Minkowski Portal Refinement and Speculative Contacts in Box2D

New Methods in Rigid Body Simulation

Joshua Michael Osborn Newth

San Jose State University

May 2013

Abstract

A physics rigid body simulation engine is a fundamental component of today's videogames. There have been recent advancements in the fields of collision detection and collision resolution with supposed performance gains. In this paper we evaluate Minkowski Portal Refinement,a new collision detection routine against GJK, the current standard. We also evaluate the speculative contacts technique for performance and behavior against Box2D's "conservative advancement" continuous collision detection approach.

Table of Contents

List of Figures

Acknowledgements

The author would like to thank Professor Pollett for his instruction and guidance in this thesis and at SJSU. The author would also like to thank his defense committee, Professors Mak and Teoh, for their participation in this defense. The author would like to thank Erin Catto for providing the physics simulation community with *Box2D*. Finally, the author would like to thank his wife Michelle for her companionship and support through the long nights and weekends working on this research project.

Ad majorem Dei gloriam.

1. Introduction

A modern videogame is a technological marvel. In the author's experience, there are few other fields that are the confluence of so many human endeavors, from engineering, math and science all the way to art. This paper concerns one important component of most videogames, the physics engine.

A physics engine provides the simulation of forces, colliding object geometry, user inputs, and gameplay mechanics in a videogame. If an angry bird soars through the air to cause the physically plausible destruction of rickety buildings and their ursine inhabitants, it is a physics engine that makes this possible.

The engine must insure that bodies do not interpenetrate and that collisions create believable changes to the motion of bodies. In order to achieve this, a physics engine must perform *collision detection*, to determine when two entities are about to touch, are touching, or are penetrating, and *collision response*, to prevent motion in the direction of penetration and to correct any penetration that has already occurred. A sophisticated engine must go further and support "continuous collision detection" which addresses the challenge of simulating extremely short-lived contact events that might otherwise be missed in the coarse timestepping of the discrete simulation.

This paper attempts to explore new techniques in the areas of collision detection and continuous collision detection. The first is through the collision detection routine called Minkowski Portal Refinement and the second is through speculative contacts, a

useful approximation that presents some important performance gains over classical approaches.

2. Background of Research

Physics engines are too broad a topic for a single paper, and the reader is directed to the wealth of resources available: (Eberly), (Hecker), and (Catto). However, this paper provides an introduction to the current state of rigid body simulation to prepare the newcomer for the research discussion.

While *accurate* simulation of the interplay of forces and geometry in the motion of bodies in space is perhaps beyond computational reach and certainly beyond the realtime requirements of a videogame, physics engines make a set of simplifications that provide believable simulation:

> A. The world is modeled as a collection of rigid, impenetrable solid bodies (typically convex hulls or "meshes" of triangles) and their motion is influenced by the action of external forces, such as gravity, buoyancy, and wind, and the action of constraint forces that couple the motion of two or more bodies, of which the most important are the impulsive forces that prevent interpenetration between colliding bodies. We may further simplify the description by saying that rigid body simulation relies on collision detection, i.e., determining when and where contact between two bodies occur, and collision response, i.e. determining a believable approximate response of each body to these collisions.

B. Rigid body physics simulations operate in a series of discrete timesteps. At each timestep, forces are applied, velocities are updated, potential collisions between rigid bodies are detected or corrected and finally the rigid bodies are integrated to their new positions, orientations, and velocities.

These two simplifying assumptions are at the heart of all modern videogame engines, and the implementation of systems to enforce the former and cope with the latter make up much of the work of physics simulation. Although there are many interesting areas of research in physics engines, this paper is confined to the issues of collision detection and continuous collision detection.

2.1 Physics Simulation Step

A physics engine operates in a series of timesteps, ideally short enough that the approximation of physical behavior does not deviate too wildly from reality. This matches the structure of videogames, where between each rendering of the current game state to the screen, the world is updated by the fraction of time that has past since the last screen update. If a game is updating the screen 30 times per second, then a single physics timestep must simulate $1/30^{th}$ of a second. This is accomplished in a tight algorithmic loop as shown in Figure 1:

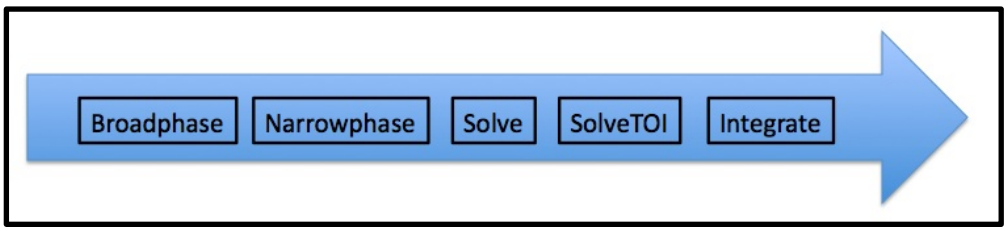

Figure 1. Basic Sequence of a physics step

Each phase is of critical importance to correct operation, and where relevant to this paper these phases will be revisited in greater detail. A brief introduction will help define some frequently used terms.

i. Broadphase Collision Detection – The brute force approach of performing collision detection on each pair of N bodies scales as N*(N-1))/2. Because the algorithms that determine precise collisions can be expensive, the broadphase is an important optimization that culls this list of all possible pairs to only those potentially interacting pairs. The broadphase uses the bodies axial aligned bounding boxes (hereafter "AABB") to quickly identify which pairs have the potential to interact and which may be safely ignored.

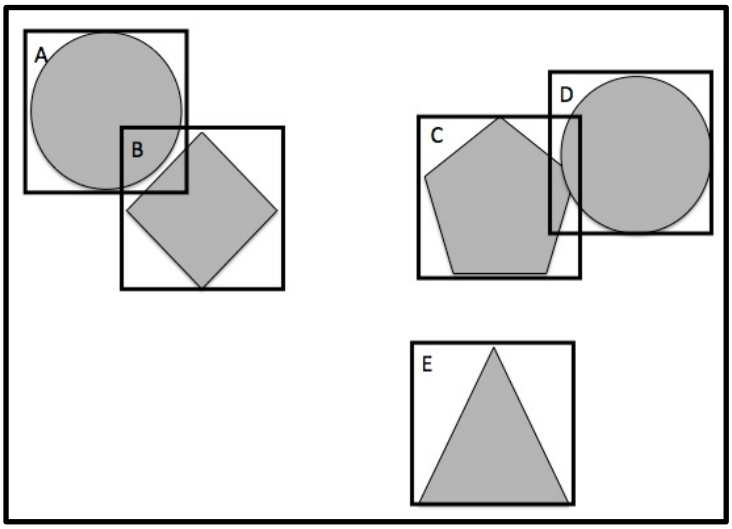

Figure 1. Broadphase and AABBs

In this figure, the broadphase algorithm is able to quickly reject any pair containing E as its AABB does not overlap any other AABB. The pairs {A,B} and {C,D} have overlapping AABBs and must be checked for collision in the narrowphase.

ii. Narrowphase Collision Detection – In this phase, the pairs of AABBs identified in the broadphase are considered in turn and precise identification of "contact points", i.e. exact locations of collision, is made. Contact points are of supreme importance to this paper and will be covered in further detail. In Figure 2 above, both pairs {A, B} and {C,D} would be checked for narrowphase collision. It should be clear that {A,B} are not colliding and would generate no contact points while the pair {C,D} are colliding and would generate a contact point.

iii. Solve – Here, the contact points are used, along with external forces to adjust the velocity of each body in the simulation.

iv. Solve TOI – The term TOI or "time of impact" is of paramount importance and will be covered in greater detail later. For now, it is best to describe this step as where body motion that has been incorrectly computed due to the discrete nature of the simulation will be accounted for and corrected.

v. Integrate – The bodies are moved from their current position to their position at the end of the time step using the velocities computed during the solve step. This process is called "integration" and is typically performed with the simple "forward Euler integration" (Wikipedia) as:

$$P_2 = P_1 + V_2 \Delta t$$

where the P terms are positions, V_2 is the body's velocity and t is the timestep. This is a matrix equation and represents both translational and rotational terms.

It is important to note that this presents the simplest and most intuitive formulation of a physics engine's time step but by no means is this the only possible formulation. Indeed, the most commercially successful and well-known physics engine, the Havok physics engine, performs these same operations but in a different order.

3. Research Overview

Thus armed with a preliminary understanding of a physics engine's operation we are ready to describe this paper's research focus.

3.1 MInkowski Portal Refinement

This paper is an exploration of two recent developments in rigid body simulation. The first is a new collision detection algorithm called Minkowski Portal Refinement. Collision detection is what occurs in the narrowphase component of the physics step, where "contact points" are discovered. The term "contact point" is perhaps inadequate, as for rigid body simulations it is necessary to determine not only the contact location, but also the "contact normal" or "collision normal" (the two terms are interchangeable) and the "penetration depth". The normal is the line of action between two convex hulls. If we translated one of the bodies by the penetration depth away from the other they would no longer penetrate. Because our simulation assumes that bodies are impenetrable, any penetrations are handled as an error requiring subsequent correction.

