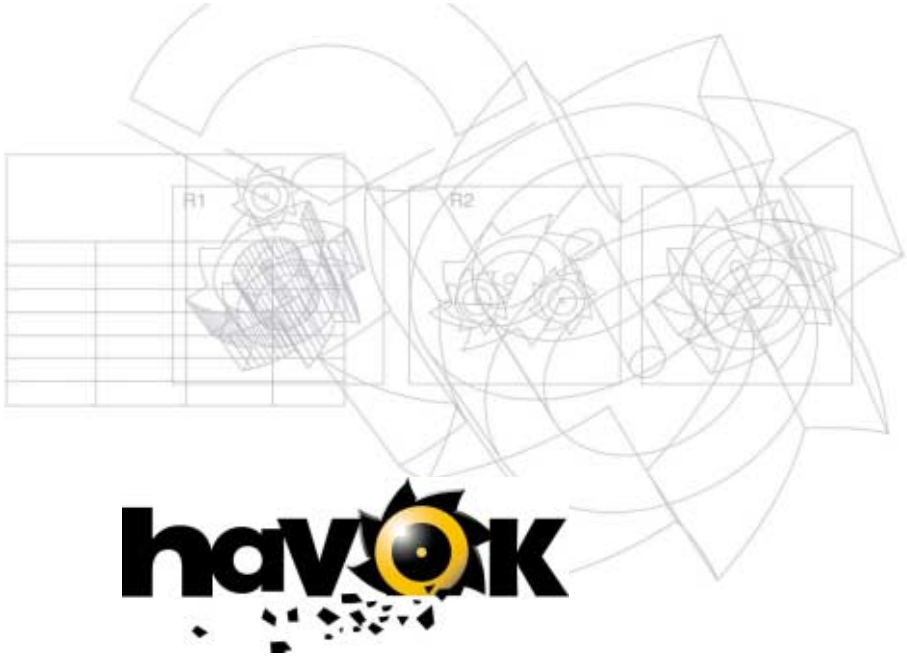

Physics Primer

Version 1.0

Table of Contents

Introduction

Welcome to the Havok physics primer. This document is designed to give a broad overview of physical simulation and specific details about how Havok technology performs these simulations. We will not refer to particular Havok products, but rather aim to give a general understanding of the terminology, methodology and behavior associated with the Havok physics engine.

Physical simulation

Physical simulation is certainly not a new phenomenon. Computers have been used to simulate ballistic motion (i.e. rocket trajectories for military purposes) since the 2nd world war; one of the world's first computers, Collossus, was built primarily for this purpose. Finite element (FE) and computational fluid dynamics (CFD) methods have been employed for decades to aid in industrial design, to simulate anything from vehicle crash test performance to airflow over semiconductors. These last 2 methods are typically very expensive in terms of the CPU resources required – they are concerned with accuracy and as such the calculations need to be performed with a high level of detail.

Both methods employ some form of grid which breaks up the object / scene being simulated into chunks or *elements* (see Figure 1) and performs calculations at the element level (therefore the greater the number of elements the more accurate the simulation, but the more calculations required.)

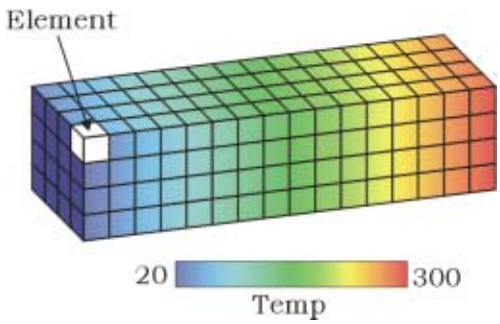

Figure 1: A finite element based temperature simulation, with an individual element shown.

Most of these high-end simulations employ some form of *boundary condition*, or information regarding the bounds of the simulation in order to take into account the rest of the scene around the object being simulated. For example, we might want to examine the result of an impact on the side wing of a car. Rather than model the entire car, we model only the wing itself and set up boundary conditions that capture the fact that the wing is actually attached to other semi-rigid parts of the car –the wing is simulated in isolation (simply because of the expense of simulating the whole car at the level of accuracy required.) In effect we are not interested in the entire scene or system (i.e. the car) but only in the object itself (the wing) and are prepared to make large

assumptions about the rest of the scene in order to get good accuracy for the object simulation.

At another level we might want to see how interacting entities behave to give system wide behavior – this is *discrete event simulation* where the aggregate behavior of a system of events over a period of time is of interest. For example, you might construct a discrete event simulation of a bank teller (see Figure 2) and the length of queue that forms based on certain conditions (average time spent by customer, speed of bank teller etc.)

Figure 2: discrete event simulation of a bank teller and customers

It's of no interest in this case whether the individual customers bang into each other, or trip over the queue guide ropes – we are only interested in figuring out the average length of the queue and the average throughput of the bank. We talk of averages here, because we're dealing with statistical behavior rather than individual behavior (with a good model with good assumptions we can predict the wait time for a customer within a statistic margin of error, but would never be able to predict exactly what would happen to the customer; who they would meet in the bank or whether they would slip on a banana skin.)

The continuum of simulation

This leads us to a sort of continuum of physical simulation; at one end we have the highly accurate but highly localized finite element based simulation and at the other we have discrete event simulation, where the behavior of individual entities is not of interest, but rather the behavior of the system as a whole.

Havok physics simulation is somewhere in the middle of these; we are not really concerned with the specific and accurate behavior of an object or sub-part of an assembly although we do want to know how each object interacts and behaves (but we might not be interested in the pressure at a particular point inside the object or the heat distribution through the object). At the other end of the spectrum, we're not really looking for aggregate system-wide behavior without caring about the behavior of the individual elements – it's these elements that we're interested in! With Havok, the entire scene is modeled; we are interested in the behaviors of collections of physical objects of various shapes and substances with a view to creating *immersive* and interesting environments.

Realism vs. Believability

Havok's core aim is to provide a simulation that **appears** realistic. In many cases we've had to make assumptions and take short cuts in order to simulate

the scene in as fast a time as possible – but these short cuts have always attempted to trade off accuracy and not believability. One of our allies in this is *chaos*! The world around us is inherently chaotic – we're all familiar with the butterfly effect (a butterfly in Canada flaps its wings and causes, by an unlikely chain of events, a typhoon in India). In the case of physics simulation, we are dealing with such a large number of parameters (positions, orientations, forces, velocities etc.) for a large number of objects that even the slightest change in starting condition can yield enormous differences in the resulting simulation. This is mostly the reason why animations of physical systems (like clothing, wind, smashing objects) rarely look realistic unless a) the animator is particularly skilled and has lots of time on their hands and b) a different animation is used each time the effect is required. In Figure 3 below, a wall is struck from 3 different locations. If, in a game, a single animation were used each time, the impact position would have no effect on the destruction sequence, giving unrealistic and unbelievable behavior.

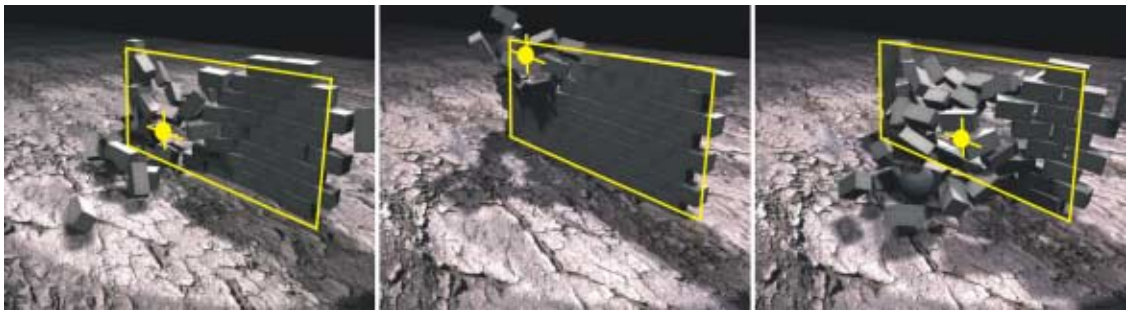

Figure 3: chaotic behavior of a smashing wall impacted at different locations

An enormous amount of time is dedicated to mimicking chaos in animation playback, particularly for real-time systems where the animations may be played many times (we are very unforgiving of looped or repeated animations.) In a game, if a character falls in exactly the same way each time it is killed, our belief that we are playing in a real environment (albeit as fantastical as the game scenario dictates) is negatively affected. With a physics engine, you get this expected chaotic behavior by default and this is the primary reason for using this technology.

Scale

Physics, as a body of knowledge, is enormous. What we are concerned with here would more accurately be described as a *mechanical simulation of the interactions of objects at real world scales*. We are dealing with Newtonian mechanics, that is, the well understood laws of motion, popularized by Sir Isaac Newton, that describe the behavior of objects under the influences of other objects and external forces. Since then we've discovered that these laws break down at really small (i.e. subatomic) and really large (i.e. planetary) scales.

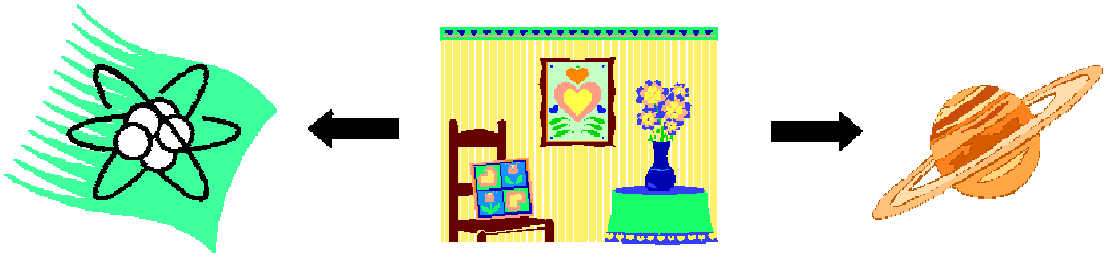

Figure 4: Havok operates at a "real world" scale and is inappropriate for both sub-atomic and galactic / planetary scales!

