The inner workings of Real-Time Physics Simulation Engines

Björn Laurell
Mälardalens University
Västerås, Sweden
bll05001@student.mdh.se

ABSTRACT

In this article we give an introductory overview of the structure of real-time physics engines and how they work. We cover topics like collision detection and response, and methods for numerically solving force equations. We present and discuss methods such as Runge-Kutta and Euler and their suitability for real time physics engines and common approaches to accurately detecting collisions between objects. We also discuss the implications of real-time constraints often found in interactive physics engines and try to discuss some of the design choices that have to be made when developing an interactive physics engine.

General Terms

Overview, Algorithms, Performance

Keywords

Physics simulations, numerical integration, collision detection, collision response, real-time, physics, real-time constraints

1. INTRODUCTION

In today's modern world of digitalized systems there are a multitude of applications where real-time physics simulations plays a big role, whether it is in vehicle training simulations or realistic 3D games. Though regardless of the application, at the core they all conform to a few basic principles, and still there are a vast number of design choices that has to be made which radically changes the final system. In this overview paper we will delve into and dissect the inner workings of a real time physics engine and illuminate some of the design choices a developer of a real-time physics engine has to face.

1.1 Related work

There exists a wide selection of textbooks on the topic of physics simulations. In this paper we have opted to refer to a specific textbook, *Physics for Game Developers* [2], which focuses on the implementation of physics engines for computer games. However since he does not go into detail about collision detection we have also referred to *Inter Active Computer Graphics - A top-down approach using OpenGL* [1] and *Comparison of Collision Detection Algorithms* [8] which covers the areas of representation of geometry and collision detection respectively.

1.2 Engine overview

According to J. Hecker[6] a physics simulation engine can be divided into 4 distinct subsystems, Contact Detection, Contact Resolution, Force Calculation, and integration of current state. We will delve into each one of these subsystems followed by a short discussion on the impact real-time constraints have on the design of a physics engine.

2. CONTACT DETECTION

The handling of collisions is one of the places where computerized physics simulators differ from theoretical physics. In the real world collisions between objects happens all the time without us having to worry about how the objects know that they have collided, however in a computer objects are merely numbers in memory, they have no physical reality in the sense of real world objects. This means that we have to keep track of them and decide whether 2 objects collide with each other or not (if we did not objects would merely pass through each other undisturbed).

There are many variations on how to implement collision detection algorithms. It all depends on what type of objects your engine should be able to handle, how you represent those objects, and how accurate versus what performance you need to get out of your simulator.

2.1 Spheres

The simplest 3 dimensional shape is the sphere. It can be fully geometrically described using a vector pointing out the center point of the sphere in relation to the systems Origo and a radius which describes the distance between the center point and all points along the outer hull of the sphere.[1] Two spheres are considered having contact with each other if points in the outer hull occupy the same space and is considered intersecting with each other if also points inside the object are overlapping. Both of these events can be considered events that will need to be handled as collisions (depending on application) either as the same type of resulting response or with different responses. (Collision response will

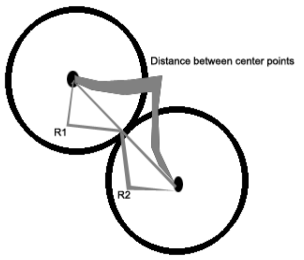

Figure 1: Sphere - Sphere collision

be handled in section 3) When two spheres are just touching each other the distance between the center points are exactly equal to the sum of their respective radii. If they are allowed to move even closer to each other we notice they start to intersect each other. As this happens we also notice the center points are approaching each other, thus the distance between them get smaller and smaller. From this we can deduce that if we have two spheres, S_1 and S_2, with the positions of their center points described by the vectors, p_1 and p_2, and with the radii, r_1 and r_2, they are intersecting when $\|(p_1 + p_2)\| \leq (r_1 + r_2)$ (figure 1. [2]

2.2 Boxes

Another commonly used shape is the box. A box consists of 6 rectangles joined together perpendicular to neighboring rectangles. Also each pair of oppositely located rectangles is congruent with each other. There is several ways to represent a box. One way to represent a box is by having one vector that point the position of one of the corner points and a set of 3 perpendicular vectors representing the directions and lengths of the edges of the various sides of the box. Another way to represent a box is by a set of 4 points that are diagonally opposite from each other. Other ways to represent boxes are by using 6 planes or 3 pairs of parallel planes and one method common in rigid body simulations is to use a center point, 3 half-edge lengths and an orientation matrix denoting the orientation of the local coordinate system.