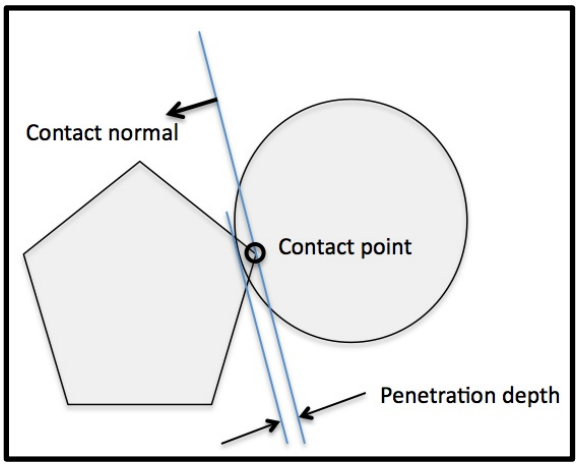

Figure 2. Contact Point, Contact Normal, and Penetration Depth

3.2 Speculative Contacts

The second area of work is the implementation of a new method of continuous collision detection called speculative contacts. This paper will discuss the shortcomings of classical approaches to continuous collision detection and why speculative contacts are an interesting approximation that offer potential performance gains over the current state of the art.

Continuous collision detection is the challenge of simulating continuous, intermediate, transitory events in a series of discrete timesteps, the simplest of which is described as the “bullet through paper” problem. A fast moving bullet traveling towards a stationary thin object can, due to the discrete nature of the simulation, appear to be first on one side at the start of simulation (T=0) and then on the other side at the end of step (T=1).

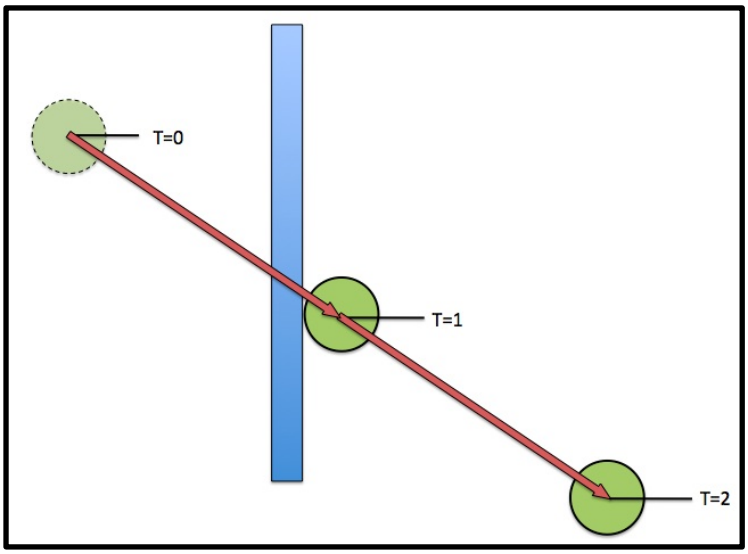

Figure 4. Bullet through Paper

A human observer "knows" that this is wrong because it is understood that bodies sweep from position to position rather than teleporting, but these discrete jumps are legal in a simulation. If the timesteps were extremely small then this behavior could not occur, but this is computationally too expensive. Instead, additional work must be taken at each step in the physics simulation to sweep bodies through their motion and look for these types of transitory violations in the simulation. There are several common methods for detecting the Time of Impact (hereafter "TOI") and correcting the collision response. This correction is performed in the "SolveTOI" phase. The current approach can be computationally expensive and is not readily parallelizable, a serious drawback given the increasingly multicore architectural landscape (Newth).

4. Preliminary Work

The background research for this thesis was intended to the provide the author with a basic understanding of physics engines. To this end, I wrote a discrete iterative rigid body physics engine called Havok2D in honor of, but not to be confused with, the

commercial Havok Physics engine. Subsequent work was conducted to study available engines and source code accessibility for use. This preliminary work, the algorithms used, and the mathematical basis for iterative rigid body solvers will be described in detail to provide context for the subsequent research.

As stated, physics simulation may be greatly simplified to:

1) Collision detection

2) Collision response.

We will consider each area in turn.

4.1 Collision Detection via GJK

Collision detection between arbitrary polygons (or solid volumes, in 3D) is often slow so rigid body physics system enforce an important simplification: All shapes must be convex hulls, for which we offer the intuitive, graphic description, rather than a formal definition. For the arbitrary polygon shown in blue below, the red represents the convex hull for the same set of vertices.

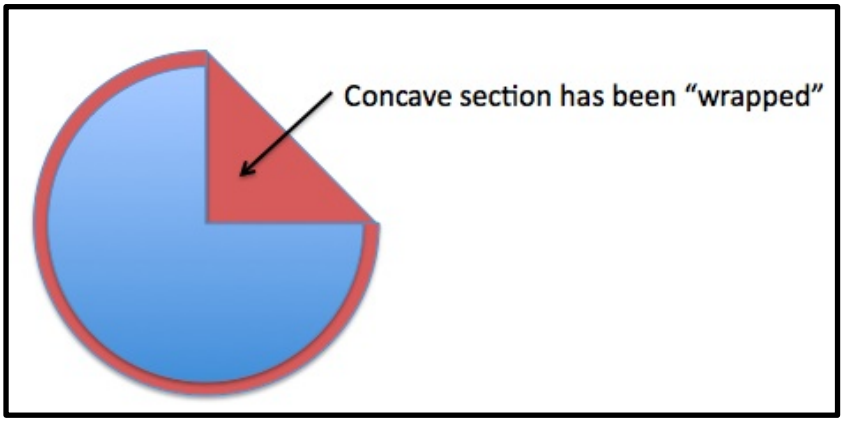

Figure 5. The Convex Hull of a Concave Polygon

Working with convex hulls is of utmost importance to the collision detection algorithms commonly used, including the current workhorse “GJK” its inventors Gilbert, Johnson, and Keerthi (Catto).

4.1.1 Minkowski Difference

An important concept in GJK and other collision detection algorithms is the Minkowski Difference. Using this object we can convert the question “Determine the arbitary point in space where two bodies overlap or draw nearest one another” to the simpler question “Determine if the Minkowski object contains the origin”.

The Minkowski Difference can be thought of as an object where each point is the result of a subtraction of a point in object B from a point in object A (Wikipedia):

$$Minkowski(x,y) = \{(A_x - B_x, A_y - B_y) | (A_x, A_y) \epsilon A \; and \; (B_x, B_y) \epsilon B\}$$

By way of illustration, here are a collection of shapes and their various Minkowski differences:

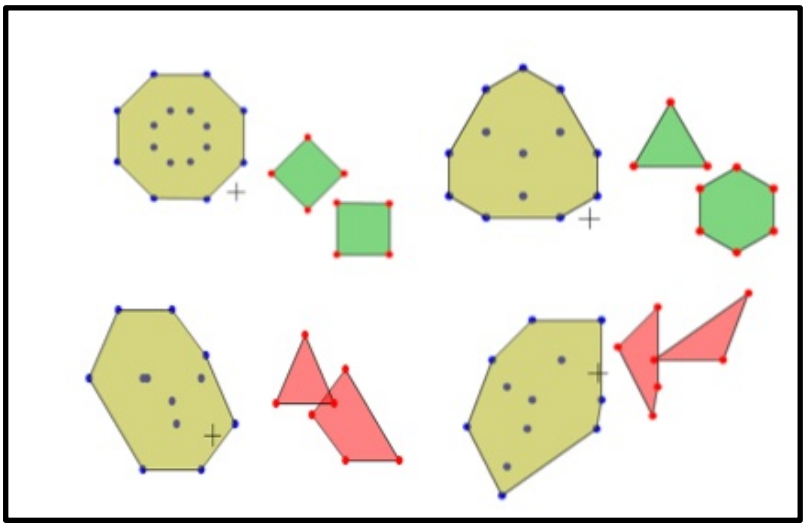

Figure 6. Minkowski Differences

These figures were generated using the applet found at the Physics2D website (Physics2D). Although it is not immediately obvious what the Minkowski shape "should" be for a given pair of shapes, the Minkowski may be explicitly computed in the following manner: Iterate over all the "points" that make up the area of object A. For each point, subtract each of these points, subtract each point that make up object B. Plot the result of these vector subtractions to arrive at the Minkowski object for the pair A and B.

Recall the Minkowski Difference allows us to the restate our problem as "Determine if the Minkowski object contains the origin". It is able to do this because if there is any point on object A that touches object B, then for at least some subset of points in the difference calculation we will compute:

$$Minkowski(x, y) = (A_x - B_x, A_y - B_y) = (0, 0)$$

Using this object, we may provide a simple description of the GJK algorithm as a method to construct, using only points on the surface of the Minkowksi object, a simplex containing the origin. A simplex is the smallest number of points for a given dimentions that can "contain" another point. In two dimensions, a triangle is the simplest polygon that contains a point. In 3D, the simplex is a tetrahedron. Our implementation is in 2D, but the algorithm extends easily to 3D.

While it is possible to generate the Minkowski object by iterating over all the points of object A, and for each point, subtracting each of the points in B, in practice this

is never done. Collision detections are able to generate only the next relevant Minkowski Difference vertex directly from A and B using the support function.

4.1.2 The Support Function

The support function is a function that takes a convex hull (or convex polygon) and a support vector representing a direction of search. The function returns the point on the body that is farthest in the sense of a plane sweep using the support vector. In Figure 7, we see in 2D the line sweeping across the polygon, finding point 1 as the last point encountered on the shape. This is the support point for the given vector.