New physics systems have been devised to work with these scales (e.g. relativistic and quantum), but these are way beyond the scope of the Havok physics engine. As indicated in Figure 4 Havok works at the scale of objects we interact with on a daily basis (chairs, cars, buildings, footballs). By default the Havok engine works in units of meters and kilograms.

One of the most common mistakes we see people make is to start by creating a box 100×100×100 units (meters remember) and wondering why it takes so long to fall. A box of this size (basically an aircraft hanger) when viewed at a distance sufficient to be able to see the entire box (say 1km away) will appear to fall at the same speed as an aircraft hanger dropped from a height and viewed 1 kilometer away. Slowly.

Look at it this way: gravity at the Earth's surface is approximately 10 meters per second per second, written 10 m/s^2. Therefore for every second, objects increase their speed by 10 meters per second, or 10 m/s. So we have the following speed progression:

Time Elapsed	Speed	Distance Traveled	Acceleration
0 seconds	0 m/s	0 meters	10 m/s^2
1 second	10 m/s	5 meters	10 m/s^2
2 seconds	20 m/s	20 meters	10 m/s^2
3 seconds	30 m/s	45 meters	10 m/s^2
4 seconds	40 m/s	80 meters	10 m/s^2

Table 1: calculating the speed of an object falling under gravity

How was this calculated? At the very start the object is stationary, and therefore its speed is 0 m/s. After a period of 1 second, accelerating at a constant 10m/s^2 it will have reached a speed of 10 m/s. But, its average speed over that period was 5 m/s (i.e. start speed = 0 and end speed = 10). Therefore the distance traveled during the 1-second interval was 5m. In fact the true formula for distance traveled d given acceleration a and starting speed v after time t has elapsed is:

$$d = vt + \tfrac{1}{2}at^2 \quad \text{and if } t = 1 \text{ second then} \quad d = v + \tfrac{1}{2}a$$

So you can see that even after 4 seconds has elapsed the large box (aircraft hanger) has not even fallen a distance equal to its own height, so will appear to be moving slowly. It is very important to maintain a sense of scale at all times when working with the physics engine.

Changing Scale

In the previous section it was stated that the Havok engine works at a scale of meters (and kilograms for weight measurements). This is not strictly accurate! In fact, Havok does not care what the units used are, it only cares about the numbers. You just need to be very careful to be consistent. So if working in meters then make sure that gravity is set to a value that is in meters (if you want Earth-like gravity then use 9.8 m/s^2).

If you prefer to work with inches, then you need to be aware that ALL quantities specified must be in inches. So for example, if you create a cube of size 100 units (as in the previous example), but don't specify gravity in inches, the cube will still fall at the same speed (even though you now think of it as being about 7 feet in length on each side). If gravity is set to 9.8 units, the physics engine will be effectively simulating a gravitational pull of 9.8 inches per second per second (less even than the moon!). Look at it this way:

❶ cube is 100×100×100 meters, gravity is 9.8 m/s^2, cube falls slowly.
❷ you switch to thinking in inches, cube is now 100×100×100 inches, but nothing has actually changed, because gravity is now 9.8 $inches/s^2$
❸ you convert gravity to inches: gravity set to 386 $inches/s^2$, now the cube falls at the expected speed

[**note**: 1 inch = 0.0254 meters; gravity = 9.8 / 0.0254 = 386 $inches/s^2$]

It is all relative: the physics engine works with dimensionless units at all times (does not care whether it is inches, meters or miles). It is up to you to remain consistent and convert values to the correct units as appropriate. To confuse matters though, the Havok engine has been designed to be most accurate when dealing with numbers as close in magnitude to 1 as possible (i.e. values like 10000000 are bad as are values like 0.0000001). Therefore for real world scenes when creating objects of 1×1×1 size it is more useful to be working in meters than centimeters or kilometers (or inches / miles) in that you will most often be simulating objects larger than sugar cubes and smaller than football fields. It is for this reason that we say the physics engine works with meter scales by default.

ⓘ Try to be very careful with scale, particularly when using 3D modelers. Often modelers will have their own mechanisms for displaying units in dialog boxes (e.g. 3ds max allows you to specify the units being used and automatically converts all values displayed to those units – however internally it always works in inches, including when exporting geometry.)

What does a physics engine do?

A physics engine like Havok has 3 basic tasks to perform:

1. **Collision detection**: track the movements of all the objects in the scene and detect when any of them have collided.
2. **Update system**: determine an appropriate response for objects that have collided by resolving the collision according to the object properties and for all other (non-colliding) objects update them according to the forces acting on them.
3. **Interface with display**: once the new positions of all objects have been determined we usually need to display them to the physics engine will report the updates to the 3D display system.

Note: a physics engine knows and cares nothing about how the objects it is simulating are displayed. It simulates the motion and interaction of these objects based on a physical (not graphical) description of the objects, and this information may be used to generate a display that "tracks" the simulation. This will be covered in a little more detail later in the section covering *proxy* objects.

Given that we are talking about simulating a continuously evolving state (i.e. objects are moving and colliding and reacting all the time in general) we need to map this to a series of snap shots in order to generate an animation for display. Typically for games we are interested in knowing the state of the world 60 times a second (this is how frequently many graphics systems redraw the screen). For movies we might be interested in 25 frames per second. What this really means is that the physics engine must be capable of evolving the world by $1/60^{th}$ of a second (or $1/25^{th}$ of a second for movies) knowing the state of all the objects at the start of this time interval and knowing the external forces acting on these objects. As an example we'll look at the simple case of a canon ball and we'll assume we're interested in animation at 60Hz (Hz = cycles or frames per second).

Simulating a canon ball

Let's forget about collisions for now, and consider only the simulation of a canon ball immediately after it has been fired from the canon. We know the ball's position (and orientation, but we'll ignore this for now), its speed and acceleration, we know its weight and we assume we know the state of the environment (i.e. air resistance, wind force, gravity). Armed with this knowledge we can start to make predictions using Havok.

Figure 5: expect ballistic motion of a canon ball is a parabolic arc

Figure 5 illustrates what we would like to achieve. Over a period of time the canon ball's rate of ascent should slow due to gravity, and it should eventually fall to the ground having traveled through a classic parabolic arc (assuming no air resistance).

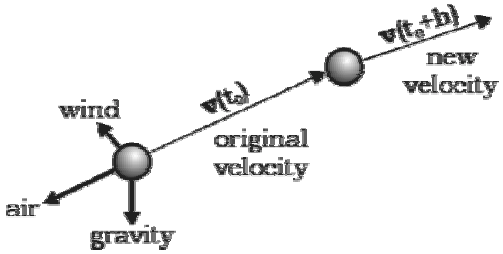

Figure 6: given an initial condition and knowing the forces acting we can estimate the new state of a body in motion

At a given point in time we can examine the state of the ball (its speed *v* and acceleration *a*) and knowing the external forces acting on it we can make **a guess** as to its change in position after a period of time has elapsed (call this period *h* seconds), as shown in Figure 6. This guess is a combination of a number of factors:

a) We assume that Newton's laws of motion govern the motion of the ball.
b) We assume that in the time period *h* all the external forces acting on the ball are constant (so air resistance and wind and gravity do not change during this time.)
c) We assume that the math we use to calculate the new position is accurate.

In general a) is usually a good assumption (except at relativistic or quantum scales which we can assume should be handled by other systems). b) and c) however cause problems and are closely linked to the time period *h* over which we're performing the calculations. We'll now examine the effect of the size of this time period on the accuracy of the simulation.

Time steps

In general, the forces acting on an object are rarely truly constant (gravity is pretty close to being constant all the time but most other forces like wind, air resistance etc. are not). So taking the canon ball example, imagine there was a windy layer in the atmosphere that the canon ball passes through as shown in Figure 7.

