Collision detection in Chrono
Collision shapes
Collision shapes

- Collision shapes are defined respect to the **REF frame** of the body
- Spheres, boxes, cylinders, convex hulls, ellipsoids, compounds,...
- Concave shapes: decompose in compounds of convex shapes
- For simple ready-to-use bodies with predefined collision shapes, can use:
  - ChBodyEasySphere,
  - ChBodyEasyBox,
  - etc.
Specifying collision shapes

• Typical steps to setup collision:
  body_b->GetCollisionModel()->ClearModel();
  body_b->GetCollisionModel()->AddSphere(myradius);
  ...
  body_b->GetCollisionModel()->BuildModel();
  body_b->SetCollide(true);

• Collision ‘families’ for selective collisions:
  // Change from default collision family (0)
  body_b->GetCollisionModel()->SetFamily(2);
  body_b->SetFamilyMaskNoCollisionWithFamily(4);
Collision tolerances

• Set these tolerances before creating collision shapes:
  
  ```cpp
  ChCollisionModel::SetDefaultSuggestedEnvelope(0.001);
  ChCollisionModel::SetDefaultSuggestedMargin(0.0005);
  ChCollisionSystemBullet::SetContactBreakingThreshold(0.001);
  ```

• Envelope (outward)
  • Represents the search volume for potential collision
  • Allows numerical schemes to anticipate collisions ahead of time
  • With zero envelope, the solver may first ‘see’ a collision with bodies already interpenetrated → inaccurate and shaky simulation

• Margin (inward)
  • Defines a range of penetrations within which faster collision detection algorithms can be safely used

• Contact breaking threshold
  • Distance beyond which contact between two shapes previously in contact is discarded
  • Bullet-specific setting (related to contact persistence in Bullet)
Recommendations

• Collision shapes and visualization assets do not need to match.
  • one may have a detailed visualization shape for rendering purposes, yet the collision shape is much simpler to avoid a slowdown of the simulation.

• Avoid shapes that are too thin, too flat, or in general that lead to extreme size ratios

• Use collision families to control what shapes interact through contact

• Collision tolerances:
  • *Too large collision envelope*: too many potential contacts, high CPU time, high waste of RAM
  • *Too small collision envelope*: risk of tunnelling effects, unstable simulation of stacked objects

  • *Too large collision margin*: shapes are ‘rounded’ too much
  • *Too small collision margin*: when interpenetration occurs beyond this value, an inefficient algorithm is used
Collision detection primer
Collision detection basics

• Collision detection implies:
  • Deciding what to test
  • Performing collision tests
    • Determining whether a collision occurred
    • Determining when a collision occurred
    • Determining where a collision occurred
  • Processing results
    • “Collision handling”

• A naïve approach is O(n²)
  • Check for collisions between objects by comparing all possible combinations
Two-phase collision detection

1. Broad-phase
   • Find pairs to compare
   • Use bounding volumes (AABB, OBB, spheres)
   • Goals:
     • efficiently determine pairs of objects that cannot collide
     • accuracy is not a major concern

2. Narrow-phase
   • Compare individual pairs
   • Use exact shape geometry
   • Goals:
     • efficiently and accurately determine pairs of objects that do collide
     • completely characterize existing collisions (from a geometric point of view)
Broad-phase algorithms

• Dynamic AABB trees
  • well optimized, general-purpose broad-phase algorithm
  • structure adapts dynamically to the size of the scene and its contents
  • fast object addition/deletion
  • handles well scenes with many objects in motion

• Sweep and Prune (SAP)
  • good general-purpose broad-phase algorithm
  • best performance for dynamic world where most objects have little or no motion
  • limitation: requires scene of fixed size, known beforehand

• Hierarchical grids
  • Good general-purpose broad-phase algorithm, based on binning
  • Relatively easy to parallelize
  • limitation: with few levels, performance decreases when object size varies very much

• Several other...
Narrow-phase algorithms

• Analytical methods for simple primitive shapes
  • Example: sphere-sphere collision
    \[ \delta = |\vec{C}_1 \vec{C}_2| - (R_1 + R_2) \]
    \[ \hat{n} = \frac{\vec{C}_1 \vec{C}_2}{|\vec{C}_1 \vec{C}_2|} \]
    ...

