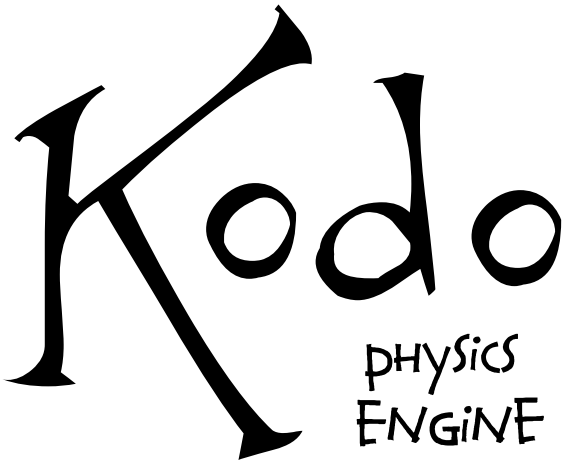
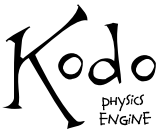


Rigid Body Physics In 2 Dimensions

Author: Anders Ekermo
Bachelor's Thesis in Game Development
Gotland University
Spring 2006
Supervisor: Mikael Fridenfalk



Abstract

The aim of this project is to produce the groundwork of a rigid body physics engine specifically for use in games. The engine is intended for simulations in two dimensions and code is written in C++. This document will discuss the reasoning behind the implementation, including but not limited to the underlying structure, primitive types, intersection detection and nonpenetration constraints. The project also aims to show some of the improvements made possible by using two dimensions instead of three.

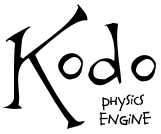


Table of Contents

1. Introduction

This document describes the process in which the Rigid Body Physics- Engine **Kodo** was created. Work started around April 2005, and while Kodo will hopefully continue to evolve even after this, the document only aims to chronicle the process up to May 2006.

It should be noted that most of the work on the original Kodo (from here on referred to as the "old version") was lost in August 2005. Though the old version did not reach the complexity and usability of the current Kodo, it was created in a very different manner and will sometimes be referred to in this text. Of course, since the data no longer exists any such references should not be taken as very important unless they are backed up with other information.

1.1. Motive

Realistic physics is a feature that seems to appear more and more in games and middleware today (see *1.2. Previous Work*). As such, it is a rather safe trend to follow and since a lot of sales in commercial videogames are largely due to "the latest technology", it is really all the reason needed. As Rollings/Adams[1] put it (in the similar context of 3D graphics in games), "... 'because it sells better' is a valid response. It doesn't mean we have to like it, though."

Wu/Hilmer[2] and Watt/Policarpo[3] has some more insightful reasons, including:
Universal rules, everyone intuitively "knows" physics so gameplay can be added without requiring the player to learn more about the interface.
Interactivity, since everything is handled by the system, more objects can be interactive without having to do more work.
Emergent behaviour, things do not need to be explicitly implemented to become possible. Of course, this is a double-edged sword as new ways to "break" the system appears.

Erleben[17] also points out that since computer graphics have improved in realism over the years, the lack of realism in other areas (such as physics) becomes more obvious and irritating.

Most of today's commercial physics engines focus exclusively on 3D environments, a safe choice since almost all commercial action games (games where real-time physics would be of use) are 3D today. It can be assumed, however, that physics would create opportunities for innovative and interesting gameplay in 2D games as well. Kodo is being made to simplify the implementation of such ideas.

1.2. Previous Work

There are some notable examples of middleware for 3D physics simulation, most notably those by Havok[4] and Ageia[5]. Non-commercial examples include Open Dynamics Engine[6] and Tokamak Game Physics SDK[7].

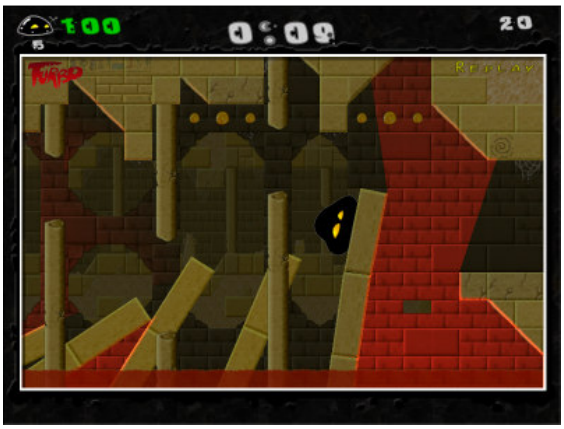


Fig 1: Gish, IGF 2005 Game of the year
Screenshot used with permission

Far too many games use physics to list them all here, although some notable examples of games based on innovative use of 2D physics include Chroniclogic's Gish[8], winner of the Seumas McNally Award for Independent Game of the Year at the Independent Games Festival in 2005, Mark Haley's Ragdoll Kung Fu[9] and a large part of the games in the Experimental Gameplay Project[10].

1.4. Structure

The Kodo API includes template primitive classes, rigid bodies sorted after these primitives and managers keeping track of the interaction between rigid bodies. These classes obey a top-down object-oriented hierarchy, allowing the programmer to decide how much functionality to use without having to worry about dependencies.

Since Kodo is intended to be used in games, plausibility has been deemed more important than exact realism. Design decisions and implementation priorities favour stability, scalability and speed before mathematically correct solutions and extra features.

1.5. Acknowledgements

I owe a great deal of thanks to my co-worker Christian Murray who has been very helpful with new ideas and reading material – even though he had nothing to do with this project. I would also like to thank my friend Theodor Berg, who was very helpful in proofreading this document.

2. Basic Components

Kodo is created in C++ and all programming work is done in Microsoft Visual Studio .net 2003[11]. It is created to be used either integrated into a project or as an external library, either static or dynamically linked. The code of Kodo is written to be open-platform and has been verified to compile with the free compilers Borland C++ Compiler[12] and GCC[13], although this does not necessarily apply to the demonstration projects.

2.1. Code Convention

Kodo is written using a loose code convention. The aim of the convention is to make the code more readable, so most rules are considered rules of thumb. For example, it is recommended to avoid writing lines longer than 100 characters or to omit scopes, but in cases where doing this can avoid large empty spaces, it should be done anyway.

The naming convention is based on Apps Hungarian notation, with notations for the most commonly-used types. Templates, classes, functions and files are all introduced with a similar comment header, no external documentation has been written so far but should it become needed, it will most likely use these headers as reference. Groups of similar functions use only one header, and functions implemented in the class definition do not have any header.

2.2. Basic Functionality

Basic functions are included in Kodo. Many of these simply call their respective C equivalent; the point is to simplify porting to different platforms by making sure all function calls within the kodo namespace are equal.

The functions include basic trigonometry and memory manipulation functions, additional math functions and a few number manipulation functions (Clamp, Min/Max, Swap etc).

2.2.1. The Array

Kodo has a simple, internal array-class for managing semi-dynamic-length lists of data. The array class works by keeping a dynamically allocated array that is a little longer than what is needed. The class keeps track of how much data is currently used, how much is allocated and how much to grow it when the allocated data is used up. This way, it will reserve a little more memory than it actually needs, but the average overhead for adding data will be smaller.

2.3. 2D Space

Among the base components of Kodo are a couple of template classes used to manage 2D geometry, a vector class and a matrix class. A 3D vector class is also implemented, but it is currently unused and does not contain anything not in the 2D vector class. Note that the math classes and primitives are templates, but the higher-level functionality of Kodo uses 32-bit floating-point variables to represent scalar values.

2.3.1. The Vector

Vectors are stored in coordinate form and can be used to represent directions or points in 2D space. They have all the basic math operators implemented, note that the division operators simply multiply the values with the reciprocal of the divisor since a multiplication operation is usually faster than a division. Since this does not work for integer values, the functions "SafeDivide" and "SafePostDivide" perform "proper" divisions.

$$\mathbf{A}_{\perp} = (-\mathbf{A}_Y, \mathbf{A}_X)$$

Eq 2.1: Perpendicular vector

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{A}_X\mathbf{B}_X + \mathbf{A}_Y\mathbf{B}_Y$$

Eq 2.2: Dot product

$$f(\mathbf{A},\mathbf{B}) = (\mathbf{A} \cdot \mathbf{B}, \mathbf{A}_{\perp} \cdot \mathbf{B})$$

Eq 2.3: A oriented with B

The vector class provides a function for quick access to the counterclockwise perpendicular vector (Eq 2.2.). It also implements the dot product and perpendicular dot product functions; these are used heavily in physics to determine the projection length of one vector on another (Eq 2.1.)

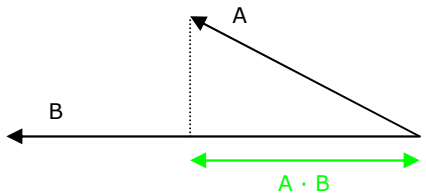


Fig 2: The Dot Product, simplified

Finally, because of the properties of 2D rigid bodies (No mirroring or scaling, See *4.1.2. DOF and Body Space*), it is enough to store the orientation of a rigid body with a 2D unit vector, representing (Cos(α),Sin(α)) where α is the angle of rotation. The vector has functions to transform a vector from one orientation to another (Eq 2.3.).