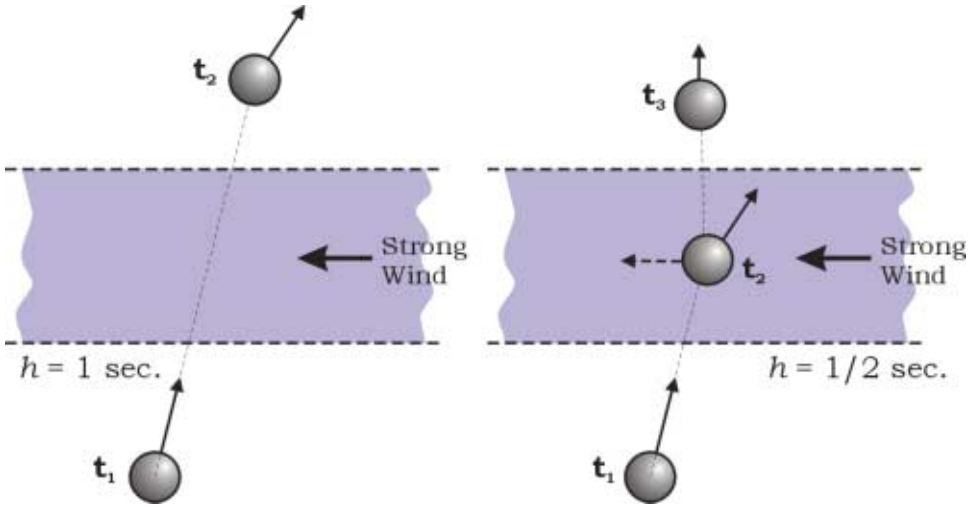

Figure 7: effects of differing time steps on simulation outcome, the big problem being the assumption of constant force acting on the object during the time period

In the simulation on the left we assume we're taking steps of 1 second (which is really pretty big for a physics simulation, but used here to illustrate the point). We know all the forces acting on the ball at time t_1 so we use some math to predict the new position and velocity at time t_2 after 1 second has elapsed. During this period we will have assumed that the wind force acting on the ball was constant. In this example, we'll calculate the new position, which will be at a height above the region of high wind (so we'll effectively have missed the windy bit by taking too large a jump). In the second example on the right, we're using time steps of ½ seconds. In this case after determining the new position at time t_2 we find the ball in the middle of the windy region. This region causes a large wind force to act on the ball which will taken into account during the next time step, at which point we re-evaluate the math and determine a new position for the ball at time t_3 which is different from the position determined in the simulation on the left (i.e. the wind has blown the ball to the left a bit and has reduced the velocity of the ball), *even though the same amount of time has been simulated in each case*.

In general, the smaller the time step taken, the more accurate the result at the end of the time step – so if you want to step forward in time by a large time step t it is better to split this into n steps of a smaller time interval t / n.

This is also true of the math. As the simulation becomes more complex the math required to calculate the new positions and velocities of objects in a simulation also becomes more complex, and as a result the guesses produced by the math give less and less accurate results.

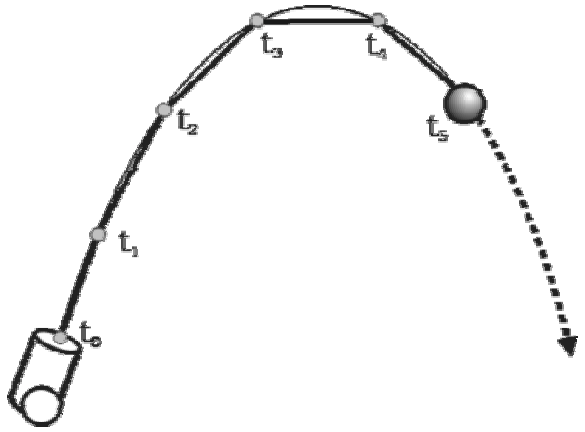

Figure 8: evolution of a physics system captured in a series of "snap shots"

So the principle is to take small time steps, evaluate all the forces acting on the objects, determine the new positions and velocities (and other parameters) of the objects at the end of the time steps and then start over. What we end up with is a series of snapshots of the state of the system as it evolves as shown in Figure 8.

Integrators

As mentioned earlier, the math becomes less accurate as larger time steps are being used. The physics engine is implementing a fast *numerical integration* of a series of *differential equations* describing the motion of objects. An integrator is an algorithm that attempts to estimate the new state of a variable or parameter (e.g. position) knowing information like the rate of change of the parameter (e.g. velocity). There are various different integrators available. They vary in CPU load and accuracy of result. The following table gives an overview of the integrators provided with the Havok engine:

Integrator	CPU load	Accuracy
Euler	low	low
Midpoint	medium	medium
Runga Kutta (RK45)	high	high
Back Euler	medium	medium-high

Table 2: integrators and their properties – faster usually means less accurate

As can be seen the higher the accuracy required the more CPU power required. We've found that for movie production you are better off with an accurate integrator (like RK45) and taking small time steps, but for real-time performance you should usually start with Euler, the fastest but least accurate, and only move to a more accurate integrator if you are not happy with the accuracy. Later we will talk more about integrators and in particular their effect on stability in constrained systems.

The Simulation Loop

We'll examine now the structure of the physical simulation (shown in Figure 9) and how we integrate this with a 3D display.

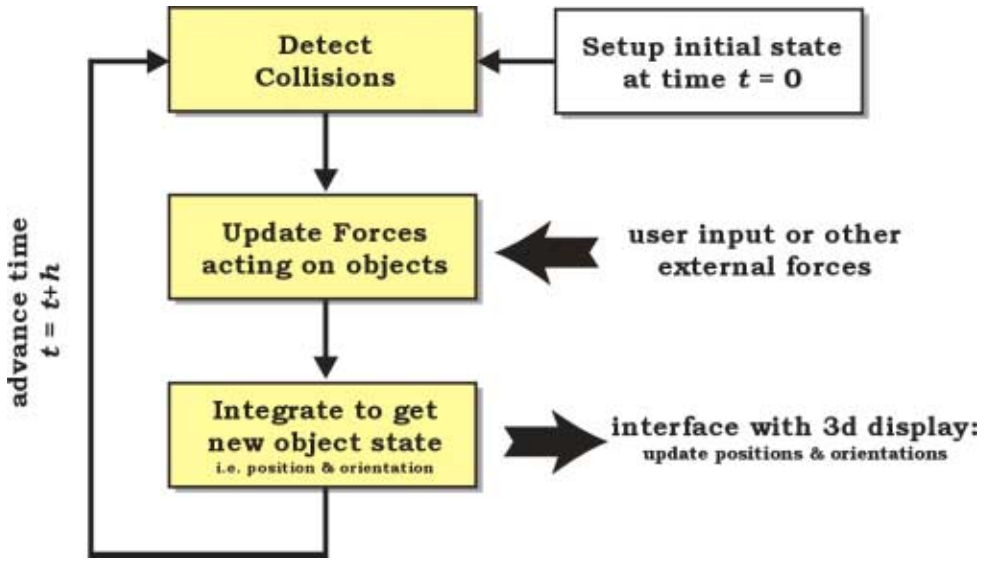

Figure 9: the structure of a physics simulation system

Having setup the initial conditions for a given scene, we begin the main simulation loop which basically steps through 3 phases:

a) **Detect Collisions**: at each step we need to determine which objects have collided. These will result in new collision forces and friction forces being introduced into the system.
b) We then **update all the forces** acting on the objects either as a result of collision detection or as a result of input from outside the simulation (here is where we would add input from a user in a real-time game e.g. the user presses the accelerate key should cause the vehicle being driven to accelerate).
c) Having accumulated all the forces we then, using the selected integrator, **determine the new state of the objects** (position, orientation, velocity, acceleration etc.). This information is then used to update the 3D display.
d) **Advance time** by the step size *h* and determine if these new positions have resulted in collisions between any of the objects.

This assumes that at each step in the simulation we actually want to update the display. The next section deals with what happens when this is not the case.

Sub Steps

Assume we absolutely need to update the display once every $1/60^{th}$ of a second (i.e. we are either playing a real-time game that refreshes the screen at 60Hz. or

we are creating a movie to be played back at 60 frames per second.) Ignoring the load on the CPU, this effectively means that we want to step the physics engine at intervals of 1/60th of a second. In many cases this does not present a problem, but if, hypothetically, the accuracy of the simulation was not sufficient (remember: smaller time steps mean better accuracy) then we'd like to decrease the time step even further, let's say to 1/120th of a second. But this would mean we generate twice the number of images we are interested in, which is wasteful. To get around this, the Havok engine allows you to specify the number of sub-steps to take.

The sub step parameter specifies the number of steps the physics engine takes *before updating the 3D display*. This gives control over the granularity of the physics simulation independent of the display update frequency. So if sub steps = 0 then no physics steps are taken, with sub steps = 1, a single simulation step is used for each update to the 3D display. With sub steps = 2, 2 physics steps are taken and then the display updated.

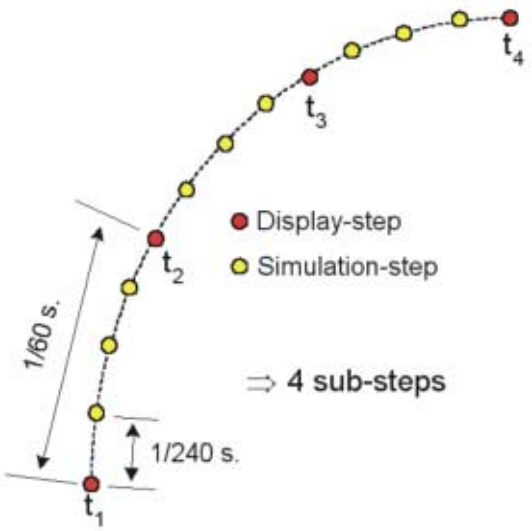

Figure 10: sub steps allow simulation frequency to be decoupled from display frequency

In Figure 10 we have specified that the physics simulation should step at intervals of 1/240th of a second, but that we only update the display once every 1/60th of a second. This has been achieved by instructing the physics engine to employ 4 sub steps. Therefore for each 4 steps we update the display only once. By setting the number of sub steps we can control the accuracy of the physical simulation independent of the display.

Lower Limit on CPU

An unfortunate but unavoidable side effect of using a physics simulation is that there is a definite lower limit on the CPU time you can give to the physics. In contrast, there's no real lower limit on the CPU time given to graphical display (let's assume you don't have hardware acceleration). As you decrease the CPU

allocation to the display you can simply draw fewer polygons, or remove fogging, or turn off lighting, but at no time will the display actually "break".

