2D Physics Engine

Implementation of rigid bodies, soft bodies and dynamic fluids

Dominic Bodman
University of Derby
Derby, England
d.bodman1@unimal.derby.ac.uk

Abstract—This paper covers the theory behind and implementation of several physical systems including rigid bodies, soft bodies and dynamic fluids. This will cover the physics and mathematics behind the simulation as well as information on architecting and optimising the implementation. The focus is on real-time applications that do not require great accuracy such as games.

Keywords—physics; games; rigid body; soft body; fluid

I. INTRODUCTION

Physics is a popular gameplay mechanic in many games. Half-Life 2 [1] was a popular game that introduced the idea of physics-based puzzles and since then almost every modern game provides some kind of physics simulation either for gameplay purposes or simply for atmosphere.

Rigid-body simulators are in abundance and one can find a freely available one for their project fairly easily. However, there is more to the physical world than purely non-deformable, unbreakable bodies. There are considerably fewer engines that provide support for deformable bodies or fluids and even fewer that provide all of these options in a stable and interactive manner.

This project aims to rectify this by providing a 2D physics engine that provides not only rigid body simulation but also soft body and dynamic fluids, all of which can interact with one another. The engine aims to be reasonably stable at real-time speeds (A requirement for game physics) and easy to use for the end-user.

II. RELATED WORK

One of the most well-known 2D physics engines in games is the Box2D engine [2]. It is an open-source rigid body simulator that supports many rigid bodies and joint types and features many optimisations to improve performance. The author of the engine, Erin Catto, is well known in the game physics community and some of the ideas he has presented at GDC have been incorporated into this project.

The dynamics of deformable bodies and fluids provide for new and interesting behaviours for a player and also give an even greater sense of realism to the game world as water flows and splashes realistically and a rubber ball compresses and bounces around in a satisfying manner. Games like “Feed me Oil” [3] and “Gish” [4] have proven that physics-based mechanics beyond the now standard use of rigid bodies can serve as a fun and engaging mechanic for players.

With an available and accessible engine for providing these kinds of behaviours, more games can take advantage of them in new and innovative ways.

III. METHODOLOGY

This paper covers the theory behind and implementation of several physical systems including rigid bodies, soft bodies and dynamic fluids. This will cover the physics and mathematics behind the simulation as well as information on architecting and optimising the implementation.

A. The Simulation

The simulation is the core of this implementation and serves as the interface between the physics engine and the host application.

It provides methods for adding and removing physical objects and advancing the simulation by one step.

The simulation can use any timestep provided by the host application, however it is more stable with smaller time steps. A simulation rate of 100Hz is about the optimum time step; providing a balance between keeping all of the physical systems stable while still being slow enough to actually be a feasible update rate.

In order to assist with keeping the simulation running at a consistent rate, the simulation has been implemented with multithreaded applications in mind. By running the simulation on a thread at a 100Hz update rate, the main thread can run at a different rate to accommodate other tasks such as UI handling and rendering.

The simulation writes the state of all physical objects and systems to a buffer at the end of every step. This state can be read by the slower-updating thread in a thread safe manner due to a specific sequence of locks and operations that prevent
the simulation state from being accessed simultaneously by multiple threads.

The result is a physics engine that can safely run at a consistent update rate in a multithreaded application with no concessions required to be made by the end-user.

B. Rigid Bodies

Rigid bodies are such named because they represent bodies of mass that are non-deformable and unbreakable. The concept is a simplification for the sake of real-time simulation since nothing in the real world is truly 100% rigid [5]. Rigid bodies allow for the simulation of complex shapes as they move in space and interact with each other.

In the engine, rigid bodies are represented as objects with the following quantities:
- Mass
- Position
- Velocity
- Acceleration
- Moment of Inertia
- Rotation
- Angular Velocity
- Angular Acceleration

This structure is derived from the kinematic properties of motion as described in [5] and [6].

