Event based collision detection

Benjamin Schnieders

15.07.2012

Abstract

This web article presents an even-driven update logic for many spacial simulations, which only updates if certain events are received. It is compared to a common frame-based update system. Using a simple particle collision model, both approaches are presented and occurring errors are demonstrated. With showing which one to prefer in certain situations, the article is concluded.

1 Introduction

Many systems in scientific applications are based on entities interacting to spatially close entities, from traffic systems, where close cars have an effect on a particular driver's decision, to the simple particle simulation used for this article as demonstration. Such systems shall usually detect if a collision occurs, and react on is in a specific manner. Many systems need the information when a collision will happen next, for example to avoid it, or, in certain cases, to enforce that it happens. Especially for real-time systems, often simple approaches are used to detect collisions, although they lack precision. A common way to simply detect a collision is repeatedly test all entities whether they touch. This test and an appropriate reaction is done every simulation update or frame, and thus is a frame-based approach. Another way to detect collisions is to predict for all entities when the next collision will happen, and only handle these collisions at the moment they occur. As a prediction generates events to be consumed in the future, this is a predictive- and event-based approach.

1.1 About this article

This is a web-based article, with a web-based demo application linked to it. This is the more printer-friendly, downloadable PDF version, which also should not suffer from displaying problems. However, the hyperlinks to the connected website will most probably not work.

1.2 Test Model

For testing and demonstration, a two dimensional particle system is set up, in which distinct particles move, colliding with another and deflecting off the walls, each collision being fully elastic. See this page for an example. Be sure to turn on JavaScript, as it powers the demo. Browsers lacking support for recent CSS versions may have rendering difficulties, in these cases, switching to a standard compliant browser may be considered.

1.2.1 Model Parameters

The testing framework is controlled by certain parameters. These will determine the outcome - depending on the simulation updating approach, as not all approaches are deterministic.

- The size of the testing board and the particles. This determines to a certain extend, how many collisions there will be in a given time frame. A larger board or smaller particles will lead to fewer collisions.
- The initial particle speed. Faster particles will generally lead to more frequent collisions.
- The number of particles. An increase in particle count will lead to a quadratic increase of collisions. The initial particle positions are chosen randomly, collisions may occur in the initial layout.

These are the intrinsic parameters of the particle model itself. Other interesting properties can be derived from the model parameters, for example the particle density: This value describes how much area of the board is filled with particles, and can be calculated from the board size and the cumulative area of all particles:

$$\rho = \frac{\sum \pi \cdot particle.radius^2}{board.x \cdot board.y}$$

1.2.2 Validation

The various physical laws underlying the collisions should be respected with appropriate numerical precision. As the borders of simulation are reflecting and thus changing the impulse vector (i.e., the board does not wrap around) keeping track of the global impulse vector and validating its constancy is not possible. However, checking the total energy of our system for consistency is easily feasible. As our system only contains kinetic energy, the formula is:

$$\sum_p \frac{p.mass \cdot p.\vec{velocity}^2}{2} = const$$

1.3 Simulation

Simulation is done on multiple levels. There is graphical feedback for the user, mathematical proof of correctness and modeling the collisions itself.

1.3.1 Graphical output

The rectangular board is depicted as filled and bordered HTML <DIV>, where one world unit is represented as one pixel. Particles are presented as HTML <DIV>s, with corners properly rounded to display a circle. Each particle is simply drawn at the position it currently is. Graphical feedback can be turned off or its update rate may be reduced, if it should compete with the collisions physics for CPU time.

1.3.2 Updating

An update is generally transforming a former representation of the model to a recent one. For this model, a state is the board with all particles and their velocities. Transforming a past state to a recent one requires to apply each particle's velocity, respecting the time that passes. All collisions that will happen in the given time period, whether among particles or a particle and the border, have to be taken into account.

2 Frame-Based updating

Frame based updating refers to a strategy to do at most one update per frame, preferably less. A frame length is usually imposed by the graphics update. For example, graphics rendering at 100hz would lead to one hundred simulation updates as well, which also means that the time passing between two frames should not exceed 10 milliseconds.

A frame-based update method will always calculate an up-to-date state, which is pretty convenient, as this updated state can then immediately be drawn by the graphics engine. The integration of a physics or gameplay simulation is then simply done at an appropriate place in the main loop. However, it relocates events occurring between two frames o the end of the frame. Thus, the frame based update method relies on the assumption that, by picking an update rate high enough, the error resulting from making a calculation at maximum $\frac{1}{update\ frequency}$ seconds too late, is neglectable.

To eliminate errors, infinitesimal small time steps between updates are needed, which are not practical, as selecting a very high update rate will lead to a heavily increased workload. For optimal performance, one would want to update as seldom as possible. Adapting the rate to a finer resolution when needed gains accuracy, but detecting and ranking the error that was introduced in an update is not trivial. Also, in a real-time simulation, one can not go back in time and enhance resolution during times of fewer updates.