  • Can be defined for several primitive shape pairs (sphere-box, box-box, sphere-capsule, etc.)
  • Most efficient and accurate

• Separating Axis Theorem (SAT)
  • Test intersection of object projections on a set of different axes
Narrow-phase algorithms

• Gilbert-Johnson-Keerthi (GJK) algorithm
  • Solves proximity queries for arbitrary convex objects (as long as they can be described in terms of a support mapping function)

• Iterative process applied to the Minkovski difference of two polyhedra
  (A and B intersect ⇔ A-B contains the origin)
Narrow-phase algorithms

- Minkovski Portal Refinement (MPR)
  - Developed by Gary Snethen in 2006
  - Like GJK, relies on convex shapes that can be defined in terms of a support mapping function
  - Unlike GJK, does not provide the shortest distance between separated shapes
  - Simpler implementation and more numerically robust than GJK

http://xenocollide.snethen.com/
Collision detection algorithms in Chrono

• Chrono::Engine
  • Relies on Bullet ([http://bulletphysics.org](http://bulletphysics.org)) for collision detection
  • Broad-phase: dynamic AABB trees
  • Narrow-phase: GJK

• Chrono::Parallel
  • Custom collision detection
  • Broad-phase: uniform binning (experimental 2-level grids)
  • Narrow-phase: hybrid (analytical/SAT – MPR)
  • Option for Bullet collision detection
Contact material properties
Specifying contact method at system construction (1/3)

- The contact method is implicitly specified by the type of physical system constructed
- The base class ChSystem uses the complementarity approach to treat contacts (if any)

```cpp
class ChApi ChSystem : public ChAssembly, public ChIntegrableIIorder {
    /// Create a physical system.
    /// Note, in case you will use collision detection, the values of
    /// 'max_objects' and 'scene_size' can be used to initialize the broadphase
    /// collision algorithm in an optimal way. Scene size should be approximately
    /// the radius of the expected area where colliding objects will move.
    /// Note that currently, by default, the collision broadphase is a btDbvtBroadphase
    /// that does not make use of max_objects and scene_size, but one might plug-in
    /// other collision engines that might use those parameters.
    /// If init_sys is false it does not initialize the collision system or solver
    /// assumes that the user will do so.
    ChSystem(unsigned int max_objects = 16000, double scene_size = 500, bool init_sys = true);
}
```

- ChSystemDEM is a derived class which employs the penalty approach to treat contacts

```cpp
class ChApi ChSystemDEM : public ChSystem {
    /// Constructor for ChSystemDEM.
    /// Note that, in case you will use collision detection, the values of
    /// 'max_objects' and 'scene_size' can be used to initialize the broadphase
    /// collision algorithm in an optimal way. Scene size should be approximately
    /// the radius of the expected area where colliding objects will move.
    ChSystemDEM(bool use_material_properties = true, /**< use physical contact material properties */
                 unsigned int max_objects = 16000, /**< maximum number of contactable objects */
                 double scene_size = 500 /**< approximate bounding radius of the scene */);
};
```
Specifying contact method at system construction (2/3)

• Bodies must be constructed to be consistent with the containing system:

```cpp
ChBody(ChMaterialSurfaceBase::ContactMethod contact_method = ChMaterialSurfaceBase::DVI);
```

```cpp
enum ContactMethod {
    DVI,    /// constraint-based (a.k.a. rigid-body) contact
    DEM    /// penalty-based (a.k.a. soft-body) contact
};
```

• ChBody getter and setter methods for contact material:

```cpp
/// Access the DVI material surface properties associated with this body.
/// This function performs a dynamic cast (and returns an empty pointer
/// if matsurface is in fact of DEM type). As such, it must return a copy
/// of the shared pointer and is therefore NOT thread safe.
std::shared_ptr<ChMaterialSurface> GetMaterialSurface() {
    return std::dynamic_pointer_cast<ChMaterialSurface>(matsurface);
}

/// Access the DEM material surface properties associated with this body.
/// This function performs a dynamic cast (and returns an empty pointer
/// if matsurface is in fact of DVI type). As such, it must return a copy
/// of the shared pointer and is therefore NOT thread safe.
std::shared_ptr<ChMaterialSurfaceDEM> GetMaterialSurfaceDEM() {
    return std::dynamic_pointer_cast<ChMaterialSurfaceDEM>(matsurface);
}

/// Set the material surface properties by passing a ChMaterialSurface or
/// ChMaterialSurfaceDEM object.
void SetMaterialSurface(const std::shared_ptr<ChMaterialSurfaceBase>& mnewsurf) { matsurface = mnewsurf; }
```
Specifying contact method at system construction (3/3)

• ChSystem virtual method for constructing a body with consistent contact material:

