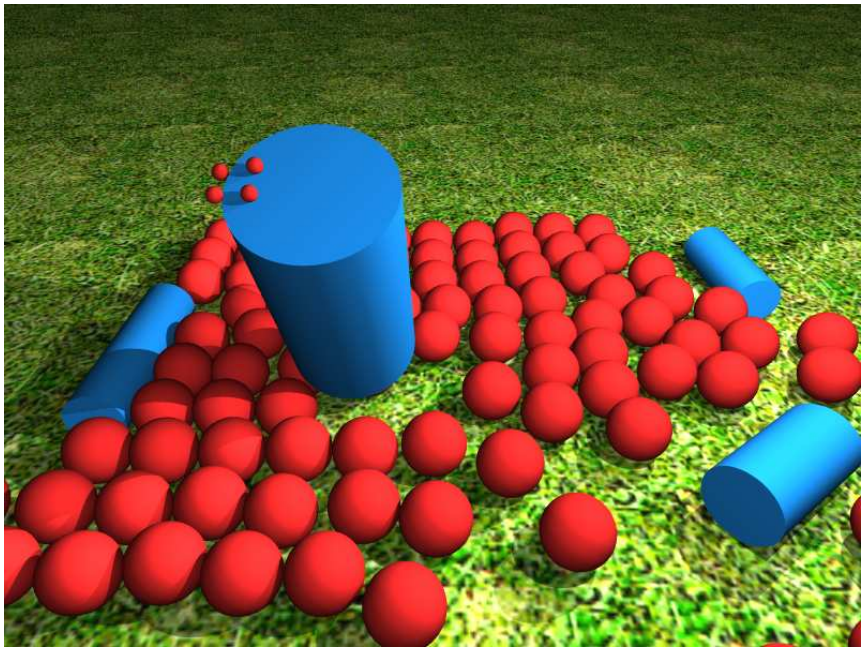
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Collision Detection for Cylinder-Shaped Rigid Bodies

Michael Sünkel



Bachelor's Thesis

Collision Detection for Cylinder-Shaped Rigid Bodies

Michael Sünkel

Bachelor's Thesis

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	M. Sc. Klaus Iglberger
Bearbeitungszeitraum:	01.07.2009 – 15.09.2010

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 15. September 2010 .....

Abstract

Collision detection is an indispensable part of every rigid body simulation tool. Without the collision detection phase, the collision between rigid bodies could not be detected and could therefore not be resolved. However, in order to be able to simulate a large number of interacting rigid bodies, the efficiency of the collision detection phase plays a major role. For this reason, efficient collision detection algorithms are still an active field of research.
In order to speed up collision detection, various optimization strategies have been proposed. In general, the collision detection phase is subdivided into a *coarse* collision detection phase, whose purpose it is to quickly reject all pairs of rigid bodies that are not in contact, and a *fine* collision detection phase that efficiently calculates contact points between pairs of potentially colliding rigid bodies. Whereas there exist a huge number of coarse collision detection algorithms, such as bounding volumes, space partitioning strategies, and grid-based approaches, that try to reduce the overall complexity of the collision detection process, there is only a smaller number of available algorithms for an efficient fine collision detection.
One general approach to simplify the fine collision detection phase is the use of *geometric primitives*, whose geometry is known exactly and can therefore be exploited more easily. The task of this thesis is the integration of suitable fine collision detection functions for the *cylinder geometry* in the *pe* rigid body physics engine in order to enable efficient collision detection with cylindric rigid bodies. The advantage of a cylinder primitive in comparison to triangle meshes is a perfect representation of the geometry in rigid body simulations and a faster and more accurate collision detection. The new feature of the *pe* will be demonstrated by means of some simulation examples.

Zusammenfassung

Die *Kollisionserkennung* ist ein unverzichtbarer Teil eines jeden Programms, welches Starrkörper simuliert. Die Kollision zwischen Starrkörpern könnte ohne die Kollisionserkennungsphase nicht erkannt und dadurch auch nicht gelöst werden. Bei der Simulation einer großen Zahl an sich gegenseitig beeinflussenden Starrkörpern spielt die Effizienz der Kollisionserkennungsphase eine wesentliche Rolle. Deshalb sind effiziente Kollisionserkennungsalgorithmen immernoch ein aktives Forschungsgebiet.
Es wurden verschiedene Optimierungsstrategien vorgeschlagen, um die Kollisionserkennung zu beschleunigen. Im Allgemeinen ist die Kollisionserkennungsphase unterteilt in eine *grobe* Kollisionserkennungsphase, welche den schnellen Ausschluss aller Paare von Starrkörpern bezweckt, die nicht in Kontakt stehen, und in eine *feine* Kollisionserkennungsphase, welche Kontaktpunkte zwischen möglicherweise kollidierenden Starrkörpern effizient berechnet. Während eine Vielzahl von groben Kollisionserkennungsalgorithmen existiert, wie etwa Bounding Volumes, Space Partitioning-Strategien und gitterbasierte Ansätze, welche versuchen, die Komplexität des gesamten Kollisionserkennungsprozesses zu reduzieren gibt es nur eine kleine Zahl an verwendbaren Algorithmen für eine effiziente Kollisionserkennung.
Ein üblicher Ansatz, um die feine Kollisionserkennungsphase zu vereinfachen, ist die Verwendung von *geometrischen Primitiven*, deren Geometrie exakt bekannt ist und somit einfacher verwertet werden kann. Die Aufgabe in dieser Arbeit ist die Integration von geeigneten feinen Kollisionserkennungsfunktionen für die *Zylindergeometrie* in die *pe* Starrkörper Physikengine, um eine effiziente Kollisionserkennung mit zylindrischen Starrkörpern zu ermöglichen. Die Vorteile einer Zylinderprimitive gegenüber Dreiecksnetzen sind zum einen eine perfekte Darstellung der Geometrie bei Starrkörpersimulationen und zum anderen eine schnellere sowie genauere Kollisionserkennung. Die neue Fähigkeit der *pe* wird anhand von Simulationsbeispielen demonstriert.

Contents

1 Introduction and Goals

1.1 The *pe* - A physics engine

Figure 1: A scene in *pe* designed by Klaus Iglberger [10]

> A physics engine is computer software that provides an approximate simulation of certain simple physical systems, such as rigid body dynamics (including collision detection), soft body dynamics, and fluid dynamics, of use in the domains of computer graphics, video games and film. Their main uses are in video games (typically as middleware), in which case the simulations are in real-time. The term is sometimes used more generally to describe any software system for simulating physical phenomena, such as high-performance scientific simulation. [1]

The *pe* physics engine (in the following just called *pe*) is a powerful simulation tool designed by Klaus Iglberger at the university of Erlangen-Nuremberg. This framework realizes movements and collisions of rigid bodies. First you can make a setup of a scene and add objects to the defined domain. Every object gets its own properties like the global position, stretch, material, velocity and orientation.
To visualize a scene there are two possibilities: During the development of new features for the *pe* it is very nice to work with a module called *Irrlicht*. The big advantage of this framework (for further information about this framework see [6]) is the real-time illustration of the scenario. Another option is to visualize a scene by rendering output files with *pov-ray* ([7]). This ray-tracer is better for presentation of results, e.g., the creation of nice pictures, using textures, etc..
The *pe* works just with rigid bodies. If the bodies would be deformable the collisions and their effects would become too complex. Compared to many other physics engines the *pe* gives the priority to physical precision instead of computational speed. As in every implementation the goal is to optimize and to minimize the calculation time but the physical result must not get worse. The calculations of all physical phenomena and their consequences are realized in a fixed number of time steps. The bigger the number of time steps the better the approximation of the result. In *pe* all objects are checked pairwise if there are collisions with another object. These tests are performed in a multi-level approach, which is described in chapter 2.2 on page 8 in detail. If the calculation results in a collision of two bodies one or more contact points have to be computed. A contact point has to be calculated if the distance of two bodies is smaller than a defined value called *contactThreshold* (for instance 10^{-6}). Such a contact point consists of the two corresponding bodies, the global position, the normal pointing from body two to body one and the penetration depth. In real world there are contact surfaces but it is sufficient to take a small and limited number of contact points to get a realistic behavior of the scene.

