There is a big difference between a collision engine and a physics engine. They do not do the same thing, although the physics engine generally relies on a collision engine.

The collision engine is then split into two parts: collision detection and collision response. The latter is generally part of the physics engine. This is why collision engines and physics engines are usually rolled into the same library.

Collision detection comes in two forms, discrete and continuous. Advanced engines support both, as they have different properties. In general, continuous collision detection is very expensive and only used where it is really needed. The majority of collision and physics is handled using discrete methods. In discrete methods, objects will end up penetrating each other, and the physics engine then works to push them apart. So the engine does not actually stop a player from walking partially through a wall or the floor, it just fixes it up after detecting that the player is partially in the wall/floor. I'm going to focus on discrete collision detection here, since that's what I have the most experience implementing from scratch.

Collision Detection

Collision detection is relatively easy. Every object has a transform and a shape (possibly multiple shapes). Naive approaches would have the collision engine do an O(n^2) loop through all object pair and tests if there's overlap between the pairs. In smarter approaches there are multiple spatial data structures (e.g., for static vs dynamic objects), a bounding shape for each object, and multi-part convex sub-shapes for each object.

The spatial data structures include things like KD-Trees, Dynamic AABB trees, Octrees/Quadtrees, Binary Space Partitioning trees, and so on. Each has their advantages and disadvantages, which is why some higher end engines use more than one. Dynamic AABB trees for instance are really really fast and good for handling lots of moving objects while a KD-Tree may be more suitable for the static level geometry that objects collide with. There are other options as well.

The broad phase uses the spatial data structures and an abstract bounding volume for each object. A bounding volume is a simple shape that encloses the entire object, generally with the goal of enclosing it as "tightly" as possible while remaining cheap to do collision tests with. The most common bounding shapes are Axis-Aligned Bounding Boxes, Object-Aligned Bounding Boxes, Spheres, and Capsules. AABBs are generally considered the fastest and easiest (Spheres are easier and faster in some cases, but many of those spatial data structures would require converting the sphere into an AABB anyway), but they also tend to fit many objects rather poorly. Capsules are popular in 3D engines for handling character-level collisions. Some engines will use two bounding shapes, such as an AABB for the first level of detection and then an OABB or Capsule for the second.

The last phase of collision detection is to detect exactly where the geometry is intersecting. This usually implies using the a mesh (or a polygon in 2D), though not always. The purpose of this phase is to find out if the objects really truly do collide, if a fine level of detail is required (say, bullet collision in a shooter, where you want to be able to ignore shots that just barely miss), and also to find out exactly where the objects collide, which will affect how the objects respond. For example, if a box is sitting on the edge of a table, the engine must know at what points the table is pushing against the box; depending on how far the box is hanging over, the box may begin to tilt and fall off.

Contact Manifold Generation

Algorithms used here include the popular GJK and Minkowski Portal Refinement algorithms, as well as the Separating Axis test. Because the popular algorithms typically only work for convex shapes, it is necessary to break many complex objects into convex sub-objects, and do collision tests for each individually. This is one of the reasons why simplified meshes are often used for collision, as well as the reduction in processing time for using fewer triangles.

Some of these algorithms not only tell you that the objects have collided for sure, but where they collided -- how far they are penetrating each other and what the "contact points" are. Some of the algorithms require additional steps, such as polygon clipping, to get this information.

Physical Response

At this point, a contact has been discovered, and there is enough information for the physics engine to process the contact. The physics handling can get very complex. Simpler algorithms work for some games, but even something as seemingly straight-forward as keeping a stack of boxes stable turns out to be quite difficult and requires a lot of work and non-obvious hacks.

At the most basic level, the physics engine will do something like this: it'll take the colliding objects and their contact manifold and calculate the new positions required to separate the collided objects. It will move the objects to these new positions. It'll also calculate the velocity change resulting from this push, combined with restitution (bounciness) and friction values. The physics engine will also apply any other forces acting on the objects, such as gravity, to calculate the objects' new velocities, and then (next frame) their new positions.

More advanced physics response gets complicated quickly. The approach above will break down in many situations, including one object sitting on top of two others. Dealing with each pair by itself will cause "jitter" and the objects will bounce around a lot. The most basic technique is to do a number of velocity-correction iterations over the pairs of colliding objects. For example, with a box "A" sitting on top of two other boxes "B" and "C", the collision A-B will be handled first, causing box A to tilt further into box C. Then the A-C collision is handled, evening out the boxes somewhat, but pulling A down and into B. Then another iteration is done, so the A-B error caused by the A-C correction is slightly resolved, creating a bit more error in the A-C response. Which is handled when A-C is processed again. The number of iterations done is not fixed, and there is no point at which it becomes "perfect," but rather just whatever number of iterations stops giving meaningful results. 10 iterations is a typical first try, but it takes tweaking to figure out the best number for a particular engine and a particular game's needs.

Contact Caching

There are other tricks that turn out to be really handy (more or less necessary) when dealing with many types of games. Contact caching is one of the more useful ones. With a contact cache, each set of colliding objects is saved in a lookup table. Each frame, when a collision is detected, this cache is queried to see if the objects were previously in contact. If the objects were not previously in contact, then a "new collision" event can be generated. If the objects were previously in contact, the information can be used to provide a more stable response. Any entries in the contact cache that were not updated in a frame indicate two objects that separated, and a "separating object" event can be generated. Game logic often has uses for these events.

It's also possible for the game logic to respond to new collision events and flag them as ignored. This is really helpful for implemented some features common in platforms, like platforms that you can jump up through but stand on. Naive implementations may just ignore collisions that have a downward platform->character collision normal (indicating the player's head hit the bottom of the platform), but without contact caching, this will break if the player's head pokes up through the platform and then he begins to fall. At that point, the contact normal may end up pointing upward, causing the player to pop up through the platform when he shouldn't. With contact caching, the engine can reliably look at the initial collision normal and ignore all further contact events until the platform and player separate again.

Sleeping

Another very useful technique is to mark objects as being "asleep" if they are not being interacted with. Sleeping objects do not get physics updates, do not collide with other sleeping objects, and basically just sit there frozen in time until another non-sleeping object collides with them.

The impact is that all the pairs of colliding objects that are just sitting there doing nothing don't take up any processing time. Also, because there is not a constant amount of tiny physics corrections, stacks will be stable.

An object is a candidate for sleeping when it has had a near-zero velocity for more than a single frame. Note that the epsilon you use for testing this near-zero velocity will probably be a bit higher than the usual floating point comparison epsilon, as you should expect some jitter with stacked objects, and you want whole stacks of objects to fall asleep if they're staying "close enough" to stable. The threshold will of course require tweaking and experimentation.

Constraints

The last major bit of many physics engines is constraint solver. The purpose of such a system is to facilitate the implementation of things like springs, motors, wheel axis, simulated soft-bodies, cloth, ropes and chains, and sometimes even fluid (though fluid is often implemented as an entirely different system).

Even the basics of constraint solving can get very math intensive and goes beyond my expertise in this subject matter. I recommend checking out Randy Gaul's excellent article series on physics for a more in-depth explanation of the topic.

Source: [How does a collision engine work? - Game Development Stack Exchange](https://gamedev.stackexchange.com/questions/26501/how-does-a-collision-engine-work)