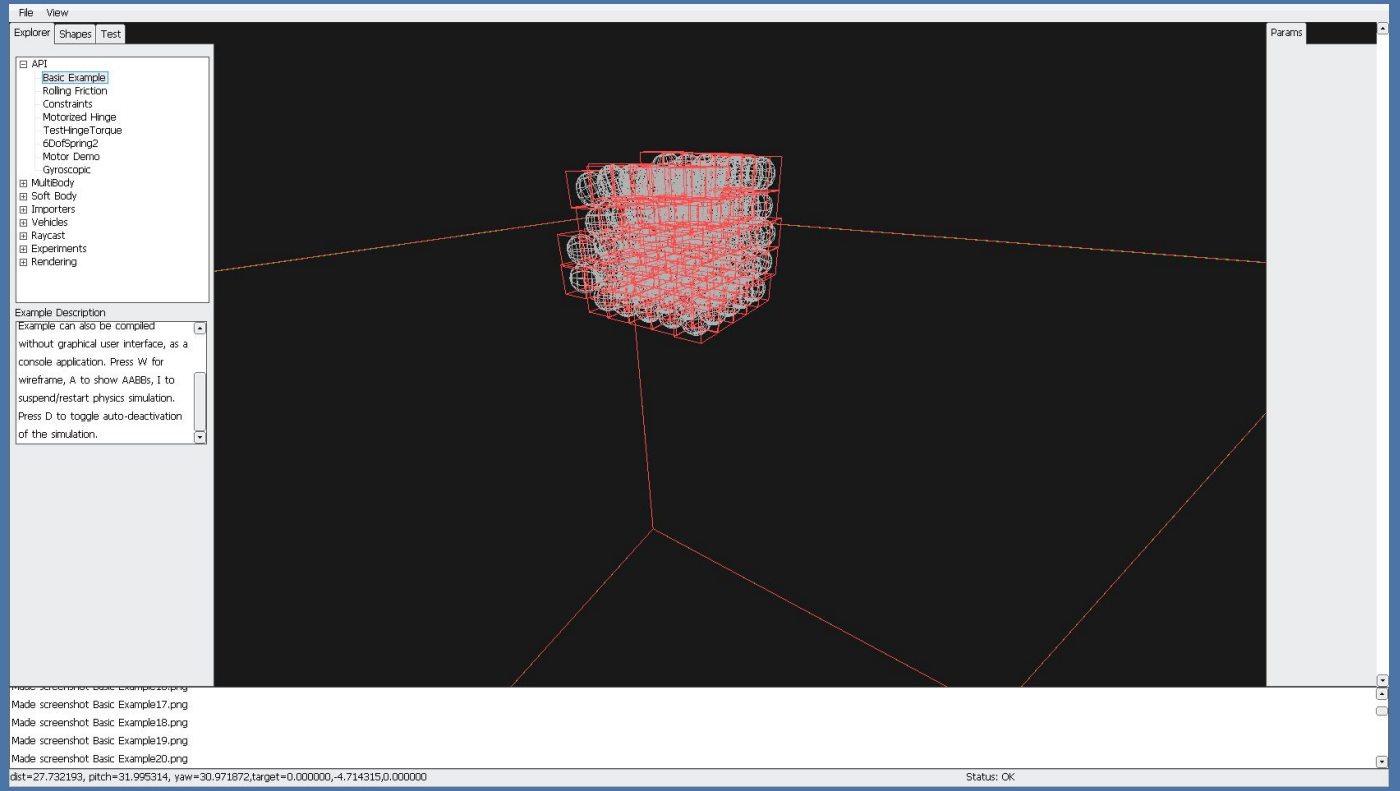
There is no (A1, B1) for hollow shapes, unless we create solid parts using convex decomposition, or use SDF.

Internal Edge Collision



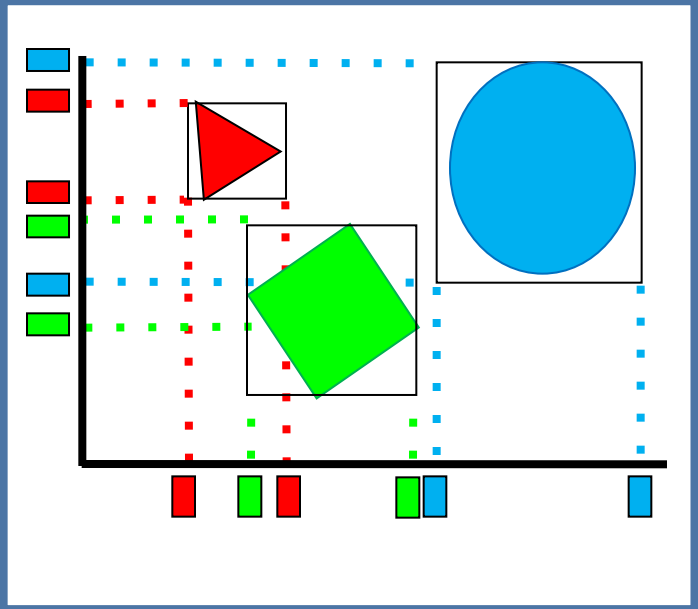
See Contact Generation for Meshes, Pierre Tediman, http://www.codercorner.com/MeshContacts.pdf