The rigid body object responds to applied forces, which generate acceleration and torque. Linear acceleration is found using the Newton’s Second Law of motion:

\[ a = \frac{F}{m} \]  

(1)

Where \( F \) is the force applied to the body and \( m \) is the mass of the body. The torque is dependent on the force and the location of its application [6]:

\[ \tau = r \times F \]  

(2)

Where \( \tau \) is the torque due to the force, \( r \) is the location of the application of the force and \( F \) is the force. The cross operator represents the cross product between two vectors. It’s worth noting that, in 2D, the cross product of two vectors (In this particular context) is also known as the perpendicular dot product and therefore produces a scalar value (Which is correct for 2D rotation) [7].

With the torque known, the angular acceleration is defined as:

\[ \alpha = \frac{\tau}{I} \]  

(3)

Where \( \alpha \) is the angular acceleration due to the torque, \( \tau \) is the torque and \( I \) is the moment of inertia. The moment of inertia is essentially the angular equivalent of mass. Just as mass describes how much force is required for a given acceleration, the moment of inertia describes the resistance of a body to rotation about an axis [6].

The calculation of the moment of inertia for a polygon must take into account the constant density property of rigid bodies; that is the mass is evenly distributed across the body [8] [6]. To simplify the process, polygons can be decomposed into triangles and the individual moments of the triangles added up to find the total moment of inertia for the entire polygon.

In order to move a body over time, the acceleration quantities (both linear and angular) must be integrated in order to find velocity quantities. Those velocity quantities can be further integrated to find changes in position and rotation [5].

The integration method used in this implementation is known as Euler implicit integration. This is a variation on standard Euler’s method for integration of ordinary differential equations [9]. In the context of motion, this produces these equations:

\[ v_{t+1} = v_t + (a_{t+1} \times \Delta t) \]  

(4)

\[ r_{t+1} = r_t + (v_{t+1} \times \Delta t) \]  

(5)

Note that \( a_{t+1} \) and \( v_{t+1} \) terms are used in the calculation. This is what makes this method implicit as it requires the calculation of the next state of those variables, not the current state. This differs from the explicit method where only terms for the current time step are used in calculations. This difference improves the stability of the integration [9].

For angular motion, the equations are as follows:

\[ \omega_{t+1} = \omega_t + (\alpha_{t+1} \times \Delta t) \]  

(6)

\[ \Omega_{t+1} = \Omega_t + (\omega_{t+1} \times \Delta t) \]  

(7)

Where \( \omega \) is angular velocity and \( \Omega \) is the rotation of the body [6].

C. Collision

Rigid Bodies also provide a polygonal shape that is used for detecting collisions with other bodies.

The engine provides a dedicated collision module that handles the detection and resolution of collisions between rigid bodies. This module is known as the Collision Manager.

Collision resolution is covered in the next section, but before collisions can be resolved, certain information must be extracted.

Simply detecting collision is not enough; the point of contact and the amount of intersection must also be found to allow for correct resolution [10]. To do this, an algorithm called the Separating Axis Theorem (SAT) is used.
The SAT algorithm detects collisions by attempting to find an axis along which two bodies are separated (Hence the name) [11]. For all of the test axes, the vertices of the bodies are projected into a linear format and the minimum and maximum bounds of the projection are found. If the two projections of the two bodies do not overlap, the bodies are not intersecting and the algorithm terminates. The axes to test are simply the normal vectors of the body polygon edges [11].

Finding the depth of intersection is simply a case of measuring the amount of overlap along each test axis and using the largest. The point of contact is found by projecting the intersecting vertex from one body on to the closest edge from the other body. This vertex/edge pair is known as a contact feature or incident/reference pair [10].

In order to find the contact information for every single pair of colliding bodies, a naïve approach would be to brute force check every body against every other body in the system. This introduces a huge bottleneck and in order to support a large number of colliding rigid bodies, a few improvements need to be made to the collision systems that help to considerably improve performance.

The source of the bottleneck is the O(n^2) loop used to test every body against every other body. In a scenario with 1000 rigid bodies, the collision system will perform 1,000,000 collision tests every step. While the collision detection method is relatively fast, it is not feasible to be performing it 100,000,000 times every second (Assuming a simulation rate of 100Hz).