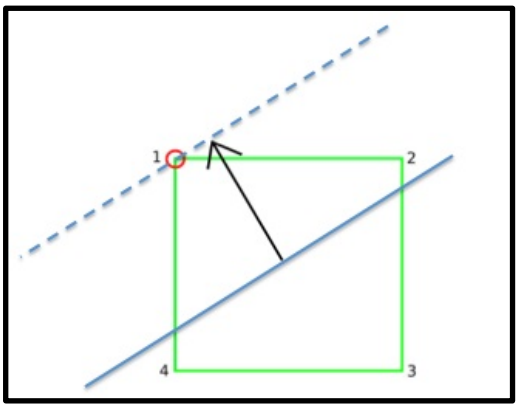

Figure 7. The Support Function

To compute a given search point for the Minkowski object along direction a given direction *d*, we simply use the support function for object A along d and the support function for object B along *-d*:

$$Support(Minkowski, \vec{d}) = Support(A, \vec{d}) - Support(B, -\vec{d})$$

4.1.3 Evolving the Simplex

The core of the GJK algorithm is the iterative evolution of the simplex. If the simplex contains the origin, collision has occurred. If the simplex is fully evolved and does not contain the origin then no collision has occurred. Obviously any real implementation must take in to account floating point issues and tolerance cases, but the simplest version is presented below.

```
bool GJK_contains_origin()
     dir = get_any_direction()
     point = get_support(dir)
     add_point(simplex, point)
     //set search direction to be opposite current support
point
     dir = -point
     Loop:
          point = get_support(dir)
          if point_doesnt_pass_origin(simplex, point, dir) then
               //no collision
               return false
          else
               update_simplex(simplex, point)
               result = check_simplex(simplex, dir)
               if result == CONTAINS_ORIGIN
                    return true
               if result == NO_CONTAINMENT
                    return false
               else
                    dir = update_direction(simplex)
                    GOTO Loop
```

Figure 8. GJK Code Listing

To evolve the simplex we repeatedly choose a new direction to search for the next candidate point to add to the simplex and remove points that do not contribute to our simplex.

4.1.4 Evolving the Simplex: Two Point Simplex and Voronoi Regions

If the simplex is empty the algorithm can start at any search direction so the first point is trivial to compute. The second point chosen using the support function in the opposite direction.

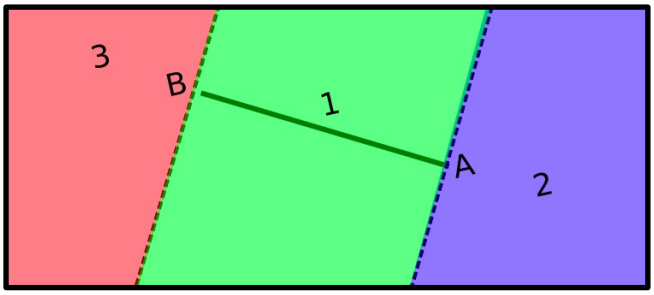

Figure 9. Two Point Simplex (Line)

In general we can think of the simplex evolution as a strategy to divide the plane into searchable regions and then to use simple geometry to determine the next best search area. To readers familiar with Voronoi regions, these pictures should be obvious, but to readers unfamiliar with Voronoi regions a brief explanation should hopefully suffice. For further reading on Voronoi regions the reference (de Berg, 151) provides a simple introduction.

Essentially, each region defines an area of the plane that is “closest to” a given feature (edge or vertex) of the polygon. Our two point simplex in Figure 9 forms a line. We know point A was chosen precisely because it was the farthest point on the Minkowksi in that direction, so there is no point in searching further in region 2. Likewise we can discard Region 3 as we know there is no point “further” in that direction. That leaves the two regions to either side of the line. We are trying to contain the origin, so

we choose the normal to our line that lies in the direction of the region containing the origin and compute a third point.

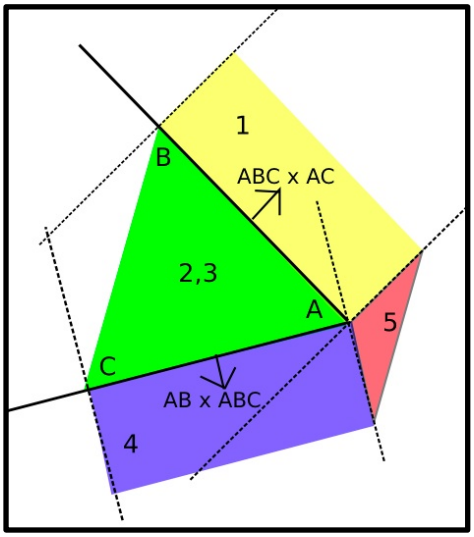

Figure 10. Three Point Simplex (Triangle)

We continue with the same general procedure. Depending on the region, we may contain the origin (in the green region) or we may discard the “non-contributing” point and further evolve the simplex. For example, if the next search region is region 4, we know that vertex B does not contribute to our simplex and it may be discarded. We return to a two-point simplex and continue the algorithm.

In the simplest implementation of GJK, the output is simply a yes or a no. For physics engines, the algorithm must be extended to provide contact position, normal and penetration depth. For this information it is easiest when penetration has not yet occurred, and frequently physics engines will return a contact point when objects enter a “convex radius” which forms a shell around the polygon.

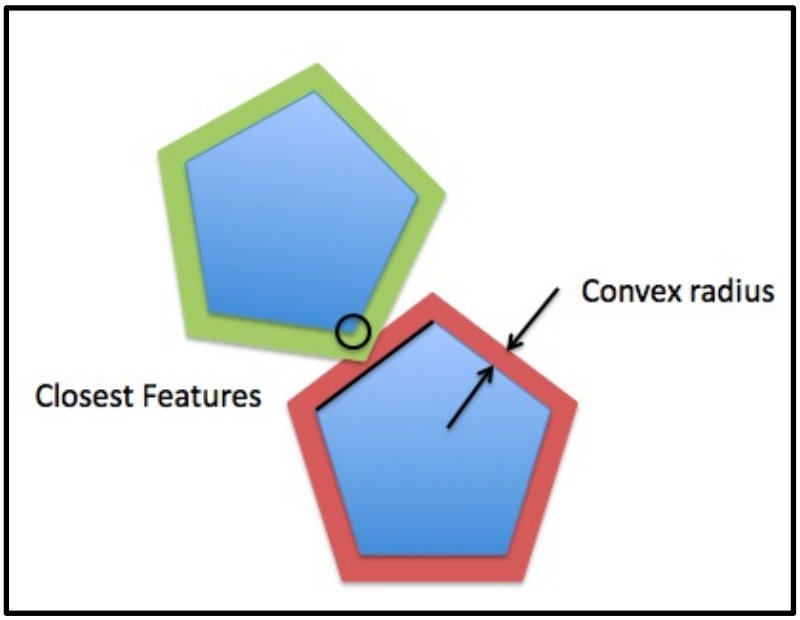

Figure 11. Convex Radius and Closest Features

To acquire this information the simplex is evolved, not until it contains the origin, but until it collapses back to a 1- or 2-point simplex representing the closest pair of features between two objects, such as a point and a line as shown in Figure 11. Although the algorithm becomes more complex, contact points, contact normals, and penetration depths are well defined.

In configurations where deep penetration has occurred another algorithm such as "Expanding Polytope Algorithm" is used to determine the contact information. Although a discussion of EPA is not needed for this paper, it is worth observing that it can be significantly more expensive than GJK and much effort is made to avoid these configurations.

Once contact points are computed correctly, they are used to construct “contact constraints” which apply impulses (instantaneous velocity changes) to the rigid bodies to prevent interpenetration.

4.2. Collision Response

The second component of physics simulation is collision response, where the contact points discovered in collision detection are used to adjust the motion of rigid bodies to maintain the various constraints our simulation places on them. Some background is necessary.

An accurate and complete mathematical description of the motion of a physical object under the influences of various forces is sometimes called “equations of motion”. There are various ways to determine the equations of motion, from the original formulation by Newton, where all forces (both external and internal) are explicitly solved, to Lagrange’s equations, with its concept of zero work constraint forces. Although these methods are all describing the same physical phenomenon and must therefore be equivalent the Lagrangian approach lends itself more readily to computation and simulation.

At the most rudimentary level we model the motion of a body with mass, inertia, and linear and rotational velocities in a series of timesteps. At each timestep, we compute the correct velocity for that integration as the result of the previous frame velocity, the action of external forces, and the action of “internal” constraint forces, i.e.,

forces that do no work and thus add no energy to the system, but enforce the particular requirements of the simulation.

The following is in large part a synopsis of Chris Hecker's articles in Game Developer Magazine (Hecker) and Erin Catto's paper (Catto, 3-14) and his 2008 presentation at the Game Developer's Conference (hereafter "GDC").

4.2.1 Euler Integration of Position

At the top level, we are trying to compute a body's linear and angular velocities for a given timestep. We can use this to update the body's position using Euler integration:

$$P_2 = P_1 + V_2 \Delta t$$

and

$$V_2 = V_1 + a \Delta t$$

We know

$$F = M a$$

therefore,

$$a = M^{-1} F$$

4.2.2 Constraints and the Jacobian

Next we consider a formulation for constraint forces and introduce some important concepts. The math is quite dense but the benefit of transforming the problem

from Cartesian space to “constraint space” is that we can reduce all constraints in a system to a common representation suitable for fast computation.

We describe a constraint as a linear function that constrains the position and motion of a pair of rigid bodies. The position constraint depends solely on configuration:

$$C_k(x_i, q_i, x_j, q_j) = 0$$

and our velocity constraints are written as:

$$\dot{C} = JV = 0$$

We commonly allow a small bias term to allow our constraints to do some work (i.e., adjust the velocity in the direction of the constraint):

$$\dot{C} = JV = \beta$$

In the case of contact constraints, this is called Baumgarte stabilization and can be thought of using the positional error (the penetration depth) to improve our velocity constraint. Further detail is not relevant to this paper.