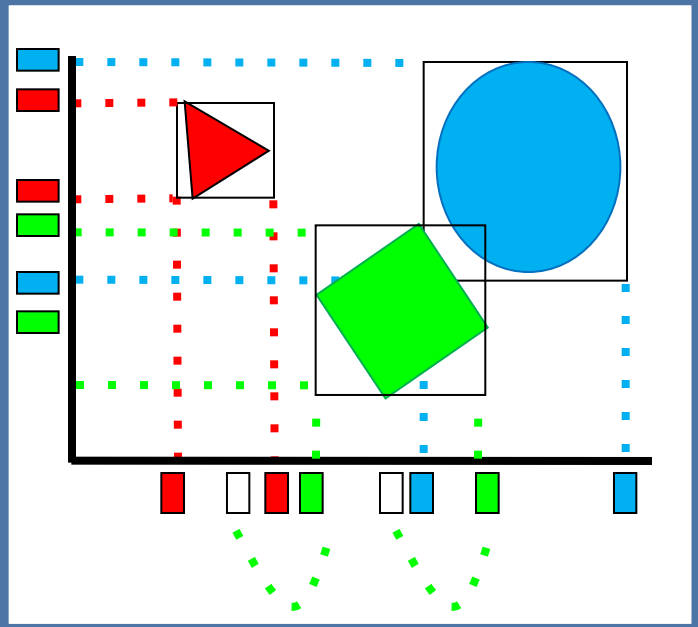
Broad Phase Acceleration Structures



Incremental sweep and prune

Update 3 sorted axis and overlapping pairs

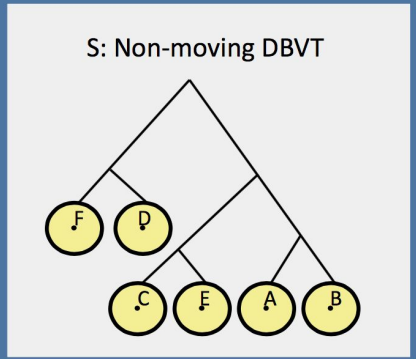


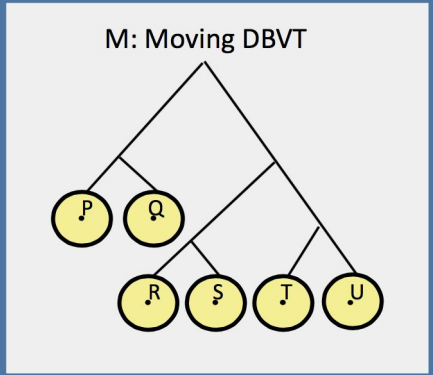


Dynamic BVH AABB tree

Key Ideas:

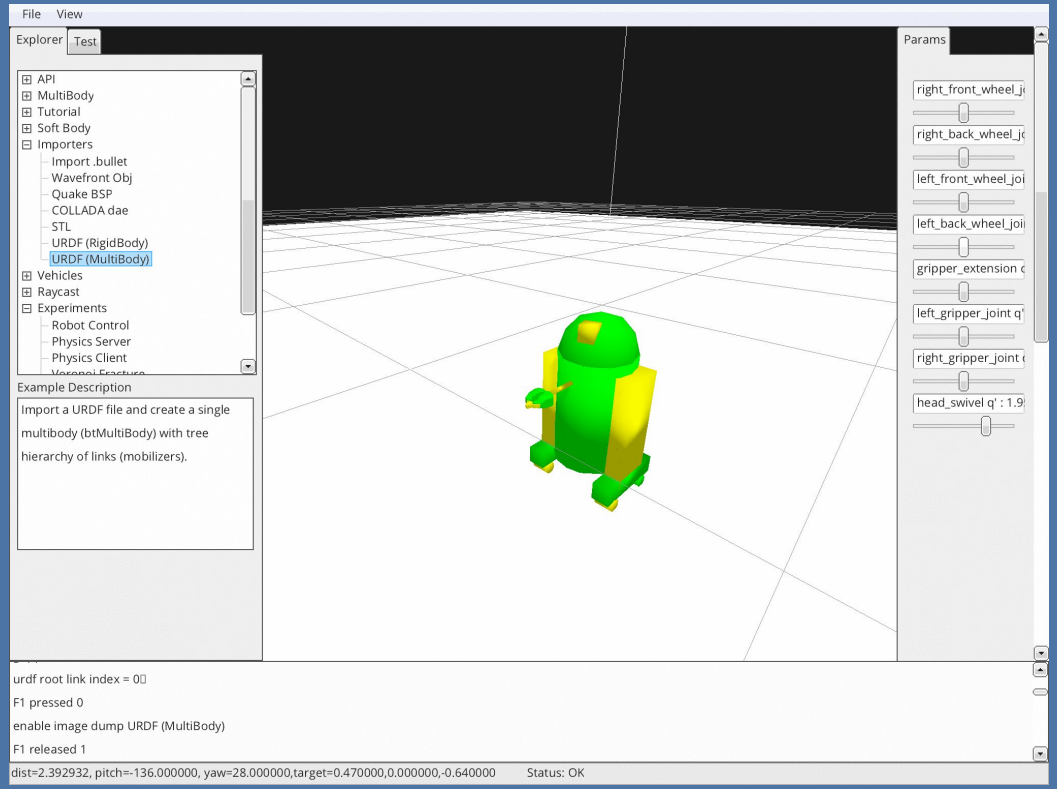
- Don’t rebuild tree from scratch,
  Fast incremental node insert and removal
- Maintain multiple trees,
  objects move from one to the other.
- Accelerates pair search, CCD, ray casts
  Compound Shapes, Soft Body





See also “Fast, Effective BVH Updates for Animated Scenes”, Daniel Kopta et.al.

Small Break



Press F1 in Example Browser, will generate png files, skipping ‘n’ frames. See ‘skip’ in Bullet/examples/ExampleBrowser/OpenGLExampleBrowser.cpp
Install imagemagick
convert -background white -alpha remove -layers OptimizePlus -quality 99 -delay 10 -loop 0 *.png animated3.gif