It is required to perform a simpler, broad-phase collision step to determine which pairs of bodies require full collision testing. This is done by dividing the simulation domain in cells and sorting bodies into lists that describes the bodies that occupy each cell. Each body then need only perform the full collision test against each body occupying the same cell as itself [12]. This rule applies also to updating persistent contacts with accumulated impulses; dividing the list of persistent contacts across cells helps with the updating process. This immediately reduces 1000 x 1000 tests by roughly two orders of magnitude.

Another step can be inserted prior to the full collision test to further improve performance. Each step, a body can calculate the axis aligned bounds of itself; a bounding box (AABB). This bounding box can be used to determine pairs of bodies that are potentially colliding. Any body pairs whose AABBs do not overlap are definitely not colliding and so can be immediately culled.

Finally, the collision system can be setup to perform the collision tests for the cells of the domain partition on separate threads. Since bodies only ever need to test against bodies in the same cell, the cells can be processed in isolation, which makes them a ripe target for multi-threading. This means that as the number of CPU cores available to the simulation increases, the capacity of the collision system also increases.

On an Intel Core i7 processor with four physical cores (eight software threads), the collision system was able to handle 2000 rigid bodies resting in a large pile at a steady 60Hz.

\[ (v_1 - v_2) \cdot n_{collision} \geq 0 \]  

\( v_1 - v_2 \) is the relative velocity at the point of contact and \( n_{collision} \) is the normal of the collision. \( v_1 \) and \( v_2 \) are the velocity of the contact point on each body involved. This takes into account both linear and angular velocity to produce a total linear velocity at that point [13]. The collision normal is simply the normal of the reference edge found by the collision detection system.

The inequality specifies that the relative velocity at the contact point that is acting towards that collision normal is greater-than or equal to zero i.e. the bodies are non-moving or moving apart at the contact point [13].

To solve the equation, impulses must be found that adjust that relative velocity such that they satisfy the inequality [13]. An impulse is a quantity much like a force only it affects velocity rather than acceleration. It represents a very large force being applied over a very small time period, which is perfect for modeling the instantaneous response required for collision resolution [13].

The direction of the impulse for each body acts away from the contact point. The magnitude of the impulse is the same for each body. The impulse needs to take into account not only the relative normal velocity but also the mass and moment of inertia of both bodies to achieve the correct motion when bodies of different masses collide with each other [13].

The impulse is scaled by the inverse of the sum of the body masses (Taking into account both mass and moment of inertia). The impulse, when applied to a body, is scaled by its own inverse mass and inverse moment of inertia. This means that each body receives the correct amount of impulse. For example, in a collision between a high mass body and a low mass body, the high mass body will scale the impulse such that it is quite small (1 / a large mass = a small scalar) while the low mass body will not scale the impulse down as much and so will end up moving more than the large mass body as a result of the collision.

For the full details and breakdown of impulses, see [13].
It’s worth noting that simple friction can be implemented using the same impulse system as described above only using the relative tangential velocity (i.e. velocity acting perpendicular to the collision normal) in place of the relative normal velocity [10]. This can be scaled by a coefficient of friction to dictate the strength of the friction impulse [10].

E. Handling Intersection

To keep the simulation running at real-time speeds, bodies are allowed to intersect and it is another of the contact solver’s jobs to resolve this intersection as well. This is where the intersection depth, calculated during collision, detection comes into play.

When the standard normal collision impulse has been calculated, an extra “bias” impulse is added to it. The magnitude of this bias impulse is proportional to the amount of intersection and a control scalar that determines how quickly intersection is resolved. The full equation for the bias impulse is [10]:

\[
J_{\text{bias}} = \beta \frac{d(\text{intersection})}{dt}
\]  

(9)

Where \( \beta \) is the bias control scalar.

A bias control value of between 0.1 and 0.3 has been found to be reliable and stable [10]. This extra impulse will push bodies out of each other over the course of many steps, which resolves the penetration, but does so slowly enough so as to not disturb the stability of the simulation.

F. Persistent Contacts

A single impulse per contact is sufficient for bodies acting in an environment with no external forces. However, when gravity is introduced, this method doesn’t hold up. Gravity introduces the concept of resting bodies when two or more bodies fall and settle, with collisions being detected and needing to be resolved every frame. Testing shows that only applying a single impulse in this scenario does not produce a stable simulation.