When it comes to boxes and testing for collisions there are 2 types of boxes to consider, axis-aligned boxes and oriented boxes. In both cases there is several different ways that you could test for collisions, and the 2 types of boxes differ in how you check for collisions. With axis-aligned boxes you assume that all boxes are aligned with a fixed set of axises and all are oriented in the same way along each axis. When it comes to oriented boxes they can all be oriented along arbitrary sets of axises for each box. Because of this collision detection between axis aligned boxes is easier to perform then between oriented boxes. Geometrically 2 boxes intersect if at least one of the faces of each box intersects with a face on the other box. [8] This reduces the problem of intersection down to 36 rectangle - rectangle intersection tests. This break down can be made both for axis aligned boxes and oriented boxes. Common methods for speeding up collision detection for axis aligned boxes is hierarchical spatial hashing and BSP-trees [7].

The basic principle of testing for rectangle - rectangle intersection is to check whether firstly one of the rectangles is intersecting the plane of the other, if they don not, then there's no possibility that they can intersect. Secondly we have to check if the corners of the projection of one rectangle onto the plane of the other is inside the edges of the other. If we can establish both these points then we can conclude that they intersect.

2.3 Arbitrary shapes and polygons

This is all good and well, but most objects we would like to represent in our simulation engine are usually not a box or a sphere. If we would like to use more complex objects like a person or a teapot, simple spheres and boxes are inadequate. This is a problem which can be solved by approximating these shapes using sets of polygons. Polygons are multi edged faces, which can have an arbitrary amount of edges. Given enough polygons of a smaller and smaller size we can approximate arbitrary shapes to arbitrary accuracies. Now however the problem of increased complexity appears. The effort for collision testing shapes that can have any number of edges and faces would grow exponentially in complexity. This can be resolved though by the fact that more complex polygons can be divided into sets of triangles, which is the least complex polygonal shape. This means we will only have to support triangle - triangle collision testing in order to support collision testing for arbitrary shapes. Triangle - triangle intersection testing can be performed following a similar principle as rectangle - rectangle intersection.

2.4 Handling large scenes

As the number of objects in a scene increases the number of intersection tests that have to be performed increases exponentially. Collision detection already is one of the main performance drains in rigid body simulation systems, according to James K. Hahn[5] up to 95% of CPU time is taken up by this. Various methods for handle this complexity and speed up intersection calculations exists. One method is to encase more complex shapes in boxes and/or spheres, so called bounding volumes, and perform collision tests on these simpler shapes. See figure 2 for an example of how a bounding box could work. If the real object is fully enclosed inside the bounding volume for any collision test which the test against the bounding volume reports no collision then the object inside cannot have collided with the object tested. However in cases where 2 bounding volumes has collided the objects inside may not have collided. Thus the common approach is to divide collision testing into 2 phases, a broad collision phase in which only bounding volumes are tested, and a narrow phase where more detailed collision testing is performed on the objects which was reported colliding in the broad phase. The assumption here is that 2 arbitrary objects is more likely to not be colliding then colliding there for you can cull a lot of objects from the more detailed testing.

The choice of whether to use bounding boxes or bounding spheres is one that depends on the general shapes of the real objects we are representing. A box can usually be closer fitted against an oblong object then a sphere can against the same object, thus reducing the amount of falsely reported collisions in the broad phase, however collision testing against boxes is a lot more time consuming then testing

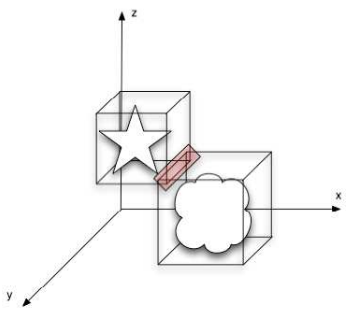

Figure 2: Example bounding box [8]

against spheres.

3. CONTACT RESOLUTION

Once you have determined that a collision has taken place how do you handle it? There are several ways to handle collision response one of the most common is the impulse-momentum principle, which takes its foundation in Newtonian physics, another is a class of methods called penalty methods. The fundamental assumption in the impulse momentum principle is Newton's principle of conservation of momentum. In other word that the sum of the bodies momentum before the impact is equal to the sum of the bodies momentum after the impact. Momentum is the product of an objects velocity and mass. The change in momentum for an object is equal to the applied impulse. A bodys or particle systems motions can be decomposed to the movement of the systems center of gravity and movement around the center of gravity. The movement of the center of gravity corresponds to the objects linear momentum and the rotary motions around its center of gravity corresponds to the objects angular momentum.