2.3.2. The Matrix

The Matrix class represents a 3×2 matrix used to create affine transformations in 2D space. It is quite similar to the vector in that it implements basic operators and the "SafeDivide" functions.

The matrix has some basic math functions for using matrices in transformations, but since the extra 2 scalars are not needed in rigid body transformations, it is largely unused in the Kodo core functionality.

3. Intersection

To have any kind of interaction between bodies, being able to determine their relationship to each other is necessary, at the very least if they are intersecting or not.

Ericsson[14] defines collision detections as a matter of finding *if*, *when* and *where* two objects collide. For reasons discussed in *4.2.2. Simulation*, finding the time of collision can be omitted.

3.1. Primitives

To model anything other than particles, some way of defining the area occupied by a rigid body is needed. Wu/Hilmer[2] suggests using either Convex Hull Unions, Sphere collections (which would really be a degenerate convex hull union) or simply using the mesh (which, in a 2D world, would usually mean using the sprite). Baltman/Radeztsky[24] use a method with particles bound together with constraints, an interesting method though not exactly the aim of this project.

Using spheres only is a solution that is beautiful in its simplicity and allows itself for space partitioning trees. However, it is not very intuitive for complex models and can easily consume a lot of memory. Using the display data (sprite or mesh) is probably the most intuitive solution. It requires no extra memory and is very easy to comprehend for non-programmers, although the display data is very seldom optimized for collision detection. Also, extracting additional information from a collision (for example where the collision originally took place) proves difficult, and more often than not, it is desired to keep the collision and display data apart.

This leaves the convex hull unions, somewhat of a middle ground between the two previous ones, and the favoured solution of most middleware physics engines. The main benefits of using convex primitives to represent a body are that they use relatively little information and are intuitive, while still being developed for fast and precise collision detection. The main drawback is that the complexity of the individual intersection detection methods is increased, but since the simulation takes place in only two dimensions, it is still kept simple.

All primitives are made into template classes to achieve maximum reusability.

3.1.1. Circle

The circle is a 2-dimensional version of the 3-dimensional sphere. It is defined as a center point and a radius.

3.1.2. AABB

The Axis-Aligned Bounding Box [AABB] is a rectangle aligned to the principal axes, defined by a center point and a dimension vector holding the half width/height. Since it cannot rotate, it cannot be used for rigid bodies (see *4.1.2. DOF and Body-space*), but it can be valuable as bounding volume as collision detection is usually very simple.

3.1.3. OBB

The Oriented Bounding Box [OBB] is a rectangle with arbitrary orientation. The orientation is defined by a unit vector holding the "right" – direction, the positive direction of axis 0.

The OBB class contains a method for determining the optimal (smallest) bounding box for an arbitrary set of points. The method is based on a heuristic brute-force algorithm as suggested by Ericsson[14], but is made a lot simpler since there are only two dimensions and three DOF. In simple terms, bisection is used to find the orientation angle that will produce the smallest possible bounding box.

The dimensions of both the AABB and the OBB could be defined as Min/Max instead of Center/Dimensions. Especially in the case of AABBs, new data is usually presented in this manner so it might seem like a better idea – but the collision detection usually requires the data to be in the latter format.

3.1.4. Line

A simple line segment is defined by a start- and endpoint. Since it is a degenerate OBB, not much of it has been implemented but it will most likely be useful if rope physics is needed.

The line is defined as Start/End instead of Start/Direction/Length since the data is almost always presented that way. It should be noted that the old version stored all of the data in one class, but not counting the obvious drawbacks of having redundant data, re-normalizing the direction vector for every new line became expensive. Most

intersection methods do not need normalized data anyway.

3.1.5. Plane

A plane in two dimensions is just a ray cutting the space in half. It is defined just like a 3D plane by a normal vector and a distance from Origo. This primitive is used for defining the sides of a convex polygon.

3.1.6. Polygon

Polygons are collections of arbitrary points forming the convex hull. Obviously, they can form the closest-fitting bounding volume, but in return they have the most expensive intersection tests.

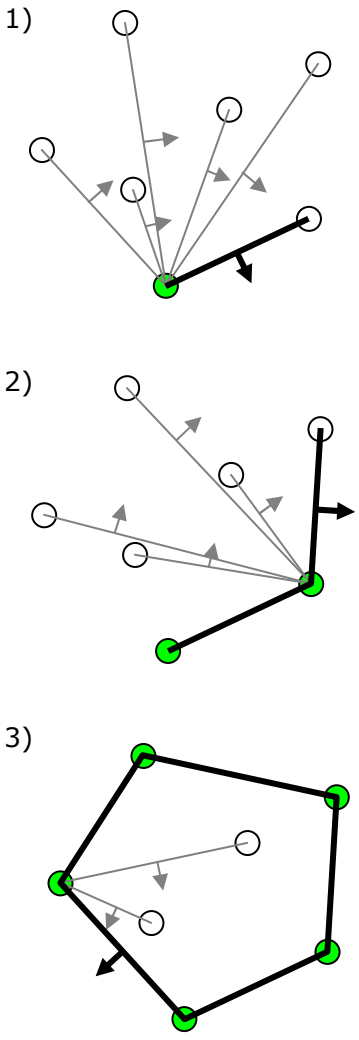


Fig 3.1: Finding the convex hull; each point is evaluated until the original point is reached

The polygon class holds an edge count, pointers to lists holding the corner points (in counterclockwise order) and edge planes, and a boolean indicator of whether it owns the memory used for the lists or not. The points and planes are stored in local-space, described in *4.1.2. DOF and Local Space*. Like the OBB, it also holds a position and an orientation. These could be deemed unnecessary as the points themselves could simply be stored in world-space at all times. It is, however, usually more effective to convert the other primitive in the intersection test-pair to the polygon's local space than to convert them both to world-space. Also, a simple operation like moving or rotating the polygon would be a lot more expensive if every point needed to be individually transformed.

Seeing as any points can be entered in the polygon, some way of making sure it is convex is needed. Archimedes originally pointed out that any two points in a convex body can be connected with a line that does not move the body. With this in mind, determining if a body is convex is a simple matter of testing, for each plane defining the body, if every point is either behind or on the plane (as mentioned by Ericsson[14]). The convex hull of an arbitrary set of points can be found using this idea in a brute-force manner. First, any point on the convex hull is determined (any endpoint on an axis projection will do), then, a line is drawn to every other point in the set and the side with no points behind it is added. Since Kodo only concerns two dimensions, a simpler solution would be to just pick the line with the greatest angle in relation to the previous side, but extracting the angle might be a more expensive procedure than just testing all the points. Ericsson[14] also has a few suggestions for faster procedures, but this is not usually done in real-time and optimizing it is not necessary.

3.2. Collision Detection

Collision is detected between two primitives of the same or different types. The kodo namespace holds a number of template functions to determine intersection between each possible primitive used in the rigid bodies.

This section discusses the narrow-phase collision detection, the identification of actual collisions between pairs of objects. Section *5.1. Sort and Sweep* talks about broad-phase collision detection, how smaller groups of potentially colliding objects are identified.

3.2.1. Separating Axis Algorithm

The separating axis theorem [SAT], described by Gomez[21] and Burns/Sheppard[19], is a general procedure for determining intersection between convex objects in *n* dimensions. When two convex objects do not intersect, they can be projected onto an axis that shows this separation. In three

dimensions, the axes that need to be considered are n^2 (All normals and all possible cross products), but in two dimensions only n axes need to be considered (the normals of the primitives).

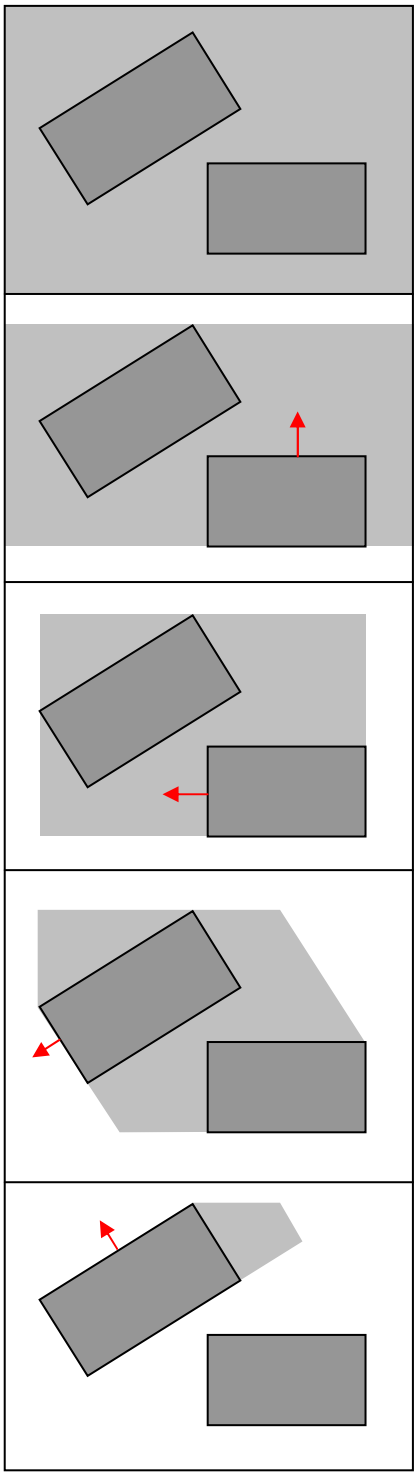


Fig 3.2: SAT eliminates most possible collisions early and difficult cases later on

The main advantages of using SAT are that the same general procedure can be applied to a lot of different shapes with effective results, and that it gives a lot of opportunities for early-outs. An intuitive solution might look for signs of intersection (intersecting edges etc.) instead of the other way around, but in games, any given pair of objects are more likely to be separated than intersecting, and such a solution would very seldom be able to exit early. As shown in the picture, most objects are culled in the first or second SAT test.