Physics is different. A physical simulation must be maintained at a stable state (given that each simulation result depends completely on the previous simulation step). If, in one step, we produce a very inaccurate result, then the next step is likely to be even more inaccurate, and you end up in a spiral of decreasing accuracy until eventually the results are garbage (the simulation is said to have "exploded"). Therefore you can't simply reduce the number of objects or turn off friction to compensate for a reduction in the available CPU resources – you simply have to ensure that your scene can be simulated stably even with the lowest expected CPU bandwidth.

Energy Management

One of the major factors determining the load on the CPU in a physical simulation is the number of objects that are active or moving i.e. being physically simulated. In a typical scene a large number of objects are not actually moving at all, and in theory could be ignored until interacted with. Energy management is concerned with determining which objects in a scene are not doing very much and removing these from the physical simulation (known as *turning the object off* or *deactivating* the object) until such time as they begin to move again. The important elements of energy management are:

- When should an object be turned off?
- When should an object be turned back on?

Both questions above are tricky to provide a general answer for and typically the correct answer is highly context dependent. Usually objects are deactivated when they haven't moved much recently and are reactivated when hit by other moving objects. However, temporarily removing objects from the simulation remains the best single way to reduce the load on the CPU and it is well worth experimenting with the parameters provided by the Havok engine for automated object deactivation.

Collision Detection

This is possibly the most crucial part of any physics engine. Collision detection typically accounts for over 90% of the CPU time required for a physical simulation. Given that we are interested in large numbers of interacting objects we have a potential explosion in the numbers of collision tests we need to do. At a worst case, with n objects in a scene we need to guarantee that every possible pair (sometimes called a *collision pair*) is checked. This requires $n(n - 1)/2$ tests (for each object n we test with every other object but not itself, $(n – 1)$, giving $n(n - 1)$ tests, but given that a test for A colliding with B is the same as one for B colliding with A we divide by 2). So for 4 objects we need to do 6 tests; for 100 objects we do 4950 tests. This gets expensive pretty quickly. Given that the physics engine needs some detailed information about each collision in order to be able to resolve it correctly, the collision tests themselves are expensive.

There are a number of ways to speed up this process:

a) Reduce the number of collisions that require detailed collision results to be generated (i.e. use a simpler collision test first).
b) Reduce the complexity of the objects being tested for collisions.
c) Reduce the number of objects.

Obviously always try to achieve c) first. The number of objects active in a given scene is the primary source of CPU load. If objects can be removed this speeds up the simulation. In the case of b), the difficulty in providing detailed collision results is directly associated with the complexity of the objects themselves. We'll address this in the next section. Firstly we'll deal with how the Havok system attempts to achieve a), a reduction in the number of complex collisions.

Multiphase Collision Testing

The Havok system employs a series of collision tests, each getting progressively more complex, but after each test we eliminate as many objects from the test process before moving to the next level. Assume we have a simple test A and a complex test B. Test A is very fast to run but is not very accurate (but as long as it is conservative this is not a problem – a conservative test may return TRUE i.e. there is a collision, when in fact there isn't but will never return FALSE i.e. there is no collision when in fact there is). Test B is slow to run but highly accurate. The process involves:

- First test all objects with test A eliminating those that definitely do not collide, but not necessarily eliminating all non-colliding pairs.
- Then test with test B which will take the reduced number of collision pairs and perform the complex collision test on these.

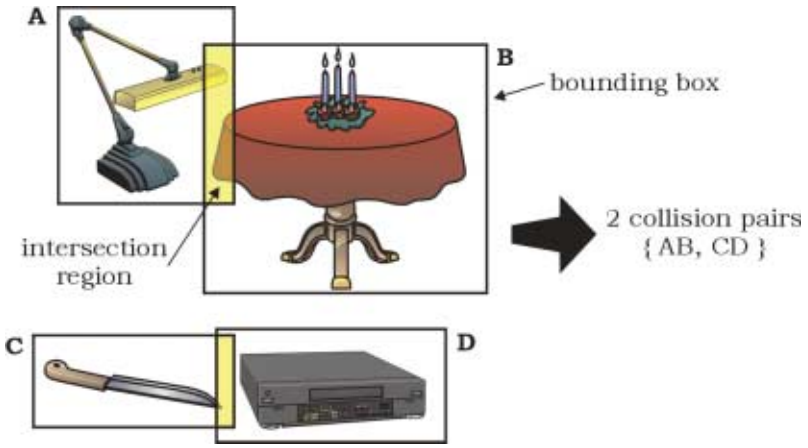

Figure 11: first pass in a collision detection system – attempting to eliminate as many collision pairs as early as possible

As an example consider the tests shown in Figure 11. Here we see an example of the first pass of a collision system. In this case we're using *bounding boxes*

or boxes that enclose fully the objects contained within. We test first to see if any of the boxes overlap (this is a much faster operation than doing full collision testing on arbitrary shapes), and if they do, that collision pair is passed to the next, more complex pass. Out of the potential list of 6 collision pairs = { AB, AC, AD, BC, BD, CD } we find that only 2 pairs have overlapping boxes = { AB, CD } and therefore only these 2 collision pairs need further testing. This is sometimes called *trivial rejection*.

Note that although, for example, pair AB has not been trivially rejected, this does not mean that A and B are actually colliding – only that they might be colliding. We do know, however, that A and C definitely do not collide because their bounding boxes do not overlap.

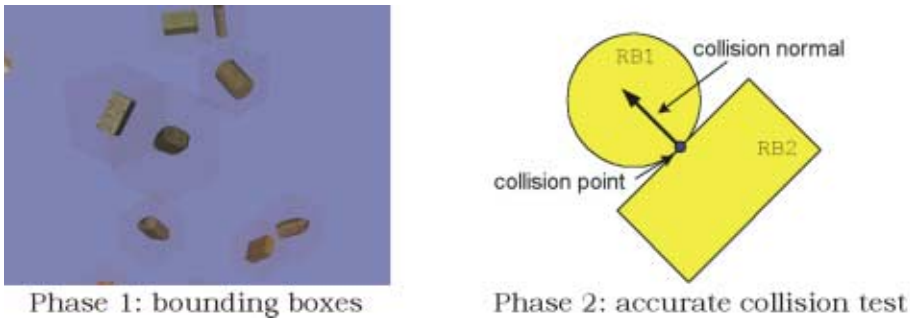

Figure 12: multiphase collision testing

The next phase will usually perform a more accurate rejection test or the final collision test (as shown in Figure 12) to determine information like the point of collision, the normal to the objects at the point of collision etc. which require substantially more work than simply testing for box overlaps.

The good news is that you don't have to worry about all this. The Havok system automatically implements this multi-phase collision detection approach and creates bounding boxes for all objects in the simulation automatically. The only control the user has over the complexity of the collision detection is in the shapes of the objects themselves. The next section deals with the various types of object shapes you can use and their complexity for collision detection.

Rigid Bodies & Collision Geometries

The shapes of the objects being tested for collisions have a major impact on the speed of the collision test. If we can make assumptions about the shape or even simplify the geometry for collision testing we can save a lot of CPU time.

One of the first assumptions that physics engines make is in assuming that all the objects in the scene are perfectly rigid (i.e. can never change shape). This results in a *rigid body simulation*. If all bodies are rigid then we can take advantage of the fact that the geometry of the objects do not vary from step to step and we can memorize these shapes and previous collision results to speed up the next collision test we perform using those objects. This naturally means

that objects made out of cloth, liquids or any deformable material cannot be simulated. Handling these sorts of objects will be dealt with later.

Insisting that all objects are perfectly rigid in a physical simulation is actually pretty bad news for the system (except for collision detection). All objects no matter how hard, when colliding, deform even if infinitesimally at the point of contact and then return to their original shape when they bounce away from the collision. When totally rigid objects collide, very large forces are generated in order to keep them apart. This is an unfortunate side effect that needs to be addressed for realistic collisions.

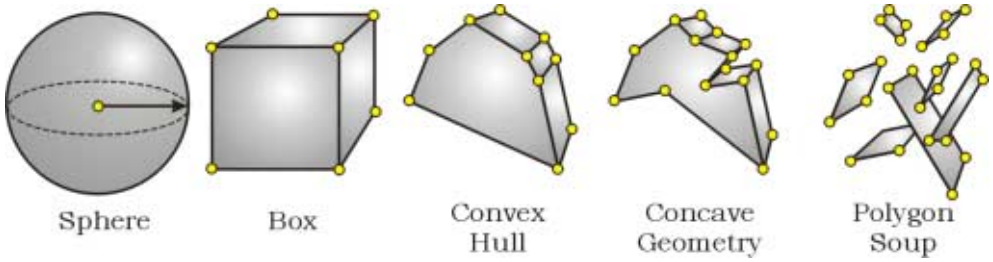

Figure 13: Some of the geometry formats supported by Havok.

In the Havok physics engine objects are classified according to their shapes, and particular shapes have particular properties that make it easier to deal with them during collision detection. The following list classifies object shapes in order of increasing complexity (see Figure 13 for diagrams of some of these):

- **Implicit**: we have a mathematical representation of the object and base the collision test on this. Implicit objects supported by Havok include:
  - spheres
  - planes
  - polygons

- **Polygonal**: we have a description of the object in terms of the polygons (usually triangles) making it up. In this case we classify the objects, in order of increasing complexity, as:

  - **Convex**: imagine wrapping the object in a cellophane wrap – if the cellophane touches every part of the surface (i.e. there are no hollows in the surface that the cellophane does not reach) then it's convex. Alternatively, pick a point inside the object and any direction: follow this direction out of the object and if you only ever pass the through the object's surface once, it's convex. Figure 15 depicts some rigid body simulations using entirely convex objects.