        /// Create and return the pointer to a new body.
        /// The returned body is created with a contact model consistent with the type
        /// of this ChSystem and with the collision system currently associated with this
        /// ChSystem. Note that the body is *not* attached to this system.
        virtual ChBody* NewBody() { return new ChBody(ChMaterialSurfaceBase::DVI); }

• ChSystemDEM

        /// Create and return the pointer to a new body.
        /// The returned body is created with a contact model consistent with the type
        /// of this ChSystem and with the collision system currently associated with this
        /// ChSystem. Note that the body is *not* attached to this system.
        virtual ChBody* NewBody() { return new ChBody(ChMaterialSurfaceBase::DVI); }

• Example: construct a system with specified contact method and create a body with consistent contact material

        ChSystem* system;
        switch (contact_method) {
            case ChMaterialSurfaceBase::DVI:
                system = new ChSystem();
                break;
            case ChMaterialSurfaceBase::DEM:
                system = new ChSystemDEM(use_mat_properties);
                break;
        }
        auto object = std::shared_ptr<ChBody>(system->NewBody());
        system->AddBody(object);
ChMaterialSurface and ChMaterialSurfaceDEM

Complementarity

```cpp
/// Material surface data for DVI contact
class ChApi ChMaterialSurface : public
ChMaterialSurfaceBase {

public:
  float static_friction;
  float sliding_friction;
  float rolling_friction;
  float spinning_friction;
  float restitution;
  float cohesion;
  float dampingf;
  float compliance;
  float compliancef;
  float complianceRoll;
  float complianceSpin;
```

Penalty

```cpp
/// Material surface data for DEM contact
class ChApi ChMaterialSurfaceDEM : public ChMaterialSurfaceBase {

public:
  float young_modulus; ///< Young’s modulus (elastic modulus)
  float poisson_ratio; ///< Poisson ratio
  float static_friction; ///< Static coefficient of friction
  float sliding_friction; ///< Kinetic coefficient of friction
  float restitution; ///< Coefficient of restitution
  float constant_adhesion; ///< Constant adhesion force
  float adhesionMultDMT; ///< Adhesion multiplier used in DMT model.

  float kn; ///< user-specified normal stiffness coefficient
  float kt; ///< user-specified tangential stiffness coefficient
  float gn; ///< user-specified normal damping coefficient
  float gt; ///< user-specified tangential damping coefficient
```
Specifying collision material (1/2)

• Easy but potentially memory-inefficient:
  body_b->SetFriction(0.4f);
  body_b->SetRollingFriction(0.001f);

• Using a shared material:
  // Create a surface material and change properties:
  auto mat = std::make_shared<ChMaterialSurface>();
  mat->SetFriction(0.4f);
  mat->SetRollingFriction(0.001f);
  // Assign surface material to body/bodies:
  body_b->SetSurfaceMaterial(mat);
  body_c->SetSurfaceMaterial(mat);
  body_d->SetSurfaceMaterial(mat);
  ...

• Note: ChMaterialSurfaceDEM can only be set through a shared pointer
Specifying collision material (2/2)

```cpp
auto object = std::shared_ptr<ChBody>(system->NewBody());

system->AddBody(object);

object->SetIdentifier(objectId);
object->SetMass(mass);
object->SetInertiaXX(400.0 * ChVector<>(1, 1, 1));
object->SetPos(pos);
object->SetRot(rot);
object->SetPos_dt(init_vel);
object->SetWvel_par(init_omg);
object->SetCollide(true);
object->SetBodyFixed(false);

switch (object->GetContactMethod()) {
  case ChMaterialSurfaceBase::DVI:
    object->GetMaterialSurface()->SetFriction(object_friction);
    object->GetMaterialSurface()->SetRestitution(object_restitution);
    break;
  case ChMaterialSurfaceBase::DEM:
    object->GetMaterialSurfaceDEM()->SetFriction(object_friction);
    object->GetMaterialSurfaceDEM()->SetRestitution(object_restitution);
    object->GetMaterialSurfaceDEM()->SetYoungModulus(object_young_modulus);
    object->GetMaterialSurfaceDEM()->SetPoissonRatio(object_poisson_ratio);
    object->GetMaterialSurfaceDEM()->SetKn(object_kn);
    object->GetMaterialSurfaceDEM()->SetGn(object_gn);
    object->GetMaterialSurfaceDEM()->SetKt(object_kt);
    object->GetMaterialSurfaceDEM()->SetGt(object_gt);
    break;
}
```