1.2 Goals and troubles

The task of this work is to extend the *pe* by a new geometric primitive. Among already existing geometries like plane, sphere, capsule and box it is also convenient to integrate the cylinder geometry. This task is important to the effect that the use of geometric primitives usually has many advantages compared with triangle meshes. A geometric primitive defines a body by a few parameters. For instance the cylinder is characterized by its global position, radius, length and orientation. With these data the cylinder is presented exactly. But if there are used triangle meshes to describe the cylinder instead there is always an error in description of the circular lateral surface. To reduce this error you have to create a very huge number of triangles and that could increase the computation time in the collision tests enormously.
Based on existing data structures the goal is to find accurate and efficient algorithms, which generate the correct number and the position(s) of the contact point(s) of a cylinder with another geometric primitive.

But the cylinder is very hard to handle because there are edges at the *end caps* (also called *discs*). If the cylinder collides with another cylinder or a box inevitable edge-edge collisions can occur. Another reason why the cylinder is hard to handle is that you are not able to know where to create the contact point(s) in many cases, e.g., if two cylinders collide parallel heuristics have to be used to decide where the motion was before (as described in figure 20 on page 20. That leads to (a) wrong placed contact point(s) and ends up with an incorrect result.

2 Collision Detection in physics engines

Collision Detection is a very general problem in everyday life. All living and thinking organisms have to solve these simple tasks. For a human being it is not a real problem to walk with a full cup of tea in one hand, to shoot the ball in a football game, etc. But these kind of tasks are very complicated to manage for a computer. Often complex algorithms have to be used and the computational costs grow horrendously.

2.1 Inaccuracy by time step discretization

In every computer simulation the treated problem has to be discretized because the computer cannot calculate an infinite number of steps to solve the problem continuously. Hence there is a user-defined number of time steps. Depending on some circumstances that can lead to wrong results, especially in case of collision detection tasks.

If the number of time steps is insufficient it is possible, for example, that a fast flying body runs through another body without any physical effects. This problem appears, if the real contact happened between two time steps and the computer missed this information. In the most likely case the current two bodies penetrate each other. Often it is not possible to make sure from which side the bodies got their real contact(s) or where the normal must point to. This problem is affiliated to the applied *a-posteriori-approach*. As the name implies this method analyses the scene about possibly happened contacts between the bodies for each time step. An alternative could be the *a-priori-approach*. In this procedure physical backgrounds like velocities and rotation speeds have to be used. With these data it is possible to determine the exact time when the two bodies will collide. But the first-mentioned method is preferred for a computer simulation because collision handling and physical calculations can be processed in separated tasks. In the real world living organisms use a mixture of both approaches. The more practice and knowledge the bigger will also be the part of the a-priori influence.

In order to minimize the errors caused by the time discretization you can choose a very small time step size relative to the motion speed. But a small time step size causes more calculations over a specific simulation time and that costs many more calculations. Hence the choice of a good step size depends on the current problem or scene.

2.2 Two step Collision Detection

The big generic name Collision Detection has to be separated into two special fields: Collision Detection and Collision Determination.

- The ***Collision Detection*** itself comes to a decision if a collision between two objects is possible or not (overlap test, first step). This part also contains two sub-steps:

  First there are tested the bounding volumes of the two bodies (coarse overlap test), e.g., by using axis-aligned bounding boxes (AABBs) and bounding volume hierarchies. If a possible collision is found by the AABB method a much more precise test has to be performed (fine overlap test). This detailed test comes to a final decision if the two bodies do overlap or not.

- In case of a positive result in the fine overlap test the collision has to be handled. This part is called ***Collision Determination*** and consists of contact point calculations.

Content of this thesis are all fine overlap tests and contact point calculations of a cylinder with any other geometric primitive.

The efficient implementation of these fine collision test functions is of utmost importance in applications such as computer graphics or automotive simulations. In a lot of computer games speed is a very important property. In order to illustrate the game in real-time there have to be accepted some simplifications. Many complex objects are described by a geometric primitive instead of triangle meshes, e.g., to speed up the flow of play. In general the aim is again to find a compromise between physical accuracy and calculation speed and this is depending on the corresponding problem again.

3 Collision algorithms of a body with a cylinder

This section begins with a detailed description of the cylinder geometry and its properties. Then the other primitives and collision detection algorithms are described. The first two collision types are cylinder-plane and cylinder-sphere. The development of the corresponding algorithms has been successful and the results are visualized in chapter 4 on page 26.
But the other three types, i.e., cylinder-capsule, cylinder-cylinder and cylinder-box, became very complicated. During the development of these features it was not possible to find algorithms, which describe most of the cases sufficiently. Hence chapters 3.4-3.6 primarily consist of illustrating the big complexity and difficulties. In other physics engines like ODE [5] these collision types are also discussed at the moment. More than thousand lines code at every type of collision and many reported bugs show the big interest on this topic but also the big complexity and immense effort. A table with features of ODE is shown in [2].

3.1 The Cylinder Geometry

The task was to develop algorithms for pairwise collisions of cylinders with any other existing geometric primitive. Every given position is taken in world coordinates if it is not mentioned otherwise.

In the following the cylinder geometry is described:

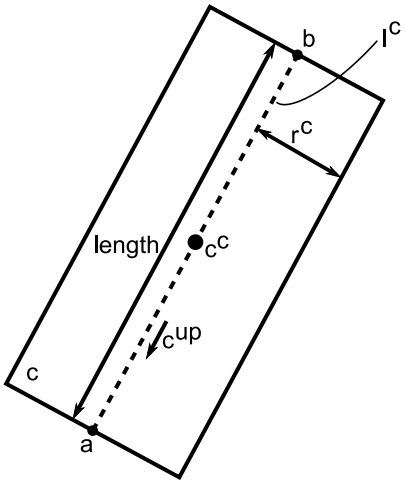


Figure 2: Cylinder and its components

The cylinder c (also notated as cy if another primitive starts with the same character) is represented by its center line l^c containing the center of mass c^c and the radius r^c. In *pe* there exist some access functions for reading properties of cylinder objects like $getRotation()$, $getPosition()$, $getLength()$ and $getRadius()$. The normalized vector c^{up} is computed by using the rotation of the cylinder and points along the center line. In every implemented cylinder collision algorithm the center points of the two end cap circles a and b are of great interest in order to get the center line segment l^c. In the picture above, e.g., the point a can be computed by adding $c^{up} * 0.5 * length$ to the center of mass c^c where $length$ is the length of the center line.
The problems of the cylinder primitive are the discontinuities at the end caps. At these edges the normal has a jump and there have to be found solutions for many problems, which will be discussed afterwards.

3.2 Plane - Cylinder

The plane p is described by the following formula:

$$\vec{n} * \vec{p} - d = 0 \tag{3.1}$$

Here $\vec{n} = (n_1, n_2, n_3)^T$ is the normal vector (accessible by $getNormal()$) and $\vec{p} = (x, y, z)^T$ is any point on the plane. The parameter d (accessible by $getDisplacement()$) is the distance to the origin along the normal. The plane is a very important primitive because it can limit a scene, e.g., it is used as the ground.

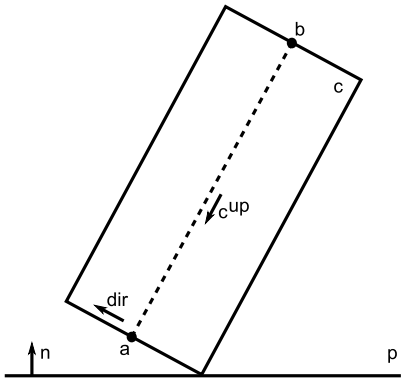


Figure 3: Overlap test for cylinder and plane

3.2.1 Collision Detection

The test if a plane p and a cylinder c have collided (fine overlap test), which is demonstrated in chapter 3 on page 10, is very simple. First the center points of the end caps are needed. This step is described in section 3.1 on page 9. Then the closer point (in this case a) to the plane (of these two) is taken and a perpendicular vector to c^{up} is computed, which points towards to the plane. That is achieved by computing the cross products in this form:

$$(\vec{n} \times \overrightarrow{c^{up}}) \times \overrightarrow{c^{up}} \tag{3.2}$$

The cross product in the brackets yields a vector, which points into or out from the paper (that depends on the direction of c^{up}, which can point towards the plane or away from it). Another cross product of this new vector with c^{up} results in a vector, which points towards the plane (or to the negative direction which is dependent on the signs again). With the corresponding vector dir (and $-dir$) it is possible to compute the four corner points, which are candidates for a contact with the plane. If the distance of at least one of these four points is smaller than *contactThreshold* a collision took place and collision determination has to be performed.