Ericsson[14] and others suggest using temporal coherence to speed up the search for the separating axis. This is not done in Kodo since it would require extra storage space and more processing even for simple tests, and since the number of potential separating axes is much lower in 2D, it is unlikely that exploiting the stiffness of a system will speed it up a lot. If an implementation uses a lot of high-complexity primitives (that is, polygons), temporal coherence might be useful.

3.2.2. Circle tests

Burns/Sheppard[19] points out that since circles have a theoretically infinite number of sides, SAT cannot be used as-is. It is possible to use the axes created by using the circle's center and the corners of the polygon, but this is an overly complex procedure for such a simple shape.

James Arvo[22] developed a different method for sphere-box testing. The function works by measuring the distance from the circle center to the closest point of the box and comparing this to the radius of the circle. This distance is easily measured by checking for each axis the distance from the circle center to the box's side – and then using Pythagoras' Theorem to compute the actual distance.

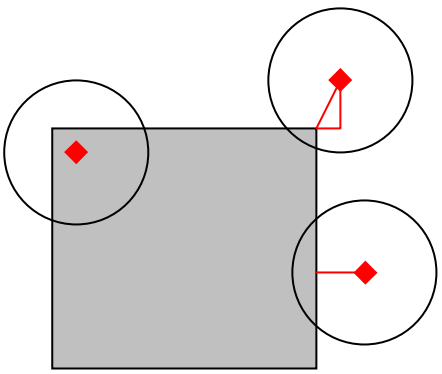


Fig 3.3: Arvo's Algorithm does a separation test for each axis

A similar method can be used in circle-polygon testing. Instead of testing every axis formed by the circle center and a point of the polygon, it is enough

to find the side with the largest distance to the circle center and test that for intersection.

3.2.3. Polygons and SAT

As previously mentioned, polygons have the most expensive intersection tests since they are the most flexible primitives. Though a straightforward SAT works fine, it can quickly become expensive when the side count rises. The main problem is quickly finding the extreme vertex when the polygon is projected on an axis.

Ericsson[14] has a few suggestions, including a Dobkin-Kirkpatrick hierarchy and hillclimbing, a simple heuristic algorithm. The Dobkin-Kirkpatrick hierarchy works by starting out from a simplex (a triangle, in this case) and finding the "correct" vertex in increasingly complex shapes until the original polygon is reached. Hillclimbing works by simply traversing the edge of the polygon to find the point where both neighbours have a smaller projected distance.

As Kodo is 2D it does not need a very complex solution, Hillclimbing is used since it is very straightforward. The Dobkin-Kirkpatrick hierarchy has the benefit of working in near-constant time for a given object, but it requires extra storage space per primitive.

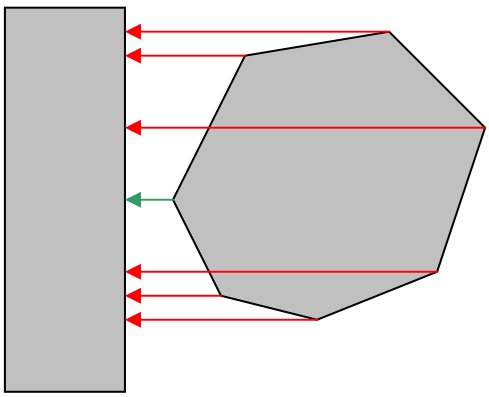


Fig 3.4: Hillclimbing works since only one vertex can be closer than both of its neighbours

A helper method exists in the polygon to optimize its composition, it is meant for automatic pre-processing of input data and not real-time usage. The method arranges the polygon in such a way so that the corner with the sharpest angle (i.e. the vertex most likely to be the extreme vertex) is always examined first. The method also uses the optimal-OBB method to find the orientation that will produce the smallest AABB when the polygon is un-oriented, since the AABB is used in broad-phase testing (see *5.1. Sort and Sweep*)

3.3. Intersection Information

When processing physics, typically a lot more information is needed than whether two objects collide or not. As most of this information is readily available when performing the actual intersection test, it is better to store it then than trying to find it again later.

The information is stored in a template class and extracted in the aforementioned test functions. To avoid wasting time writing unnecessary data, nothing is written until a collision has been confirmed.

3.3.1. Normal

The collision normal is the normal of the surface with the least projection, always directed from the first body towards the second. The normal is needed for several collision handling reasons, but it also represents the best axis for separating the bodies.

3.3.2. Points and Depths

The point of intersection and intersection depth can easily be extracted since they, like the normal, are by-products of the Separating Axis Algorithm. For simplicity, the primary intersection point is always the corner with the deeper projection along the normal.

Any collision between two objects will be a vertex/vertex, side/vertex or side/side. In the case of a side/side collision, the single intersection point gets replaced by a line segment. Catto[23] describes a method of extracting the contact area in 3 dimensions; this method can be simplified and used in 2 dimensions. After the normal and the primary intersection point has been found, the neighbouring point is checked to see if it is clipped by the separating axis. If it is, the collision is marked as just having a single intersection point, otherwise the point is clipped to the rest of the polygon and the resulting point is considered the second end of the contact area. The primary contact point is always the first one.

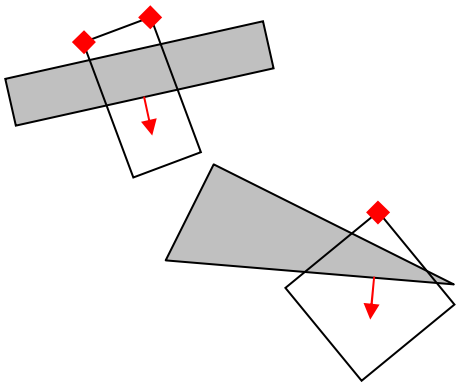


Fig 3.5: The intersection point might appear outside of one of the bodies

The primary contact point is always a corner of one of the bodies. This can create problems with tunnelling (objects moving through each other due to infrequent collision testing) as illustrated in Fig 3.5. The problem can be minimized by treating all such collisions as single-point, since the abovementioned algorithm can create strange results if the original point is not inside both of the bodies. Should it prove to be a big problem, it would be wiser to treat the tunnelling problem instead of cluttering the intersection detection with additional tests.

4. Rigid Body Dynamics

While the primitive intersection functions are the tools for building them, the rigid bodies and the classes that manage them are the core of the rigid body physics engine.

Dynamics, the study of the effects of forces and masses causing kinetic energy to change, is what separates physical simulations from more "traditional" collision methods. By using Newtonian physics instead of arbitrary methods on in-game objects to control movement, simulating conditions similar to the real world becomes an emergent feature of the system.

4.1. Rigid Bodies

A rigid body is, quite simply, a body that does not deform. In the real world, true rigid bodies do not exist, but in simulation they are very useful; if the shape of a body is considered constant, only the movement of the body needs to be taken into account. This might seem like a big limitation at first, but complex objects can be modelled by tying several rigid bodies to each other with different methods, and the rigid body model is actually very flexible.

All rigid bodies have some form of volume and a collection of properties that determines how the rigid body reacts to forces. These properties and related functions are stored in a base class – however, since the volume may have any of the different shapes mentioned in part *2.1. Intersections*, subclasses are made for each type. It might seem un-intuitive to let the body "be" a volume instead of "having" it, but to do this the body would have to have either redundant data or a dynamic primitive container. Arguably, neither is a prettier solution and both are computationally slower.

4.1.1. The Rigid Body State

The rigid body state is defined according to Baraff[18] as the body's position [**X**] and orientation [**R**], linear velocity [**v**] and angular (rotational) velocity [ω]. Since the state needs to be cached and restored, these properties are put in an internal class in the rigid body, a "current" and a "cached" object is created.

The **Position** is a 2D vector and defines the position of the rigid body's **center of mass** [$\mathbf{X}_C$]. Considering the rigid body as a collection of particles, the center of mass is the mean position of these particles [$\mathbf{r}_i$] weighted by their individual masses [m_i] and divided by the total mass [M] (Eq 4.1.). Thus, in a symmetric rigid body with evenly distributed mass, the center of mass would be in the exact center of the body.

$$\mathbf{X}_C = \frac{\sum m_i \mathbf{r}_i}{M}$$

Eq 4.1: The center of mass

The **Linear Velocity** is also a 2D vector and defines the change in position of the center of mass in a single unit of time.