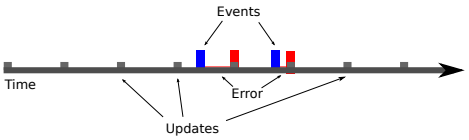


Figure 1: Frame-based updating: *Updates are done at a fixed rate, events such as collisions may happen any time. The resulting event processing delay leads to errors in the simulation.*

To compensate for that in real-time applications, one may be tempted to project the simulation one update into the future. This again only works if the time between two updates is constant ant known, which is usually not the case, as real-time applications commonly try to perform as much work as possible, thus updating without breaks. Even with break, shifted machine load may lead to delays that were not compensated for.

2.1 Occurring Errors

As updates are done after fixed time periods, multiple errors are introduced to the system. Particles may penetrate the wall or another particle before actually colliding, making a simulation run unrepeatable. Particles may also collide for

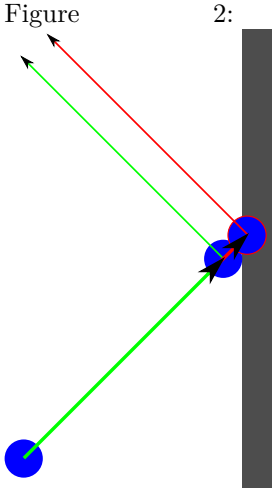
more than an update, if the penetration can not be resolved within an update. The last type of error is a lack of feedback - a particle is updated once an update, but may experience a collision after updating it, but caused by the particle itself.

2.1.1 Wall and particle penetration error

A particle that is detected to be hitting a wall during the current update is with all probability not actually touching the wall, but already penetrating it to a certain extent. This difference in time between the actual collision and the update time leads to an abridged movement, as shown in image 2.1 Although this defect does not influence the overall energy or impulse, it destroys reproducibility, unless the update intervals can be exactly reconstructed.

If two particles collide, the situation is comparable to wall penetration: A collision is detected when the entities are overlapping already. If the penetration depth is less than the diameter of the particles, the resulting impulse can still be correctly calculated, but the origin of movement differs, as in the wall penetration example. Image 2.1.1 shows this in detail. If the penetration depth is larger than the diameter, a collision will be detected, but as both particles already move away from another, no impulse can be generated, and the calculation is most likely not carried out. With very high velocities, small particles or a low update rate, particles might move through another without a collision at all. As a very unlikely case, two particles might occupy the exact same position, from which one cannot calculate correctly the resulting impulse as well.

Figure 2:

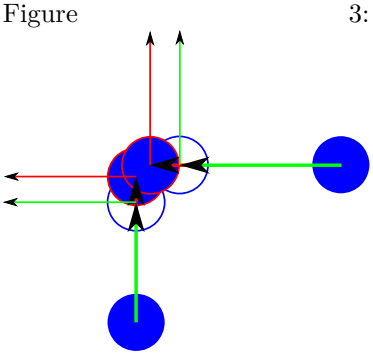


Wall Penetration: *The green arrow describes correct deflection off the wall, the red arrow shows the effect of wall penetration.*

A prolonged penetration does, depending on the implementation, not necessarily lead to an increased error. As particles colliding will exchange impulses in any case, their resulting relative velocities will not indicate a collision again in another update.

2.1.2 "Lacking Feedback" error

Figure 3:



Particle penetration: *Particles colliding that penetrate another will generate different resulting velocity vectors from correctly colliding particles.*

Collisions between two particles can be calculated quite simply. Due to the delayed reaction of the frame-based model, situations may happen that multiple particles collide simultaneously. There is no direct mathematical model describing the collision of multiple objects simultaneously, instead, all individual two particle collisions are calculated with no time passing until all of them are resolved.

The frame-based update method though would loose a big advantage, being that each update it always has to process a fixed load only, if one would implement a multiple object

collision into it. In the normal case, with no multiple objects collisions, the load is n^2each update, as all n objects are checked against all other objects for a collision, and the resulting impulse is generated. If, in one update, three particles *may* collide, the load immediately rises to $2 \cdot n^2$, as during this update all particles are checked against each other, impulses are generated, and then all objects have to be checked again in the same update. For$m$-ary collisions, the load may rise to $m \cdot n^2$. One can construct cases that cannot resolve, for example a row of particles jammed between the walls, with one particle having an impulse against one of the others. A setup like this is constantly colliding, and no matter how many follow-up collisions are regarded, no solution can be found. A loop detection algorithm may be used to see if any particle receives the same impulse twice with no time in between, issue a warning and stop computation.