3.2.2 Contact point calculation

If the fine overlap test resulted in a collision, the collision determination step has to be executed now. Here the same steps are necessary as in the fine overlap test but additionally it is necessary to care about the number of contact points and where to place them correctly.
There are three cases how the contact points have to be constructed if the cylinder and the plane overlap:

- In the first case there is just one contact point, e.g., if the cylinder is moving towards the plane and is not (numerically exact) parallel or perpendicular to the plane. That is solved by reducing the three dimensional cylinder to a rectangle, which is perpendicular to the plane. Then the four corner points are candidates for a contact point cp and have to be tested (Fig. 4).

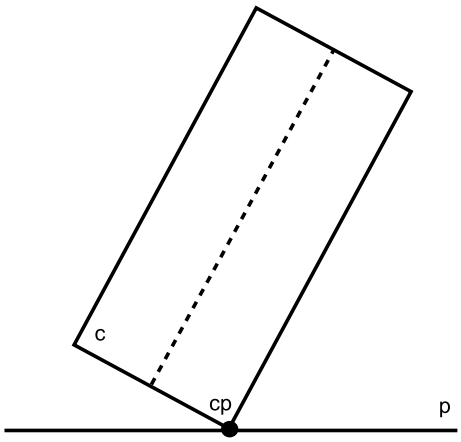


Figure 4: Contact point calculation of a cylinder-plane collision – one contact

- Another possibility is that the center line is parallel to the ground plane. That happens for instance if the cylinder is rolling over the plane. Here we need two contact points, which are computed in the same way as in the first case, with the difference that there are two contact points, which penetrate the plane or have a smaller distance than *contactThreshold* (Fig. 5).

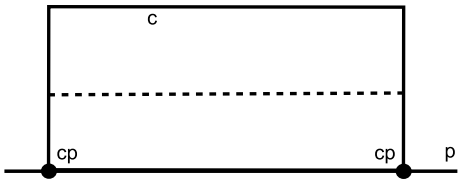


Figure 5: Contact point calculation of a cylinder-plane collision – two contacts

- In the most complicated case the cylinder is *standing* on the plane and the center line is (numerically exact) perpendicular to the plane. There have to be used three contact points in order to have a stable and realistic simulation. The deepest point has to be found and then the other two with an 120 degree angle in both directions need to be computed. This is done by using quaternions (Fig. 6).

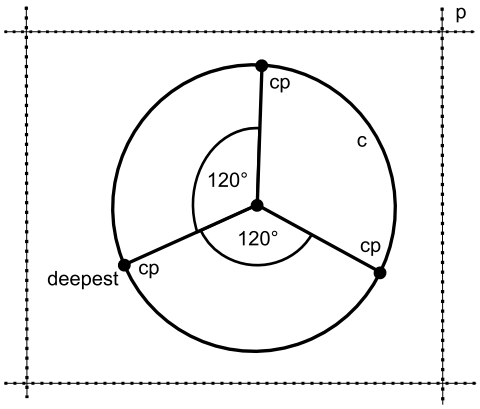


Figure 6: Contact point calculation of a cylinder-plane collision – three contacts

3.3 Sphere - Cylinder

The sphere s is described by its center point c^s and the radius r^s. These data you can get with the access functions *getPosition*() and *getRadius*(). The invariance of the rotation is a very nice property of the sphere, which makes it easy to handle.

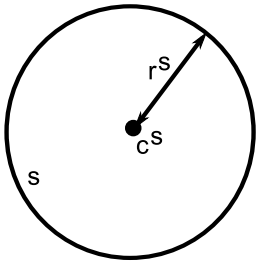


Figure 7: Sphere and its components

3.3.1 Collision Detection

The most efficient way to attend collisions of spheres and cylinders is to transform the global position of the sphere center from world coordinates to local coordinates of the cylinder. The result is a three dimensional vector named $rPos$. The first dimension $rPos[0]$ is along the center line of the cylinder like a projection of the sphere center to the (extended) center line of the cylinder. The sum of the squared entries of the other two dimensions $rPos[1]$ and $rPos[2]$ is equal to the shortest distance $d1$ or $d2$ from the sphere center to the (extended) center line.
At the start of the algorithm it is necessary to notice whether the sphere center is between or outside the two planes defined by the end cap circles of the cylinder.
In the first case (Fig. 8 on page 13) there are two examples when the sphere touches the cylinder at its lateral surface. This part is trivial because a simple comparison of the sum of the radii of the two bodies and the distance of the sphere center to the center line of the cylinder indicates whether a collision has occurred or not.

$$\sqrt{rPos[1]^2 + rPos[2]^2} < (r^s + r^c + contactThreshold) \tag{3.3}$$

To optimize the calculation time we use the squared values, which give the same result but prevent a very expensive square root operation:

$$(rPos[1]^2 + rPos[2]^2) < (r^s + r^c + contactThreshold)^2 \tag{3.4}$$

In the other case (Fig. 9 on page 13) it has to be computed if the sphere touches one of the end caps of the cylinder. Here a cross product approach gives the necessary direction (if *endp* is the closer endpoint to the sphere):

$$(\overrightarrow{c^{up}} \times (\overrightarrow{c^s} - \overrightarrow{endp})) \times \overrightarrow{c^{up}} \tag{3.5}$$

With this vector you can build a segment on the end cap, which has length $2 * r^{cy}$ and contains the nearest point on the end cap to the center of the sphere. This point can be computed by the function *getClosestLineSegmentPoints*, which takes two line segments and gives back the two nearest points of these (this basic primitive test is described in detail in [12]). That also works if a segment is degenerated to a point like in this case. Then the easy comparison of the distance of these two points and the radius of the sphere decides on a collision. If the distance is bigger than the sphere's radius (increased by *contactThreshold*) no collision is possible and otherwise the bodies collided.

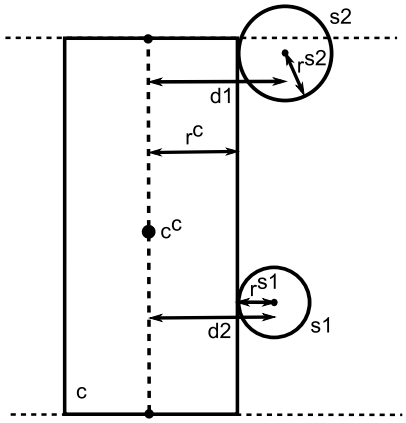


Figure 8: Overlap test for sphere and cylinder – case 1

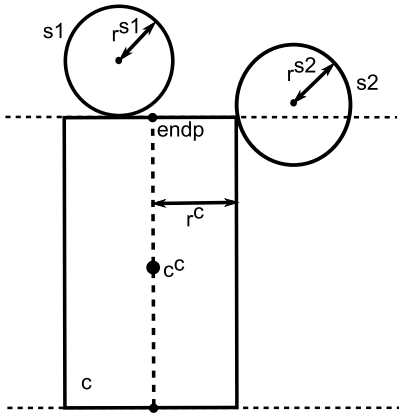


Figure 9: Overlap test for sphere and cylinder – case 2

3.3.2 Contact point calculation

If the fine overlap test resulted in a collision, the collision determination step has to be executed now. Here the same steps are necessary as in the fine overlap test, but additionally you have to care about the number of contact points and where to place them correctly.

In the first case where the sphere is within the height of the cylinder the relative height of the sphere center to the cylinder's center is calculated (as shown on Fig. 10 on page 14). That gives the point on the center line of the cylinder, which is nearest to the sphere center. This point and the sphere center can now perform a sphere-sphere collision with radius r^c, r^s and distance $d1$ or $d2$. The normal $\vec{n}$ is defined by a normalized vector starting from the contact point directed to the sphere center. As discussed in the beginning the bodies usually penetrate each other, e.g., $d1 - r^c - r^s$ gives a negative result. That is corrected by adding the half of this negative value along the normal and to put the contact point there. Now the contact point is equally penetrated in both bodies.