To provide some context and to cement the notion that the Jacobian is dependent solely on configuration, we present:

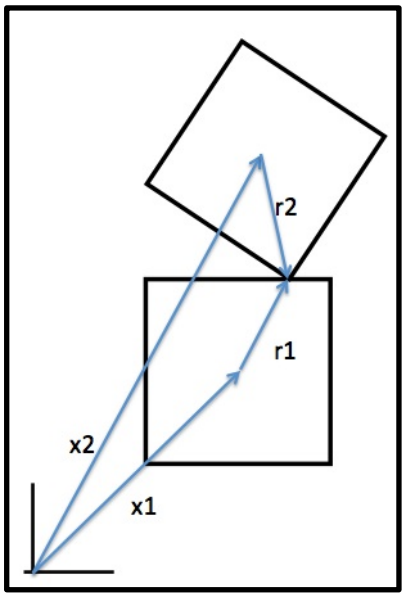

Figure 12. Computing the "No Penetration" Constraint

And state that the normal constraint for a contact point is:

$$C_n = (x_2 + r_2 - x_1 - r_1) \cdot n$$

The above formula describes the contact point from the center of the two bodies. If no penetration occurs, this point should be the same, and this sum should be zero. The error here is the penetration depth. As it is somewhat challenging to isolate the Jacobian we simply present it here without derivation as:

$$J_n = (-n^T \; -(r_1 \times n)^T \; n^T \; (r_2 \times n)^T)$$

We also know that constraint forces must do no work, so this suggests:

$$F_c = J^T \lambda$$

This states our forces must be orthogonal to our velocities. If internal forces acted in the direction of velocity, they would add energy to our system, making it unstable.

The lambda in the above equation is sometimes referred to as a Lagrangian multiplier, and represents the fact that we know the directions of the forces but must still compute their magnitudes. We solve for these multipliers at each timestep in our simulation.

Constraints for friction, motors, rope, rigid links, ragdolls joints, and so on can be described in this form using different Jacobians to map between “Cartesian space” and “constraint space”. Once we are able to express the Jacobian we can compute an impulse that operates on the body to preserve the constraints.

4.2.3 Computing Velocity

With an understanding of Jacobians and lambdas and constraint space, we revisit the our velocity computation:

$$V_2 = V_1 + a\Delta t$$

Where we substitute for our acceleration term and divide the force in to external forces (like gravity) and the constraint forces:

$$V_2 = V_1 + M^{-1}(F_{ext} + F_c)\Delta t$$

We further simplify the problem by computing an intermediate velocity (which violates constraints) as:

$$\tilde{V_2} = V_1 + M^{-1}F_{ext}\Delta t$$

and we describe the final velocity as:

$$V_2 = \tilde{V}_2 + M^{-1}P_c$$

This states the resultant velocity is an intermediate velocity summed with the impulses which can cause instantaneous changes in velocity.

Finally we must compute the impulsive change in velocities due to the constraints as:

$$P_c = J^T\lambda$$

where lambda, the constraint multiplier, is computed as:

$$\lambda = m_c(-JV_1 + b)$$

where b is related to the bias term described earlier and

$$m_c = \frac{1}{JM^{-1}J^T}$$

This may be thought of as the “effective mass” of the bodies.

Although tangential to this paper, it is interesting to note that the effective mass thus calculated includes both linear and rotational mass (i.e., moments of inertia) which has important implications for the stability of constraints. Essentially, bodies with large inertial terms will appear many times more “massive” to their constraints, making it hard for these constraints to stabilize quickly (Strunk).

4.2.4 Iterative and Global Solvers

These bodies may simultaneously on one or more other bodies at each time step. Therefore some strategy is needed to reach a globally correct solution for the set of constraint equations in the simulation. This leads to two different types of solvers: Global solvers, which attempt to solve all constraints simultaneously as a linear complementarity problem, and iterative solvers, which solve each constraint sequentially for multiple passes to allow the system to converge to the globally correct solution (Catto, 15). Iterative solvers, while less accurate, are much faster than global solvers and are commonly used in physics engines.

Although the math is quite dense and care must be taken when formulating this algebra in a manner solvable by a computer, this solver lies at the heart of a constraint based physics system. In addition to the basic collision detection and collision response mechanisms, this research implementation includes a number of extensions (such as Baumgarte Stabilization and lambda clamping) to improve numerical stability, convergence, and performance. These are classical improvements to cure the well-known ills of iterative rigid body solvers.

The final result is “Havok2D” (not to be confused with the Havok Physics Engine), a very simple 2D physics system that performs convex hull collision detection and response with an iterative constraint solver.

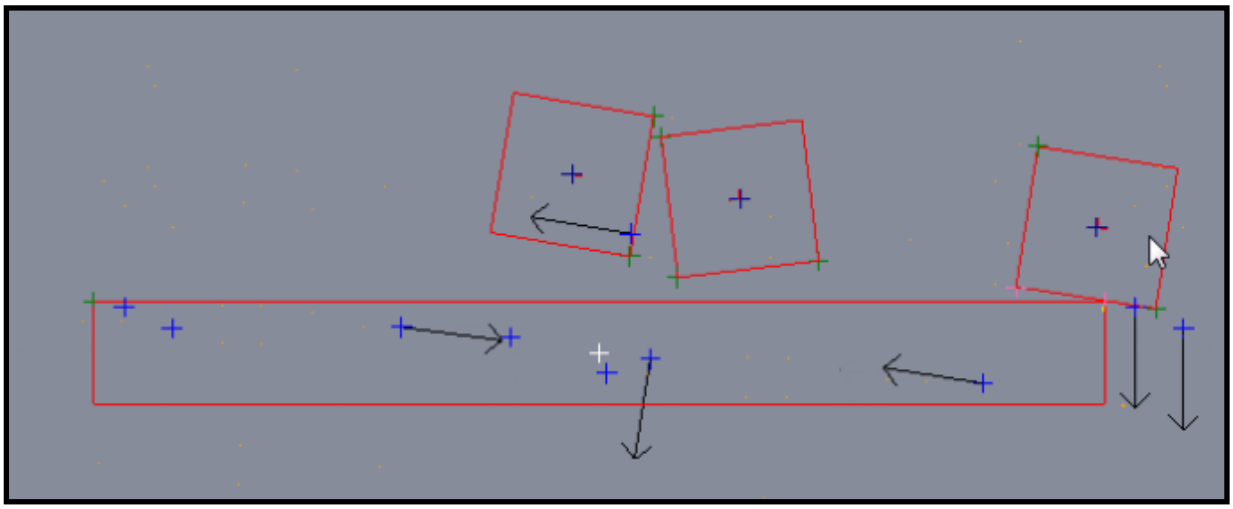

Figure 13. Havok2D, A Simple Physics Engine

5. Box2D

Although Havok2D proved a useful starting point for this research, to better understand the state of the art in physics engines, all further implementation was added to Box2D, a full-featured, open source, optimized physics engine, written by Erin Catto (www.box2d.org). Box2D allows this research to take place in the larger context of a sophisticated physics engine rather than as extensions to a very feature-limited and unsophisticated Havok2D. All further research and discussion takes place in this context.

6. Minkowski Portal Refinement Collision Detection Algorithm

The first component of this thesis is a 2D implementation and analysis of the relatively new collision detection algorithm “Minkowski Portal Refinement”, first implemented and described by Gary Snethen in his paper (Gems, 165–178) and further discussed on his website (http://xenocollide.snethen.com).

6.1 Preliminary Comparison with GJK

GJK, as discussed above, has some drawbacks. The first is that although its description is rather straightforward, its implementation can be fraught with error. In its simplest formulation, it has only two exit cases: The creation of a completely evolved 2-Simplex (triangle) that does or does not contain the origin, or a completely evolved 1- or 0-simplex that does not contain the origin. In practice, the algorithm is generally extended to compute contact point, penetration depth, and contact normal, and thus introduces many more exit conditions. Evolving the simplex is no trivial matter. It requires calculation of Voronoi regions and consideration of many different branch cases in a particular order. A correct implementation is a one-time challenge to a programmer, and once adequately tested, may be used as a black box without further consideration but simpler algorithms are easier to make correct and this difficulty should be noted. Last, GJK fails to provide the normal, depth, and point when penetration actually occurs and engines using GJK must resort to the use of additional algorithms such as EPA to handle this case.

MPR is a collision detection algorithm that supposedly addresses some of the shortcomings of GJK, such as algorithmic complexity and a reduced number of branching tests (Gems, 177). The first component of this research is an attempt to integrate MPR in to the collision detection pipeline of Distance2D (a standalone collision detection demonstration application that uses Box2D's collision detection system) to understand the benefits and drawbacks of MPR as compared to GJK.

Like GJK, MPR uses the Minkowski Difference object implicitly, using the support function for bodies A and B. Instead of searching for the simplex that contains the origin, MPR seeks to refine a “portal” through which a vector starting at some deep “interior point” must pass on its way to the origin.

6.2. The Algorithm

The following pseudocode is reproduced from Snethen’s “XenoCollide” article (168):

```
Bool MPR_contains_origin()
//Phase 1: Portal discovery
(1)    find_origin_ray()
(2)    find_candidate_portal()
(3)    while (origin ray does not intersect candidate)
(4)       choose_new_candidate()

//Phase 2: Portal refinement
       while (true)
(5)        if (origin inside portal) return true
(6)        find_support_in_portal_direction()
(7)        if (origin outside support plane) return false
      //this is a tolerance case
(8)        if (support plane close to portal) return false
(9)        choose_new_portal()
```

Figure 14. MPR Code Listing

Figure 15 below shows the Minkowski difference for the colliding objects, and illustrates in 2D the various phases of collision detection. We can see the origin is inside the difference, so we know this represents a collision between the two shapes.

