

Re: Floating point errors in collision routines

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.cpp/2004-12/0240.html

From: John Nagle (nagle_at_animats.com)

Date: 12/02/04

Date: Thu, 02 Dec 2004 18:28:09 GMT

Trying to converge a floating point calculation all the way to zero is fundamentally futile. You get characteristic underflow before you get zero.

In general, bear in mind that subtracting two large numbers to get a tiny result is inherently prone to roundoff error. You must design your algorithms to avoid that.

Some collision detection algorithms, especially early versions of GJK, are badly behaved when two faces are very close to parallel. Of course, if you're doing physics right, objects in contact come to rest in face-parallel situations, so the near-parallel situation gets explored very thoroughly. Newer collision engines, like (I think) Gino's SOLID, get it right.

Developing collision algorithms that don't have floating point precision problems yet don't "cheat" on contact analysis is hard, but quite possible. See my US patent #6,067,096, which covers the first "ragdoll physics" that worked. It's necessary to be very careful about the limitations of floating point. But once you get it right, it works very well.

As I usually tell people, either buy a physics engine (probably Havok), figure out some way to make gameplay work with lousy physics, or expect to spend a few years on the problem.

John Nagle
Animats

Dave wrote:

> *Hi folks,*
>
> *I am trying to develop a routine that will handle sphere-sphere and*
> *sphere-triangle collisions and interactions. My aim is to develop a*

comp.lang.c++. Re: Floating point errors in collision routines

- > *quake style collision engine where a player can interact with a rich*
- > *3D environment. Seem to be 90% of the way there! My problems are*
- > *related to calculations where the result tends to zero (or another*
- > *defined limit.)*
- >
- > *Have loads of cases where this kind of interaction occurs but this one*
- > *is as tricky (?) as any...*
- >
- > *When a moving sphere approaches a triangulated plane I am using a*
- > *sphere-plane collision routine to see if a collision will occur. All*
- > *goes well until I start to have near parallel interactions (e.g. when*
- > *the sphere has sufficient energy to collide with, and then slide along*
- > *the plane.) During the next pass of the collision routine the velocity*
- > *of the sphere is perpendicular (in a single plane) to the face normal*
- > *of the previous colliding plane. A dot product between the velocity of*
- > *the sphere and the normal of the colliding plane should be zero. It's*
- > *close to zero but how close is close enough to warrant no further*
- > *collision? A "need to collide" case occurs where the sphere is just*
- > *above the plane with a velocity almost parallel to the plane BUT just*
- > *on a collision course. The above dot product will be very close to*
- > *zero again, but this time there should be a collision with this plane!*
- >
- > *Since I need to account for glancing off faces, approaching faces at*
- > *right angles, interacting with steps, falling off edges etc the number*
- > *of vector operations to determine the final position and velocity of*
- > *the sphere need to be as accurate as possible.*
- >
- > *I guess my question (sorry for the waffle) is how to minimise rounding*
- > *errors in floating point operations?*
- >
- > *So far I've tried to account for the errors (or eliminate them) by:*
- >
- > *- Using double precision decelerations and calculations*
- > *- Using the epsilon error on the variables to account for known*
- > *internal storage errors*
- >
- > *Unable to eliminate or account for the errors I then set about trying*
- > *to use flags to ignore colliding objects from future collision during*
- > *the current recursive collision routine call. Helped in some cases so*
- > *started to try to account for all near critical iterations using*
- > *flags.*
- >
- > *Getting close to a general solution now but my once clear simple*
- > *routine has turned into a logical nightmare!*
- >
- > *When working with numbers where accuracy is paramount should I be*
- > *using a different approach? I've considered the following:*
- >
- > *- Using a quantised system where all values have to fall along a*
- > *virtual 3D grid*
- > *- Using sin and cos tables to support the above goal of quantisation*

comp.lang.c++: Re: Floating point errors in collision routines

- > – *Using integer mathematics*
- > – *Use a high precision custom storage class for calculations (speed is off the essence so might not be an option?)*
- >
- > *Perhaps I am worrying too much! My current "effort" seems to handle interactions in a similar, if much more shaky, way to Unreal Tournament 2004. Kind of clunky at times, stops occasionally and seems to have problems jumping on steps!*
- >
- > *I'm working on Windows 2000 using Visual Studio 6 and .net enterprise architect 2003 (c++). It's a DirectX application if that makes any difference?*
- >
- > *Any advice would be really appreciated.*
- >
- > *Many thanks,*
- >
- > *Dave*