The second case works exactly like described in 3.3.1 on page 12. Finally the function *getClosest-LineSegmentPoints* provides the sphere center and the closest point to the end cap of the cylinder. A vector, which connects these two points is used as the normal pointing from the cylinder to the sphere (as shown on Fig. 11 on page 14).

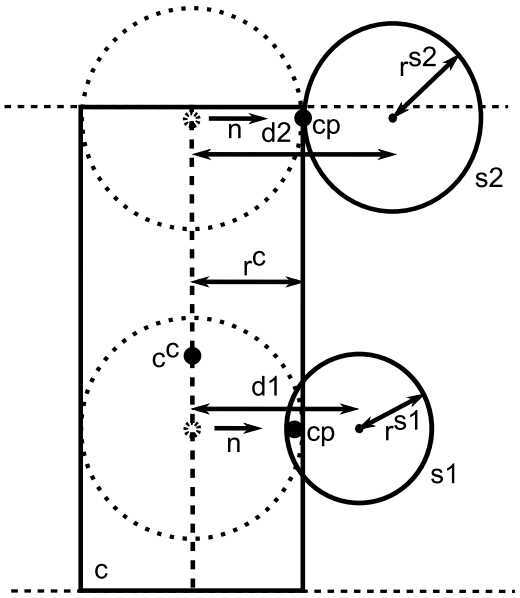


Figure 10: Contact point calculation of a cylinder-sphere collision – case 1

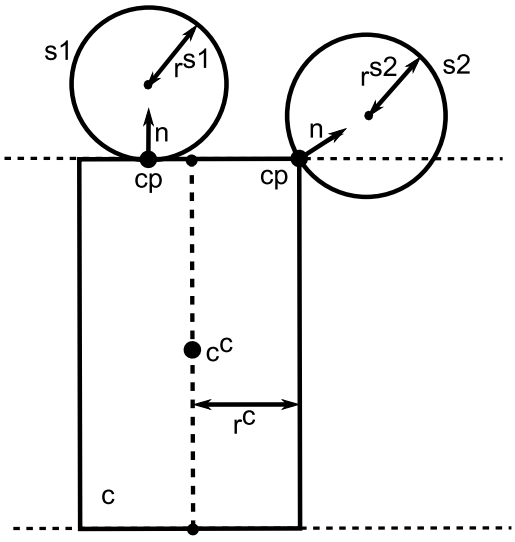


Figure 11: Contact point calculation of a cylinder-sphere collision – case 2

3.4 Capsule - Cylinder

The capsule *ca* is also an easy geometric primitive. It is a body similar to a cylinder, but with additional hemispheres at the end caps. It is described by the center line l^{ca} containing the center of mass c^{ca} and the radius r^{ca}. The points a and b are computed as described in chapter 3.1 on page 9.

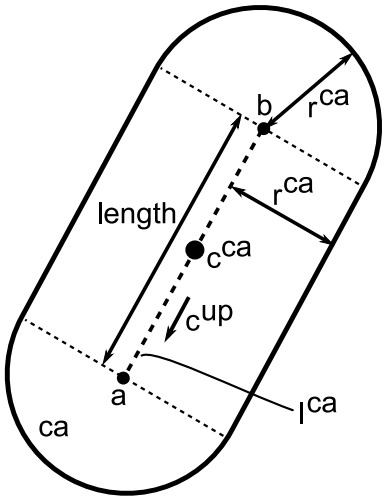


Figure 12: Capsule and its components

The capsule is also called *sphere-swept line*, because it is a body, which is described by a sphere moving along the center line.
The access functions for reading properties of capsule objects are equal to the functions of the cylinder (*getRotation*(), *getPosition*(), *getLength*() and *getRadius*().

3.4.1 Collision Detection

The collision detection step begins with an early-out approach of the two center lines. If the capsule and the cylinder's capsule do not overlap a collision of capsule and cylinder is not possible. This test is very easy to perform using *getClosestLineSegmentPoints*. But it is not possible to make a final decision if the capsules do overlap because one of them is a real cylinder.
Hence the overlap test has to be considered more precisely. But using the line geometry is not that easy because the bodies are both rotated. This leads to the big problem to find the right direction if you expect a contact at one of the end caps of the cylinder.

3.4.2 Contact point calculation

Also the computation of contact points became very hard. Most of the problems, which appeared during the development of the necessary algorithms are described in 3.5 on page 16 quite precisely. After a long period of looking for working algorithms regarding line geometry it was time to think of other approaches.
Another option for a collision handling between a capsule and a cylinder is to mutate the complex cylinder geometry to other bodies. These changes imply some errors because the cylinder cannot be reproduced exactly. But the modifications reduce the computation costs immensely and make it possible to find an algorithm for the whole collision.

- One possibility is to change the cylinder to a capsule, which fits the cylinder as good as possible (Fig. 13). Then you can reduce the cylinder-capsule collision handling to an easier capsule-capsule collision.
- Another idea is to present the cylinder using many spheres (Fig. 14) and to reduce the current problem to a sphere-cylinder algorithm.

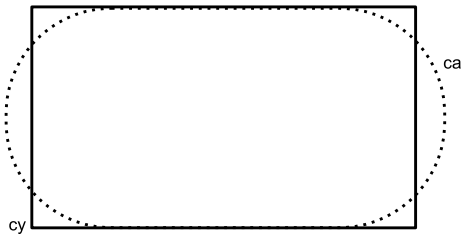


Figure 13: Cylinder collision handling – simplifications – idea 1

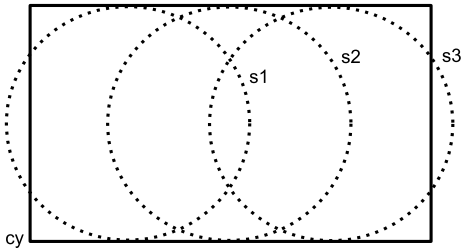


Figure 14: Cylinder collision handling – simplifications – idea 2

But caused by these simplifications the algorithm makes the physical precision worse, which disagrees with the rules of the implementation in *pe*.

3.5 Cylinder - Cylinder

Like in previous tests with other primitives the overlap test and contact point calculation are very similar to each other. This collision handling is very hard to implement because there are many difficulties to cope.
The study of [11] clarified the complexity of using line geometries combined with colliding cylinders. This paper shows the fine overlap test of two cylinders. This could also be used for the capsule-cylinder overlap test. But there are no hints for the handling of contact points and its complexity.

3.5.1 Collision Detection

An easy part of collision detection is given for two infinite cylinders. It is possible to use the *getClosestLineSegmentPoints* function and a trivial comparison of the sum of the radii and the shortest distance of the two line segments comes out with a decision if an overlap has occurred or not. Similar to this case it is the same with finite cylinders if they have their two nearest line segment points both on the line segments where none of them is an endpoint of the corresponding cylinder (Fig. 15).
But there are problems if a nearest point is an endpoint of a cylinder. Then the disc of a cylinder touches the other cylinder and the direction, which points towards the other cylinder, has to be computed. In the *worst case* both cylinders overlap at their end caps and the problems get harder.

3.5.2 Contact point calculation

Of course the problems discussed in the previous section also appear in the contact point calculation part.
In the trivial case (shown on Fig. 15) the calculation of the contact point is very easy to handle. The function *getClosestLineSegmentPoints* provides the points cp^{c1} and cp^{c2}, which are the closest points from the two center line segments. If the two points are no endpoints of the cylinders you can compare the distance d with the sum of the two radii r^{c1} and r^{c2}.
Many difficulties appear at the creation of contact points for the more complex parts of the algorithm. There are four possibilities:

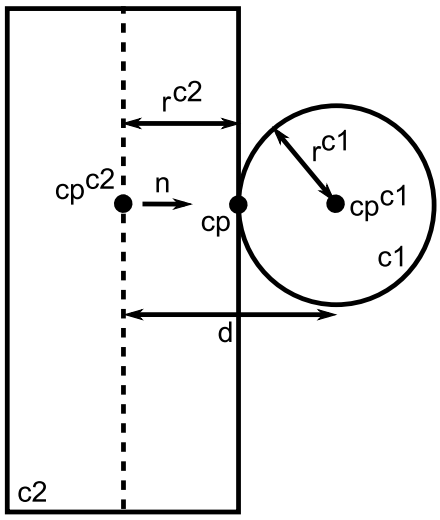


Figure 15: Contact point calculation of a cylinder-cylinder collision – trivial case

- The case with just one contact point (Fig. 16) contains the trivial case (Fig. 16c).