The **Orientation** of the body is yet another 2D vector and it defines how the body is rotated around the center of mass, containing the direction of the "right", or positive direction on axis 0, of the rigid body.

Finally, the **Angular velocity** of the object is a scalar defining how many radians it will rotate around the center of mass in a single unit of time. Since the orientation is a vector, it might seem unnecessary to perform two expensive trigonometric functions every time the system is updated. The angular velocity represents the arc-length of the rotation and needs to be a scalar for simplicity, flexibility and stability. While it might seem like a good idea to store the orientation as a scalar as well, it is needed in vector form for most intersection queries and, in a graphical implementation, for presentation.

$$\mathbf{P} = M\mathbf{v}$$

Eq 4.2: Linear Momentum

$$L = I\omega$$

Eq 4.3: Angular Momentum

Baraff[18] advocates storing the linear [**P**] and angular [L] momentum (Eq 4.2. and 4.3.) instead of storing velocities. These are constant in a closed system (as implied by Newton's first law of motion) and provide a better relation to Newton's second law of motion (Eq 4.4.), but velocities must be extracted adding an extra step to moving a body. In a game, it is unlikely

that a completely closed system will be treated and having easy access to velocity is arguably more important than having easy access to momentum.

$$\mathbf{F} = M\mathbf{a} = M\mathbf{v}' = \mathbf{P}'$$

Eq 4.4: Newton's 2nd law of motion; Force equals the rate of change in momentum

Note that since position and orientation exists in all bodies, the actual primitive objects are not stored in the rigid body objects. Instead, the subclasses hold the extra information needed for each primitive type and creates the primitive on demand.

4.1.2. DOF and Body Space

As previously mentioned a rigid body can move around freely in two dimensions and rotate around its center of mass. Since a rigid body cannot deform, the points in it have a fixed relation to each other and are stored in "local space", or "body space" – the center of mass is considered Origo and the axes are the orientation and its perpendicular vector.

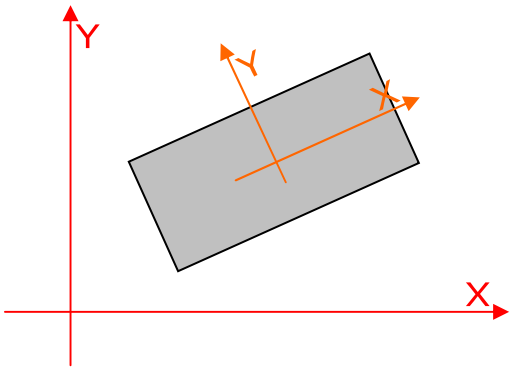


Fig 4.1: Local space of a rigid body

Chasles' Theorem (Hecker[16]) breaks up motion into linear and angular components this way, and the velocity of a point [$\mathbf{v}_i$] in a rigid body can be calculated by adding the linear velocity to the perpendicular vector of the point's relative position multiplied by the angular velocity (Eq 4.5.).

$$\mathbf{v}_i = \mathbf{v} + \omega\mathbf{r}_{i\perp}$$

Eq 4.5: Velocity of a point

The Degrees of Freedom [DOF] determines in how many ways a body is free to move. In two dimensions, the body can be moved up/down or left/right and rotated, making 3 DOF. A three-dimensional rigid body can move in three directions,

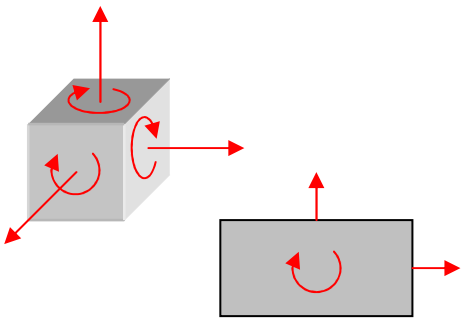


Fig 4.2: Degrees of Freedom in 2D and 3D

but it can also rotate in three dimensions instead of one, so it has a total of six DOF and needs to store an axis of rotation as well as the angular velocity and orientation.

4.1.3. Forces and Torques

Forces [**F**] and Torques [τ] are applied according to Newton's second law of motion and its application on angular velocity as described by Watt/Policarpo[3] among others (Eq 4.6.).

$$\tau = I\dot{\omega} = I\omega' = L'$$

Eq 4.6: Newton's 2nd law of motion applied to angular velocity

Since the forces and torques do not immediately affect the rigid body state but only its rate of change in momentum (written as Mass * Linear Acceleration [**a**] and Inertia [I] * Angular Acceleration [$\dot{\omega}$]), they are added up and stored in the rigid body. A helper function handles adding forces to any point of the rigid body, using Chasles' Theorem to determine the added torque.

4.1.4. Additional types

A rigid body can be formed from a circle, an OBB or a polygon, but it can also be created from a point or a combination of these objects, a "compound object". A point-body is only a degenerate sphere, but some calculations have been optimized to allow for simple handling of particles or other kinds of objects that does not need as accurate physics as the others.

Compound objects contain a list of pointers to sub-objects. The sub-objects are defined by structures containing the same kind of data as their rigid body counterparts, except in the object-space of the compound object. This makes adding a new object cheap but removing one relatively expensive, but the compound objects are primarily made for creating concave objects and not for breakable ones. To simplify object "welding" features (like arrows getting

stuck in a wall), methods in the compound object allow adding primitives as well as other bodies.

Intersections involving compound objects are solved by iterating through the sub-objects and testing every possible pair for collision. If more than two intersection points are found, the two with the greatest penetration depths are kept and the collision normal is calculated as the average of the two points' normals. This can lead to problems if the normals are opposing each other, but it is a problem with the rigid body concept and careful design should keep it to a minimum.

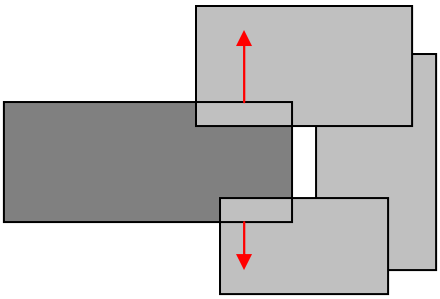


Fig 4.3: Compound object unable to produce an average normal

4.1.5. Physical Properties

Besides volume, the rigid body has a collection of constant physical properties, for simplicity grouped in an internal class of the rigid body. These properties (as with volume) do not necessarily have to be constant at all times, but as long as they are the rigid body will behave the same in similar situations.

The **Mass** of the object determines its resistance to change in linear velocity and the **Moment of inertia** its resistance to change in angular velocity. In three dimensions, the inertia depends on the axis of rotation and must be stored in matrix form (or similar). Since there is only one possible axis of rotation in two dimensions, a scalar is enough. As will be shown later, both mass and inertia are used mostly in divisions so the reciprocals are stored as well. Mass can be automatically generated by providing the density of the object, and the inertia can be generated from the mass and the shape of the body.

Inertia, as explained by Watt/Policarpo[3], is the "sum of the masses of each particle weighted by the square of the perpendicular distance to each axis" (Eq 4.7.). A few methods for calculating the inertia of symmetric homogeneous shapes are provided, but the inertia of arbitrary shapes (polygons, compounds) can be estimated by evaluating a sample of evenly distributed particles in the body. Since a lot of points are needed for an accurate value, this procedure can become expensive and should not be called in real-time.

$$I = \Sigma m_i \mathbf{r}_i^2$$

Eq 4.7: The moment of Inertia

The **restitution** and **static/kinetic** friction are scalars in the 0..1 interval and determine how much energy is lost in a collision. Normally, some kinetic energy is converted as objects are deformed in collisions, but since rigid bodies do not deform, this energy is assumed to be lost completely. As described by Burns/Sheppard[19], the restitution (or elasticity) determines the loss of momentum along the collision normal and the friction along the vector perpendicular to the collision normal. The static friction determines the friction for a resting body and the kinetic friction for a moving one. Read more about this in *4.3.3. Friction and Restitution.*

4.1.6. System flags

The rigid body holds a 32-bit bitfield with flags defining the body and a 16-bit bitfield with flags defining requirements of bodies it can collide with. The top 16 bits of the rigid body are reserved by Kodo and used for the following things;

LOCK_MOVE and **LOCK_ROTATE** simply prevent the rigid body from moving or rotating. In calculations, the mass or the moment of inertia of the rigid body is considered to be infinite if the respective flag is set.

ENABLED simply determines whether the object is enabled or not. A disabled body will not collide with other bodies.

RESTING should not be changed by the user as it is set and unset by the system when the object enters or leaves a resting state. Read more about this in *5.2. Resting*. The **NEVER_REST** flag disallows the rigid body from ever entering the resting state.

IS_COLLIDING should not be changed by the user either. It is cleared by the system in the start of the integration function (see *4.2.2.1. Order of Processing*), and set if the body collides with another body.

The final four flags determine how the bottom 16 flags are compared. An object can be set to "any" (any flag will do), "all" (all flags must be present), "equal" (all flags and no others) or "no test" (always pass). Note that if no flags are set, "any" and "all" will fail. Also note that if rigid body A is allowed to collide with body B, B should also be allowed to

collide with A, otherwise the ordering of the bodies will determine if they collide or not, and this may lead to unexpected results as the processing order is not user-controlled.