One can reduce the average load by checking and updating the objects more intelligently, keeping an additional stack of entities to process: Knowing that the collision property is symmetric, each object i is only checked once against all others j, where i is immediately processed if a collision was detected, and j is pushed onto the stack of objects still to be processed. Keeping $j > i$, using a little more memory can reduce the complexity to $\frac{n^2}{2} + c$, with c being the number of collisions each update. For multiple object collisions, we adapt the algorithm slightly, so that each update we process each object i against all objects j, then process all impulses for the collided objects c on the stack, and finally check all objects (and, if necessary, process) c against all other objects for a new collision.

3 Predictive updating

Predictive update is a method that decomposes the simulation into linear interpolatable phases and events separating them. For the particles model, all movement is linear, and each collision is an event. The key point of said algorithm is to calculate when the next event will happen, rather than updating each frame and checking if an event happened in between the last frames. This calculation is quite costly compared to the simple collision check, but only has to be done once per collision.[1]

For all particles, the algorithm checks whether they will collide and whether this collision will take place before any other collision, including particles colliding with walls. From the current time to the time of the first collision, particle movement can be linearly interpolated with no extra overhead, knowing that no collisions can happen during this time. These updates may be called lazy updates, because the prediction prevented the need to do costly calculations at this time. See Image 3 for details. During lazy updates, the algorithm may already calculate the outcome of the collision and the time of the next collision afterwards.

3.1 Collision time calculation

To accurately calculate when the next collision will occur, one first translates the movement of two particles into a relative movement to one particle. Taking

[1] Whereas frame-based updating will faithfully check hundreds of entities against each other for collisions each frame without one particle moving at all.

the dot product of the normalized relative velocity and the position difference yields the time in seconds when the two particles are closest to another. From that time and the radii of the particles one can easily calculate the exact time and position of the collision. The whole algorithm is described as pseudo code in Algorithm1.

Algorithm 1 Collision Prediction Algorithm

```
foreach Particle p1, Particle p2, p1 ≠ p2
{
    Δp̃os = p2.pos − p1.pos
    Ṽrel = p1.velocity − p2.velocity
    if(‖Ṽrel‖ ≤ ϵ)
        continue; //parallel or no movement
    V̂rel = Ṽrel / ‖Ṽrel‖
    Distance_ClosestApproach = V̂rel · Δp̃os
    if(Distance_ClosestApproach ≤ 0)
        continue; //closest encounter lies in the past
    if(‖Δp̃os‖² − Distance²_ClosestApproach ≥ (p1.radius + p2.radius)²)

        continue; //at closest encounter, farther away than summed radii
    Distance²_Collision = (p1.radius + p2.radius)² − Distance²_ClosestApproach
    timeToCollision = Distance_Collision / ‖Ṽrel‖
}
```

3.2 Updating

A big advantage of the predictive method is, that the collision time calculation does not have to be done every frame. It can be delayed until the time of the next collision. For the movement update one has two possibilities - either, during the prediction phase one calculates the position of all particles at the collision time and then interpolates linearly between them until the collision takes place, or, as a memory saving technique, one simply adds up the velocity fractionally every frame. The latter method may cause smaller errors in the magnitude of $\frac{1sec}{\#updates} \cdot speed$ - as the collision can be earliest handled during the frame after the event took place, particles may move farther than actually planned. It is advised to estimate the highest number of collisions per movement update in-

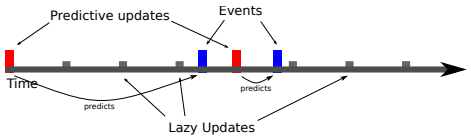


Figure 4: PREDICTIVE UPDATING: *Updates are done at a fixed rate, events such as collisions may happen any time. When these events will happen is predicted as early as possible; after that, lazy updates can be done until the next prediction is needed. The effect of an event is applied in retrospect at exactly the time it occurred.*

terval (or measure it) and buffer at least as many steps in advance; this way, all movement updates should be able to resort to exact interpolated positions.

3.3 Quality

The predictive update method tries to overcome all the problems presented for the frame based updating method. The algorithm is designed to allow reproducibility and to circumvent realtime simulation errors. In the following, the actual quality of the method shall be investigated.

3.3.1 Errors

Predictive update avoids the errors made by the real-time frame based update strategy by simulating ahead - simulation takes place in the future, so the real-time service just has to interpolate the last saved state of the world until now. Each event may be handled at precisely the time it occurs. In order to keep real-time output error-free, one has to ensure that the simulation will always keep being ahead of the real-time display, otherwise, displayed particles may penetrate another or leave the board, if interpolation from two calculated points becomes extrapolation of the last result. This behavior may still be preferred over a lagging or hanging simulation, as it is only visually flawed, not mathematically, as long as the extrapolation happens only visually, i.e., the events are not postponed to the next frame. Keeping this in mind when implementing movement updates, one can indeed easily create a simulation framework that allows for reproducible simulation runs.