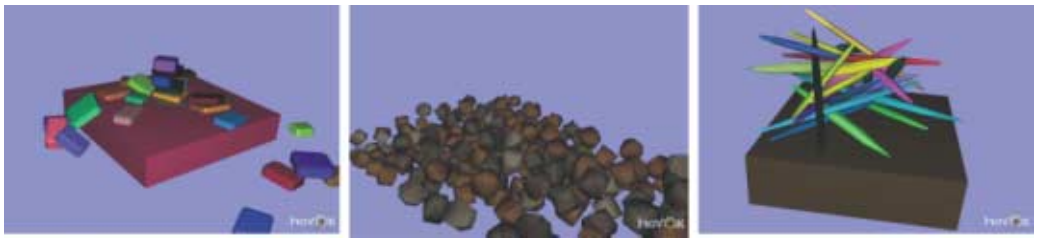

Figure 14: rigid body simulations using only convex shapes

- **Concave**: an object described by a closed surface (no holes or strange self-intersections like Klein bottles). Concave objects are always assumed to represent volumes (i.e. the polygon mesh representing the surface has no holes or gaps in it).

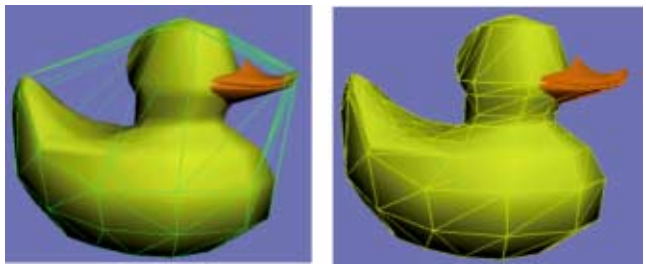

Figure 15: convex and concave representations of a rubber duck

- **Polygon Soup**: a collection of polygons, not necessarily connected, all grouped and classified as a single object. This is the most expensive format to detect collisions with, but is also the most general.

In Figure 15, the duck is shown represented as a convex object (the skin defined in the image on the left) and the true concave geometry on the right.

Proxy Objects

The good news is that even though an object in a simulation may appear complex, it need not be simulated with this complexity. For example, a car chassis displayed with detailed NURB panels and including wing mirrors and bumpers etc. might be simulated as a box.

This is one of the main ways to reduce the CPU costs of a simulation. These simplified objects representing the more complex display geometry are known as *proxy objects* or *proxy geometries*. Artifacts resulting from the use of proxies may be visible and in general you should attempt to make the proxy object as close a fit to the real display geometry as possible.

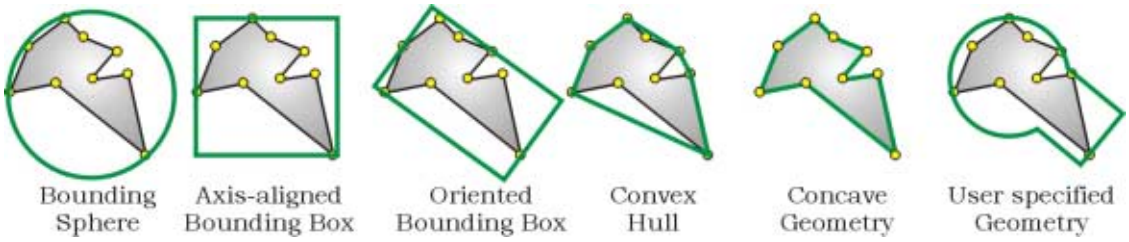

Figure 16: collision proxies for a complex shape

In Figure 16 we show a series of proxy objects for a complex geometry (the gray jagged thing which we'll assume is represented as a polygon soup). You can choose any proxy geometry you like for any shape of object but it makes sense to only use a simpler proxy geometry than the actual object itself. In many cases you will actually create a specific geometry (like the last one in Figure 16) that is a combination of a sphere and a box.

The final choice of proxy will represent a tradeoff between speed of collision and accuracy required and is typically very context specific. In Figure 17 the effect of the choice is plainly visible; a sofa geometry (with a pretty high polygon count) is being represented by 2 different proxies, a sphere and a collection of boxes. The collection of boxes is more expensive to compute collisions for but the sphere has a tendency to roll around, so is probably unsuitable in this case.

Figure 17: on the left we show a complex sofa object and 2 choices of proxy, on the right we show the simulation in progress (with the proxy objects displayed – although this would normally not be the case). The sphere is less suitable than the collection of boxes.

Interpenetration

Objects in a physical simulation are assumed to be solid and (except in the case of polygon soups) have a defined volume. It makes no sense in the real world to think of one solid object either inside or penetrating another solid object (it's not possible to embed a cup in a table). Similarly in the simulation of the real world, a physics engine is intolerant of objects that are *interpenetrating*. It also causes disbelief on the part of the viewer of the simulation – we don't expect solid objects to pass through each other. Therefore, a lot of the work performed by the physics engine is to attempt to prevent interpenetrations. This can be pretty tough, particularly when there are large numbers of objects stacked up all pushing downwards under the force of gravity (it's nearly as if they wanted to interpenetrate most of the time!)

When Havok encounters a pair of interpenetrating objects it simply ignores that collision pair and does not attempt to do anything clever – it will always try to prevent it happening in the first place of course. Interpenetrations occur most frequently in the following cases:

- When **large forces** are acting on objects forcing them to penetrate another object close by (often caused in large stacks).
- If the **time-step is too large** objects can become embedded in other objects before the physics engine has a chance to do anything about it.
- If the **user sets the position** of a physical object in such a way as to cause an interpenetration (see next sections for more details).
- If the **collision tolerance** is too small. See the next sections for more information on this.

The time step used in a simulation has a large impact on the collision detection engine's ability to accurately detect collisions. If large time steps are being taken 2 things can go wrong:

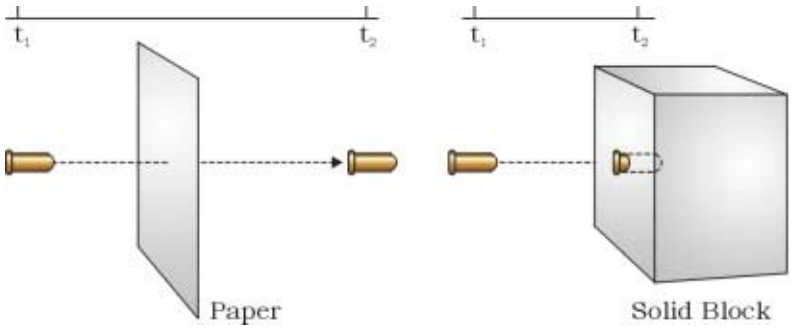

Figure 18: large time step issues in collision detection: on the left a collision is missed completely, on the right an interpenetration results.

- Collisions are missed – the so-called "bullet through paper" effect (see Figure 18.) Objects that are moving quickly will pass right through thin objects particularly if the time step is too large (i.e. a time t_1 the bullet is in front of the piece of paper, but at the next time step, t_2, the bullet has been placed on the other side of the piece of paper.)
- Interpenetration: the time scale is so large that objects become embedded in each other from one time step to the next – too deeply for the physics engine to be able to recover gracefully.

In general there is not much you can do except decrease the time step. For interpenetrations this will often help, but for the "bullet through paper" effect this will usually not solve the problem. Try to avoid very thin objects and extremely fast moving objects.

Inside & Outside

One major consequence of the assumption of solidity and the problems of inter-penetration is that you cannot consider most objects to be hollow. A common mistake is to create a box for a room and begin to place objects within the box, thinking these are being placed inside the room. A box, by default, is a solid geometry and any objects placed inside will be tagged as interpenetrating and will likely simply fall through the base of the box if simulated (i.e. collisions between the box and the objects will have been disabled by the physics engine).

ⓘ A way around this is to represent the box using a polygon soup geometry (unconnected triangles). This will have the desired effect.

Set Position & Deforming Geometry

Physics engines like to control the scene completely. By this we mean the physics engine is carefully storing the state of all objects in the scene and updating this state from frame to frame. If some external system then makes some changes to the scene the physics engine needs to update its state accordingly. This is often OK except when the changes made to the scene actually violate the simulation stability that the physics engine has been attempting to maintain.

Two good examples of this are users setting object positions or orientations (sometimes called *object warping*) and *deforming meshes*. Taking the first example it's easy to see how arbitrarily setting the position of an object in the physics scene may cause problems:

- If the object is moved to a position that causes an interpenetration; without additional information the physics engine will simply have to turn off collisions between the moved object and the object it has been moved into.
- If other objects are resting or stacked above the object that is moved; hopefully the stack will simply restructure to fill the gap made by the moved object.
- If other objects are attached to the moved object by *springs* or *constraints*: this is the worst case. By moving the object abruptly, the connections to the other objects are stretched instantaneously and can cause the system to explode.

Objects may be moved outside of the control of the physics system if they are being key framed in some way. For such objects, the Havok system needs to be informed that these objects can be expected to move unexpectedly – Havok will track these objects specifically and will attempt to resolve the situation, though not always successfully. In general try to avoid direct manipulation of objects. It is always better to apply *forces* and *impulses* either *linear* or *angular* to the objects to "push" them towards the desired goal.

The case of deforming meshes is subtler, but is related to the difficultly in handling warping objects. An example of this is shown in Figure 19.