4.1.7. Additional Data

The rigid body contains a 32-bit integer to hold any kind of user data, for example a pointer to the game object it is associated with. This user data is never read or written by the Kodo engine.

The rigid bodies also hold information about a grown version of themselves, the format of this depends on the type of the body. This grown version is used to determine what other rigid bodies are potentially touching the body and to preserve space, no extra information is stored in polygon bodies or compounds. Instead, the bounding box/bounding circle is used.

4.1.8. Additional Functionality

A couple of helper methods are included for finding the area and testing a point for intersection. These are also used in the mass/inertia auto-generation methods. To make duplication of rigid bodies easier, a virtual "clone" method will create a dynamically allocated copy, and another method in the base class will duplicate only the state, flags and physical data.

4.2. Physics Manager

The rigid bodies hold all information regarding the individual objects in the physical simulation, but the information about what rigid bodies exist and how they interact with each other is contained in the Physics Manager. This is the context in which the simulation happens and the core of the Kodo physics engine.

Due to the hierarchal design, the Rigid bodies themselves have no knowledge of the physics manager and could potentially be included in several physics managers at once. The exception to this would be simultaneously running two managers on the same bodies in separate threads. The manager uses some temporary attributes of the bodies and values could become unreliable.

4.2.1. Basic Structure

The physics manager holds pointers to the rigid bodies it "owns" in a double-linked list. To save time and space, the linked list itself is stored in an array and the links are 16-bit unsigned integers instead of pointers. In order to make sure both adding and removing rigid bodies is reasonably fast, the array is not re-formed when objects are deleted. Instead, another array holds the positions of the unused entries.

The linked list actually has two entry and exit points, one for active bodies and one for resting ones. This is described further in *5.2. Resting*.

The physics manager also has methods for adding forces and torques to all of its bodies, or to a subset defined by a flag parameter. To simplify adding of gravity, it can also add a force multiplied by the individual mass of each object.

4.2.2. Simulation

Moving a single rigid body in a somewhat physically correct manner is a rather trivial task; one simply has to integrate the position of the body over a series of short timesteps and display the result. Treating a collection of rigid bodies quickly introduces some problems, mainly that of collisions. By testing for intersections at each frame, it is easy to find out if a collision has occurred between two rigid bodies sometime during the timestep, but not quite so easy to know exactly where and when the collision occurred.

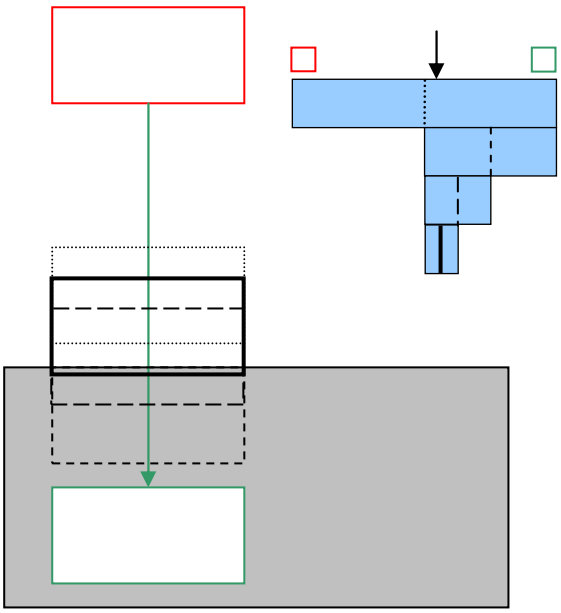


Fig 4.4: Using bisection to find the time of collision

Using geometry to find the exact solution might be possible, but is extremely complicated for anything other than the simplest of shapes. Baraff[19] and Del Palacio[20] restore the cached states of the bodies, use bisection to find the time of collision within an acceptable threshold and process it, they then forward the entire system to that time and do it again until the entire timestep has been completed.

This method was used in the old version, and while it produces accurate results, it has some problems. First

of all, a lot of re-evaluation has to be done every frame, causing it to have a very large speed variance. Contact forces become harder to compute since the error threshold becomes very small, and since numerical integration is used, the final state of a body actually depends on how many intersections were in that frame. The concept of arbitrary threshold values is introduced, and the demands on initial conditions are higher – if bodies are already intersecting at the start of a frame, it is not possible to find the time of collision and left unchecked, this problem will cause the system to freeze.

While it may be nice to rest assured that no two bodies will ever penetrate each other at the end of the frame, this comes at a high cost. As Kenny Erleben[17] put it, "You may as well recognize that penetrations are a way of life. So instead of trying to get rid of them, accept that they exist and fix them instead."

With this in mind, the unrealistic properties of rigid bodies are compensated with another unrealistic property- bodies are allowed to occupy the same space. Collisions are handled as they were detected, and to prevent objects from sinking through each other, they are separated at the end of the frame. This design is much more in line with the plausible-rather-than-realistic- mentality.

4.2.2.1. Order of Processing

The entire execution of a timestep is put in a single function taking the length of the timestep as a parameter, and set of flags allowing the user to discard some of the functionality as another.

A straightforward solution would be to integrate the velocity and transformation of each particle, process collisions, determine contacts and finally separate the intersecting bodies. Guendelman et al[15] points out that this makes threshold values for contact velocities needed, and suggests that contact resolution should occur in between the velocity and transformation updates. Their idea is to build a system that favours keeping stable bodies stable over having a fully physically correct behaviour at all times. Building on this idea, the manager processes the bodies in the following order;

Cache the bodies and clear all collision flags.

Forward the bodies in time, regarding both velocity and transformation.

Determine collisions. Finding the collisions is a geometric problem so the velocity of the body does not matter. The collisions are found in the bodies' final states and stored in a list.

Restore velocities to their cached states. Collisions are assumed to occur at the start of the timeframe to induce stability. As such, the initial velocities are used in collision response.

Resolve collisions. All collisions are resolved several times, more on this in *4.2.2.2. Iterations*.

Forward velocities and do **contact calculations**. Contact is resolved according to Guendelman[15], by treating active collisions a number of times with zero restitution. This will cause unstable objects colliding multiple times to settle. In essence, the body is extrapolated to see if it strives to settle or not.

Restore transformations and **separate bodies** that are intersecting.

Forward transformations using the new velocities. For bodies that are stable, the new velocities should have been reduced to zero by the contact calculations and the bodies will remain still.

4.2.2.2. Iterations

Several iterations are used – when integrating the time to forward bodies as well as when processing collisions. The numbers of iterations are stored as variables in the physics manager so that they can be set at run-time.

Since integration is handled numerically, a smaller timestep will produce a smaller cumulative error, which is why the original timestep is split up and processed in smaller parts. Read more about this in *4.2.2.4. Integration*.

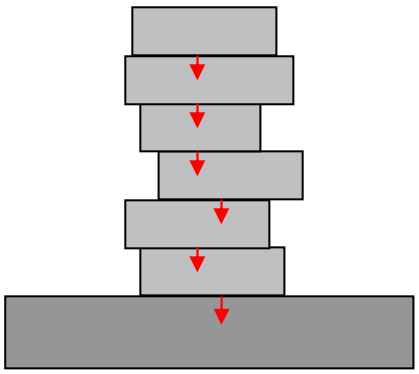


Fig 4.5: A stack of boxes showing the problems with pairwise testing

Collisions are processed several times due to the problem with pairwise collision handling. Consider figure 4.5, the bottom box in the stack should apply the combined force of the boxes resting on it on the block below instead of just the force due to gravity and its own mass. As it is now, the box can just as

easily be pushed away from the side as the topmost one. What is worse, since the topmost box descends with the same speed as the box below, it will not register a collision – in the worst-case scenario, only the bottom-most box will.

As mentioned by Guendelman[15], a solver considering the entire system at once instead of just the individual pairs could be more efficient, but processing the collisions several times will increase stability as at least one new box will be "stopped" with every iteration.

Guendelman[15] also suggests a shock propagation model to stabilize the system before finalizing the integration step. A similar model is constructed by sorting the collisions and processing them from lowest to highest, using zero restitution and treating the lower of the two bodies involved in the collision as an infinite-mass object with no velocity. This can greatly improve stability in a system with a lot of stacking, without raising the number of rest iterations – but only when gravity is the major force. Also, since collisions are not grouped into a contact graph as suggested by Guendelman[15], the sorting algorithm can become unnecessarily expensive when a lot of separate stacks are formed. For these reasons, shock propagation is disabled by default.

4.2.2.3. Random ordering

Another improvement to the pairwise handling model is to evaluate collisions in a random order, as suggested by Guendelman[15]. Since collisions change the velocities of the bodies, there is an inherent bias in the ordering. In order to diminish this bias, a new permutation of the collision list is used every time it is looped through.

Creating a random permutation can become computationally expensive when performed so many times per frame; instead the list is iterated using a step-size that does not share any factors with the total list-size – a semi-random permutation. The step-size is picked from a list of primes every time a new permutation is needed. It being a prime is not necessary, though it is simpler to check for divisibility since no factors need to be considered.