To better support these persistent contacts, a few steps must be taken. The first is to iterate on the impulse generation process; going through each contact, finding the impulse, applying the impulse. Rather than attempting to calculate that impulse analytically, this numerical method is used instead that solves contacts individually on a local level and eventually converges on the global solution [10] [14].

The previous step prevents the resting bodies from becoming a compressed, jumbled mess, but still does not allow for desirable structures such as stacks or pyramids of shapes [10]. The reason for this is that the solver must start from scratch in finding the correct impulses for all contacts every step [10]. As well as this, the solver is currently not allowed to correct “bad” impulses as it can only generate repulsive impulses [10].

To solve these issues, a solution known as “accumulated impulses” is employed [10]. Full details can be found in Catto’s GDC 2006 presentation [10], but to summarise: At the end of each step, the final calculated (accumulated) impulse for each contact is clamped (To prevent attractive impulses) and saved. The solver can then, at the beginning of the next step, pre-emptively apply that saved impulse to each contact before starting the iteration process. This allows the solver to re-use all of the calculation that has occurred previously and therefore generate a more stable solution. As well as this, due to the clamping that occurs on the saved accumulated impulse, the solver is now allowed to apply both repulsive and attractive impulses. This allows it to recover from any errors that may get propagated through a system of resting bodies.

While it was not implemented, the system used for solving contact constraints using sequential accumulated impulses can be used to solve other types of constraints, for example joints. See [Catt] for details.

G. Particle Systems

The other types of physically simulated objects rely on the simulation of constrained particles to form a larger dynamic system.

Particles consist of the same linear quantities as rigid bodies (Position, Velocity and Acceleration) as well as a mass. They are simulated using a scheme called Verlet integration. In the context of particle simulations, Verlet integration can be expressed as [14]:

\[
r_t + \Delta t = r_t + (r_t - r_{t-\Delta t}) + (a_t \times \Delta t^2)
\]  

(10)

Note the lack of any explicit velocity term. Instead, momentum is preserved using the backwards difference between the current position of the particle and the previous position of the particle. This means the velocity is better kept in sync with the position, which helps stability [14].

The inferred velocity allows positions to be modified and the momentum of the particle to be automatically changed in response. This property allows for position constraints to be applied to particles, which allows for certain behaviours to emerge as is explored in later sections.

H. Interaction with other Particles

Particles can be influenced by other particles through the use of constraints. Constraints limit the motion between pairs of particles so that groups of particles form more cohesive systems and can cause interesting interactions between large groups of particles [15].
Particles can adjust their position based upon their location relative to other particles. This is known as relaxation or a position constraint [14].

A simple position constraint that is applied to all particles is one that prevents particles from occupying the same physical space. If the distance between a pair of particles becomes too small, they can be displaced to maintain a minimum distance [14]. This kind of constraint is used to simulate collision between soft bodies and interactions between soft bodies and fluids.

Particles can also be constrained by forces, which are the other element taken into account during the motion simulation.

I. Architecture

Particles are not simulated on a one-by-one basis; they all belong to some kind of container system. The two systems presented here are soft bodies and fluids so particles will either be contained within one of those two. The container system defines exactly how the contained particles interact with all other particles.

In theory, each particle could be processed in isolation and interact with every particle in an identical way. In practice, however, it is advantageous to treat these interactions on a per-system basis to allow for stability tuning. For example, a soft body’s particles should not perform the collision constraint with particles of the same body as this can cause stability issues when the body is deformed. It should, however, perform the collision constraint with other soft bodies and all fluids.

All particles, regardless of the system to which they belong, can be run through the same simulation pipeline, post-custom tuning. This unification greatly simplifies the architecture of the engine.

J. Soft Bodies

Soft bodies are implemented as a system of particles with constraints between those particles to allow the body to restore to its “resting configuration” after deformation [16].

To initialise a soft body, a polygonal hull is provided and filled with particles. The shape is filled by rasterising the shape to a grid of cells. Particles are placed in the centre of these cells. The size of the cells determines the particle density of the resulting shape. Higher density shapes will look better at the cost of simulation performance.

The key constraint at work within a soft body is a spring-like constraint that gives the body its deformable, bouncy qualities [16].