The definition of linear and angular impulse is

$$\int_{t_+}^{t_-} F dt = m(v_+ - v_-)$$

$$\int_{t_+}^{t_-} M dt = I(\omega_+ - \omega_-)$$

where F is the impulsive force, M is the impulsive torque, I is the moment of inertia, ω is the angular velocity and subscripted - and + is values immediately before and after impact. This can be rewritten as

$$F = m\frac{v_+ - v-}{t_+ - t_-}$$

$$M = I\frac{W_+ - W_-}{t_+ - t_-}$$

.

The principle of conservation of momentum gives us the following equation: $m_1 v_{1-} + m_2 v_{2-} = m_1 v_{1+} + m_2 v_{2+}$ This assumes that during the exact moment of impact the only force that matters is the impact force, other forces being negligible.

When it comes to collisions there are 2 idealized types of collisions, elastic collisions and plastic collisions. In an elastic collision all kinetic energy is conserved. In a plastic collision all kinetic energy is turned into strain energy which deforms the objects. In reality most collisions are a mix to some degree of both types. In a non-rigid body simulator we have to find some way of modelling this deformation. In a rigid body simulator we instead because our objects do not deform (each point in an object retains their relative position with each other), have to calculate how much of the energy is lost and how it affects the velocities of our objects. This degree of which the collisions are elastic we call the coefficient of restitution, e. $e = \frac{-(v_{1+} - v_{2+})}{(v_{1-} - v_{2-})}$

To derive a formula for calculating the velocities after a collision we have to derive formulas for both the linear impulse and the angular impulse. If we let J be the impulse then:

$$J = m_1(v_{1+} - v_{1-})$$

$$-J = m_2(v_{2+} - v_{2-})$$

$$e = \frac{-(v_{1+} - v_{2+})}{(v_{1-} - v_{2-})}$$

Note that the impulse forces on the objects are equal but opposite of each other, this is in accordance with Newton's third law: Each action has an equal and opposite reaction. Solving for velocities after impact and substituting into the formula for e we will eventually reach:

$$v_{1+} = v_{1-} + \frac{(Jn)}{m_1}$$

$$v_{2+} = v_{2-} + \frac{(-Jn)}{m_2}$$

$$J = -v_r \frac{(e+1)}{(\frac{1}{m_1} + \frac{1}{m_2})}$$

where v_r is the relative velocity along the line of action to the impact and n is the normal to the surfaces of collision. The value of e is determined through empirical studies and varies depending on materials.

If we also factor in angular impulse, we will have to modify our formula. A modified set of equations that takes angular velocities into consideration:

$$v_{1+} = v_{1-} + \frac{Jn}{m_1}$$

$$v_{2+} = v_{2-} + \frac{(-Jn)}{m_2}$$

$$\omega_{1+} = \omega_{1-} + \frac{(r_1 \times Jn)}{I_{cg}}$$

$$\omega_{2+} = \omega_{2-} + \frac{(r_2 \times -Jn)}{I_{cg}}$$

$$J = \frac{-v_t(e+1)}{\frac{1}{m_1} + \frac{1}{m_2} + n * [\frac{(r1 \times n)}{I_1}] \times r_1 + n * [\frac{(r2 \times n)}{I_2}] \times r_2}$$

Where r_1 and r_2 is the vectors from the bodies centers of gravity to the collision point, n is an unit vector along the line of action, ω is angular velocity and I_{cg} is the moment of inertia tensor.

Given that we already know the objects masses, moments of inertia, velocities and angular velocities before impact and that we have set a value for the coefficient of restitution we can now calculate their velocities after the impact. Worth to note is that in this way of modelling impacts we do not have to directly apply a force to the force vector sum instead since we did the assumption that the impact force is the only active force during the exact moment of impact the others being negligible we can directly calculate the change in velocity instead.