3.3.2 Calculation time:

The predictive update mechanism is more computationally expensive as the frame base model. The demonstration implementation of the predictive update mechanism has about twice the lines of code of the frame based method. This additional computational expense has to be compensated by the lower updating rate the prediction allows. Nevertheless, both algorithms involve a all-vs-all-part, making them essentially of $O(n^2)$ complexity. Whether the predictive update mechanism can replace the real-time frame-based strategy is dependent on the number of events per second (the collision rate C), the update rate R needed for frame-based updating and $\Delta complexity$, the factor of how much slower one update step including collision prediction is further influence the equation.

- Frame-based: $complexity = \frac{\#particles^2}{2} \cdot R$
- Prediction-based: $complexity = \frac{\#particles^2}{2} \cdot C \cdot \Delta complexity$

Whether the prediction-based algorithm should be used is now determined by:

$$prefer\,prediction = \begin{cases} true & if\ \frac{C \cdot \Delta complexity}{R} \leq 1 \\ false & if\ \frac{C \cdot \Delta complexity}{R} \gg 1 \\ depends & else \end{cases}$$

To be able to decide between the two methods, one first has to decide, how much error the simulated system may compensate. If no error is wished, only

the prediction based algorithm is applicable. If particle penetration artifacts will not bother the simulation, but particles should not tunnel though another, one can calculate the needed update rate for frame-based updating from the maximum expected velocity and the minimal radius:

$$R_{min} = \frac{V_{max}}{Radius_{min}}$$

If an acceptable rate is found, the next step is to estimate the number of collisions or events in the simulation. For the example setup, the number of collisions is expected to rise quadratic with the density. These two factors yield the requirements for $\Delta complexity$. If it is impossible to construct the prediction algorithm with these complexity constraints[2], a predicting algorithm is inapplicable.

In the case that the particle number is high, but the simulation area or particle size may be variable, one can tune these parameters in a way to lower the particle density, and with it, quadratically reduce the collision rate.

4 Conclusion

Besides getting rid of the introduced error by the frame based updating method, the calculation time scales much better with the number of particles for the prediction-based model - as long as the particle density stays the same. Cross-checking n particles for collision takes n^2 steps, the calculation expense rises quadratically with the number of particles only; particle density has no immediate effect. For prediction, the number of collisions is relevant, the quadratic part only has to be evaluated if a collision actually happens. This means, especially for sparse, huge worlds there can be quite a runtime improvement. The number of collisions rises, in general, quadratically with the number of particles, so the complexity is divided by the square root of the number of particles.

This means, if the density is constant, the runtime of the predicting algorithm rises with $n^{3/2}$, compared to n^2 of the frame based algorithm. Both algorithms can equally gain speedup from optimized data structures, because those reduce the computational complexity during the all-vs-all, n^2 operation.

To be able to provide some confidence that the prediction stays ahead of the simulation, one can sacrifice memory to buffer more than one world state ahead. Especially in situations where multiple particles collide, no time passes between two predictions, so the simulation can easily catch up with the prediction. If memory is an issue, one can only save the changes to the previous state of the board, and, during simulation, apply only the changes rather than switching the whole board.

For simulation setups, where events are not easily predictable, such as real-time simulations with user input, the prediction algorithm is generally unsuitable. It may still be used, if the unpredictable events are limited in various ways. A user being able to move a particle for above model does not inherently mean that only frame based update is applicable. If the user movement speed

[2] $\Delta complexity$ is going to be larger than one, unless one finds optimizations applicable for prediction, but not in-time cross checking. Saving all future collision timesteps and only updating those being influenced by the first one might be such an optimization.

is limited, one can force repeated [3]updates, and increase the radius of the user controlled particle by the distance it can travel between two forced updates. This way, using only a little overhead, one can still benefit from all advantages of the prediction based approach.

4.1 Further Research

Further research can be done to improve performance, for example by regarding the last available results for the next update where applicable. Researching ways for efficient parallelization of the prediction phase is another performance improving topic.

Systems with a high rate of user or otherwise unpredictable input should be inspected, and methods to reduce calculation overhead because of these may be found. A simple method may be to transform the user input into a representation that stays constant for an amount of time, eliminating the need to update during this time. Other ways might be limiting the amount of change that can be done in a certain amount of time, and ensuring consistency between these phases.

Only some systems are linearly predictable. Even simple particle simulations become impossible to predict linearly if quadratic terms like gravity or electromagnetic forces are involved. Simulations may include entities that move along certain paths or inside corridors. These specific movement patterns may be predicted with specially designed predictors.

[3]Updates with a much lower rate than frame-based, like one update per second.