  The illustrated collision type on Fig. 16a is not that easy as it seems to be because it is very hard to get the right direction where the segments on the end disc point to.

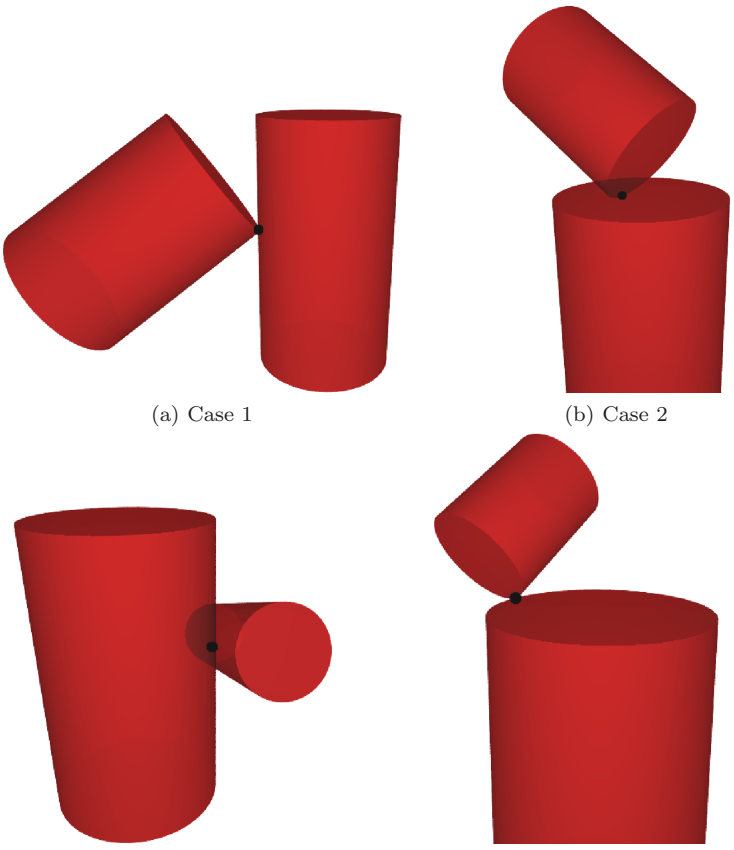
  That problem rises dramatically in case of Fig. 16b because in this case two end caps collide and that is a very costly procedure because it is very hard to find the correct direction, which points to the other cylinder in a correct way. Another possibility in this case is given if the discs collide at their edges (Fig. 16d). Then it is not possible to define a unique normal.

- If it is necessary to compute two contact points (Fig. 17) there are two cases: In the first one the cylinder center lines are parallel to each other. The contact points will be calculated as illustrated in Fig. 17a. The opposed case is given if the center lines are perpendicular to each other (Fig. 17b). The collision handling of c4 and c5 is the easiest one because you can compute quite fast the two contact points, which are set at the end of the radius of the disc of the other cylinder. The line on which these two contact points are set is a complete secant to the end cap.

  The collisions of c4 with c6 and c7 are much more complicated because the projected center lines on the end cap are no complete secants and there was not found a solution to compute the corresponding contact points correctly.

- There is just one possibility where three contact points are needed (Fig. 18): This is the case when one cylinder is standing on the other one and the center lines are parallel to each other. Additionally one of the projected circles has to contain the other one.

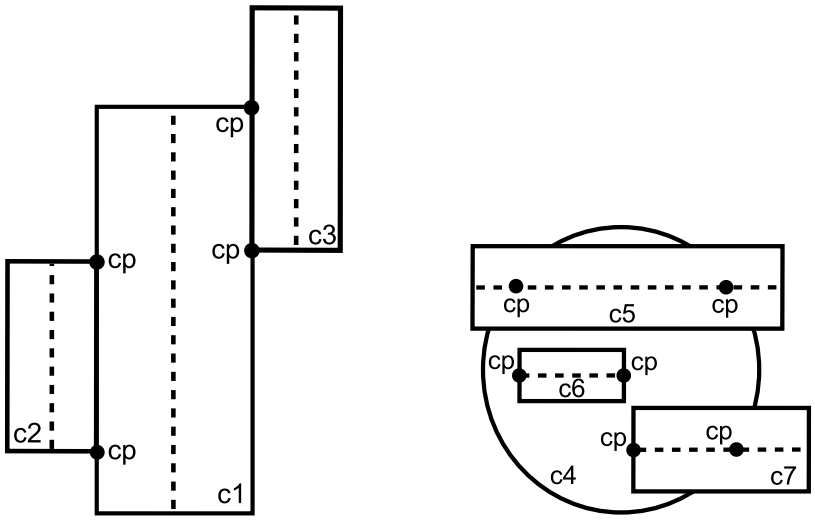
- Similar to the last case is the last possible scenario with four contact points. The only difference is that the projected circles now do overlap. To have a stable simulation it is necessary to compute four contact points as displayed in figure 19.



(a) Case 1 (b) Case 2

(c) Case 3 (d) Case 4

Figure 16: Contact point calculation of a cylinder-cylinder collision – one contact



(a) Parallel Case (b) Perpendicular Case

Figure 17: Contact point calculation of a cylinder-cylinder collision – two contacts

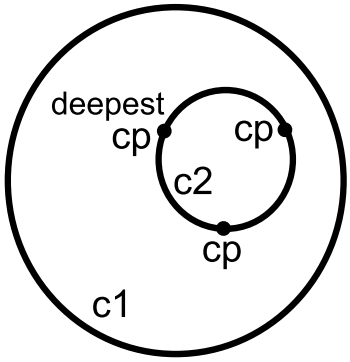


Figure 18: Contact point calculation of a cylinder-cylinder collision – three contacts

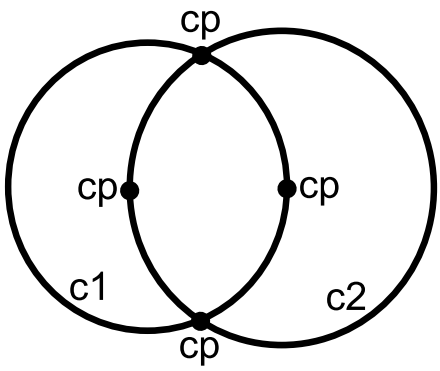


Figure 19: Contact point calculation of a cylinder-cylinder collision – four contacts

Another big problem class is illustrated on figure 20. That is an exemplary scenario where the cases Fig. 17a and Fig. 19 both are possible. Based on the former described *a-posteriori-approach*, in this case it is not possible to indicate if c2 is moving from the right side or from above to c1. For this case there could be used a heuristic Al approach using the distances $d1$ and $d2$. For instance if $d1$ is bigger than $d2$ it is more probable that c2 is penetrating c1 from the right side. But that gives just the right result if there are no very big rotation or translation speeds.

All these scenarios show how complex a collision handling of two cylinders can be. You need many requests to indicate, which case could be the right one. That is very error-prone. Then the algorithms are quite complex and the calculation effort grows.

All these difficulties made it impossible to implement a collision handling of two cylinders in a correct and efficient way. The advantages of using the cylinder as a geometric primitive cannot be utilized in this part.

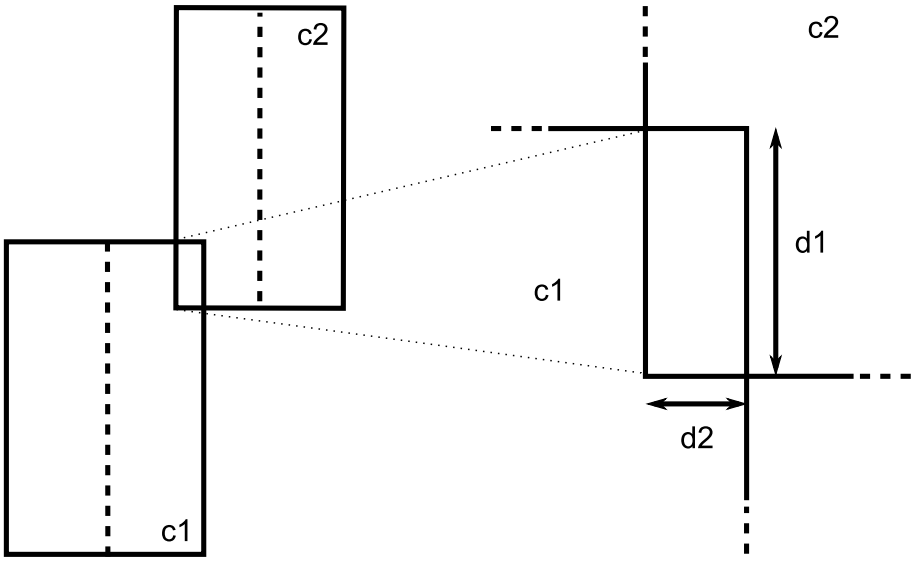


Figure 20: Contact point calculation of a cylinder-cylinder collision – Trouble

Another option for a collision handling between two cylinders is to mutate the complex cylinder geometry to other bodies (as described in chapter 3.4.2 on page 15). These changes imply some errors because the cylinder cannot be reproduced exactly. But the modifications reduce the computation costs immensely and make it possible to find an algorithm for the whole collision.