4. FORCE CALCULATION

When we are talking about the forces acting on an object we can divide the forces into two fundamental types of forces contact forces, which is applied on contact surfaces between objects, and force fields, forces that works from a distance. Examples of force fields are gravity, drag forces from motions in fluids or air and electromagnetic forces between charged objects. Contact forces occurs when objects interacts with each other or can be problem specific simplified forces applied to simulate more complex energy exchange processes, like a cannon ball getting fired out of a cannon. Exactly what forces and force fields which is used in a simulation is highly problem specific.

Force fields that occurs in a lot of problems involving motion is gravity and air drag. According to classic Newtonian physics gravity is an attractive force between objects with mass. The force between two arbitrary objects is $F = G\frac{m_1 m_2}{r^2}$ where G is the gravitational constant, r is the distance between both objects and m_1 resp. m_2 is the masses of respective bodies. For simulations where you handle objects with a massive difference in mass, for example people walking around on the earth, this formula ca be simplified as $F = mg$ where m is the objects mass and g is the larger objects specific gravity. For most practical purposes g is a vector pointing towards the center of the main body, in the "down" direction.

Air drag is actually not a force field as such according to the above definition, but drag occurs when the air molecules collides with the molecules of the object passing through the air causing friction. In this sense drag is a contact force, however for most practical purposes it is sufficient to model it as a force field working in the opposite direction of the objects movement and with magnitude as a function of the objects surface area in the perpendicular direction of movement and speed. The total drag force an object moving through air is affected by can be calculated with the following formula: $R_t = (\frac{1}{2}(\rho v^2 \cdot A))C_d$ where C_d is the drag coefficient, p is the density of the medium, v is the relative velocity to the medium and A is the characteristic area of the object perpendicular to the flow. The drag coefficient has to be determined experimentally for each material and type of object.

What we are interested in when calculating the forces that applies to each object is the total effect of all the acting forces. If we represent each force, whether it is a contact force or a force field, as a vector we can compute the compound effect of all these forces as the vector sum of all the force vectors. This is repeated for each object in the scene, every program cycle.

5. INTEGRATING MOTION

Now that we know all the forces acting on our objects we want to turn this knowledge into concrete information of where our objects are at each instant in time. Newton's Second law of motion gives us that $F = ma$. We know the masses of our objects and we know the sum of all the forces acting upon them. This means we can calculate the acceleration of our objects as $a = \frac{F}{m}$. What we would like to know though is the position of our objects. So what is the relationship between an objects position and it is acceleration. If we derivate an objects position in regards of time we get the objects velocity. If we further derivate an objects velocity in regards of time we get the acceleration of the object. Thus acceleration is the second time derivative of position. From calculus we know that the integral of the derivate of a function is the function itself. This means the integral of acceleration in regards of time is velocity, and the integral of velocity in regards of time is position.

So far all good, now we face the problem of actually finding a value for the integral though. Generally differential equations are not solvable though analytical means, although for some special cases analytical solutions exist. This means we will have to find some numerical method with which we can approximate the solution. Because our problem can be described as solving 2 first order differential equations in successive order. We can focus our search to only consider methods for solving first order differential equations. Still there exists a multitude of methods in this particular area. They can broadly be categorized as multi-point methods and single-point methods. In multi point-methods you use the information of several know sample points on the solution function to calculate each subsequent point on the function. In single point methods you use the information of one point and the derivate in that point to calculate the next points. Since in our problem the data we have in each time frame is the data from the previous frame and the second derivate of what we are trying compute single point methods better fits our requirements. This way we avoid overly complex initial value problems.

The simplest single point method is the Euler method. The algorithm to calculate V_{n+1} and P_{n+1}:

$$V_{n+1} = V_n + adt$$

$$P_{n+1} = P_n + V_n dt$$

Where $a = \frac{F}{m}$, dt is the difference in time between the current step and previous step, V_n is the velocity in the previous step and P_n is the position in the previous step. This is repeated for each object in each frame. The basics in this method is that in each step you approximate the next point as a dt long step in the direction of the incline at the beginning of the current interval (figure 3). An alternate version of this method exists which uses the derivate at the end of the interval to approximate the position rather then the beginning of the interval, this method is called the Symplethic Euler, Backward Euler or Euler-Cromer method (after its