The internal spring constraint is implemented as a position constraint between two particles:

\[ |r_0 - r_1| + (k \times r_{\text{displacement}}) = l_{\text{resting}} \]  \hspace{1cm} (11)

Where \( l_{\text{resting}} \) is the relaxed length of the spring (Which is defined as the distance between the two particles in the resting configuration of the body). \( r_{\text{displacement}} \) is the displacement vector and \( k \) is a constant for the constraint that determines the stiffness of the spring behaviour. A \( k \) value of 1 produces a very rigid shape where a \( k \) value close to zero produces a shape more giving to deformation.

This constraint is derived from Hooke’s law for determining the compression forces for a spring:

\[ F = -kx \]  \hspace{1cm} (12)

Where \( x \) is the displacement of the spring from the resting length.

The displacement required can be calculated as the difference between the current distance between two particles and the resting length. The particles are displaced along a vector that is opposite in direction to the vector from one particle to the other. The amount of displacement each particle experiences is half of the total calculated displacement. This halving derives from the fact that all particles in the body are of the same mass and therefore the hypothetical forces exerted on each particle from other particles are equal due to \( F = ma \).

Constraints are placed between neighbouring particles (i.e. all particles within some resting configuration distance) in all directions.

To help with the stability of the particle configuration (i.e. its ability to retain or return to its resting shape), constraints can also be placed for particles further away than immediate neighbours [16]. These constraints have larger resting lengths and lower stiffness coefficients so as to not make the structure too rigid. The extra constraints prevent the body from collapsing in on itself under more intense deformation. The amount of deformation supported can be increased by adding more constraints to more distant particles.

It was found during this implementation that the stability of the system deteriorates as more constraints are added over a certain distance. The effectiveness of this scheme depends on the size, shape and particle density of the body. Tuning is required to find a good balance.

These constraints are in addition to the minimum distance collision constraint common to all particles as described previously.
**K. Fluid Dynamics**

Fluids are also implemented as a system of particles, much like soft bodies.

Unlike soft bodies, however, fluids have no concept of a shape or resting configuration and the fluid particle motion is constrained via forces rather than strict position constraints.

In fluid dynamics, fluids are often represented as continuous fields of various quantities [17]. The quantities of interest for this project are density, pressure and velocity.

Continuous fields are not computationally feasible in real-time and so must be discretised in some way for example, a grid of cells [17].

This implementation, however, uses a particle-based discretization that can be incorporated into the existing particle simulation system easily. The particle-based approach has an advantage in that the mass of the fluid is easily conserved since the mass of a fluid is simply the sum of its component particles. This is advantageous as other methods of discretization require explicit management of mass conservation, which adds extra complexity [18] [19].

Sampling a continuous quantity field for each particle is done using a technique called Smoothed-Particle Hydrodynamics (SPH) [20]. This method provides an algorithm for finding any property \( A \) at any given position \( r \) [20]:

\[
A(r) = \sum_j m_j \frac{A_j}{\rho_j} W(|r - r_j|, h) \tag{13}
\]

Where \( m_j \) is the mass of particle \( j \), \( A_j \) is the value of the quantity at particle \( j \), \( \rho_j \) is the density of the particle \( j \), and \( W \) is a smoothing kernel function taking the distance from the position \( r \) to particle \( j \) and a smoothing length \( h \) as inputs [19]. The smoothing length essentially defines the resolution of the particle system; systems of higher particle density can have a smaller smoothing length as particles will be situated at shorter distances away.

The algorithm essentially states that the property \( A \) at position \( r \) is the sum of the contribution of all neighbouring particles scaled by a function of the distance to that particle.

The smoothing function \( W \) generally describes a curve that defines a smooth falloff from a distance of zero to some finite distance. An optimisation can be implemented if the \( W \) function outputs zero at a distance of \( 2h \) or further [21]. This allows the simulation to only consider the contribution of particles within a radius of \( 2h \). This is a considerable optimisation in systems of thousands of relatively sparsely distributed particles. This optimisation is similar to the spatial partitioning scheme described previously for rigid body collisions.

The full details of determining particle quantities can be found in [19].