- One possibility is to change one of the cylinders to a capsule, which fits the cylinder as good as possible (Fig. 13 on page 16). Then you can reduce the cylinder-cylinder collision handling to an easier capsule-cylinder collision.
- Another idea is to present one of the cylinders as many spheres (Fig. 14 on page 16) and to reduce the current problem to a sphere-cylinder algorithm.

But with these simplifications the algorithm does not calculate real edge-edge contacts. Furthermore, this strategy makes the physical precision worse, which disagrees with the rules of the implementation in *pe*.

3.6 Box - Cylinder

The collision detection handling of a box and a cylinder has some problems similar to the former chapters. Especially the detection of the right direction at the end caps, the hard edge-edge collisions and the many special cases, which have to be considered. The properties of a box are the center of mass in the middle, an own local coordinate system and the three half lengths in the corresponding direction of the local coordinate system.

3.6.1 Collision Detection

A very helpful basic primitive test *getClosestLineBoxPoints* is also illustrated in [12]. This function gets a box and a line segment as input and gives back the two nearest points of the input data.

- The easiest scenarios are given in Fig. 21a-21d in most of the cases. That often works because the *getClosestLineBoxPoints* function prepares the closest point on the cylinder segment $\overrightarrow{cp^c}$ to the box $\overrightarrow{cp^b}$.

  If this point is within the cylinder height ((a), (c)) the contact point is generated at the cylinder's lateral surface and the normal is a vector with direction

  $$\vec{n} = \overrightarrow{cp^b} - \overrightarrow{cp^c} \tag{3.6}$$

  If an end cap contact needs to be computed (i.e., cp^c is an endpoint of the cylinder) we need the cross product approach

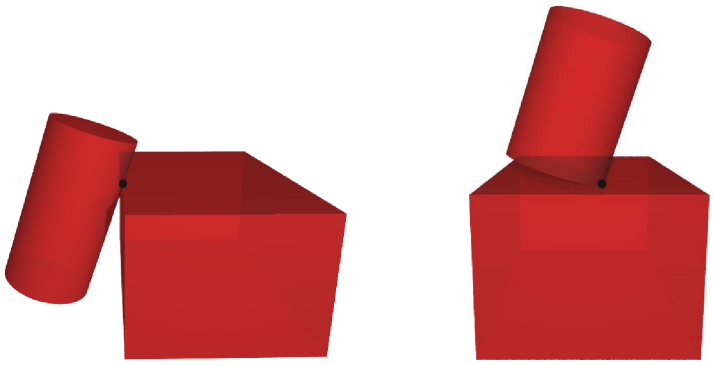
  $$\overrightarrow{dir} = ((\overrightarrow{cp^b} - \overrightarrow{cp^c}) \times \overrightarrow{c^{up}}) \times \overrightarrow{c^{up}} \tag{3.7}$$

  again.

- But the problems grow if you try to handle more complicated cases like Fig. 21e because it is not possible to utilize the advantages of the line geometry. The *getClosestLineBoxPoints* function just can help in special cases.

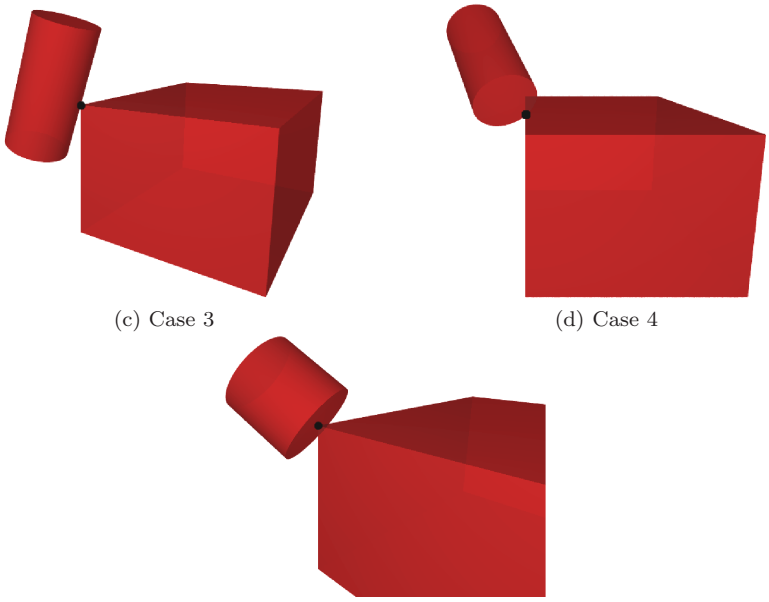
3.6.2 Contact point calculation

In the following there are illustrated the many possible scenarios, which may appear at a collision of a box and a cylinder:



(a) Case 1 (b) Case 2

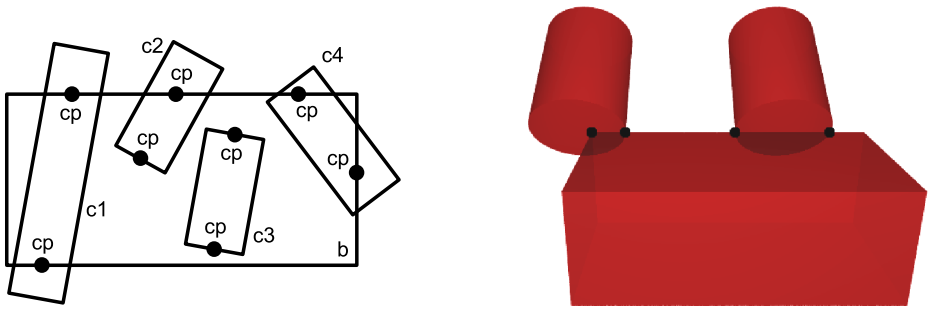
Figure 21: Contact point calculation of a cylinder-box collision – one contact (1/2)



(c) Case 3 (d) Case 4

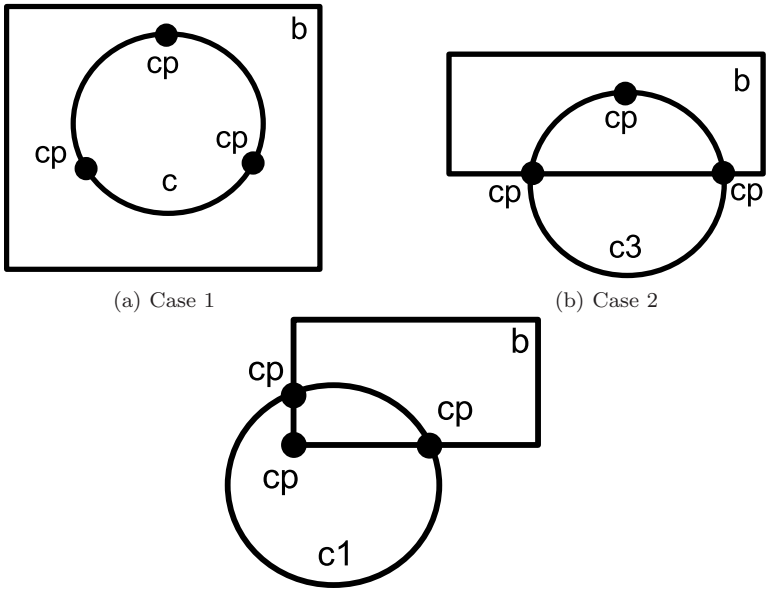
(e) Case 5

Figure 21: Contact point calculation of a cylinder-box collision – one contact (2/2)