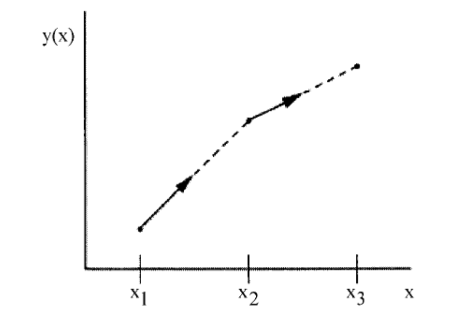

Figure 3: Eulers method Source: [3]

inventor). The formula for Euler-Cromer is:

$$V_{n+1} = V_n + adt$$

$$P_{n+1} = P_n + V_{n+1}dt$$

These algorithms are fairly inaccurate for all but the simplest functions, we ask ourselves if we can improve the accuracy by decreasing the step size. As we decrease the step size we do note that our approximation comes closer and closer to the actual function. However this is where one limitation of the internal representation in the computers we use to run our simulations on. In real world physics and mathematics our variables is calculates using real-valued numbers. However our digital computers are unable to cope with these numbers with potentially infinite decimal expansion. Instead real-valued numbers are approximated using fixed-length floating point numbers. However when using these approximative data types there will be small rounding errors in every floating point operation, which will make our stored data deviate slightly then if we used true real valued numbers. As these errors accumulates for each floating point operation we see that as we increase the step size we increase the number of operations required to calculate behavior over the same interval. This causes the accumulating rounding errors to grow in magnitude as the step size decrease until they are no longer negligible. This means we do not always get increased accuracy with smaller step sizes. Thus for some cases we will need to find other methods if we want to increase accuracy.

The theoretical foundation of the above mentioned methods is the expansion of the Taylor Series. These states that the behaviors of a function at a point $n + 1$ can be known from the function at the point n and its derivate at that point. The Taylor Series expand to an infinite number of turns. As the number of expansions approaches infinity, the expansion approaches the exact solution. The terms of the Taylor Expansion is ever decreasing, which means we can approximate the function to any accuracy by accounting for any number of terms. The terms we do not account for we call the truncation term. Since we are not taking the truncation term into consideration in our solution we do not get the exact solution, only an approximation. This discrepancy forms what we call the truncation error, its magnitude is on the scale of the unaccounted terms. In the case of Euler's method we only take the first derivate into account, hence

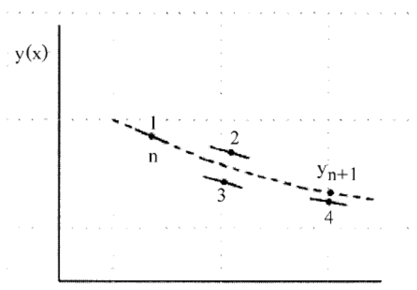

Figure 4: 4th order Runge-Kutta method Source: [3]

we call Euler a first order method. From the Taylor Series we can derivate a large number of algorithms which allows us to achieve increased accuracy at the cost of increased computational complexity. The Runge-Kutta class of methods are examples of these higher-order methods. The classical Runge-Kutta method, which is a 4th order method, can be written as follows:

$$k_1 = dx \cdot y(x, y)$$

$$k_2 = dx \cdot y(x + \frac{dx}{2}, y + \frac{k_1}{2})$$

$$k_3 = dx \cdot y(x + \frac{dx}{2}, y + \frac{k_2}{2})$$

$$k_4 = dx \cdot y(x + dx, y + k_3)$$

$$y(x + dx) = y(x) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

What you do is calculate the derivate at the start of the interval, calculates the slope at the mid point using the slope at the start of the interval to step, calculates the slope at the mid point again stepping from the start point but with the slop of the mid point, and then uses the final slope of the mid point to step to the end point (figure 4). After that you step with a weighted averaged slope to calculate the final approximation. In the classical Runge-Kutta the slopes at the mid points are given a higher relevance then the end points, by changing the weights and the number of mid points can get variations of methods belonging to the Runge-Kutta class of methods. Other such methods are Heun's method, Fehlberg's method and the Dormand-Prince method. The Euler methods can also be seen as first order Runge-Kutta methods.

As seen these methods can offer greater accuracy, however they have a higher degree of computational complexity. Within the context of computers this means they will have a longer execution time.