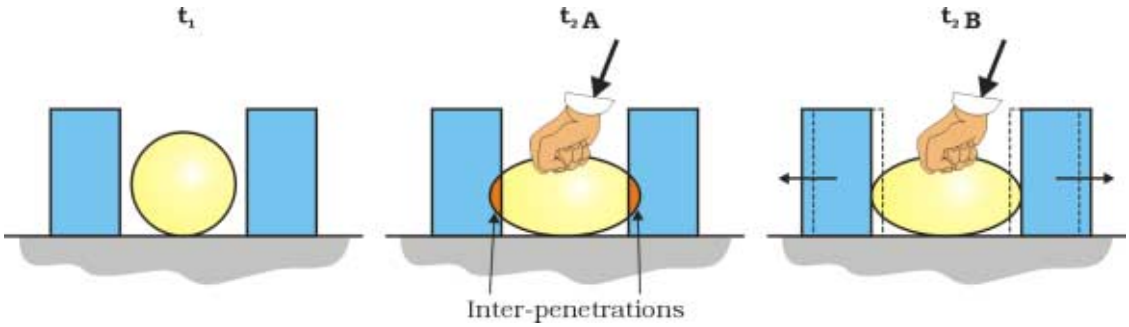

Figure 19: deforming objects like the squished sphere above need to be treated with care in a physics engine. The desired result is shown on the right i.e. the blocks are pushed out of the way when the sphere bulges out.

In this example, a sphere has being key framed to squash (perhaps to simulate the effect of being squashed from above – note we are talking about key framing here and not a soft object under physics control). Under normal circumstances this results in an interpenetration. The sphere's sides bulge out as it deforms into an ellipsoid shape, but this happens outside the control of the physics engine and therefore appears to happen instantaneously. This sort of thing can happen quite frequently, particular in situations where meshes are being generated automatically (e.g. in character animation a skin mesh around a set of bones deforms based on the position of the bones).

The correct result would involve the physics engine tracking the shape of the deforming mesh and when it spots a change in the mesh in a give time step it will use this with the shape of the mesh in the previous step (which it has remembered) to try and figure out how the surrounding affected objects should move. In this case, the deforming mesh will cause the two blocks on either side to be pushed outwards away from the sphere. The Havok system needs to be told that an object's geometry may deform outside of its control in order to achieve this result.

Collision Tolerance

One final but crucial feature of the Havok system is its implementation of collision tolerances. If you consider the previous sections on collisions and interpenetration you may have spotted that it's pretty hard for the physics engine to detect a collision without an interpenetration having taken place. Because the physics system is based around discrete time steps (of say 1/60th of a second) it is rare that a collision at a given time step will take place where 2 objects are just touching (i.e. a single point of contact).

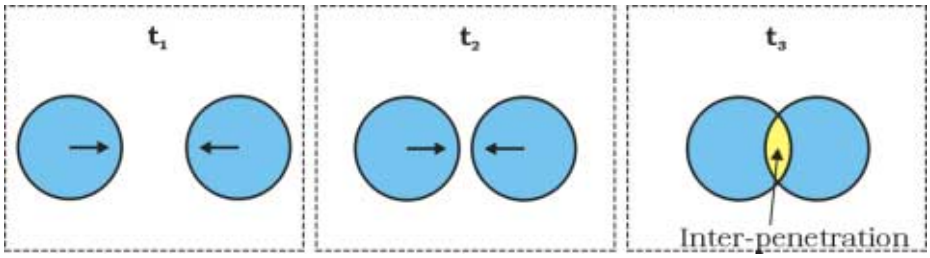

Figure 20: object collision are usually only detectable after they have occurred = interpenetrations

In Figure 20 this situation is illustrated. 2 spheres are traveling towards each other. At time steps t_1 and t_2 they are still apart, but at time t_3 they have inter-penetrated. To avoid this the Havok engine uses a system of collision tolerances. These are minimum distance values that specify how close objects can be before they are deemed to have collided. Consider the collision tolerance to be a "skin" of a certain thickness around the objects. If the skins overlap (or inter-penetrate) the objects are said to have collided and the nearest points on the 2 objects are used as collision points.

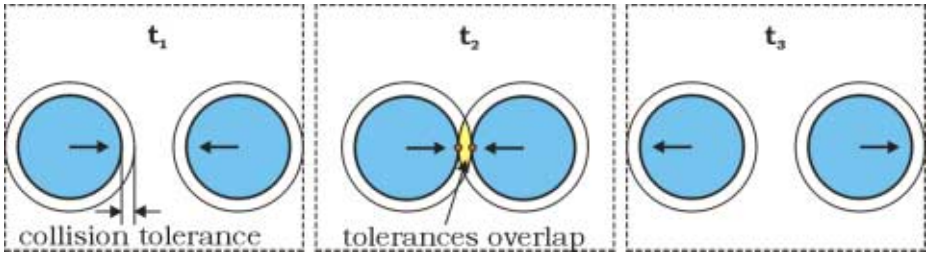

Figure 21: using collision tolerances, we can check to see if the tolerances overlap and if so consider the objects to have collided and take appropriate action.

Figure 21 shows what happens when collision tolerances are used. In this case the spheres have a small collision tolerance that really means that the spheres have a collision proxy that is another slightly larger sphere. At time t_2 each of the spheres have collided with the collision tolerance of the other sphere (note that it is not enough that only the tolerances have inter-penetrated – one of the spheres must be inside the collision tolerance of the other), the system calculates the collision information, assuming that the objects have collided (collision points are shown in Figure 21) and takes the appropriate action.

As expected, with large values of the collision tolerance, you'll begin to notice that the objects are not actually hitting. This is particularly noticeable for objects that are stacked (you'll see gaps between them). In general the tolerance should be around 2% - 10% of the size of the object itself. To remove most of these visual artifacts you should increase the size of the visual geometry by approximately half of the tolerance, or alternatively create a collision proxy that has been reduced in size by half the tolerance value.

Setting the scene

The first stage in any physics simulation is to create the scene. This is slightly different to constructing a 3D scene without physics and is constrained by collision tolerances and requirements of non interpenetration. If you construct a scene in the normal way you will run into difficulties placing objects so that they are stably resting on other objects without collision problems.

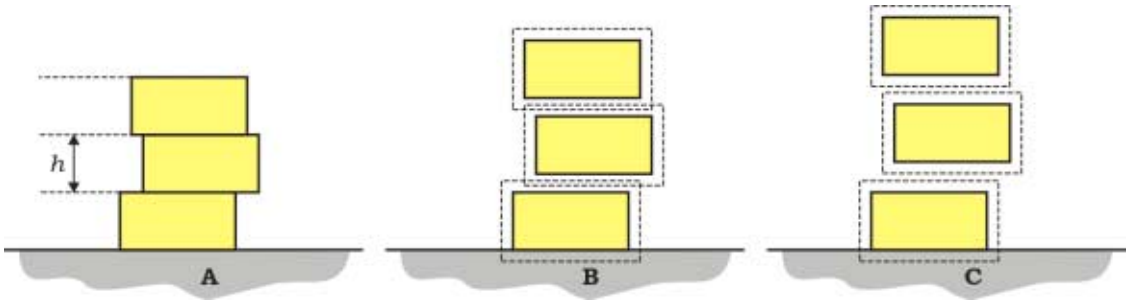

Figure 22: creating a stable stack

As an example take the case of creating a stack of boxes of height *h* on the floor. Usually you will place the base of the first box at height 0, the second at height *h*, the third at height 2*h* and so on, as shown in Figure 22 A. But given the collision tolerance issue mentioned in the previous section there will be problems with this approach. Depending on the numerical accuracy of the simulation, the faces of neighboring boxes will either be classified as interpenetrating or colliding. If interpenetrating, the boxes will fall through each other – if colliding they will immediately bounce off each other (albeit with very little energy, so you may get lucky). Another approach might be to place the boxes such that the collision tolerances overlap, but the boxes are not actually touching as in Figure 22 B. This can also cause problems, because the boxes are liable to settle for a while into a stable state (depending on external forces like gravity etc.)

The best way around this is to use the physics engine itself. As shown in Figure 22 C, you should create the boxes initially separated by a distance of more than the collision tolerance. Now simply simulate the scene for a period of time; the boxes will fall the small distance and will settle into a stable stack. Eventually they will turn off as the energy management system kicks in. Now you have a stable stack that is turned off and ready to be saved. Store the positions of the boxes – now you can use these positions to create your stack at a later point in time (making sure to turn the boxes off manually to fully recreate the state of the stack.) You are guaranteed that when the new simulation starts the stack will be stable and unmoving.

Physical Units & Values

We've already dealt with the issue of scale and switching between units of measurement. Physics engines are not simply concerned with lengths and weights however – in fact there is a very large number of different measures in use at any given time. In this section we'll list some of these and give the units

used to quantify them. Despite the apparent complexity of dealing with all these units, in many cases you will not need to worry: we find that experimentation with values is ultimately the best way to proceed when designing a scene. It is often useful, however, to fall back on an analysis of the scene and units used to at least determine the ballpark values required to achieve a desired goal.

For simplicity we'll stick with the metric system in the following sections:

- m = meters
- kg = kilograms
- s = seconds
- N = Newtons (measurement of force: 1 N = force required to change the speed of a 1kg object by 1 m/s in 1 s).
- rad = radians (1 rad = $180/\pi$ degrees where $\pi = 3.14159...$)

Note: some implementations of the Havok physics technology use degrees as units of angle. This will be clearly stated in the accompanying documentation.