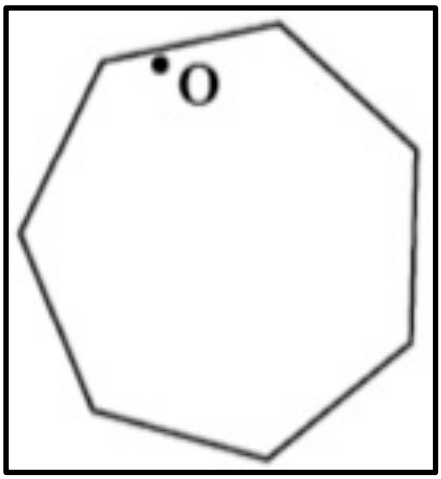

Figure 15. Minkowski Object Contains Origin

Lines (1) find_origin_ray() and (2) find_candidate_portal();

Figure 16 shows the find_origin_ray() and find_candidate_portal() step. First an "interior point" V0 is chosen. The point can be anywhere inside the Minkowski difference but for obvious reasons the geometric center (arithmetic mean of x,y coordinates of vertices) is convenient. The origin ray is the ray from this interior point V0 to O, the origin.

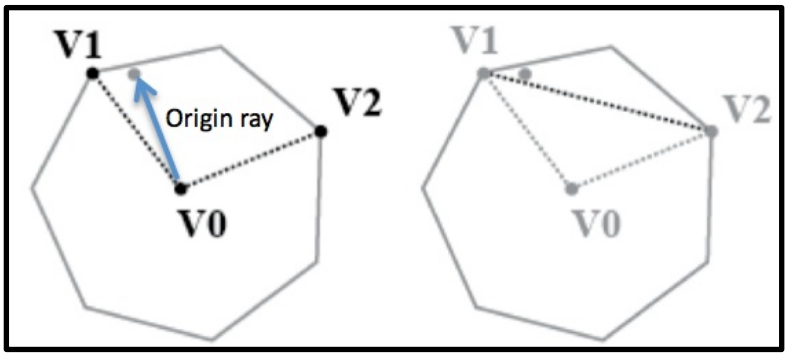

Figure 16. Origin Ray and Candidate Portal

Here, the origin ray is used as the first support direction to discover the first support point V1. Then the normal of vector V0 to V1 is used as a second support direction to

discover the third support point, V2. There are two choices for the normal to the V0V1 vector. The algorithm uses the vector that points towards the origin.

Line (3) while (origin ray does not intersect candidate) { choose_ }

The line segment V1V2 is the “portal” through which the origin ray must pass. We aren’t yet concerned with containing the origin inside an area. We are simply concerned that V0V1 and V0V2 sweep an angle that bound the origin, so we perform two half plane checks. If these pass, we have found a candidate portal and can continue with our algorithm. If not, we can compute a new support vector in the choose_new_candidate() step by choosing a support vector normal to the edge that failed the half plane check.

We now pause to observe Figure 16 and note there are three spaces where the origin can be:

1) the region inside the V0V1V2 triangle,

2) the region inside the Minkowski difference but outside the V0V1V2 triangle,

or

3) within the swept angle V0V1 and V0V2 but outside the Minkowski difference.

6.2.1 Portal Refinement

In the “Portal refinement” stage, we seek to conclusively show that the origin is inside or outside the Minkowski object:

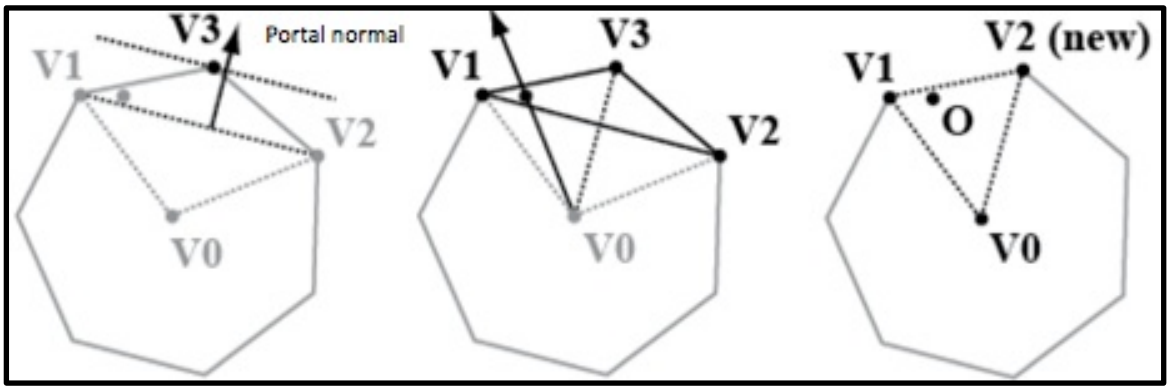

Figure 17. Portal Refinement

Line (5) if (origin inside portal) return hit;

If the current portal contains the origin (via a barycentric coordinate check or three half plane checks) then a collision has been detected and the algorithm can return a hit.

Line (6) find_support_in_portal_direction();

If the algorithm continues on, the portal normal (ie the vector perpendicular to V1V3 and away from V0) is used as a support vector and a new support vertex is computed.

Line (7) if (origin outside support plane) return miss;

If the origin lies farther than the support mapping on this new normal, we know the Minkowski object couldn't possibly contain the origin, as there exists no point on the Minkowski that is "past" the origin that could be part of a containing portal. In the case shown in Figure 17, the origin is inside this line, so the new support mapping is used to refine the portal.

Line (9) choose_new_portal();

We know the points V1, V3, and V2 form a triangle, not necessarily with edges on the Minkowksi object's surface (although in this example it does). We know that the origin ray enters this portal. We determine which edge the origin ray would exit from, either V1V3 or V2V3. We construct the edge V0V3 and we determine which side the origin lies toward (V1V3 or V3V2). We preserve the support point V0, and the support points of the existing edge V1 and V3 to construct a new, refined portal, as shown in Figure 17.

The algorithm now repeats the while (true) loop and hits the exit condition.

6.3 Results

Figure 18 below shows a collision configuration of two bodies, and the process of portal refinement or simplex evolution for these configurations.

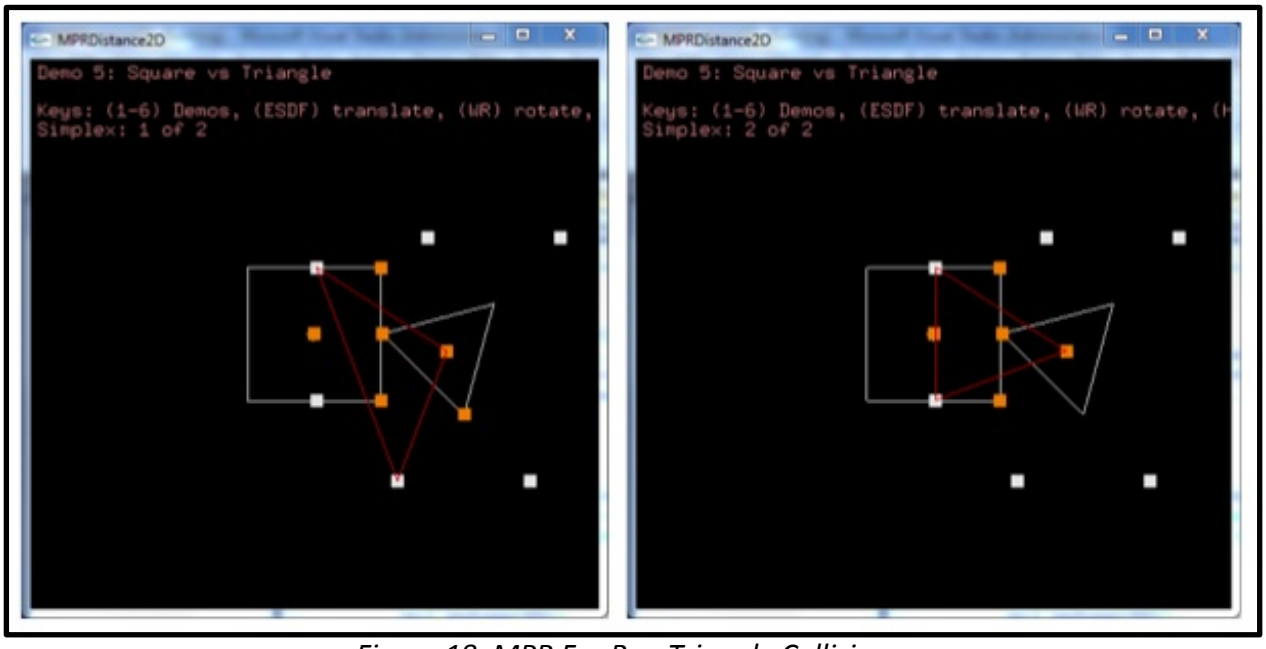

Figure 18. MPR For Box-Triangle Collision

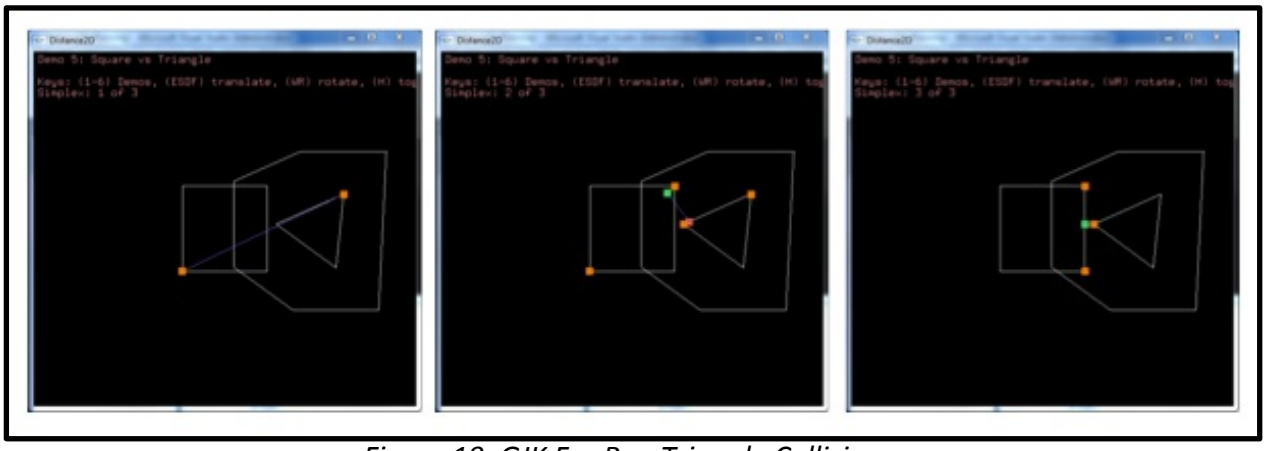

Figure 19. GJK For Box-Triangle Collision

MPR in this configuration provides a slightly faster resolution to finding the nearest points (2 portal refinements) compared with GJK's 3 simplex evolutions. This should not be indicative of the general case, as there are a bewildering set of configurations possible for arbitrary convex hull collisions, but some general observations can be made.