6. IMPACT OF REAL-TIME CONSTRAINTS

At first we would think that more accurate is always better, but within the context of real-time simulations that is not always the case. In real-time applications there are the added constraint of time. With time we both mean the actual execution time a cycle can take, and response time of the system. When it comes to the most common applications of real-time physics engines, interactive simulators and computer games, the time constraints imposed on the physics engine are usually soft in the sense of that a missed deadline is not fatal for the system. However, response time and frame rate (number of generated images per second, usually one image per program cycle) is very important. When it comes to frame rate modern computer screens have refresh rates varying between 60Hz to 100Hz, applications can run with anything between 10-100 frames per second (fps for short) generated, however anything below 25 frames per second is generally believed (depending on the application) to be perceived as sluggish and stuttering. This means we in most cases want our system to be able to go through a full loop through the program, all the way from resolving contacts, calculating forces and integrating motions to handling user input and generating elaborate 3d images of our simulation taking no longer time then about 40 milliseconds per cycle, while also providing fast response to user inputs. Both collision detection and 3d graphics generation do usually take up a lot of CPU time and since CPU time per cycle is limited that means there is less time available for the other subsystems. In most modern simulations and computer games where there exists high demand of physical realism there is also high demands on graphical realism, often graphical realism is given priority over physical realism. Given that the systems resources is limited developers have to make compromises somewhere. Usually it is more accepted to compromise on physical accuracy then graphical accuracy (dependent on the application). When it comes to games people are generally more likely to accept inaccuracies in physical behavior assuming errors are systematic enough then decreased graphical detail or loss in performance. This has a consequence that when it comes to interactive physics simulations greater physical accuracy is not always the main concern when designing a system. Most of the time you just want it to be accurate enough.

When it comes to collisions developers have to balance accuracy in detecting collisions against accuracy in performing physically correct responses. Garcia *et al.* [4] does in their article discuss that in some cases non-realistic, "cartoonish responses to be more visually appealing then physically correct responses. When it comes to the choice of integrator method game developers usually chooses the simplest algorithm that achieves their requirements for performance and accuracy. A common choices are the backward Euler method, or the classic Runge-Kutta method when increased accuracy is wanted. More accurate methods exist though then the evaluation of whether the increased accuracy is important for the game experience, since it comes at a significant additional cost. This is in contrast to the more scientific applications of physics simulations, which might or might not be real time applications, where increased accuracy is always wanted and where time and cost is less of an issue. In the case of real-time simulations for research purposes graphical detail is usually more likely to be compromised then accuracy, since accurate data and calculations is usually more important then the visual presentation of said data.

7. DISCUSSION AND CONCLUSIONS

Interactive real-time physics simulation engines is an area of engineering which still have a high degree of untapped potential, not just only for the computer game and entertainment industries, but also for more traditional production industries as an aid for better understanding physically complex systems. The hands-on approach which becomes possible with interactive systems enables us to understand complex physical processes to a degree which is not possible with more static approaches. Future development in the area is likely to become more cross-discipline as people will try to find more ways to improve upon interactive physics systems and bring in more expertise on how to model physical systems within each problem domain.

Physics engines can be viewed as tools which help us understand physical systems and enable us to explore the complexities of such systems. However as in todays system we still have to make compromises and simplifying assumptions to be able to simulate and render in real time, this has a consequence that engines specialize in a particular area depending on which assumptions are made. For future work in this field one possible aim could be to develop general purpose engines which performs equally well in all types of simulations, this will become more plausible as computer power grows. Also the possibility of performing these calculations in hardware rather then software should speed up the progress in the field.

8. REFERENCES

[1] Edward Angel. *Inter Active Computer Graphics - A top-down approach using OpenGL*. Addison-Wesley, 4th edition edition, 2006.

[2] Donald M. Bourg. *Physics for Game Developers.* O'Reilly Media, Inc., first edition edition, 2002.

[3] Thomas P. Engel. Computer simulation techniques. *Journal of Clinical Monitoring and Computing*, Volume 17:3–9, 2002.

[4] Marcos Garcia, John Dingliana, and Carol O'Sullivan. Perceptual evaluation of cartoon physics: accuracy, attention, appeal. In *APGV '08: Proceedings of the 5th symposium on Applied perception in graphics and visualization*, pages 107–114, New York, NY, USA, 2008. ACM.

[5] James K. Hahn. Realistic animation of rigid bodies. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 299–308, New York, NY, USA, 1988. ACM.

[6] Chris Hecker. Physics in computer games. *Commun. ACM*, 43(7):34–39, 2000.

[7] Samuel Ranta-Eskola. Binary space partitioning trees and polygon removal in real time 3d rendering. Master's thesis, Uppsala University, January 2001.

[8] W Anthony Young. Comparison of collision detection algorithms. tonyyoung.ca, 2004.