To read the units in the tables below apply the following conventions:

- x/y = x per y e.g. m/s = meters per second (i.e. velocity or speed). Note that x/y is sometimes written as $x\ y^{-1}$.
- x/y^2 = x per y squared or x per y per y e.g. m/s^2 = meters per second squared or meters per second per second (i.e. acceleration)

Position Based

Name	Units	Description
Position	m	A better description of this term is displacement or distance.
Velocity	m/s	Speed – how fast is an object traveling relative to some frame of reference.
Momentum	kg m/s	Velocity times mass: this is the property of an object that determines the amount of force required to change the velocity of something. For example a truck is harder to stop than a scooter even if both are traveling at the same speed.
Acceleration	m/s^2	Rate of change of velocity over time i.e. is an object speeding up or slowing down?
Impulse	kg m/s or Ns	A measure of a change in momentum (usually measured in Newton seconds). If you want to instantaneously change the velocity of an object you must apply an impulse to the object.
Force	N or $kg\ m/s^2$	The basic unit of a physics engine. This is the quantity that measures the effort required to change the speed of an object (i.e. to give an object an acceleration – either positive or negative).

Table 3

Note: Sometimes, to distinguish from their angular counterparts (detailed in the next sections), velocity, momentum and acceleration are defined as *linear velocity*, *linear momentum* and *linear acceleration*.

Orientation Based

Orientations are often one of the most difficult quantities to become familiar with. They can be specified in any number of ways, some of the most popular being a transformation matrix or an axis and angle or using one of the many "standards" like pitch-yaw-roll, or azimuth-elevation-tile (shown in Figure 23: different ways to specify the orientation of an object in space). The most important thing to understand is that orientations require a reference frame, just as distance requires a reference point (i.e. giving a location as 10 meters away is useless without information about what / where it is 10 meters away from).

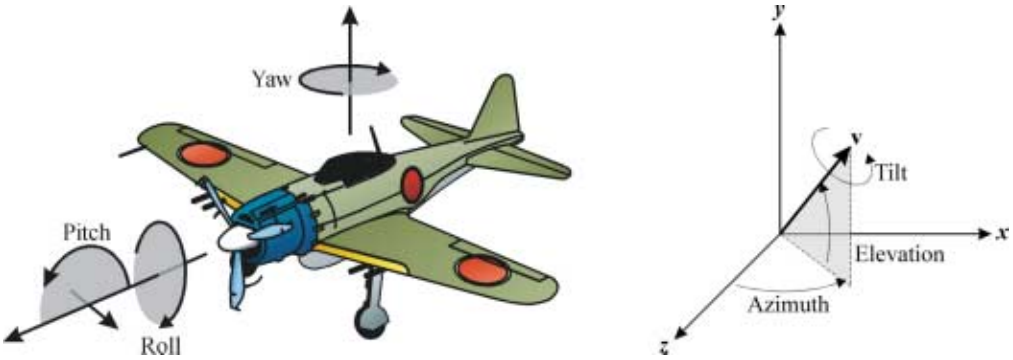

Figure 23: different ways to specify the orientation of an object in space

Coordinate Systems & Reference Frames

In order to talk about orientations we need first to mention coordinate systems and reference frames. The orientation of any object in a scene is specified with respect to a coordinate system, often called the *world coordinate system* or *world reference frame*. A 3D coordinate system has 3 independent directions or *vectors* usually termed x, y and z and an origin (position). All positions and directions are specified with respect these directions and the origin. When an object is created, we need to position it in the world and we do so by specifying its position and orientation; the object's position and orientation are termed its *local coordinate system* or *local reference frame* (i.e. local to that object). Figure 24 shows a scene with a global coordinate system and 2 objects, each with its own local coordinate systems and positions.

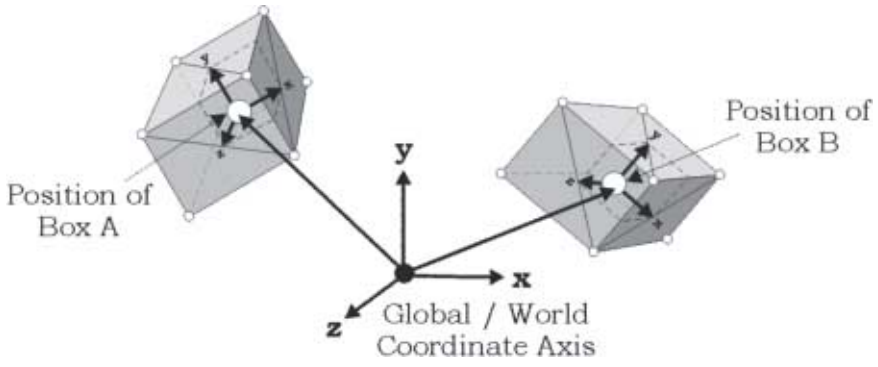

Figure 24: local and global coordinate systems

There is one final ambiguity to resolve (and one which can cause headaches when interfacing 3D engines with physics systems): what is the center of an object. If we say "place the object at the origin" (i.e. at position [0, 0, 0]) where exactly is it placed? This depends on which part of the object we line up with the origin: there are 2 options:

- The origin or pivot as specified by the geometry or modeler
- The center of mass (COM), which depends on the physical properties of the object (like the distribution of mass through the object i.e. is one end heavier than the other, like a hammer). The COM is the point around which the object will naturally spin.

The physics engine always uses the COM for specifying rotations and orientations, whereas usually the 3D engine will use the object's geometric center or pivot as defined by the modeler. In Figure 24 the boxes' geometric center is also at the COM (i.e. right in the middle of the box). The position of the box is taken to be the position of that center. The local coordinate system for each box is shown and these are defined relative to the global coordinate system.

Specifying Orientations

In Havok we specify the orientation of an object using a rotation. This rotation is defined by an axis, which is the axis or line we want to rotate around and an angle, which is the angle to rotate around this axis. The rotation used is the rotation that would take the object from its starting orientation (usually lined up with the world reference frame) to its current orientation. Note that rotation angles are given as *anti-clockwise*. This is a standard in computer graphics derived from our use of the *right-hand rule* for orientation specification. The right hand rule is simple to remember and is depicted in Figure 25.

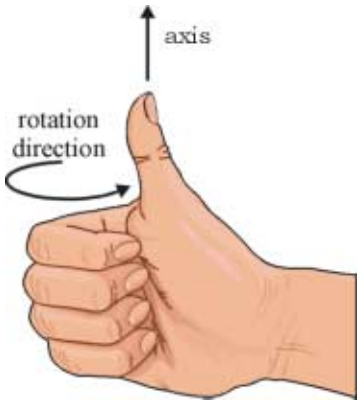

Figure 25: the right hand rule for rotations: the thumb lines up with the axis to be rotated around and the direction the fingers curl indicates the direction a positive rotation angle will rotate in.

Armed with this information (pun intentional!) we can describe the method of specifying orientations in general.

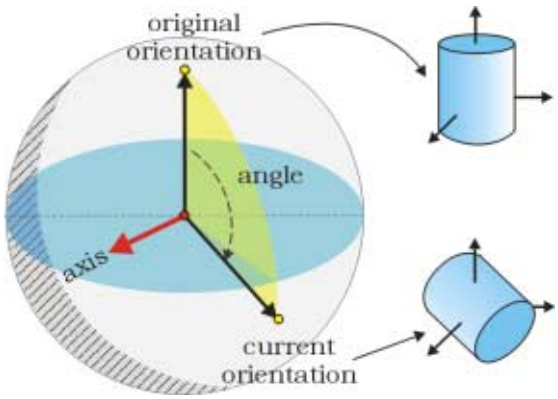

Figure 26: specifying an orientation using an axis and angle

In Figure 26 we show how a rotation takes effect given an axis and angle. The top right image of the object shows it at creation time (i.e. its local coordinate system is lined up with the world's). In the image below the object is now lined up to the required orientation.

Name	Units	Description
Orientation	Axis + angle	An object's orientation with respect to the world coordinate system.
Angular velocity	Axis + rad/s	The speed at which the object is rotating: i.e. the number of radians per second the object rotates, usually specified with the axis it is rotating around
Angular Momentum	kg rad/s	Angular equivalent of momentum. This quantifies how hard it is to increase or decrease the rotational velocity of an object.
Angular Acceleration	rad/s^2	Angular equivalent of acceleration. This is the rate of change over time of the angular velocity.
Angular Impulse	kg rad/s or N rad	Angular equivalent of impulse. This is a measure of a change in angular momentum. Apply an angular impulse to an object if you want to instantaneously affect its angular velocity.

Torque	kg m^2/s^2 Nm	This is the angular equivalent of force but needs a separate discussion below. Units are Newton meters.

Table 4

With Havok you can *passively* change the behavior of a body by applying forces, torques, impulses and angular impulses (or *actively* by setting properties like velocity or angular velocity). When applying impulse and forces it is important to specify the point with respect to the object that the impulse / force is to be applied.

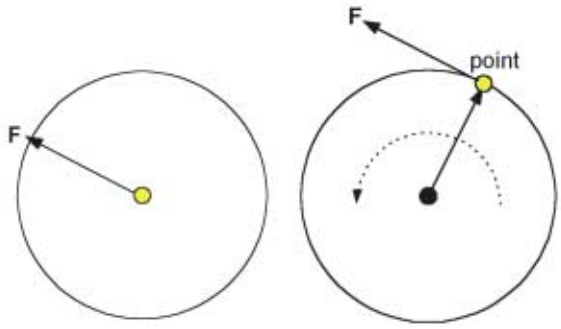

Figure 27: applying forces at COM and at an arbitrary point with respect to an object's coordinate system

If a force / impulse is applied at the center of mass (COM) of an object that object's acceleration / velocity will change but no additional rotational velocity or acceleration will be introduced. On the other hand, when a force or impulse is applied at some other point away from the COM a torque is introduced which is proportional to the force / impulse applied and also the distance from the center of mass and will alter the angular velocity / acceleration of the object. Think of it like a lever were attached from the COM to the point where you're applying the force / impulse – a longer lever makes it easier to move (or alter the angular velocity / acceleration of) a heavy object than a short lever.