MPR, like GJK, must be modified to provide contact points and normals. Although Snethen provides no such explanation in the paper itself, he outlined his method in a forums response (Snethen).

It proceeds in a similar manner as the modifications to GJK. The algorithm must be modified to further refine the portal until it does not contain the origin but also can not be improved, i.e., that there is no other support vector that produces any point on the Minkowski object that moves the portal closer to the origin. Once a fully refined portal is discovered, the portal normal can then be used to determine, as we did with GJK, a contact normal, a penetration depth (actually, penetration in to the convex radius

of the body, not in to the convex hull interior), and a pair of closest features, suitable for determining contact points.

MPR also collapses when penetration actually occurs. Like GJK, it correctly detects penetration but then must resort to a separate algorithm to find a suitable normal. Snethen instructs the user to use the contact normal discovered above to “re-run” MPR using this normal as the origin ray. This effectively computes the distance of the origin from the Minkowski surface along the contact normal.

Although this doesn’t require implementing an additional algorithm like EPA, it does represent a substantial cost (double the computation per deeply penetrated contact point), it is thus an improvement but not an impressive or persuasive one.

The claim “MPR requires fewer branching tests” is a clear win for MPR. My GJK implementation has seven distinct logic branches, depending on the size (0-, 1-, or 2-simplex) and which Voronoi region is selected. MPR has fewer (and far simpler) branches. It is perhaps unwise to compare two implementations that have not been optimized for efficient pipelining, but the assertion based on comparison of algorithm structure is sound.

Finally, the assertion that MPR is simpler to implement is undoubtedly correct. GJK proceeds by evolving a simplex, of which any of three points can be rejected at any stage, resulting in a simplex of changing size and vertex order throughout. This introduces some algorithmic complexity, and is the cause of many entertaining and hard to find bugs. MPR works by only choosing between either one of two possible choices for

further portal refinement, corresponding to which fraction of the portal the origin ray passes through. Although one-time implementation costs are generally less important, the graphical, either-or simplicity of MPR is a notable improvement over the more algebraic formulation of GJK.

7. Speculative Contacts

The second component of this thesis is an attempt to replace Box2D's classic continuous collision detection scheme incorporate with an increasingly popular approximation technique called speculative contacts (Firth).

7.1 Classical Continuous Collision Detection

The motivation for continuous collision detection is often framed as the "bullet through paper" problem – that is, the problem of simulating continuous motion with discrete timesteps.

A physically correct interaction might look like this:

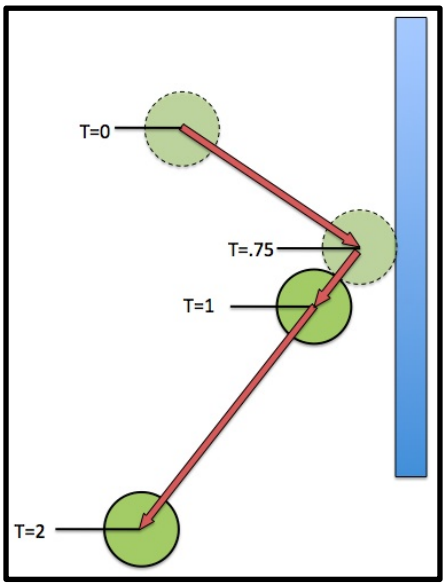

Figure 20. Correct Physical Simulation of "Bullet through Paper"

The transitory interaction between the green “bullet” and the blue wall is detected and handled correctly. The bullet’s motion is reflected in a purely elastic collision at T=0.75, and ends up in its correct position at T=1.

In a purely discrete simulation the bullet would tunnel, integrating through the wall from a valid non-overlapping position at T=0 to a valid non-overlapping position at T=1. A human observer knows that this is incorrect because bodies must sweep, rather than teleport, to a new position, but this human knowledge is not honored in a discrete simulation. This incorrect behavior is shown in Figure 21:

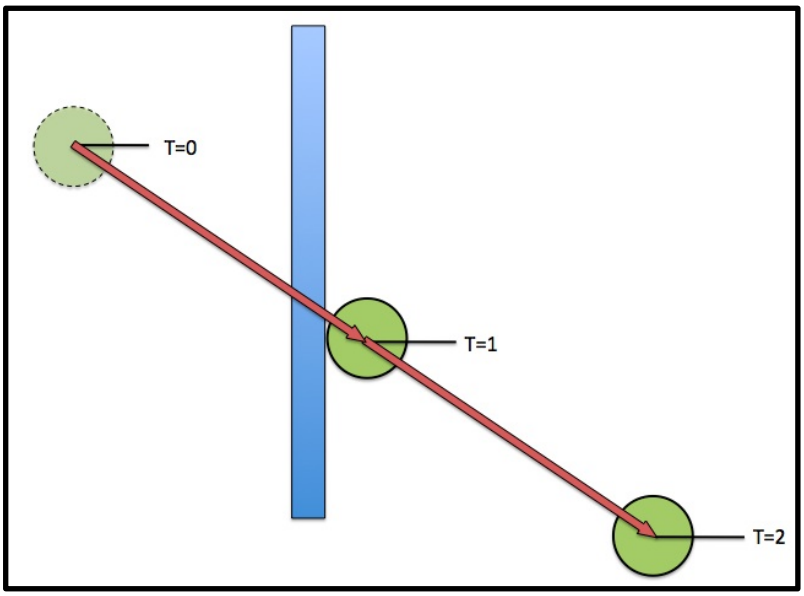

Figure 21. Tunneling: Incorrect Physical Simulation of “Bullet through Paper”

At the start of the timestep (T=0) there is no penetration so narrowphase collision detection creates no contact points. Thus there is nothing to prevent the ball from passing through the wall during this timestep.

The challenge of continuous collision detection is to detect when tunneling has occurred and to properly determine the precise time of impact, or “TOI”. In order to understand the performance improvements offered by speculative contacts I present a brief outline of continuous collision detection.

The current approach may be characterized as improvements on a technique called “Conservative Advancement” first proposed by Brian Mirtich in his PhD dissertation (Mirtich). A very readable introduction is presented in Catto’s 2013 GDC presentation (Catto). The basic idea is to use a collision detection algorithm to detect closest features between two convex hulls, generate a polynomial function as a (conservative) measurement of their separation and then apply a root-solving algorithm to it to iteratively advance to successive “safe” (non-penetrating) positions to within some tolerance of impact. This collision is then resolved.

This approach turns the TOI detection problem in to the well understood problems of collision detection and root finding, both of which have been subject to intensive optimization (Wikipedia).

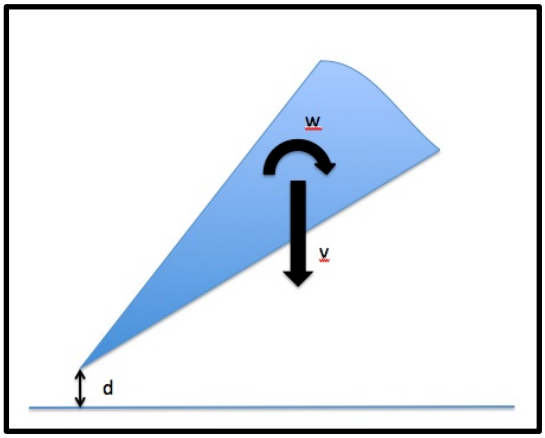

Figure 22. Conservative Advancement

In Figure 22, we see that “d” the distance between two objects is found using GJK. We then use our root solver to find a safe substep to advance the simulation. We then call GJK and determine if we are within our “contact tolerance”. If not, we again call GJK to determine closest features and take another safe substep. Gradually we are able to advance the body to within some tolerance of the fractional time of impact and correctly compute the body’s motion.

The problem is that for certain configurations this approach can result in many calls to the GJK routines and to the root-finding algorithm. This is prohibitively expensive for even a single contact point, let alone the many contact points in a busy simulation.