(a) Case 1 (b) Case 2

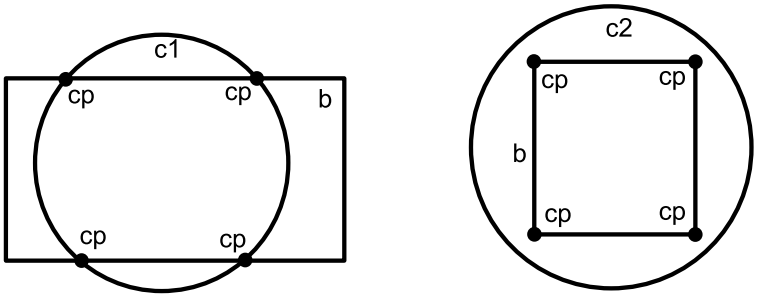
Figure 22: Contact point calculation of a cylinder-box collision – two contacts



(a) Case 1 (b) Case 2

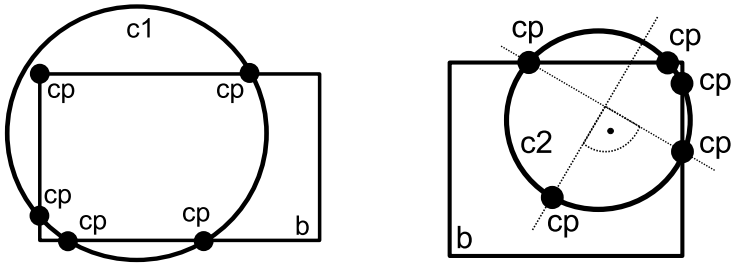
(c) Case 3

Figure 23: Contact point calculation of a cylinder-box collision – three contacts



(a) Case 1 (b) Case 2

Figure 24: Contact point calculation of a cylinder-box collision – four contacts



(a) Case 1 (b) Case 2

Figure 25: Contact point calculation of a cylinder-box collision – five contacts

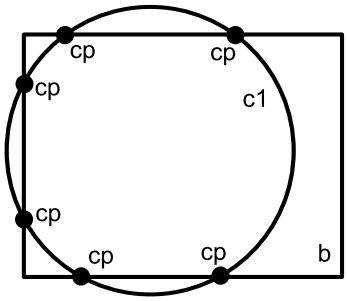


Figure 26: Contact point calculation of a cylinder-box collision – six contacts

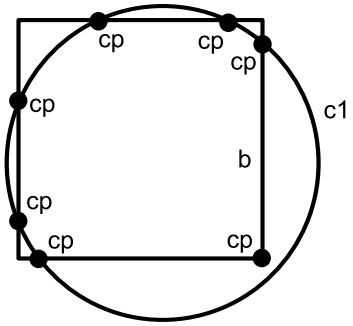


Figure 27: Contact point calculation of a cylinder-box collision – seven contacts

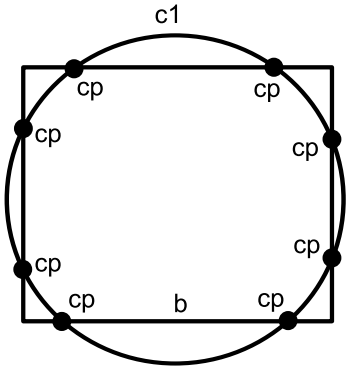


Figure 28: Contact point calculation of a cylinder-box collision – eight contacts

Another option for a collision handling between a cylinder and a box is to mutate the geometries to other bodies (as described in chapter 3.4.2 on page 15):

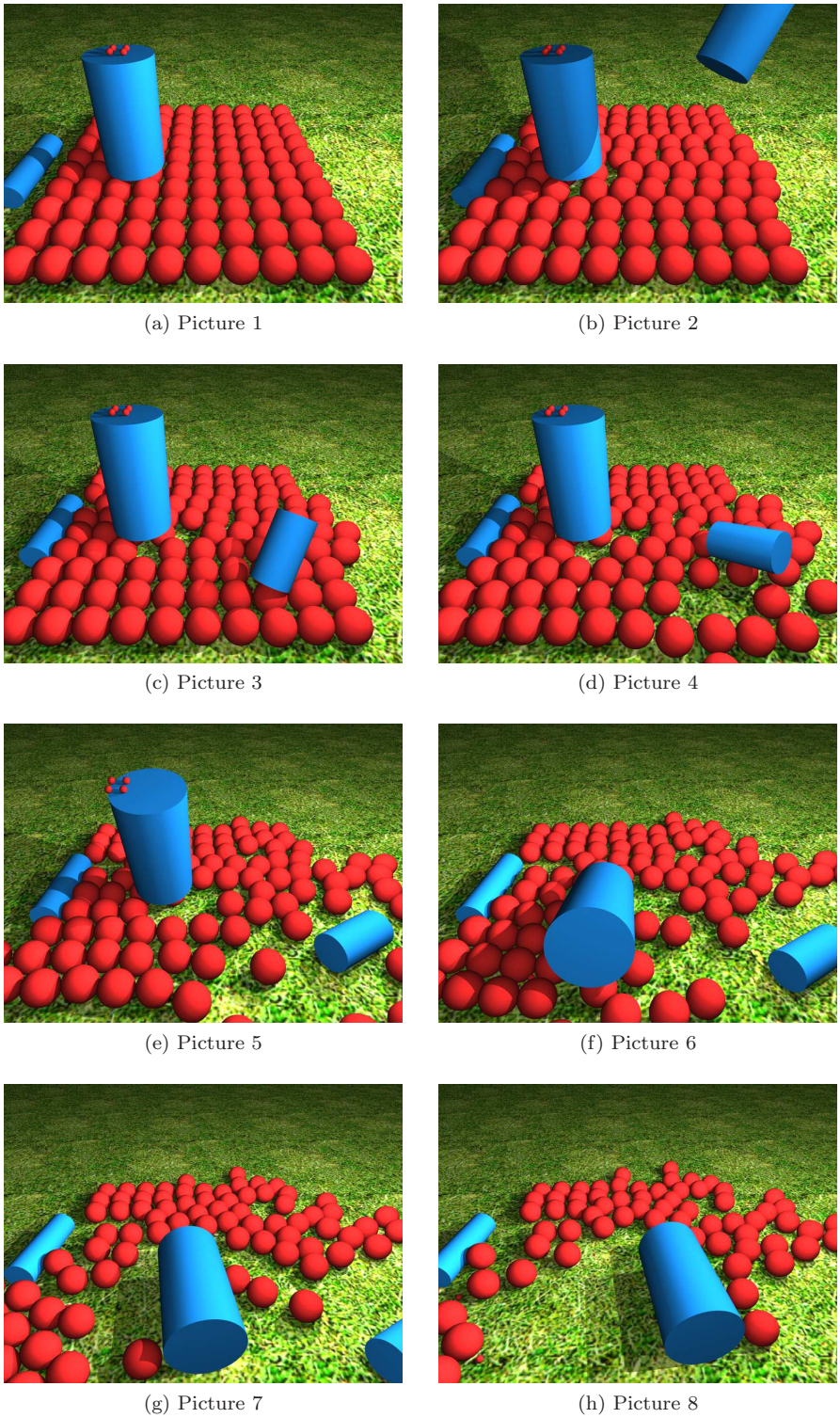
- One possibility is to change the cylinder to a capsule, which fits the cylinder as good as possible (Fig. 13 on page 16). Then you can reduce the cylinder-box collision handling to an easier capsule-box collision, which is already integrated to the *pe*.
- Presenting the cylinder as many spheres (Fig. 14 on page 16) and to reduce the current problem to a sphere-box algorithm is another idea.
- One more idea is it to subdivide the box into many spheres or capsules and to simplify the problem to a cylinder-sphere task. Here the spheres have to be set in one more dimension as in idea one because the lengths of the box are usually not equal.

Again with these simplifications the algorithm does not calculate real edge-edge contacts. Furthermore, this strategy makes the physical precision worse, which disagrees with the rules of the implementation in *pe*.