Physical Properties

These properties depend on the material(s) making up the object and affect its behavior in a physical simulation.

Name	Units	Description
Friction	Dimensionless	The friction coefficient (usually a value from 0 to 1) which specifies how "sticky" or "rough" an object is. See below for a more detailed explanation.
Restitution	Dimensionless	The restitution coefficient (usually a value from 0 to 1) specifies what percentage of the kinetic energy is lost during a collision between 2 objects. With a value of 0 all energy is lost and the objects appear to come to a complete halt when they collide. With 1, no energy is lost and the objects will bounce off each other with an equal but opposite velocity. Somewhere in between and the objects lose energy with each collision.
Mass	kg	A measure of an object's resistance to change in motion, or the amount of matter in an object. Not to be confused with weight which is the attraction of the Earth's

		gravitational pull on a certain mass. To make matters confusing though, mass is defined officially by the *weight* of a mass of platinum-iridium stored in Sevres in France.

Table 5

Dynamic and Static Friction

Friction is that quantity which attempts to prevent surfaces sliding off each other and is the key factor is allowing stable stacking (i.e. stacks or piles of objects that come to rest, held in place by the friction at the points of contact). During all collisions a certain amount of energy is lost due to friction (and mostly converted to heat). Friction manifests itself in 2 forms, static and dynamic. Static friction operates when objects are at rest; it attempts to prevent the objects from moving or sliding. If a force is applied to the object large enough to overcome static friction, the object will begin to move / slide. At this point dynamic friction kicks in and, as long as the object is in contact with another object or surface, the dynamic friction attempts to slow the object down.

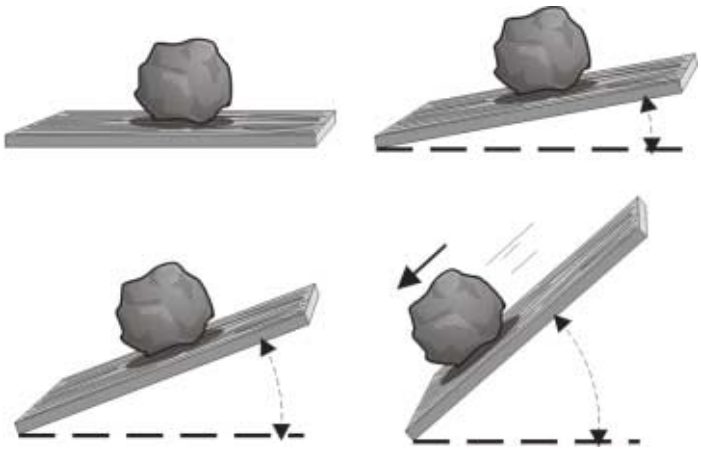

Figure 28: static and dynamic friction in action. The boulder is held in place by static friction until enough force is applied to break the contact (i.e. the plank has been raised to a sufficient height), after which the boulder begins to slide and dynamic friction kicks in which acts against the sliding action, generating heat.

Figure 28 gives an example of static and dynamic friction in action. In Havok you set a single friction coefficient value and the engine manages the transition between the friction modes for you.

Next Steps

We've briefly covered most of the main topics of physical simulation. This is a very rich subject matter and there's certainly a lot more to know but this should be sufficient to help you tackle the documentation that comes with the Havok technology. We have specifically not attempted to cover all areas of the Havok physics technology but have focused instead on the general principles that underpin all simulation. We'll briefly describe now some of these other physics technologies available from the Havok engine.

Constrained Dynamics

In many cases we want to construct physics systems that have constraints or attachments between sub parts of a larger assembly. There are many types of constraints and some of those provided by Havok include:

- Springs: forces are applied to objects connected by springs to attempt to keep the objects within a desired distance from each other (the rest length).
- Dashpots or stiff springs: springs by default are pretty stretchy and in some situations you want to connect objects more tightly, but increasing the rigidity or *stiffness* of a spring also decreases its stability, and at some point the spring will break. Dashpots allow much higher stiffness values.
- Reduced coordinate systems: systems of rigid bodies are not simulated independently but are simulated as a large system all at the same time. Some reduced coordinate constraints include hinges, ball and socket joints, prismatic joints and universal joints.

Constraints are a requirement for creating any dynamic system like a car chassis, or a locomotive engine. Forces applied to one object are instantaneously applied to all objects connected to this one, so when the piston on a locomotive pushes forward, the wheels begin to spin. See Figure 29 for some examples of constrained systems.

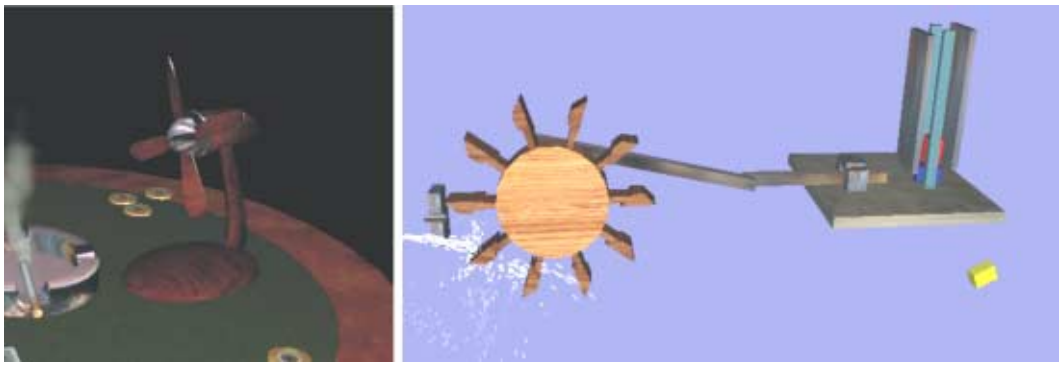

Figure 29: constrained systems in action: the fan blade is connected to the base via a revolute joint and the water machine on the right uses a series of constraints to create the water powered block pushing mechanism.

Non Rigid Body Dynamics

So far we have been assuming that all objects are rigid (i.e. the geometry or shape does not change during the simulation). In order to simulate soft, cloth or liquid objects we need to lift this restriction. Most of what has been discussed earlier still applies except for the details of collision detection. For deformable objects, collision detection becomes much more difficult – given that the object can change shape dramatically between time steps and also can attempt to collide with itself (this is particularly true of cloth, where interpenetration prevention is very expensive).

Some examples of deformable surfaces are shown in Figure 30.

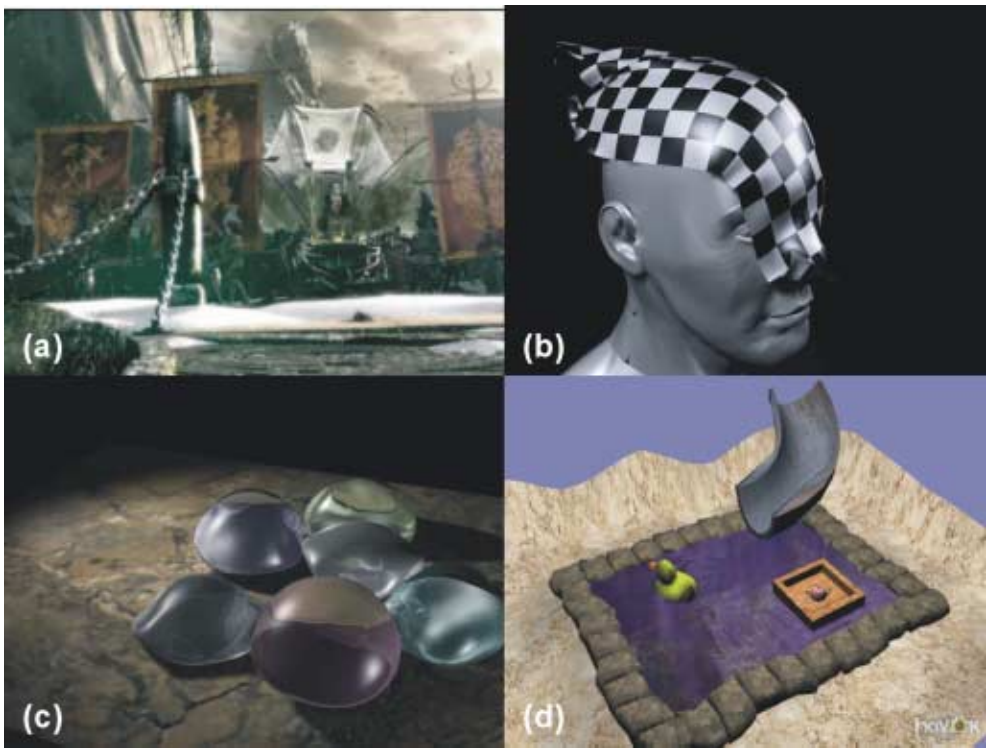

Figure 30: examples of deformable objects: (a) lots of cloth and rope elements used in a scene (courtesy of Blizzard Entertainment) (b) a piece of cloth slides over a mannequin's head (c) blobby objects land on a hard surface (d) a water pool simulation with floating raft and rubber duck.