Additionally, correctly resolving a TOI may result in new collisions which invalidate other integrated body positions. This forces the system to move back and forth between collision detection and solving, rather than a single pass to completion. An example of this is shown below:

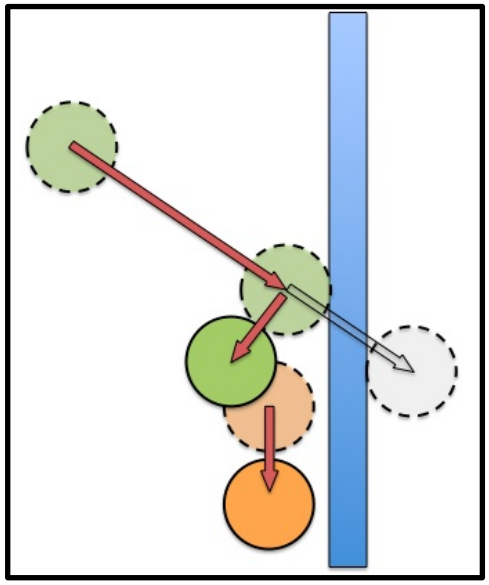

Figure 23. TOI Resolution Creating New TOIs

In the figure, the discretely solved position of the green ball is corrected by the TOI solver. This corrected position invalidates our result for orange ball which now also experiences a TOI collision with the green ball that must in turn be resolved.

This knock-on effect is an additional and serious drawback to this approach and motivates the following observation: For correct simulation, TOI events must be handled in sequence and in a singlethreaded fashion. Each TOI can in turn create or eliminate subsequent collision events in a given frame. This means that, unlike many aspects of physics simulation, TOI solving is a singlethreaded bottleneck (Firth).

7.2 Speculative Contacts

Because of the performance drawbacks of continuous collision detection, there is increasing interest in the very useful continuous collision approximation called speculative contacts. This simple approximation subtracts the component of the velocity that would cause penetration:

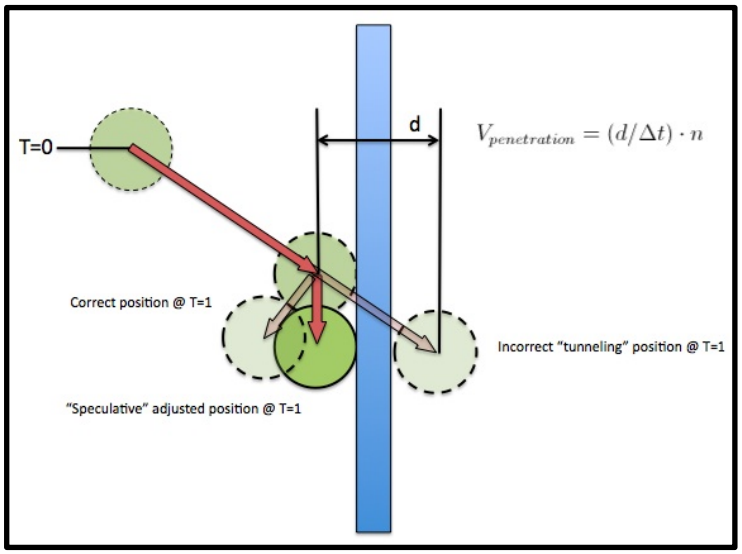

Figure 24. Speculative Contacts Velocity Subtraction

This approximation has a number of consequences. First, the simulation can settle much faster (Firth). Consider the case where two separate contact constraints are acting in opposition, as shown below:

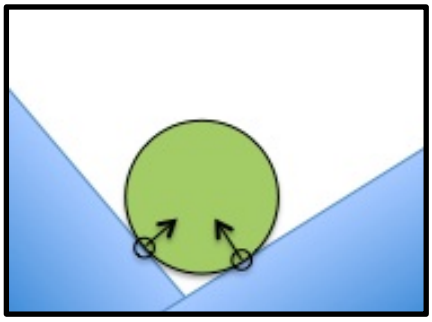

Figure 25. Speculative Contacts Improve Settling Time

The solver must iterate over these two constraints, first driving the ball's position away from one wall, which leads to greater penetration on the other wall. Then the ball's position is corrected back the other way. Ultimately, this takes many iterations of the solver before the "globally correct" solution can emerge, that is, where the ball is penetrating neither wall. These iterations take time.

With the speculative contacts approach, each contact constraint would need to be solved fewer times to reach a stable state. The ball would be detected as penetrating the first wall and this penetration velocity would be removed. The ball would then be detected as penetrating the other wall. This component of its velocity would also be removed. Thus in a single round, the horizontal velocity has been almost entirely removed and the vertical component of velocity that causes penetration would also be removed, resulting in a stable configuration.

This has important performance implications for current and next-generation systems, where much of the speed gains have been through increasing numbers of cores and better multithreaded execution rather than faster monocore designs. It is relevant to note that the performance specifications for the upcoming PS4 suggest a CPU with a slow clock speed but multiple cores (https://us.playstation.com/ps4/).

The first and most important drawback is that this approximation creates incorrect physics. For comparison, the correct position is the elastically reflected position at time T=1. The speculative contact subtracts only the velocity causing the penetration, so the body is still integrated through the full timestep in the non-penetrating direction. This "discards" time in the direction of penetration.

For videogames, this approximation is generally acceptable. Bodies falling out of the world or passing through walls is obviously incorrect, but a body slowing down slightly at a wall briefly is generally unnoticeable, especially when a single frame is only $1/30^{th}$ of a second.

The second important drawback is what is often called "ghost contacts" that is, speculative contact constraints in the simulation that subtract penetrating velocity, but in actuality, should not alter a body's velocity. This can be seen when a fast moving object is moving obliquely with respect to a wall:

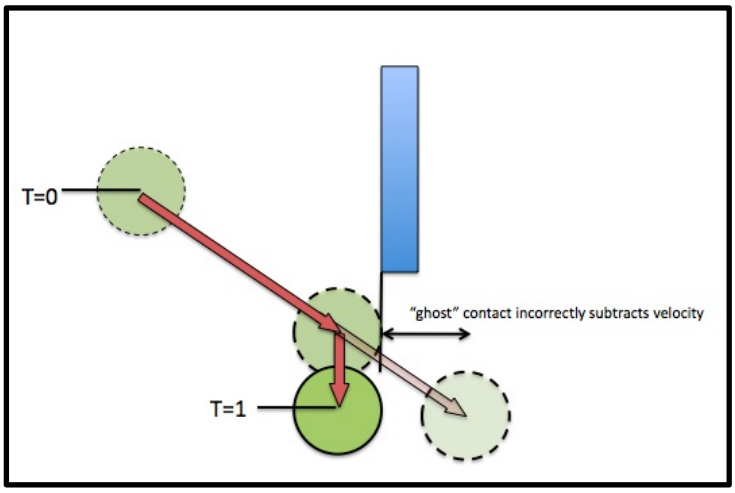

Figure 26. Ghost Contact

Because the ball is moving fast enough to pass the wall in a single timestep, its AABB must overlap the wall and create a contact. This contact detects that there is a substantial negative component of the velocity in this round, so it deletes the component of the velocity that would cause penetration. From observing Figure 26 it is clear the body's unaltered motion would not cause a penetration but its velocity has been altered anyhow.

7.3. Implementation

In principle, adding speculative contacts to Box2D is easy but a number of different subsystems have to be changed in order to make this behavior possible.

7.3.1 Broadphase

Of fundamental importance is that contact points are generated "earlier" than in a typical physics solver. The usual approach is that contact points are checked for penetration based on the configuration of the bodies at the start of the timestep. However, with speculative contacts we cannot allow penetration or tunneling, so we

must generate contacts early, prior to penetration. We do this by forcing narrowphase checks ahead of time by expanding a body's AABBs by its velocity. Box2D, while not using speculative contacts, already artificially expands AABBs by more than their velocities. This allows us to perform narrowphase checks on bodies that have not yet collided but will collide during the next few timesteps. We can use this information to do a tentative integration and remove the penetrating velocity. In Figure 27 below we see the purple AABB around both the moving ball and the stationary box. This also shows that although the bodies will not interact on this frame (nor the next, actually) the contact point has already been added to the solver.

While reusing Box2D's expanded AABBs was a useful shortcut, it does cause subsequent problems. The AABB should be expanded by the velocity of the body so that it only picks up contact points that will be created during the next frame. Box2D expands its AABBs by a different metric and thus creates quite large AABBs, which means more contact points are generated and care must be taken to compute when a contact should be considered active.

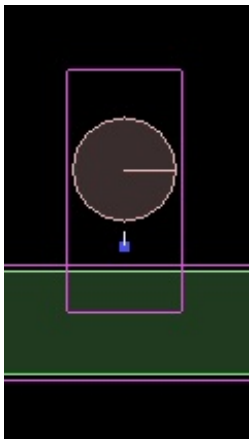

Figure 27. Velocity Expanded AABB

7.3.2 Narrowphase

The second modification is that the collision detection subroutines (called Contacts in Box2D and Collision Agents in Havok Physics) must be modified to prevent early “outs” when no penetration at the start of the time step is detected. The easiest example is a sphere falling towards a floor:

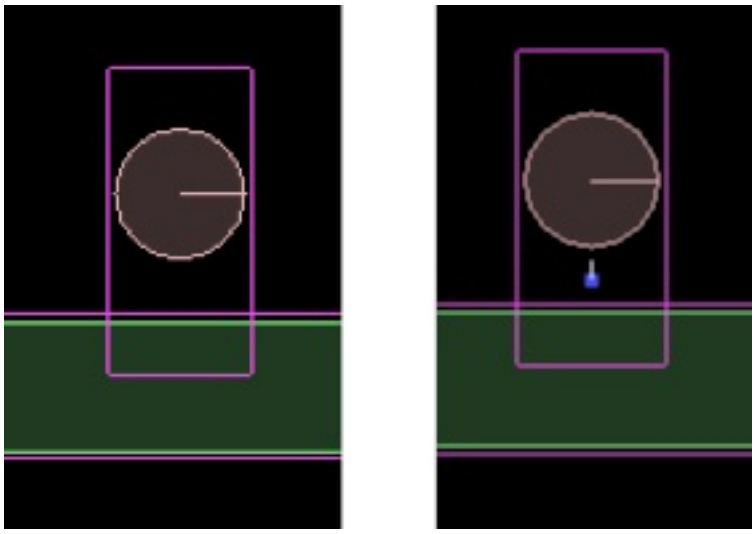

Figure 28. Narrowphase Changes For Speculative Contacts

The image on the left shows AABB overlap but no contact point. Indeed, in unmodified Box2D, no contact point will be created until the bodies penetrate. In the modified Box2D shown on the right the speculative contact has already been added to the solver, though we must add some logic to prevent this constraint from being solved too early, that is, prior to actual collision.