4.2.2.4 Integration

Most formulae in dynamics, including the ones used to move rigid bodies, are Ordinary Differential Equations [ODE]. The position of a rigid body over time, for example, is a function of the velocity – the derivative of position, and the velocity is itself a function of the acceleration – a derivative of the velocity.

Baraff[19] describes the process of solving ODEs numerically, the simplest way being Euler's Method (Eq 4.8.), in the case of velocity simply adding the current velocity multiplied by the length of the timestep to the position of the body. He does not recommend using Euler; pointing out that it accumulates an error as in Fig 4.6. Instead, Baraff[19] advocates providing the ODE solver at runtime.

$$\mathbf{x}(t_0+h) = \mathbf{x}_0 + h\mathbf{x}'(t_0)$$

Eq 4.8: Euler's method for solving ODEs numerically

Guendelman[15], on the other hand, suggests that while other integration methods are more accurate for bodies that do not interact with each other, this accuracy is lost in the presence of contact and collision. Since Kodo is concerned with interactive simulation and more concerned with plausibility and stability than accuracy, the safest solution seems to be to use smaller step-sizes by iterating the integration, and to design around the problems with the Euler-method.

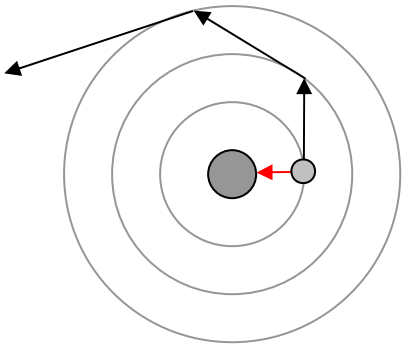


Fig 4.6: A centripetal force should keep the body at a constant distance from the center; Euler's method will cause it to spiral away

4.3. Collision Response

Collision response is handled by the physics manager, and the results are readily available should the user wish to have a collision induce additional effects. As previously mentioned, collision detection is only performed once but collision response is processed several times to increase stability in the system by allowing stacks of objects to settle. One of Kenny Erleben's[17] *Seven rules of thumb* reads "Get stacking to work", and since environments are simulated rather than specific physical cases, having a stable system is highly important.

4.3.1. The Impulse

When objects collide, their velocities are changed due to applied force. Force only affects acceleration and acceleration only affects velocity over time, but collisions between rigid bodies occur instantaneously. To solve this problem, Baraff[18] suggests a quantity that produces an immediate change in momentum, the impulse [*J*].

Baraff[18] also writes a formula for the impulse created in a collision. Del Palacio[20] and Hecker[16] explain how this works in a two-dimensional environment, the impulse being a function of the bodies' relative velocity along the normal [**n**] in the collision point (Eq 4.9.).

$$J = \frac{-(1+\varepsilon)\mathbf{v}_{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}\left(\frac{1}{M_A} + \frac{1}{M_B}\right) + \frac{(\mathbf{r}_{A\perp} \cdot \mathbf{n})^2}{I_A} + \frac{(\mathbf{r}_{B\perp} \cdot \mathbf{n})^2}{I_B}}$$

Eq 4.9: The impulse

The impulse is added to the velocities of each body the same way a force is added to the total force/torque acting on it, and a separate impulse is calculated for each collision point. Should the impulse be a negative value, this means the bodies are already separating at the contact point and the impulse is not applied.

4.3.2. Collision Information

Since the denominator of the impulse equation is constant as long as the positions and physical properties of the bodies do not change, it is calculated once and stored in the collision object. Also in the collision objects are pointers to the rigid bodies, the intersection information (from *3.3. Intersection Information*) and the other geometric information that only has to be calculated once.

This collision object is used by the Physics manager in the integration function, but since it can be useful for gameplay purposes to have information about the collision, it is made accessible to the user. The largest applied impulse is also stored in the collision object, this is not used by the manager itself but information about the "strength" of a collision can be useful for example in damage calculations.

4.3.3. Friction and Restitution

The coefficient of restitution [ε] or elasticity, describes the momentum lost along the collision normal, as shown in Eq 4.9. When two rigid bodies collide, the smallest value of the two is used as did Guendelman[15]. When doing resting collisions, a restitution smaller than 0 is used (as described by Guendelman[15]) to make the bodies slow down rather than be repelled or stopped.

The coefficient of friction [μ] describes the loss of momentum along the surface, perpendicular to the collision normal. Since the impulse created by the collision only acts along the surface normal, another impulse needs to be created solely for the purpose of adding friction. The force due to friction can be calculated with Coulomb's friction law (Eq 4.10.).

$$|\mathbf{F}_\mu| = \mu\mathbf{F} \cdot \mathbf{n}$$

Eq 4.10: Coulomb Friction

Impulses are used instead of forces since the change of momentum is instantaneous. The impulse needed to nullify all movement perpendicular to the collision normal is calculated and compared to the friction impulse exerted, using the coefficient for static friction. If the friction impulse is larger, the movement is nullified, otherwise the impulse is computed again using the coefficient of dynamic friction.

There are two problems with using the collision impulse; first, the impulse is dependent on the restitution. Second, the amount of friction added will be dependent on the number of collision iterations. Since these problems are known, applications can be designed around them.

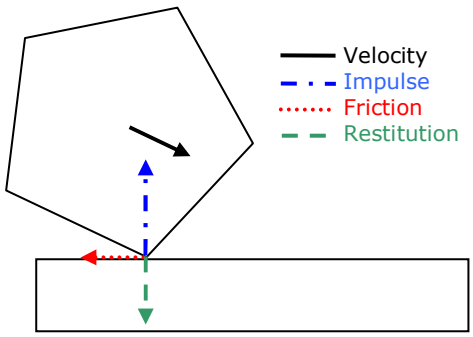


Fig 4.7: The collision (scale incorrect)

Rolling friction is given a simple treatment as well. When a body is rolling across a surface, velocity at the contact point will be zero even if it is moving. The collision response mechanism detects these cases and manipulates the friction impulse with the distance from the body's center to the contact point. This is not intended to give a physically correct solution, only to make sure that rolling bodies eventually comes to a halt.

4.3.4. Separation

Separation is the final step of the collision response. The separation depth could be used, but this value would become invalid if separation is performed several times on a body, so it is recalculated. This is not possible for Compound objects since the penetration depth is not as easily calculated for concave primitives, so the stored penetration depth is used instead. This can make compound objects unstable in multiple collisions, but the shock propagation should keep the penetration to a minimum even in these cases. If one of the bodies has the LOCK_MOVE – flag set, only the other one is moved.

By default, the bodies are not separated completely but are allowed some penetration. This is so that the collisions will be detected and processed in following frames.

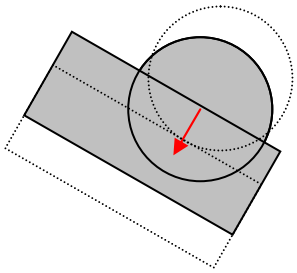


Fig 4.8: Separation

It might seem like a bad idea to move bodies an equal amount rather than to use the mass or volume ratio to determine how much each should move. For example, a concrete block sitting on an edge while being hit by high-velocity ping-pong balls would not be tipped over the edge by the force, but if they were to penetrate the block, separation might push it over the edge. However, since collisions are processed before position is integrated, object separation is only really needed when great forces are pushing bodies into each other, for example in stacks. In these cases, having lighter bodies move more when separated would make them sink into the ground if heavier bodies were to fall on them.

5. Optimization

The methods mentioned so far are more than enough to create a reasonably stable rigid body physics engine. To maintain quality in a real-time simulation, a high number of iterations per second are needed, and preferably without putting too much strain on the CPU since things like graphics and game logic must also be handled. Naturally, this becomes difficult as the amount of bodies rise.

In computer graphics, this is usually done by streamlining the output data and culling things that are not in the current view. While the idea of having a clipping box and using less exact – or none at all – integration methods for objects outside of it is interesting, it will not be treated in this document. Instead, the problem with the costly procedures of identifying possible collision pairs is considered, the broad-phase collision detection.

5.1. Sort and Sweep

Ericsson[14] suggests sorting the rigid bodies after their projection on any or all of the principal axes, and then traversing each axis testing only pairs of objects that are close to each other. The advantage of this solution is that object ordering rarely change much between frames, so sorting becomes relatively cheap while still allowing us to skip collision tests between objects with a large distance to each other.

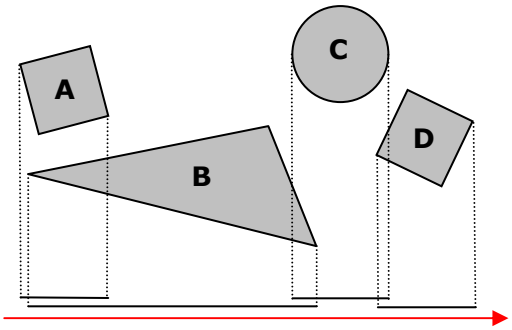


Fig 5.1: Sort and sweep- when body C is reached, A is removed from further testing

To simplify sorting, one of the principal axes is used at all times. Ericsson[14] mentions that in the case where only a single axis is used, this can be the diagonal as a broader projection is possible, but this makes projection more expensive.