The constraint forces that dictate the motion of the fluid particles are derived from the Navier-Stokes equations for fluids [17]:

\[
\frac{\rho \text{d}v}{\text{d}t} = -\nabla p + \rho g + \mu \nabla^2 v \tag{14}
\]

This can be rearranged to reveal a more familiar looking form (Analogous to \( a = F / m \)):

\[
a = -\nabla p + \rho g + \mu \nabla^2 v \tag{15}
\]

The expressions being divided by density are the internal fluid forces pressure and viscosity along with the external force of gravity [19].

Another force that can be applied to particles is the surface tension force described in [22]. This allows for the formation of cohesive fluid structures such as sheets, filaments and drops [21]. This force also helps maintain the separation of fluids in a multi-phase system [22]. It is important to only allow this force to affect particles of the same density.

These quantities and forces are calculated every frame for each particle. The forces are applied to the particles and the particles are then advanced.

Fluids can be tuned by adjusting density (particle mass) and viscosity variables.

**L. The Simulator**

The simulator is the high-level controller of all physical simulation systems and additionally acts as the bridge between the two otherwise separated systems. It handles advancing rigid bodies, resolving rigid body collision, advancing particles, constraining particles and handling interactions between particles and rigid bodies.

**M. Interaction between Rigid Bodies and Particles**

All particles can interact with rigid bodies using a simple detection and displacement system, which is extended from a description for interactions between rigid bodies and fluids found in [21]. This system requires no modification of either the rigid body system or the particle simulation system and instead can be implemented at a higher level and make use of the existing interfaces.

The first step is to determine which particles lie within the polygon [21]. This is done using an idea known as Jordan Curve Theorem, which states that for any point inside an arbitrary polygon, a horizontal line from that point to the outer bounds of the polygon will intersect an odd number of the
polygon edges [23]. This is used to determine if a point is inside or outside a polygon.

Next it is necessary to project the position of the particle on to each edge of the polygon to find potential displacement locations:

\[
r_{\text{displacement}} = e_{\text{start}} + \left( (r_{\text{particle}} - e_{\text{start}}) \cdot e_{\text{direction}} \right) \times e_{\text{direction}}
\]

(15)

The displacement location to use is the one with the shortest length. Moving the particle to the selected displacement location solves the interaction for the particle [21].

Each displaced particle applies an impulse to the rigid body using the method described in the Rigid Body section [21][13].

The result is particles that correctly push rigid bodies with a magnitude proportional to their mass while also giving way to the bodies that are denser than the group of displaced particles.

N. Further Work and Issues

Joints between bodies are one aspect that is missing from this implementation. Erin Catto provides details on extending the impulse-based contact solver into a more general constraint solver that can then be used for joints [10].

Further optimisations could be applied to the particle systems. Most notably, a similar multi-threaded scheme as is seen in the rigid body collision manager could be applied to a partitioned system of particles since particle constraints tend to work over a limited, local distance.

There exists an issue with the soft body particle collision constraint. The issue is that the minimum distance constraint only maintains the resting distance between particles for collision purposes. This means that if the body is stretched and the distance between particles increases, particles from other systems can penetrate and exist inside the soft body. This can be rectified by having the minimum distance constraint take into account the current stretched distance between particles of the same body.

Fluids can be hard to tune and tweak. A large number of parameters need to be correctly balanced to prevent the fluid simulation from exploding or destabilising. The more difficult parameters to balance are the particle density/fluid density/smoothing length balance. Further research into finding ways to keep fluids more stable regardless of these parameters has been done, notably by Clavet and his double-density relaxation scheme [21].

IV. Conclusion

This paper has presented the details behind the implementation of a 2D physics engine that provides simulation and interactions between rigid bodies, soft bodies and fluids.

By constructing two systems for handling rigid and deformable objects separately, those two kinds of systems can handle simulating and interactions between objects of the same type (Rigid and rigid or deformable and deformable).

As well as this, a method for allowing these two systems to interact by making use of functionality already available from both systems (And therefore without modifying either system) is provided which allows new and interesting behaviours to emerge.

Some optimisations are also presented to improve the performance of the engine and the stability of the simulation.

All of this together provides a reasonably robust and scalable engine that is well architected and easy to understand.

REFERENCES