4 Example scenes

This part of the thesis demonstrates the new features of the *pe*:

The first scene (Fig. 29) describes a grid of spheres, which is deformed by a cylinder rolling from the left side very fast and another colliding cylinder from above. The third cylinder, which is standing on the spheres is thrown out of balance caused by its moving base and rolls over the ground plane.



(a) Picture 1 (b) Picture 2

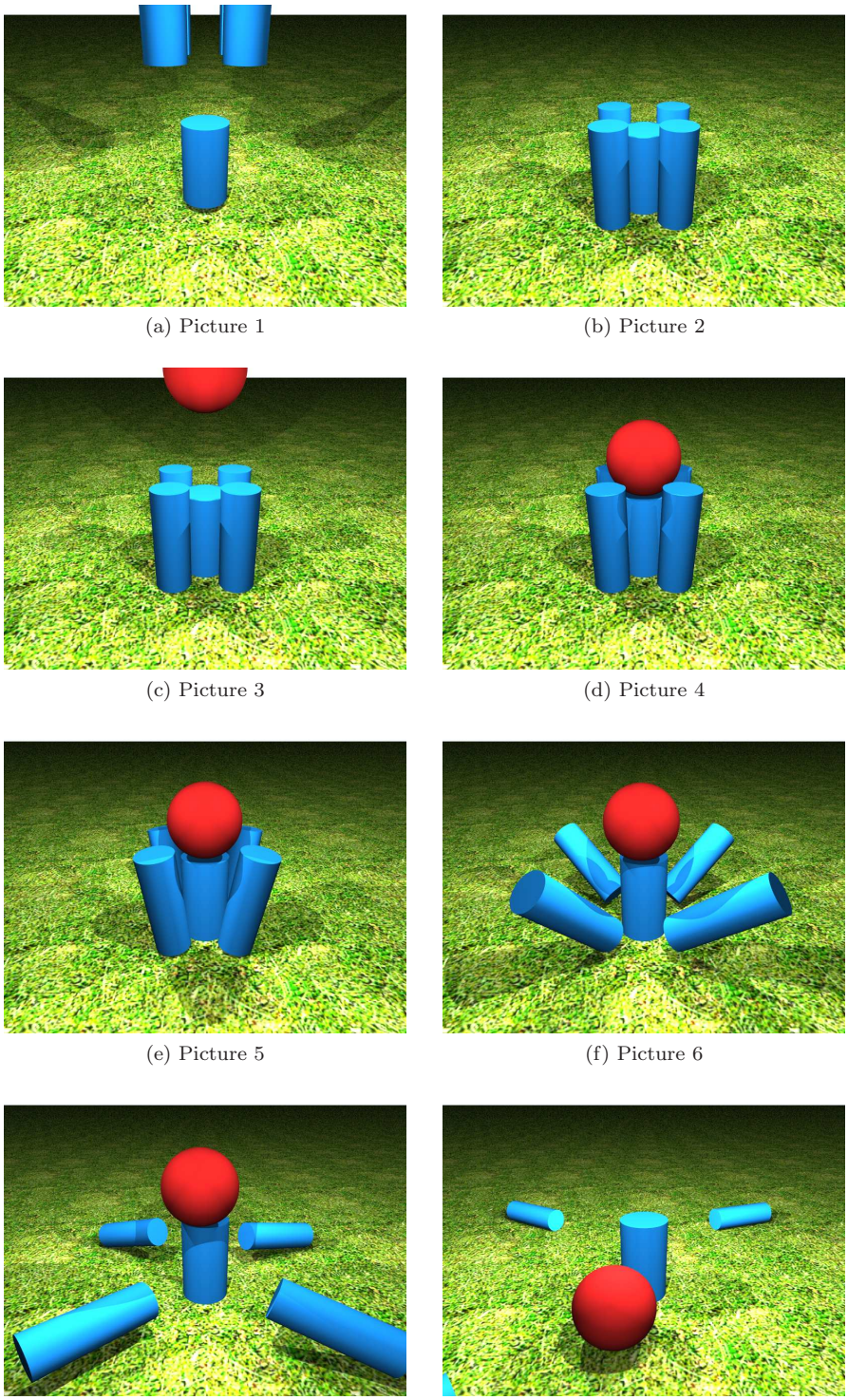
(c) Picture 3 (d) Picture 4

(e) Picture 5 (f) Picture 6

(g) Picture 7 (h) Picture 8

Figure 29: Example scene 1

Scene two (Fig. 30) shows five cylinders falling down onto the ground plane first. Then they stay standing until a sphere appears. The sphere knocks off the four quadratically disposed cylinders from their position. Then the cylinders move away from the scene and roll on the plane. Finally the centered sphere rolls down from the centered cylinder. Hence this example should also illustrate that numerical mistakes could have an effect on the behavior of the bodies.



(a) Picture 1 (b) Picture 2

(c) Picture 3 (d) Picture 4

(e) Picture 5 (f) Picture 6

(g) Picture 7 (h) Picture 8

Figure 30: Example scene 2

5 Conclusion

The goal of this thesis was to integrate the cylinder as a new geometric primitive in the *pe*. This task included the development of suitable fine collision detection functions for the collision types cylinder-plane, cylinder-sphere, cylinder-capsule, cylinder-cylinder and cylinder-box.
The algorithms for the first two mentioned types have been implemented successfully. From now on the collision detection phase in the *pe* is able to use these new algorithms for very efficient and precise simulations. The advantage of plane and sphere in comparison to the other three primitives was the small number of special cases. Furthermore the end cap difficulty of the cylinder was quite easy to handle in both collision types.
Big problems appeared during the development of the other three collision types. The advantages of using the line geometry have not been sufficient to solve all problems. Especially the end cap problem of the cylinder and the involved difficulties, e.g., the correct direction search, which points to the other body, and the very huge number of special cases made it impossible to integrate new algorithms for cylinder-capsule, cylinder-cylinder and cylinder-box in the *pe*. As support there have been studied some other physics engines ([5], [3], [4]) or books ([8], [9]) in order to get new ideas. But most of the authors just use the capsule geometry and avoid the cylinder problem. Another master's thesis ([13]) - which covers collision detection for cylinders with other primitives - contains some approaches, which could be used. But the important information is not worked out explicitly: Where is the right direction at the end caps, which points towards the other body. In this thesis it was not possible to get this important information, which is essential for the development of every case in detail.
Hence the end cap problem was the crucial difficulty, which made it impossible to solve all the tasks within the meaning of a bachelor's thesis.

Finally the cylinder is very good applicable as a geometric primitive if just cylinder-plane or cylinder-sphere collisions occur. For the more complex collision types there have to be used simplifications (as shown in chapter 3.4.2 on page 15), which reduce the physical precision. If these simplifications are not good enough, triangle meshes (as described at [12]) could be used to represent the cylinder. But then the big advantage of using geometric primitives with their few parameters is lost. Additionally triangle meshes do not represent a cylinder as good as a geometric primitive because the circular lateral surface can not be described exactly.

List of Figures

References

[1] `http://en.wikipedia.org/wiki/Physics_engine`, August 2010.

[2] `http://opende.sourceforge.net/wiki/index.php/Manual_%28All%29#Collision_Detection`, August 2010.

[3] Official homepage of Bullet Physics. `http://bulletphysics.org/wordpress/`, August 2010.

[4] Official homepage of Havok Physics. `http://www.havok.com/`, August 2010.

[5] Official homepage of Open Dynamics Engine. `http://www.ode.org`, August 2010.

[6] Official irrlicht homepage. `http://irrlicht.sourceforge.net`, August 2010.

[7] Official pov-ray homepage. `http://www.povray.org`, August 2010.

[8] David H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Academic Press, 2001.

[9] Christer Ericson. *Real-Time Collision Detection*. Elsevier Inc., 2005.

[10] Klaus Iglberger. *pe* physics engine. `http://www10.informatik.uni-erlangen.de/~klaus/`, August 2010.

[11] John S. Ketchel and Pierre M. Larochelle. Collision Detection Of Cylindrical Rigid Bodies Using Line Geometry. In *ASME 2005 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, 2005.

[12] Martin Ketzer. Geometrieprimitive und Dreiecksgitter für die pe Physikengine. Master's thesis, University of Erlangen-Nuremberg, 2007.

[13] Karen Petersen. Efficient collision detection and realtime simulation of kinematics, dynamics and sensors of autonomous vehicles. Master's thesis, Technical University Darmstadt, 2007.