In short, the algorithm works by sorting the rigid bodies, then adding each body to a temporary list and processing them, one by one. When a body is processed, it is checked against all other bodies currently in the list and bodies not needed anymore are removed from the list.

The sort and sweep algorithm has a major drawback – it is weak when clusters of objects are handled since it cannot cull a lot of collision pairs in these cases. Also, since slight movement in the bodies can change the order drastically in such situations, sorting becomes more expensive. It is therefore advised to disable the sorting when environments with only a few tight clusters of bodies are handled.

5.1.1. Sorting Axis

When sorting against a single axis only, it is crucial that the best axis is used; otherwise the clustering problem might be encountered unnecessarily. Consider a collection of 10 rigid bodies lying on a flat surface. Sorting against the axis perpendicular to the surface will yield the maximum 45 tests but testing against the surface axis will cull all of them. A simple algorithm for determining the axis with the least clustering is used, as demonstrated in Fig 5.2.

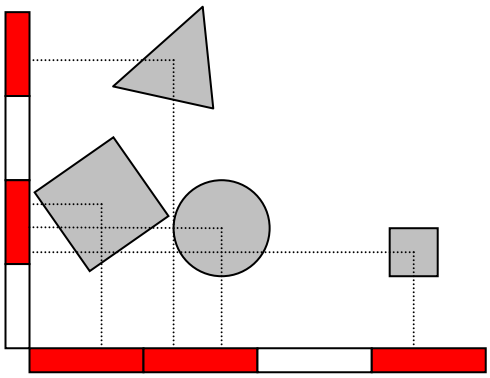


Fig 5.2: The largest spread is along the axis with the most cells tagged

Each axis is divided into a number of cells; each cell in which a body is projected is then tagged. Although this algorithm does not take into account different sizes of bodies, usually, the axis with the most cells tagged will have the largest spread.

The sorting axis is re-evaluated at a constant interval. It might seem like a better idea to do this only when the amount of collision tests rise above a certain limit, but this would put additional strain on the system when objects are clustered in both dimensions, worsening an already existing bottleneck. Since swapping the axis makes the ordering of the object completely off, efficiency might be lost in sorting if only a few bodies are moving, making one axis slightly better one frame and slightly worse the next. This is solved by only swapping the axis if the new one provides a substantial improvement over the old.

Sorting between frames is done with InsertionSort, since the algorithm has a very high efficiency for lists that are almost sorted. The bodies are sorted after their initial projection value along the axis. A MergeSort function is also implemented for use when sorting after axis switch since it handles completely unordered list faster. It might seem like a better idea to simply use QuickSort like in the Shock Propagation (*4.2.2.2. Iterations*), but MergeSort is more suited for linked lists and has a better worst-case efficiency.

5.1.2. Sweeping

Sweeping is done by creating a temporary single-linked list with the bodies, and adding and processing each body in the Physics Manager. When a new body is added, it is tested against each body already in the list. If the projections on the sort axis overlap, the bodies are tested for intersection, otherwise, the old body is removed from the list since no following bodies can intersect it, they having a larger minimal projection value.

5.1.3. Threshold

When adding new bodies to the Physics Manager, a linear search determines their position in the list. Since there is some overhead to the Sort and Sweep algorithm, it is not necessary to use it when there are few bodies in the system, and it can be automatically switched on and off depending on the body count and a threshold value.

The bodies are not sorted when the algorithm is turned on, even though a MergeSort would be able to create order in a random list faster than InsertionSort. The reason for this is that the only time this is really an issue is when sorting is turned on and off frequently, and in these cases the list will probably remain somewhat sorted in between.

5.2. Resting

In a normal system with gravity and somewhat inelastic ground, bodies will sooner or later loose all their momentum and come to rest. Such bodies move very little – if at all – between frames, and a lot of processing can be skipped for them.

Resting bodies in Kodo are tagged with the **RESTING** flag, so a body with resting enabled cannot be used simultaneously in two different Physics Managers. Additionally, the Physics Manager Rigid Body list contains two entry points instead of one, one for active and one for resting bodies. This way, functions that only concern either resting or active bodies can be a lot faster since they do not need to iterate through as many bodies, and do not have to do a flag comparison for each body.

5.2.1. Kinetic Energy

To prevent bodies from stopping in mid-air or settling in weird positions, it is not enough to consider if they are currently immobile, they must be immobile for an extended period of time. This can be achieved either by setting a threshold value and either adding the timesteps of the integration or by simply counting the frames. A large timestep might force bodies into rest that should have stayed active if they had been sampled more times, so counting the frames is a safer choice.

$$W_k = \frac{M\,|\mathbf{v}|^2}{2}$$

Eq 5: Kinetic Energy

At the end of each frame, the kinetic energy [W_k] (Eq 5.) of each body is compared to the energy the body would have gained from the applied forces and torques. If the existing energy is less, the body's rest counter is increased, and if the counter reaches the threshold, the object is tagged as resting and moved from the "active" list to the "resting" list. If, on the other hand, the applied energy is less, the rest counter is zeroed.

The problem with resting is to determine a good threshold value that will not deactivate bodies that are accelerating slowly. Theoretically, it should not have any effect on the stability of the system (practically, it has a slight effect, see *6.1. Future Work*) so the only real drawback to having a high value is that a too high value negates the purpose of the resting concept.

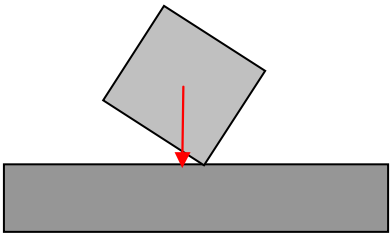


Fig 5.3: The body is only slightly off-balance so it might not accelerate fast enough to be considered active

5.2.2. Collisions and Activation

Bodies can be activated, or "unsettled", manually. When a resting body is unsettled, it might be supporting other bodies resting on it, so it is important that nearby bodies are also unsettled. This is done in a recursive function that performs intersection tests with expanded versions of the rigid body primitives. Since the function changes the list itself, a temporary list of indices is used to trace backwards if the current iterator is activated in a lower level of recursion, but in cases where this is not enough, it is necessary to start over from the beginning of the list. An expensive function, but since objects are activated it does not have to be called very often.

When two active bodies collide with each other, their rest lists are synced to the lowest value of the two to make sure all bodies in contact are inactivated at the same time. If not, they would just activate each other instantly after either becomes deactivated. If an active body collides with an inactive one, the resting body is simply unsettled.

6. Conclusion

This document has provided a somewhat simplified, but complete guide to implementing a simple 2D rigid body physics engine geared specifically for games, by describing the development of the Kodo Physics Engine. Simple Euclidian math was described, as well as primitives and intersection testing. Newtonian physics with rigid body simulation using impulses and the propagation model were described next, and finally, a couple of optimization methods to improve real-time performance.

As Kodo was written for simulation in two dimensions rather than three, a lot of operations were simplified. Some of these are regarding intersection detection, how objects are found to penetrate, but the most important one is the step from six to three DOF, and the simplification of angular velocities when objects are only allowed to rotate around a predefined axis.

The Kodo Physics Engine was developed with the plausibility-before-reality – reasoning used by Guendelman/Bridson/Fedkiw[15]. Making it possible for objects to stack and rest was considered more important than having correct movement of mobile objects; as such, the physics integration order favoured objects that were coming to rest. A stable system working in reasonably constant time was required. Instead of solving collisions at their exact time the entire system was processed only at the end of the timestep, and instead of disallowing penetration; a separating function was created to make the system strive towards a plausible state.

Kodo was created in parts, exposing much of the inner workings to allow users to customize it to their needs without having to re-write the library. The parts in Kodo include the primitive shapes of the rigid bodies, the rigid bodies themselves and the Physics Manager, the environment in which the simulation takes place. The parts are designed to have few dependencies, and the user may decide where to use existing parts and where to write new code.

A lot of research has been done in the field of real-time physics; many different solutions exist to even the simplest of problems. Kodo as a physics engine might become useful in a limited field, but as suggested by Erin Catto[23, 25], if physics is to be a core feature of a game, it is very risky to outsource it. Even in the case of Rigid Bodies using convex primitives, this text has shown that several decisions

have to be made to shape the engine. The aim with Kodo was to produce a general-purpose engine for games but as the game design spectrum is broadened, many games may well benefit from having a different set of priorities in the physics engine. Instead of being a fully useable library, Kodo might be useful together with this document as an introduction to Physical Simulation in real time.

6.1. Future Work

The original intention was to release Kodo to the public under the LGPL license once it was done. As mentioned, research has shown that it is not certain whether Kodo would be very usable as a third-party library so before this is decided, whether this method is overly complicated or not needs to be taken into consideration. Regarding the software itself, there are several planned improvements following, these will hopefully be added in the future.

Physics Manager Interface
The physics manager class needs to be converted into an interface and extended physics managers providing support for force systems need to be written to make creating such systems more streamlined and structured.

Separation Impulses
With Shock Propagation, the system is somewhat stable but separating bodies without adding velocities produces movement that is not recognized by the system. This in turn creates unreliable results in the rest calculations; Catto[25] describes procedures for using impulses for a lot more things than collisions. Separation should not be handled as an external "fix" to the system but should use impulses to force bodies apart; this would also make stacks more stable faster.