For simplicity’s sake, this narrowphase modification was performed only on the circle-polygon contact handler while the other issues with speculative contacts were addressed.

7.3.3 Solver

Once the contact point has been generated the solver must be modified to take advantage of this new information.

The preliminary implementation does the following: Using the existing Box2D integration routines, the body is integrated to a tentative (potentially violating) new position. After this tentative integration, the contact point's position is evaluated to determine the amount of penetration *d* (see Figure 24).

If *d* is positive, no penetration occurs during the timestep (i.e., the speculative contact is still inactive). This is used to notify the solver to ignore this contact point for the remaining step, a decision that leads to further complications downstream.

If *d* is negative, this penetration velocity is removed from the body velocity at the start of the solve step. This is used to notify the solver that the contact point should be considered. The position and velocity constraint created by the contact point is then computed as usual.

7.4 Results

7.4.1 Speculative contacts in action

In Figure 29 below we see the three different collision responses to the exact same initial conditions. Discrete penetration on the left allows the ball to deeply penetrate. If the block had been very thin this would likely have resulted in tunneling and the ball being ejected from the underside rather than merely corrected over several

frames. The middle shows continuous collision detection, which provides for correct collision detection and response but also requires several calls to GJK and to the root solver. It's hard to visualize transitory responses in a single screenshot but the continuous image shows no contact point precisely because it has been handled as a TOI. This means in a single frame the ball has collided, created a contact point, and resolved collision away from the floor. The rightmost image shows the collision response for speculative contacts. The ball does not penetrate. The ball's velocity is subtracted, and the ball rebounds at a reduced speed.

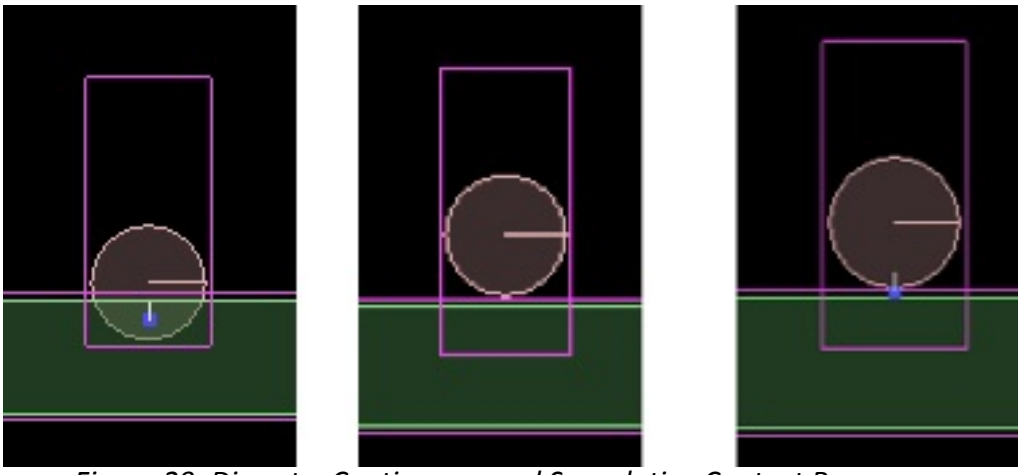

Figure 29. Discrete, Continuous, and Speculative Contact Response

Paul Firth's implementation removed all velocity in the penetrating direction and he subsequently handles the collision in a purely inelastic fashion (no bouncing response). The Box2D modifications handle the constraint as a regular constraint but using the subtracted velocity so the response may be somewhat "less" elastic without being completely inelastic.

7.4.2 Disabling contact points

One problematic behavior observed was that contact constraints generally represent two constraints. The first is the normal constraint that prevents interpenetration and the second is a tangential constraint that relates the sliding motion of two bodies in contact. The solver modification to disable the contact constraint when the penetration velocity is 0 means that bodies are moved in to contact during the step when their penetration would have occurred, but if they have no further downward velocity, the body appears to glide along the surface of the other, not penetrating but not interacting with friction. Clearly a more sophisticated heuristic about when a contact point is active is needed. This likely requires some integration of what was previously considered "collision detection" code in to the solver itself so that the solver can decide which speculative contact points should be handled in the solver for a given timestep.

7.4.3 Warm starting

An additional unforeseen problem is that speculative contact disabling doesn't work well with warm starting, an optimization used to speed up the convergence of an iterative solver.

"Warm starting" is the practice of caching the previous frame's net impulse applied to each body. These are used to initialize the impulse for the next frame. As the frames are *coherent* (very similar) the impulses applied next frame will be very similar to the impulses applied in the last frame. This allows the solver to reach the globally correct

solution in fewer iterations, speeding up the timestep. This had an unfortunate consequence for the speculative contact modifications.

The ball's response is to accelerate away from the contact. This is because the contact point is active on the frame for which penetration would occur. This contact point becomes active; the penetrating velocity is removed and the contact constraint is solved. The contact is subsequently disabled on all successive frames because it is no longer active. The correct impulse to apply for that contact constraint on the next frame should be zero but the solver is unable to re-solve this constraint and drive the multiplier to zero so this impulse stays nonzero, effectively accelerating the ball until the expanded AABBs separate, thus removing the contact constraint from the simulation.

The simplest solution is to disable warm starting globally. This has a detrimental impact on the overall performance and a more refined solution would reenable warm starting but re-zero the impulse for that contact constraint while the contact still exists but is no longer actively solved.

8. Concluding Remarks

There are many important areas for future development in rigid body simulation. They remain a fundamental component of videogames which have strict requirements for fast and believable simulation. Useful approximations like speculative contacts will help overcome the current singlethreaded bottlenecks but additional behavior must be added to correct the physical inaccuracies caused by this approach.

Faster collision detection algorithms remain an active area of study. MPR leverages the useful Minkowski object and is far simpler to implement than GJK but does nothing to address the real criticism of GJK for use in physics engines.

Physical simulation, especially faster, performance-driven rigid body simulation, is an exciting area of computer science, where new algorithms and performance improvements are needed and actively developed. This paper has only briefly touched on two areas, but there are many other areas to explore and where contributions can be made.

9. Bibliography

Eberly, David M. “Game Physics”, 2nd Edition. New York: Morgan Kaufman, 2010. Print.

Hecker, Chris. “Physics, Part 1: The Next Frontier”. *Game Developer Magazine*. Oct/Nov 1996. Print.

Hecker, Chris. “Physics, Part 2: Angular Effects”. Game Developer Magazine. Dec/Jan 1996. Print.

Hecker, Chris. “Physics, Part 3: Collision Response”. Game Developer Magazine. Feb/Mar 1997. Print.

Hecker, Chris. “Physics, Part 4: The Third Dimension”. Game Developer Magazine. June 1997. Print.

Catto, Erin. GDC_2008. www.box2d.org. Feb 2010. Web.

Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/Euler_method. April 2013. Web.

Newth, Joshua. “Multicore Architectures: Parallelism Problems and Software Solutions”. San Jose State University, 2010. Print.

Catto, Erin. “Computing Distance with GJK”. Game Developer’s Conference. San Francisco, California, March 2010. Presentation.

Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/Minkowski_addition. April 2013. Web.

Physics2D.com “GJK Algorithm” http://physics2d.com/content/gjk-algorithm, May 2013.

de Berg, Mark, et al. “Computational Geometry: Algorithms and Applications”, Third Edition. Berlin: Springer 2008. Print.

Catto, Erin. “Iterative Dynamics with Temporal Coherence”. Crystal Dynamics, Menlo Park, California, 2005. Print.

Strunk, Oliver. “Stop My Constraints From Blowing Up!”, Game Developer’s Conference. San Francisco, California, 2013.

Snethen, Gary. “XenoCollide: Complex Collision Made Simple”. Game Programming Gems 7. Ed. Scott Jacobs. Boston: Course Technology 2008. Print.

Snethen, Gary. XenoCollide. http://xenocollide.snethen.com, May 2013. Web.

Snethen, Gary. “Re: Minkowski Portal Refinement (MPR) for 2D”. http://www.bulletphysics.org/Bullet/phpBB3/viewtopic.php?f=6&t=1964&start=15. Physics Simulation Forum. Web.

Firth, Paul. *Speculative Contacts – A Continuous Collision Approach* . 2011. http://www.wildbunny.co.uk/blog/2011/03/25/speculative-contacts-an-continuous-collision-engine-approach-part-1/. May 2013. Web.

Brian Mirtich “Impulse Based Dynamic Simulation of Rigid Body Systems”. Dissertation, University of California Berkeley, 1996. Print.

Catto, Erin. “Continuous Collision Detection”. Game Developer’s Conference, San Francisco, California. March 2013. Presentation.

Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/Root-finding_algorithm. April 2013. Web.

“PS4 See The Future, Specifications PS4”, https://us.playstation.com/ps4/, May 2013. Web.