Constraints
Enforcing Constraints on Bodies is a fundamental feature of any Physics Engine; otherwise two bodies cannot be connected in any other ways than solid object compounds. This can usually be solved with penalty forces and springs but these perform very poorly with Euler integration. Catto[25] describes a simple method of implementing pin constraints that could probably fit Kodo very well.

Heightmaps
A more specialised feature, but important for games. The heightmap – or "heightfield" – is created in 2 dimensions as an array containing height values of the ground across the simulation area. Since nothing can exist below the heightmap, relatively simple collision tests can be used for complex terrain.

Frustum culling
Briefly mentioned in part *5. Optimization*, some way of only working with a subset of currently "interesting" bodies might prove useful. As it is now, this can only be achieved by having several different Physics Managers.

7. References

1. **on Game Design -** Andrew Rollings/ Ernest Adams, New Riders Publishing 2003

2. **Design Patterns for Interactive Physics -** David Wu/ Richard Hilmer, http://www.pseudointeractive.com/technology-research.php

3. **3D Games – Real-time Rendering and Software Technology Vol 1 -** Alan Watt/ Fabio Policarpo, Addison Wesley 2001

4. **Havok** – http://www.havok.com

5. **Ageia PhysX** – http://www.ageia.com

6. **Open Dynamics Engine** – http://www.ode.org

7. **Tokamak Game Physics SDK** – http://www.tokamakphysics.com

8. **Gish** – Chroniclogic 2005, http://www.chroniclogic.com/index.htm?gish.htm

9. **Ragdoll Kung Fu** – Mark Haley 2005, http://ragdollkungfu.beasts.org

10. **Experminetal Gameplay Project** – http://www.experimentalgameplay.com

11. **Microsoft Visual Studio** – http://www.msdn.com

12. **Borland Command-line Tools** – http://www.borland.com/downloads/download_cbuilder.html

13. **GCC, the GNU Compiler Collection** – http://gcc.gnu.org/

14. **Real-Time Collision Detection** – Christer Ericsson, Morgan Kaufmann 2005

15. **Nonconvex Rigid Bodies with Stacking** – Eran Guendelman/ Robert Bridson/ Ronald Fedkiw, http://graphics.stanford.edu/papers/rigid_bodies-sig03/

16. **Rigid Body Dynamics Information** – Chris Hecker, http://www.d6.com/users/checker/dynamics.htm

17. **Stable, Robust, and Versatile Multibody Dynamics Animation** – Kenny Erleben http://www.diku.dk/~kenny/thesis.pdf

18. **An Introduction to Physically Based Modelling** – Andrew Witkin/ David Baraff/ Michael Kass, http://www.cs.cmu.edu/~baraff/pbm/pbm.html

19. **Basic Collision Detection and Testing** – Raigan Burns/ Mare Sheppard, http://www.harveycartel.org/metanet/tutorials/tutorialA.html

20. **2D Rigid Body Dynamics** – Jaime Del Palacio, http://www.gametutorials.com/gtstore/c-8-free-tutorials.aspx

21. **Simple Intersection Tests for Games** – Miguel Gomez, http://www.gamasutra.com/features/19991018/Gomez_1.htm

22. **A Simple Method for Box-Sphere Intersection Testing** – James Arvo, http://www.ics.uci.edu/~arvo/code/BoxSphereIntersect.c - published in **Graphics Gems Vol 1 –** Academic Press 1990 (http://www.graphicsgems.org)

23. **Iterative Dynamics with Temporal Coherence** – Erin Catto, http://www.continuousphysics.com/ftp/pub/test/physics/papers/IterativeDynamics.pdf

24. **Verlet Integration and Constraints in a Six Degree of Freedom Rigid Body Physics Simulation** – Rick Baltman/ Ron Radeztsky Jr. http://www.only4gurus.com/v3/preview.asp?resource=7245

25. **Fast and Simple Physics using Sequential Impulses** – Erin Catto, http://www.gphysics.com/?page_id=16

26. **Atari Games** - http://www.atari.com/

URLs checked for validity 2006-05-29

Appendix A: Projects

These projects were made using the Kodo Physics Engine. Note that not all of them follow the Kodo code convention. They have all been developed with Microsoft Visual Studio .net 2003, and unlike the Kodo library, they are not guaranteed to compile with other compilers.

KodoGL

This project is not really a project in itself, but a static library used by other projects to link Kodo functionality with OpenGL and Windows. It is used to perform common rendering tasks such as creating and handling a window, loading textures and printing text. It also simplifies rendering of Kodo primitives and rigid bodies.

Destructoooor!

This was the first full project developed with Kodo; it is just a demo showing some of Kodo's functionality by allowing the user to drop primitives in one of four different scenes. The user is also allowed to tweak most of the input parameters regarding the integration, allowing for some testing of different situations. It is not meant as a stress test, the framerate is locked and the integration timestep is constant. The demo shows how shock propagation greatly improves stability even with few iterations. It also shows how penetration depth is an issue due to rising forces and larger stacks, but how this can be countered with more rest- and collision-iterations.

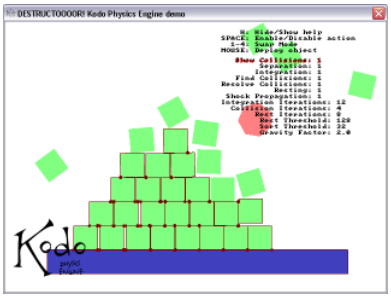


Fig A.1: Destructoooor!

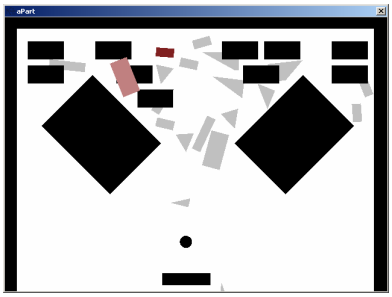


Fig A.2: aPart

aPart

As Kodo was written to enhance gaming experience, aPart was written as an experiment to see how a simple game design could benefit from correct physics. The game chosen was Breakout, originally developed by Steve Wozniak and published by Atari Games[26]. In the original game, a ball travels across the screen bouncing off the top and the sides. On the screen are also a number of rectangular blocks, when hit by the ball these disappear. The player's mission is to keep the ball from bouncing off the bottom of the screen and to destroy all the blocks, to do this the player controls a paddle in the bottom of the screen.

Rollings/Adams[1] mentions that when updating classic games, it is important to keep the things that made the original game good. For this reason, physics-related features are the only things added and the graphics are kept simple. Blocks, when destroyed, will fission in different ways and fall off the screen as opposed to simply disappearing. In addition, certain special blocks have parts that can be caught by the player, some attach to the paddle and some stick for a second and then fire straight up, destroying any blocks immediately above the player. Yet another kind of block fissions into solid parts that function as additional balls.

Technically, the original Breakout was very easy to implement due to the access to occurred collisions, compound objects and flag-based collision detection. The complexity of the code in aPart comes from the generation of block parts and the added features. It should be mentioned that aPart goes against the recommendation of not using the KRB_RESTING – flag for gameplay purposes (see *4.1.6. System Flags*), but since no other forces can affect the bodies than those putting them into unrest, and since resting is not otherwise used in the application, an exception can be made.

Design-wise, aPart adds power-ups, obstructions and eyecandy to the breakout concept, as well as unruly balls and paddles. These add to the gameplay but increase the complexity of the game, the original concept is relatively untouched, with one notable exception. Since everything is controlled by the physics engine, the ball might sometimes come to a halt in mid-air if no external forces are acting on it. To solve this problem, a very small gravitational force is added to it, but this drastically changes the player's way of predicting the ball's path. If aPart was to be extended from an experiment into a full game, a better solution to this problem should be thought of.

Appendix B: Signs and Abbreviations

AABB	Axis-Aligned Bounding Box	*3.1.2.*
DOF	Degrees of Freedom	*4.1.2.*
OBB	Oriented Bounding Box	*3.1.3.*
ODE	Ordinary Differential Equation	*4.2.2.4.*
SAT	Separating Axis Theorem	*3.2.1.*
[a]	Linear Acceleration	*4.1.3.*
[ε]	Coefficient of Restitution	*4.3.3.*
[F]	Force	*4.1.3.*
[I]	Rotational Inertia	*4.1.3.*
[J]	Impulse	*4.3.1.*
[L]	Angular Momentum	*4.1.1.*
[μ]	Coefficient of Friction	*4.3.3.*
[M]	Mass	*4.1.1.*
[n]	Normal	*4.3.1.*
[P]	Linear Momentum	*4.1.1.*
[R]	Orientation	*4.1.1.*
[r]	Local Position	*4.1.1.*
[τ]	Torque	*4.1.3.*
[v]	Linear Velocity	*4.1.1.*
[ω]	Angular Velocity	*4.1.1.*
[$\dot{\omega}$]	Angular Acceleration	*4.1.3.*
[W_k]	Kinetic Energy	*5.2.1.*
[X]	Position	*4.1.1.*